7__23-07-2022

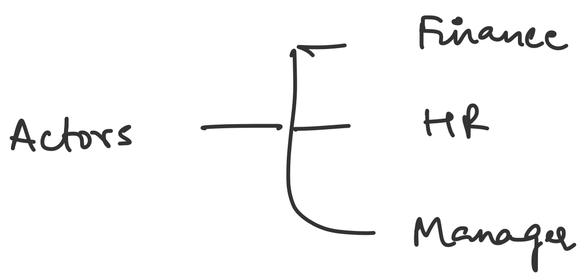
Saturday, 23 July 2022 8:04 PM

S - Single Responsibility Principle

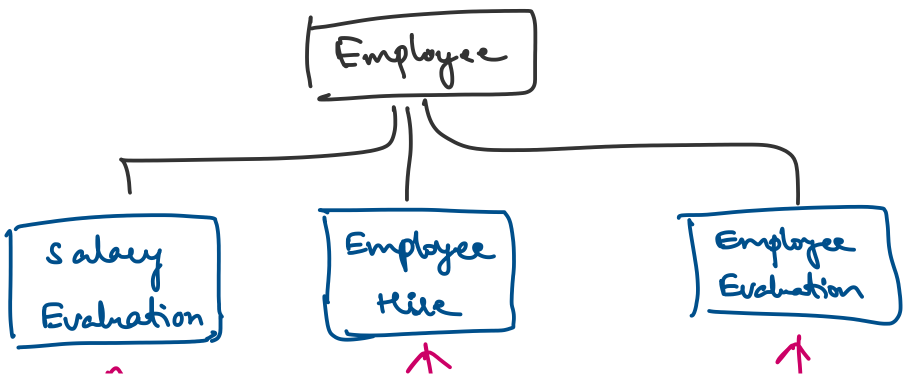
```
class Employee
{
    fn Employee Details()
    {
        =
    }                          } → Common for al

    fn calculate Salary ()
    {
        =
    }                          } → Finance Team

    fn Hire Employee ()
    {
        =
    }                          } → HR Team

    fn Evaluate Employee()
    {
        =                      } → Manager
    }
}
```

SRP

- One actor per class
- There should be only one reason to make change, irrespective of how many methods do we have.



A class should do only one thing.



$$\underbrace{\text{Finance} \qquad \text{HR} \qquad \text{Manager}}_{\text{SRP followed here}}$$

```java
public class Employee {
   String name;
   int empId;

   public Employee(String name, int empId)
   {
      this.name = name;
      this.empId = empId;
   }
}
```

```java
public class SalaryCalculator {
   public void calculateSalary(Employee emp)
   {
      System.out.println("Calculate employee salary.");
   }
}
```

```java
public class EmployeeEvaluator {
   public void evaluateEmployees(Employee emp)
   {
      System.out.println("Evaluate employees.");
   }
}
```

```
public class EmployeeHiring {
   public void hireEmployees(Employee emp)
   {
      System.out.println("Hire employees.");
   }
}



public class Program {

   public static void main(String[] args) {
      Employee emp = new Employee("abc", 101);

      SalaryCalculator sc = new SalaryCalculator();
      sc.calculateSalary(emp);

      EmployeeHiring eh = new EmployeeHiring();
      eh.hireEmployees(emp);

      EmployeeEvaluator ee = new EmployeeEvaluator();
      ee.evaluateEmployees(emp);
   }
}
```

Output:

```
Calculate employee salary.
Hire employees.
Evaluate employees.
```

<u>SRP</u>

↳ One class should change only if it is one instructions per entity (or actor).

↳ If class has more than one actors then break them into total number of actors.

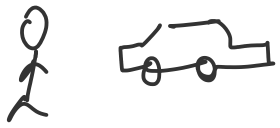
3 actors → 3 classes.

↳ SRP says – Every software module should have only one reason to change. This means that every class, or similar structure in your code should have only one job to do.

Everything in that class should be related to a single purpose. It does not mean that your classes should only contain one method, there may be many methods as long as they are related to single responsibility.

Relationship possession

Girl – I have this boy.



Person has a car.

2 classes ⟨ C1, C2

Class C_1 has a C_2

↳ C_1 possesses C_2 ✓

↳ C_1 contains C_2 ✓

Is it is false that means there is

no relationship.

```
class Person
{
    string name;
    int age
    car car;
}
```

```
class Car
{
    string model;
    String name;
}
```

Person $\boxed{\text{has a}}$ car.

↓

If it is true then person will compose car.

Another eg -

```
class Student
{
    string name;
```

```
class Address
{
    string city;
    String gender;
    Address address;
}
```

```
    String pincode;
}
```

Student $\boxed{\text{has an}}$ address.

is- A → inheritance

has- A → composition

when you inherit your parent's property then that is called as inheritance.

when you try to change your future by working hard then that is called as composition.

$$\begin{tikzcd}
 & \text{Composition} \arrow[dl] \arrow[dr] & \\
\text{Aggregation} & & \text{Association}
\end{tikzcd}$$

[illegible]

Association

```
class HumanBody        |   class Heart
{                      |   {
      =                |        =
}                      |   }
```

Human body $\boxed{\text{has a}}$ heart.

$\downarrow$

composition

We cannot have a heart without a human body, so this type of relationship is called as association.

Association — { one to one
one to many

- many to one
- Many to many

Two flows for association

- → unidirectional
- → Bidirectional

Aggregation

```
class College          |   class Professor
{                      |   {
                       |
    ==                 |       ==
                       |
}                      |   }
```

College $\boxed{\text{has a}}$ professor.

Even though there is composition but they are not dependent on each other, [illegible] relationship is called so such type of

as aggregation.

O – open closed principle (OSP)

$$\begin{tikzcd} & \text{open for enhacements | features} \\ \text{s/w app} \arrow[ur, dash] \arrow[dr, dash] & \\ & \text{closed for code changes} \end{tikzcd}$$

This principle says that our code should be open for enhancement or feature addition, but at the same time for making that change it should not be impacting your existing code, you should be doing minimal change but it should not change the entire implementation.

Appⁿ — Animal Feeder

↳ Feeding dogs

```
class Animal feeder
{
    public void feedDog ()
    {
        S.O.P. (" Feed Dog");
    }

    public void feed Cat ()
    {
        S.O.P. ("Feed cat");
    }
}
```

(+) Feeding cats as well

(+) Provide support for all of the animals.

Problem

- ↳ violating SRP
- ↳ violating OCP

Soln

- ↳ To provide interface support.

```
public interface IAnimal
{
    public void feed();
}
```

Now every animal should be a child for this interface.

```
public interface IAnimal {
   public void feed();
}



public class Cat implements IAnimal {
   @Override
   public void feed() {
      System.out.println("Feeding cats");
   }
}



public class Dog implements IAnimal {

   @Override
   public void feed() {
      System.out.println("Feeding dogs");
   }
}



public class AnimalFeeder {
   public void feedAnimal(IAnimal animal)
   {
      animal.feed();
   }
}



public class Program {

   public static void main(String[] args) {
      AnimalFeeder animalFeeder = new AnimalFeeder();
      animalFeeder.feedAnimal(new Dog());
      animalFeeder.feedAnimal(new Cat());
   }
}
```

Output:

```
Feeding dogs
Feeding cats
```

OCP says - " A software module|class is open for extension and closed for modifications".

Here " open for extension" means we need to design our module|class in such a way that the new functionality can be added only when the new requirements are generated.

" closed for modification" means we have already developed a class and it has gone through unit testing then we should not alter it until we find bugs.

As a class it should be open for extensions and we can use inheritance to do this.