SOLID Principles

Saturday, 15 July, 2023 06:16 PM

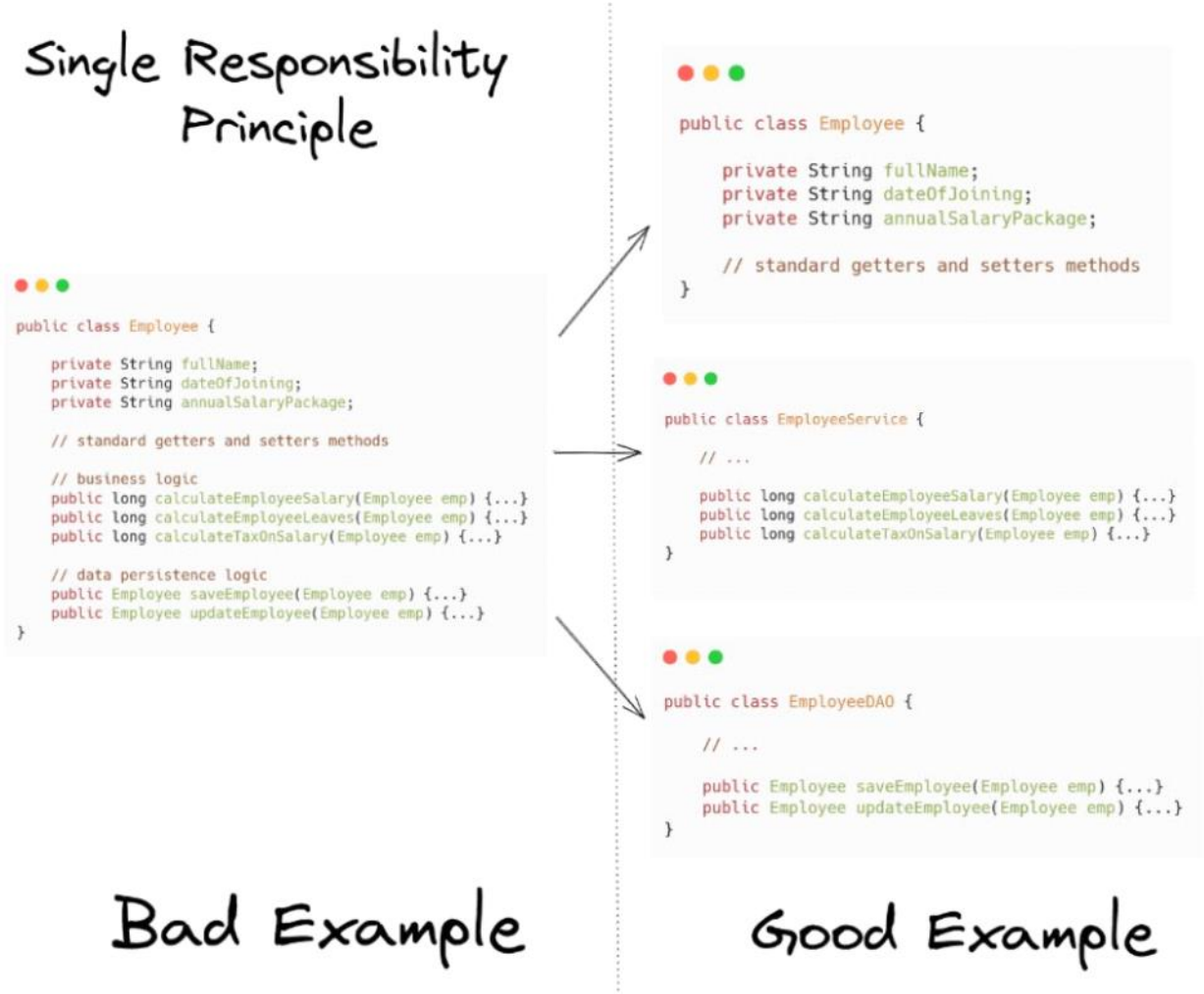
© RAJAT KUMAR
https://www.linkedin.com/in/imRajat/
https://github.com/im-Rajat

Reference : https://twitter.com/vikasrajputin/status/1593460494886576128

S = Single Responsibility Principle
O = Open/Closed Principle
L = Liskov Substitution Principle
I = Interface Segregation Principle
D = Dependency Inversion Principle

Single Responsibility

- A class should always have one responsibility and there should be only a single reason to change it.
- Don't make your class tightly coupled, hard to maintain, multiple reasons to modify this class.
- Made your class loosely coupled, easy to maintain, and only single reason to modify.



