Simplest Guide To Master SOLID Principles

Abbreviation

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

Let’s understand this more with Java examples

Single Responsibility

A class should always have one responsibility and there should be only a single reason to change it.

Bad Implementation

Below Employee class contains personal details, business logic to perform a few calculations, and DB logic to save/update.

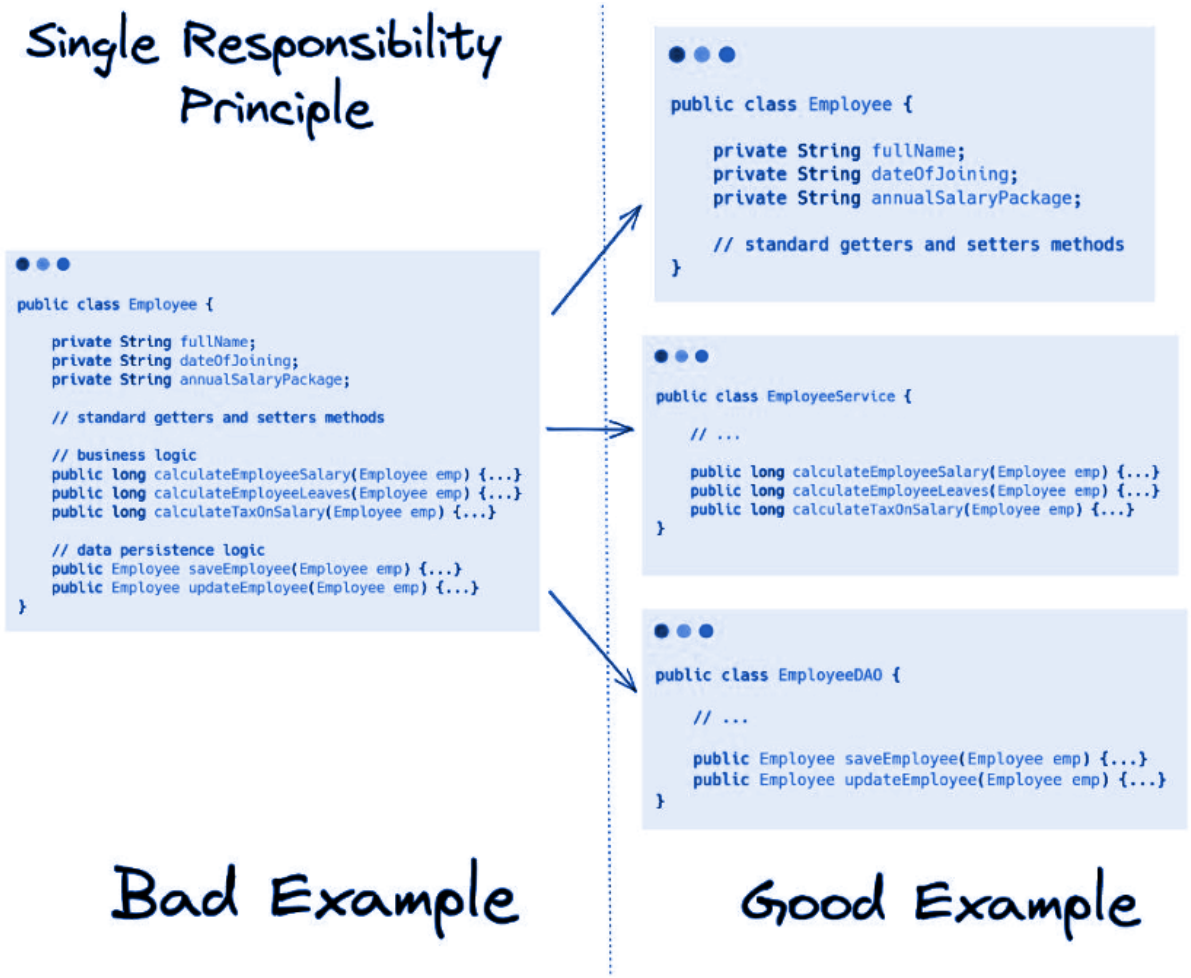
Our class is tightly coupled, hard to maintain, multiple reasons to modify this class.

```
public class Employee {

    private String fullName;
    private String dateOfJoining;
    private String annualSalaryPackage;

    // standard getters and setters methods

    // business logic
    public long calculateEmployeeSalary(Employee emp) {...}
    public long calculateEmployeeLeaves(Employee emp) {...}
    public long calculateTaxOnSalary(Employee emp) {...}

    // data persistence logic
    public Employee saveEmployee(Employee emp) {...}
    public Employee updateEmployee(Employee emp) {...}
}
```

Good Implementation

We can split a single Employee class into multiple classes as per their specific responsibility.

