- Why SOLID Principles?
- Single Responsibility Principle
- Open Close Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

## Why SOLID?

- Recall the aim of learning LLD.
- We want to make the code to be easier to understand, easier to extend and maintain.
- SOLID is a set of 5 principles, which if followed can give us the desired qualities in the code.

## Single Responsibility Principle

- Recall what we did to make the flying behaviour of hen different from that of the eagle.(Approach 1)
- We had multiple if/else statements.
- Because of which time and effort to test and extend the code were increased.
- This happened because every method is responsible for multiple types of behaviours.
- SRP says that a class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.
- A class should have only one reason to change.
- Having sub-classes for every bird is following the SRP.
- Create a design for this business requirement

```
#HRM Business Requirement Doc
#Version 1
- Create a HR management system to keep track of employees.
- System should support full time employees and interns.
- Implement save() method to save the employee data in the file system.
```

- Approach 1

```
Class Employee{
String name, email;
int id;
void save(){
// Serialize the object
// Open a file
// Write in the file
// Close the file
}
}

Employee e = new
Intern();
e.save();
```

```
Class FTE extend Employee{
}
```

```
Class Intern extend
Employee{
}
```

- What if we want to change the format in which we are writing in the file? OR What if tomorrow we want to move to sql?
- In all these cases Employee class needs to be modified and tested violating SRP.

- Quick checks to find violations of SRP
  - Multiple if/else statements: If/else statements correspond to different behaviour.
  - Unspecified Util/Helper class

```
Class Util{
void rupeeToDolar(amount){}
int roundOffDouble(double){}
int calculateIncomeTax(Employee e){}
String toString(Object){}
}
```

  - Monster methods:

```
int getIncome(){
//Generate payslip
// Convert PS to JSON
// Mail PS to the employee
return this.income.
}
```

## Open Close Principle

- Again recall the Bird example approach 1 of having different flying behaviour.
- In that approach if we need to extend the system by adding a new type of bird, we need to modify all the existing methods (by adding a new if/else in them).
- This increases testing efforts and chances of breaking already existing code.
- Open close principle says that software entities  should be open for extension but closed for modification.

- Add the following requirement to the system

```
#HRM Business Requirement Doc
#Version 2
   -  Add functionality to calculate tax of the employees.
   -  Income tax: 20% of income + Professional tax 2% of income
```

- Should we add calculateTax() method to the Employee class?
- What if the tax rules change ?
- Better to have a separate class for TaxCalculationUti with calculate(Employee e) method.

- Next requirement

```
#HRM Business Requirement Doc
#Version 3
   -  For FTE
      Income tax: 30% of income + Professional tax 2% of income
   -  For Intern
      Income tax: 15% of income (no professional tax)
```

- Approach 1: Have if/else in the calculate(e) methods.
    - Violates SRP
    - What if a new type of employee is added in future (Ex. Contract based employee)
- Better way would be to have an abstract TaxCalculationUti class with abstract method calculate(e);

- Concrete child classes FTETaxCalculationUtil and InterTaxCalculationUtil will have the implementation based on rules.
- Pros:
    - This follows SRP.
    - Adding a new employee means adding a new TaxCalculation class without modifying any existing code. => OCP followed.

## Liskov Substitution Principle

- Recall from the bird example, the example of Kiwi class.
- If Bird is an abstract class with abstract fly() method what will be the implementation of fly() method in the Kiwi class?
- Kiwi can not fly hence should not implement the fly() method.
- LSP says that “Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program”
- Which means that whatever the parent class can do, the subclass must be able to do that.

## Interface Segregation Principle

- Recall the Bird example once again when we had objects other than birds which can fly (Superman, mig-21 etc).
- We created an interface flyable. Any class that wants to support the feature of flying can implement this interface.
- Which all methods should be there in this interface?
- Fly(); flapWings(); takeOff()?
- SuperMan and mig-21 do not flap wings before taking off.
- So they will have to implement a method which they do not perform.
- The interface segregation principle says that “no client should be forced to depend on methods it does not use”
- "Many client-specific interfaces are better than one general-purpose interface."

- Can we have an Employee interface for all common methods related to the employees?
    - getEmail()
    - processPayment()
    - getSalary()
- Yes, if all the employee sub classes can perform these methods.

- Assume that we want to add unpaid interns to the system.
- If the UnpaidIntern class implements the employee interface then it will have to implement the processPayment() method as well which violates the ISP.
- Better solution is to keep thin interfaces. A payable interface will do the job here.

## Dependency Inversion Principle

- Assume that you are working in an e-commerce company.
- And you have a SqlProductRepo class for every db operations (following SRP).

```
Class SqlProductRepo{
    public Product getProductById(String id){
      //create connection with DB and fetch product
    }
}
```

- You want to add the PaymentProcessor class to implement the pay method which will take the product id as input.

```
Class PaymentProcessor{
    void Pay(String productId){
      SqlProductRepo repo = new SqlProductRepo();
      Product P = repo.getProductById(productId);
      //Process payment for this product
   }
}
```

- Tomorrow if we want to switch to Mongo then we will have to change the PaymentProcessor class as well.
- Currently, we have  a higher level module (PP) which is depending on a lower level module (SPR).
- We should avoid creating objects of concrete classes inside other classes.
- Dependency inversion principle says that one should "depend upon abstractions, [not] concretions”
- So we can create an abstraction of product repo

```
Interface ProductRepo{
   Product getProductById(String productId);
}
```

- Now, instead of creating the object of product repo inside the PP class, we will **inject** the dependency through the constructor. And in the parameters we will pass an object of the ProductRepo (which is an abstraction).

```
Class PaymentProcessor{
      ProductRepo repo;
      public PaymentProcessor(ProductRepo repo){
            this.repo = repo;
      }

      void Pay(String productId){
                  Product P = repo.getProductById(productId);
                  //Process payment for this product
       }
}
```

- Now, if we want to move to Mongo, then we just need to create another concrete implementation of the ProductRepo class (MongoProductRepo) and pass its object in the constructor of the PP.
- The code is now easy to extend and also follows the SRP.