Open Close

- Class should be Open for Extension but Closed for Modification.

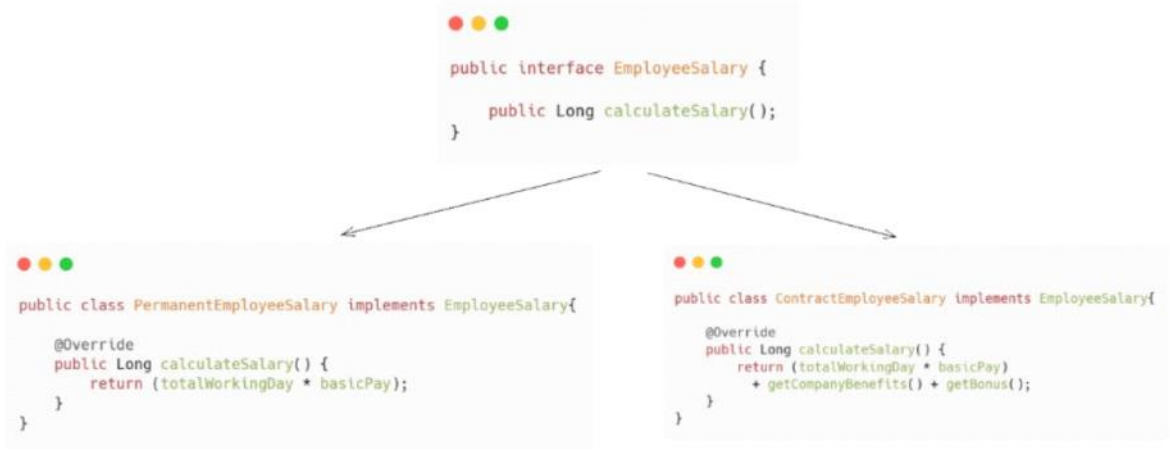
Bad Implementation

- Below EmployeeSalary class calculates salary based on employee type: Permanent and Contractual.
- Issue: In the future, if a new type(Part-time Employee) comes then the code needs to be modified to calculate the salary based on employee type.

```
public class EmployeeSalary {

    public Long calculateSalary(Employee emp) {
        Long salary = null;

        if (emp.getType().equals("PERMANENT")) {

          salary = (totalWorkingDay * basicPay) + getCompanyBenefits() + getBonus();
        } else if (emp.getType().equals("CONTRACT")) {

          salary = (totalWorkingDay * basicPay);
        }
        return salary;
    }
}
```

Good Implementation:

- We can introduce a new interface EmployeeSalary and create two child classes for Permanent and Contractual Employees.
- By doing this, when a new type comes then a new child class needs to be created and our core logic will also not change from this.