It made our class loosely coupled, easy to maintain, and only a single reason to modify.



Open Close

Class should be Open for Extension but Closed for Modification.

Bad Implementation

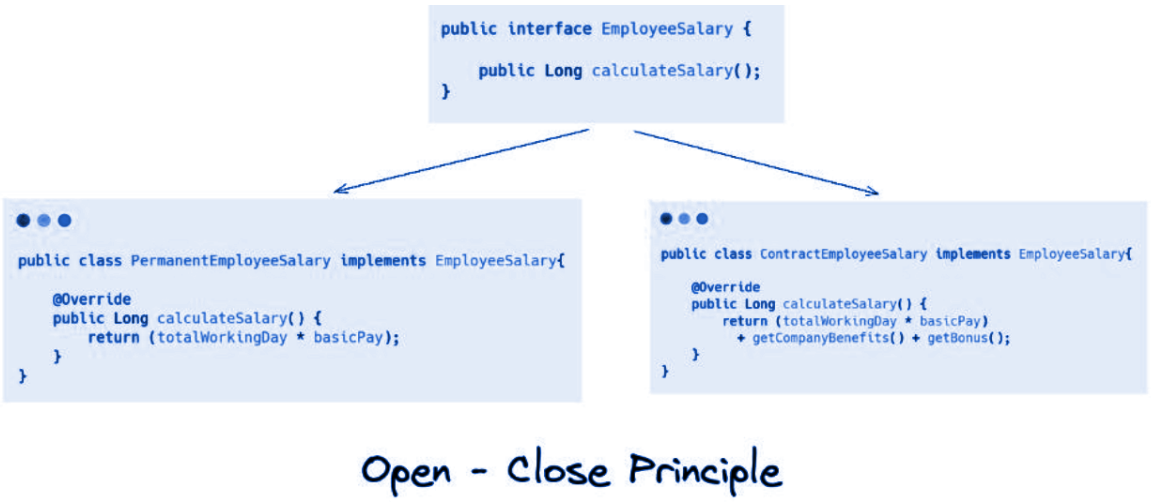
Below EmployeeSalary class calculates salary based on employee type: Permanent and Contractual. Issue: In the future, if a new type(Part-time Employee) comes then the code needs to be modified to calculate the salary based on employee type.

```java
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.