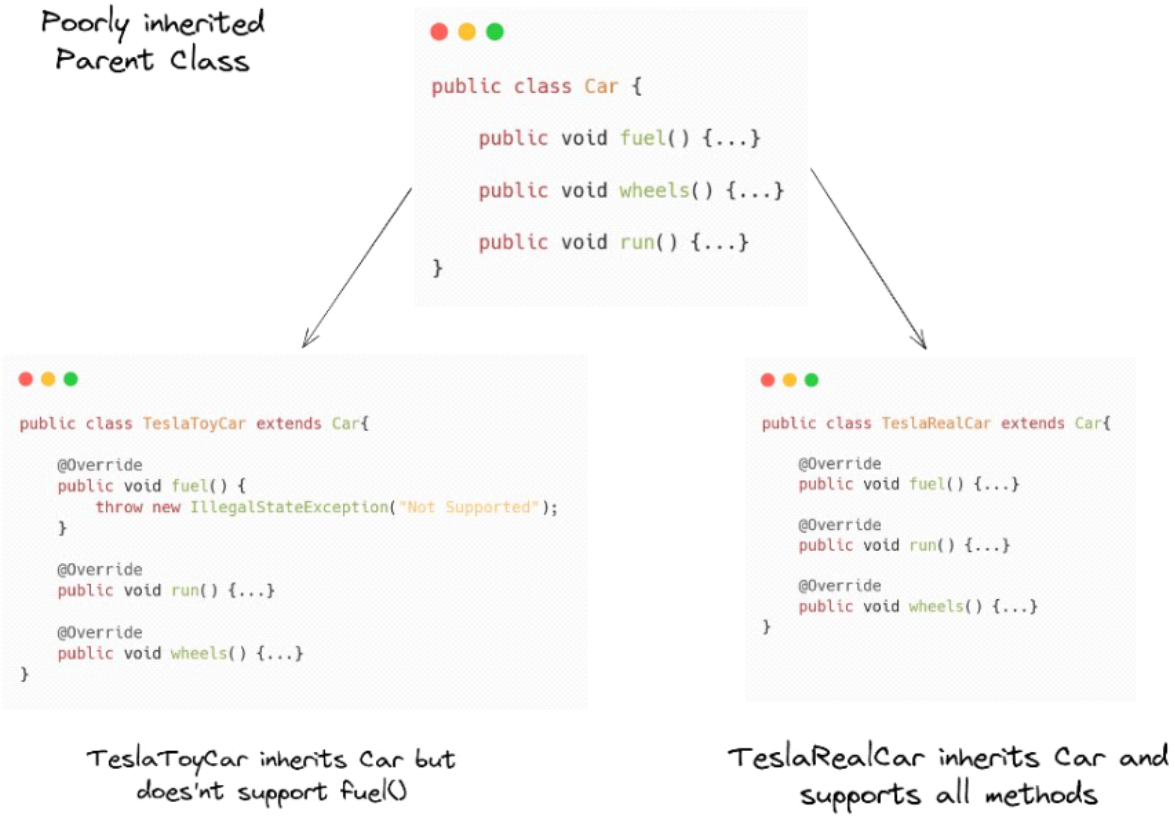
Open - Close Principle

Liskov Substitution

- Child Classes should be replaceable with Parent Classes without breaking the behaviour of our code.

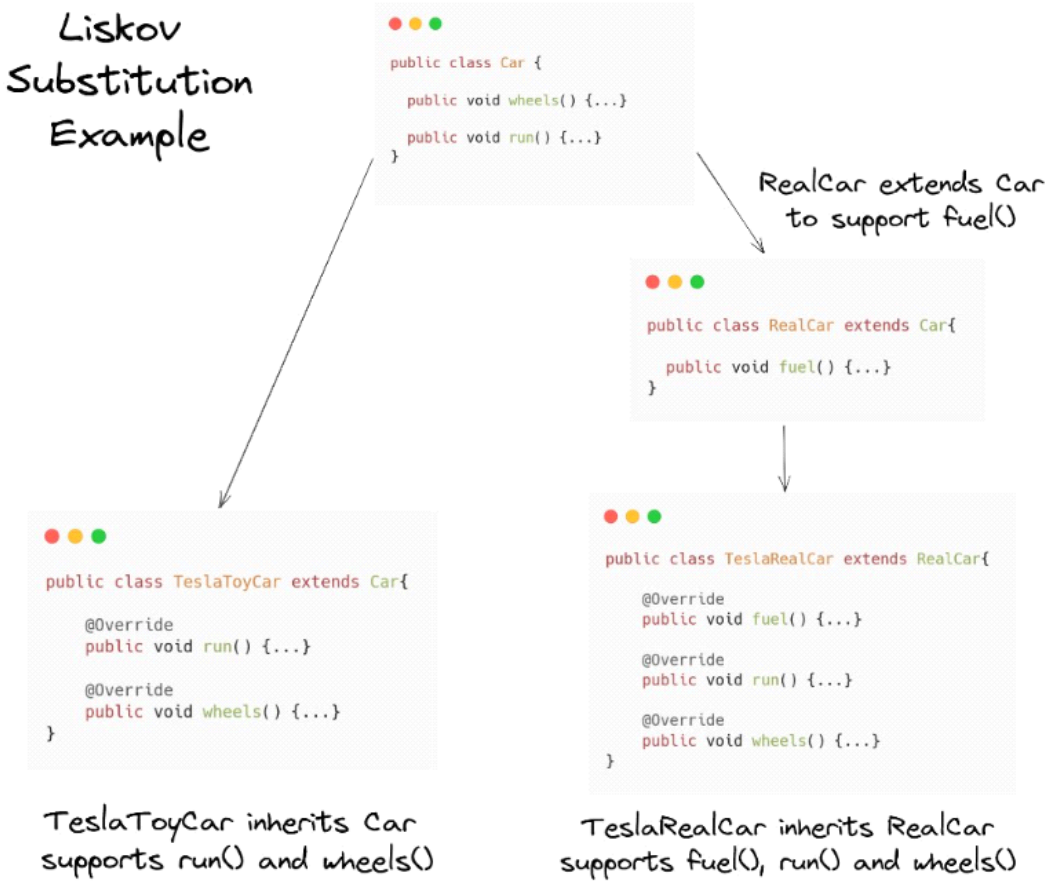
Bad Implementation

- Below, TeslaToyCar extends Car but does not support fuel() method as its toy. That's why it's violating the LS principle.
- In our code where ever we've used Car, we can't substitute it directly with TeslaToyCar because fuel() will throw Exception.