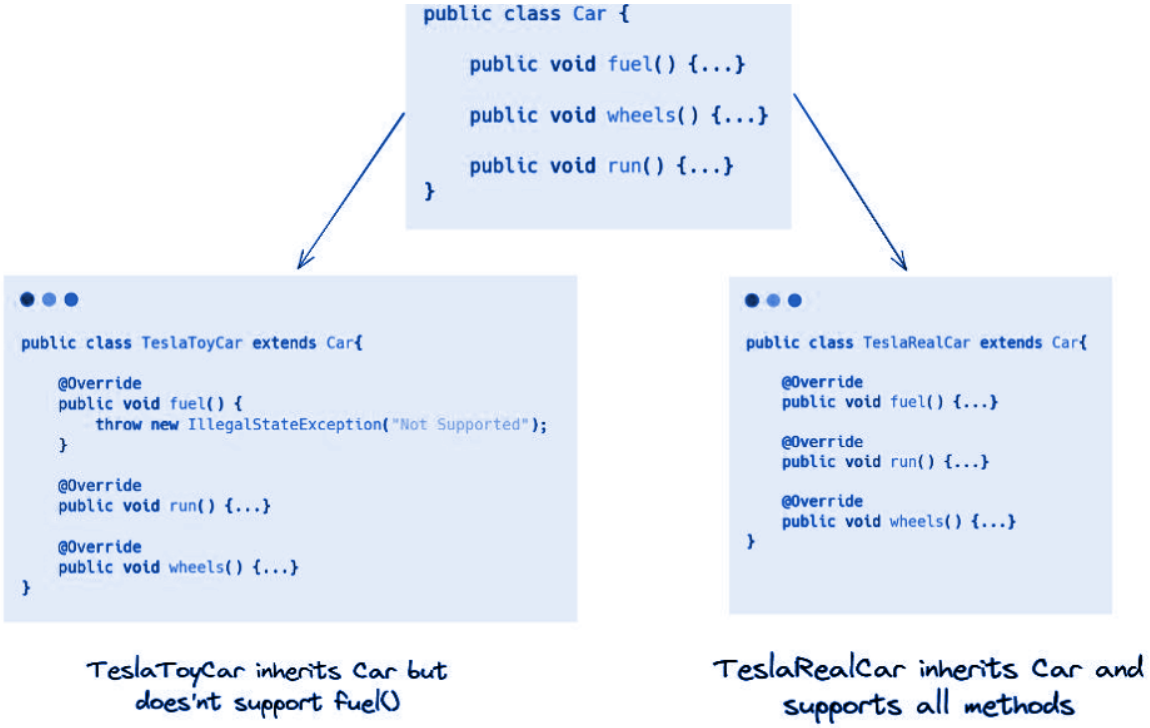
Liskov Substitution

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

Bad Implementation

Below, TeslaToyCar extends Car but does not support the fuel() method as its toy. That's why it's violating the LS principle.

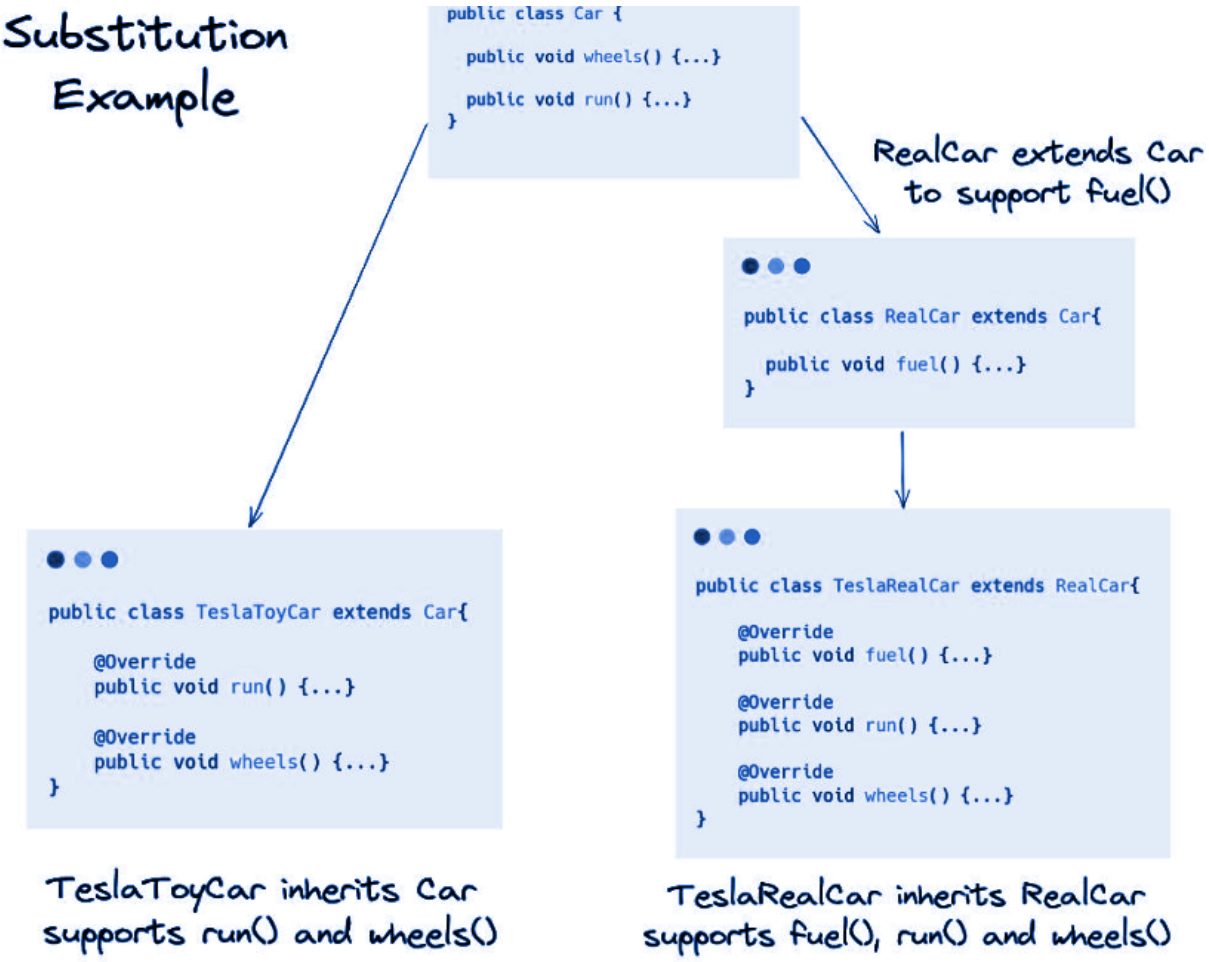
In our code wherever we've used Car, we can't substitute it directly with TeslaToyCar because fuel() will throw an Exception.



Good Implementation