Good Implementation

- Creating new subclass RealCar from parent Car class, so that RealCar can support fuel() and Car can support generic functions support by any type of car.
- As shown below, TeslaToyCar and TeslaRealCar can be substituted with their respective Parent class.

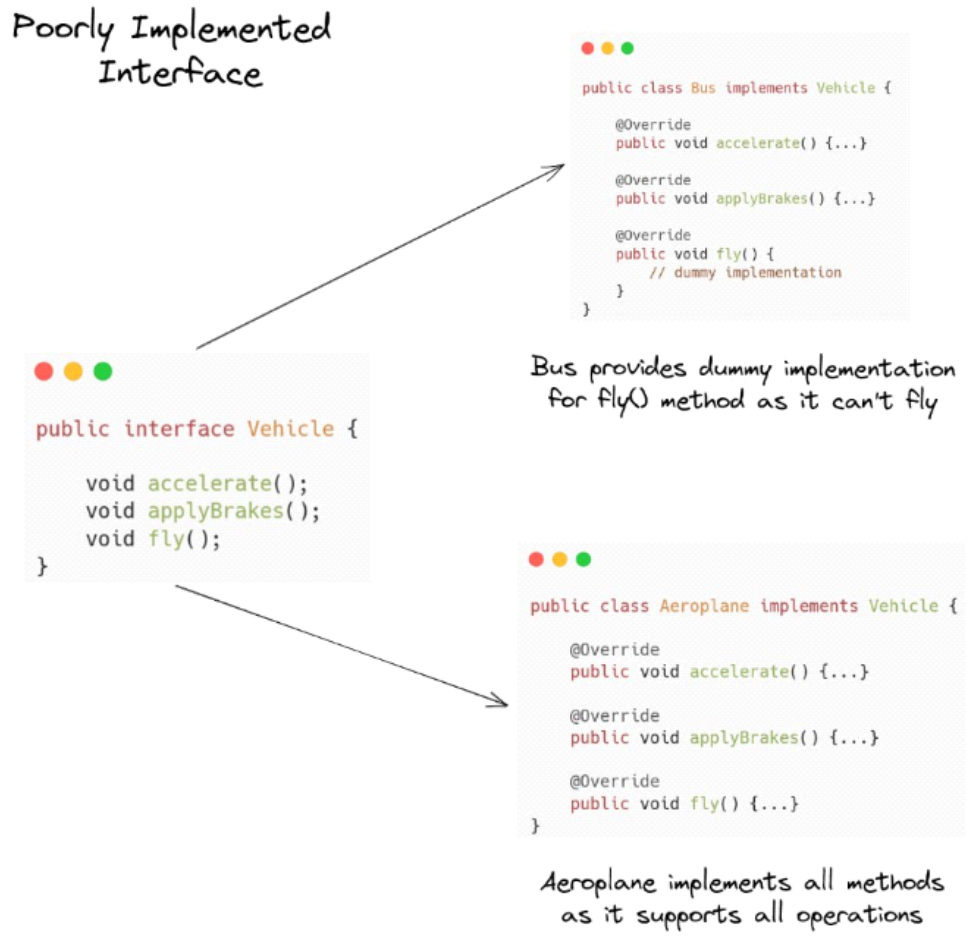


Interface Segregation:

- Interface should only have methods that are applicable to all child classes.
- If an interface contains a method applicable to some child classes then we need to force the rest to provide dummy implementation.
- Move such methods to a new interface.