Creating a new subclass RealCar from the parent Car class, so that RealCar can support fuel() and Car can support generic functions supported by any type of car.

As shown below, TeslaToyCar and TeslaRealCar can be substituted with their respective Parent class.



Interface Segregation

Interfaces should only have methods that are applicable to all child classes.

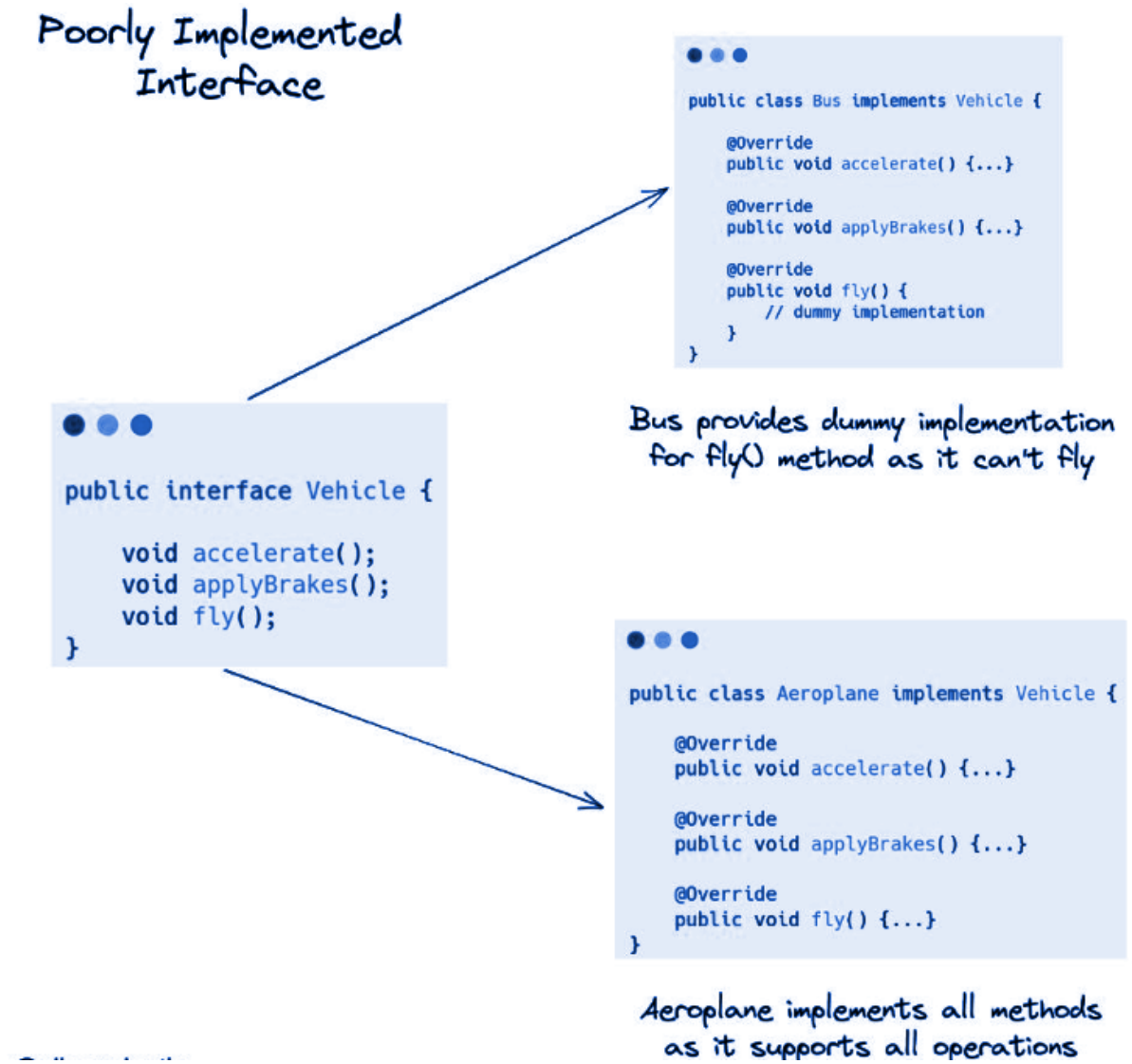
If an interface contains a method applicable to some child classes then we need to force the rest to provide dummy implementation.