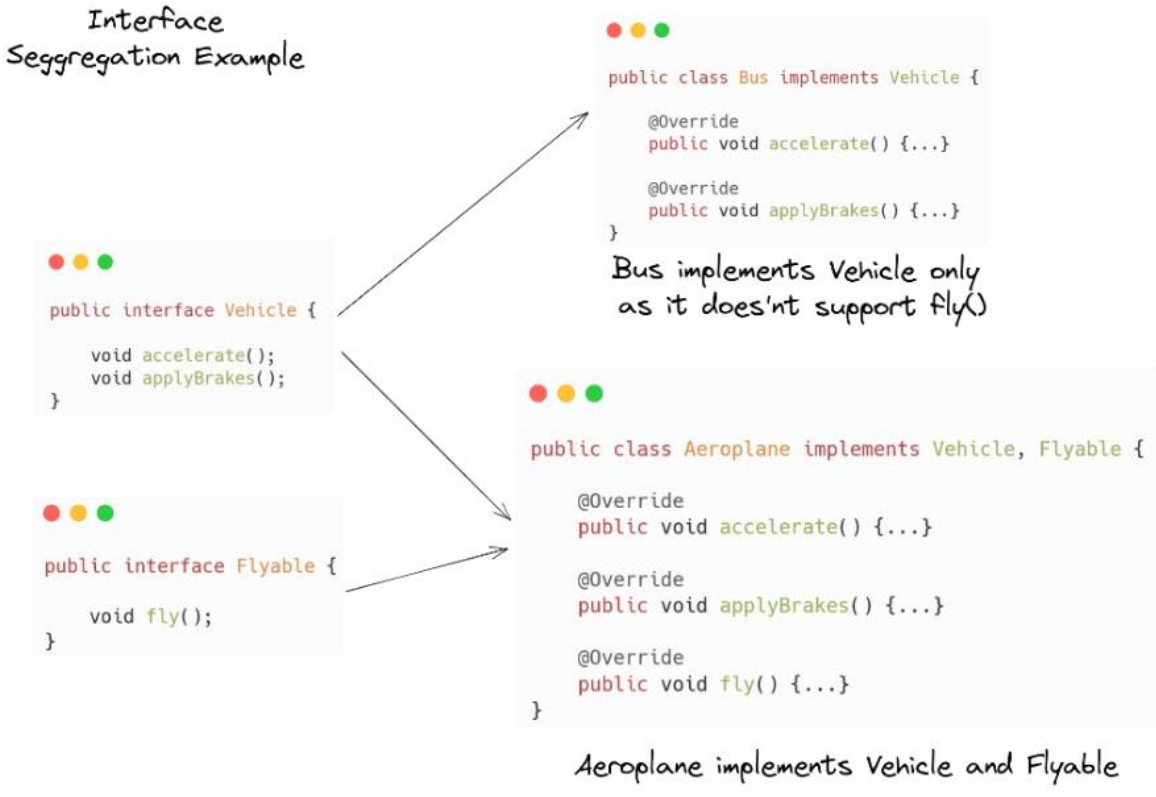
Bad Implementation:

- Vehicle interface contains the fly() method which is not supported by all vehicles i.e. Bus, Car, etc. Hence they've to forcefully provide a dummy implementation.
- It violates the Interface Segregation principle as shown below:



Good Implementation:

- Pulling out fly() method into new Flyable interface solves the issue.
- Now, Vehicle interface contains methods supported by all Vehicles.
- And, Aeroplane implements both Vehicle and Flyable interface as it can fly too.



Dependency Inversion

- Class should depend on abstractions (interface and abstract class) instead of concrete implementations.
- It makes our classes de-coupled with each other.
- If implementation changes then the class referring to it via abstraction won't change.

Bad Implementation

- We've got a Service class, in which we've directly referenced concrete class(SQLRepository).
- Issue: Our class is now tightly coupled with SQLRepository, in future if we need to start supporting NoSQLRepository then we need to change Service class.

```
class SQLRepository{
  public void save() {...}
}

class NoSQLRepository{
    public void save() {...}
}

public class Service {

    //Here we've hard-coded SQLRepository
    //in-future if we need to support NoSQLRepository
    //then we need to modify our code

    private SQLRepository repository = new SQLRepository();

    public void save() {
        repository.save();
    }
}
```

Good Implementation

- Create a parent interface Repository and SQL and NoSQL Repository implements it.
- Service class refers to Repository interface, in future if we need to support NoSQL then simply need to pass its instance in constructor without changing Service class.

```java
interface Repository{
    void save();
}

class SQLRepository implements Repository{
    @Override
    public void save() {...}
}

class NoSQLRepository implements Repository{
    @Override
    public void save() {...}
}

public class Service {
    private Repository repository;

    //Here we're using interface as reference
    //not the concrete class so our code
    //can easily support other child classes
    //of the same interface.
    //For eg: NoSQLRepository class

    public Service(Repository repository) {
        this.repository = repository;
    }

    public void save() {
        repository.save();
    }
}
```