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:

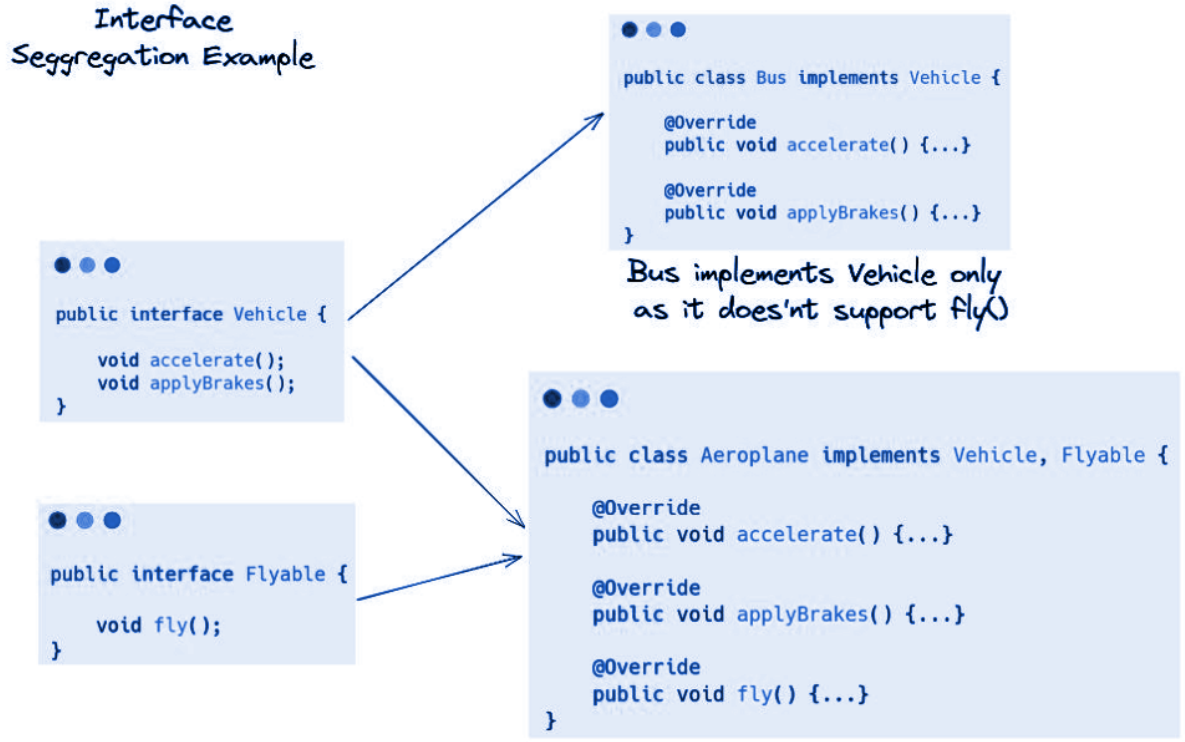


Good Implementation

Pulling out fly() method into the new Flyable interface solves the issue.

Now, the 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 a 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 the Repository interface, in future if we need to support NoSQL then simply need to pass its instance in the constructor without changing Service class.

```
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();
    }
}
```