SOLID Principles

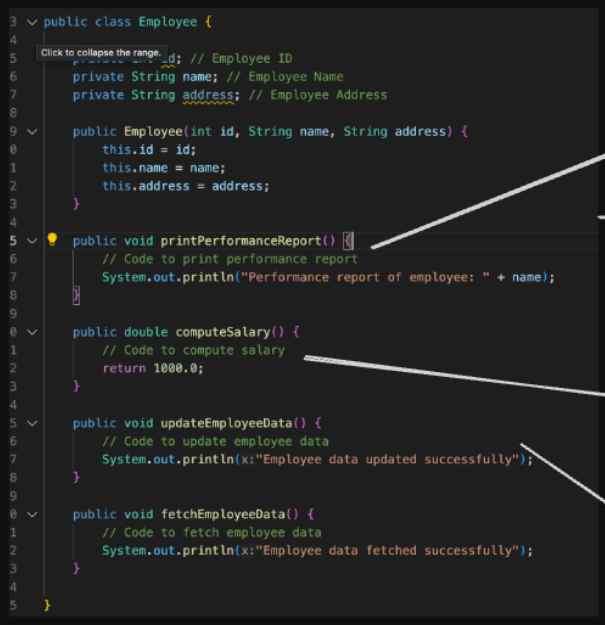
S - Single responsibility principle (SRP)

O - Open Closed Principle (OCP)

L - Liskov substitution Principle (LSP)

I - Interface segregation Principle (ISP)

D - Dependency inversion Principle (DIP)



```java
public class Employee {

    private int id; // Employee ID
    private String name; // Employee Name
    private String address; // Employee Address

    public Employee(int id, String name, String address) {
        this.id = id;
        this.name = name;
        this.address = address;
    }

    public void printPerformanceReport() {
        // Code to print performance report
        System.out.println("Performance report of employee: " + name);
    }

    public double computeSalary() {
        // Code to compute salary
        return 1000.0;
    }

    public void updateEmployeeData() {
        // Code to update employee data
        System.out.println(x:"Employee data updated successfully");
    }

    public void fetchEmployeeData() {
        // Code to fetch employee data
        System.out.println(x:"Employee data fetched successfully");
    }

}
```

If the report format changes, we might need to update this function.

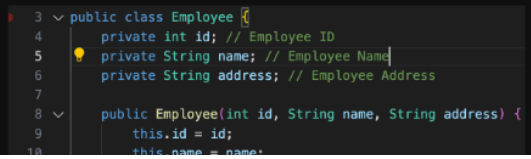
This class is trying to do too many things.
k

Say, if the taxation changes, we might need to update this function.

If data storage requirements change, we might need to update this function

Because the above class is doing too many things, there are too many reasons to update the code present in the class. This violates Single responsibility principle.

Single responsibility principle states that, there should one and only one reason to change a piece of code.



```java
public class Employee {
    private int id; // Employee ID
    private String name; // Employee Name
    private String address; // Employee Address

    public Employee(int id, String name, String address) {
        this.id = id;
```

```java
        this.address = address;
    }

    public int getEmployeeId() {
        return id;
    }

    public String getEmployeeName() {
        return name;
    }

    public String getEmployeeAddress() {
        return address;
    }

    public void setEmployeeAddress(String address) {
        this.address = address;
    }

    public void setEmployeeName(String name) {
        this.name = name;
    }
}
```

This class is now only responsible for Basic employee data creation and fetching

```java
public class EmployeeSalaryCalculator {
    public double computeSalary(Employee e) {
        // Code to compute salary
        return 1000.0;
    }
}
```

This class is only responsible for computing salary

```java
public class EmployeePerformanceReportGenerator {
    public void printPerformanceReport(Employee e) {
        // Code to print performance report
        System.out.println("Performance report of employee: " + e.getEmployeeName
    }
}
```

This class is only responsible for printing performance report

We have segregated the classes in a way such that their is one core responsibility every class has, hence there is only one reason to change logic in a class.

OCP - Open closed principle

A class should be open for extensions but closed for modifications.

```java
public class NotificationSender {

    public void sendNotifications(List<String> notificationTypes, String message) {
        for (String notificationType : notificationTypes) {
            switch (notificationType) {
                case "EMAIL":
                    EmailNotification notification = new EmailNotification();
                    notification.sendEmailNotification(message);
                    break;
                case "SMS":
                    SMSNotification smsNotification = new SMSNotification();
                    smsNotification.sendSMSNotification(message);
                    break;
                case "PUSH":
                    PushNotification pushNotification = new PushNotification();
                    pushNotification.sendPushNotification(message);
                    break;
```

This class will be modified every time we introduce or remove any type of notification.

This is a clear violation of OCP

```
        }
    }
}
```

How to improve ? We should try to think that if futher alteration in the requirements can unnecesary impact code of a class, it is violating OCP

```java
public interface Notification {

    void sendMessage(String message);
}

```

Interface to represent any notification

Individual type of notification implementing the interface

```java
public class EmailNotification implements Notification {

    @Override
    public void sendMessage(String message) {
        System.out.println("Email notification: " + message);
    }
}

```

```java
public class PushNotification implements Notification {

    @Override
    public void sendMessage(String message) {
        System.out.println("Push notification: " + message);
    }
}

```

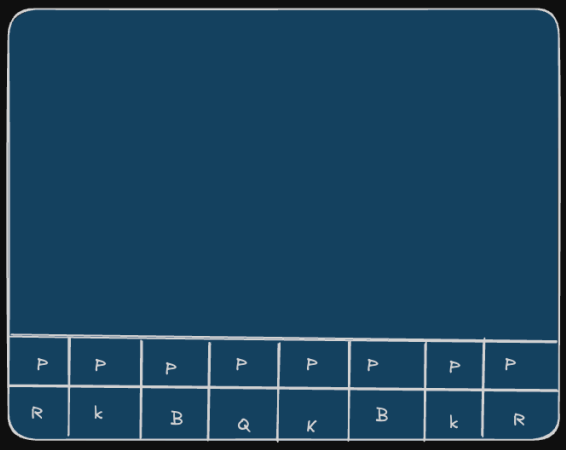
```java
public class NotificationSender {

    public void sendNotifications(List<Notification> notifications, String message) {
        for (Notification notification : notifications) {
            notification.sendMessage(message);
        }
    }

}

```

Instead of being tightly coupled to type of notifications, we now depend on Notification interface.

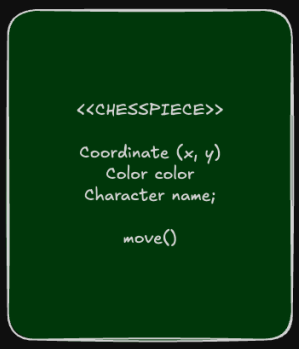
Even if we introduce new type of notifications or remove any older one, the sender class is not impacted.

It is a good idea to depend on abstractions rather than concrete classes.

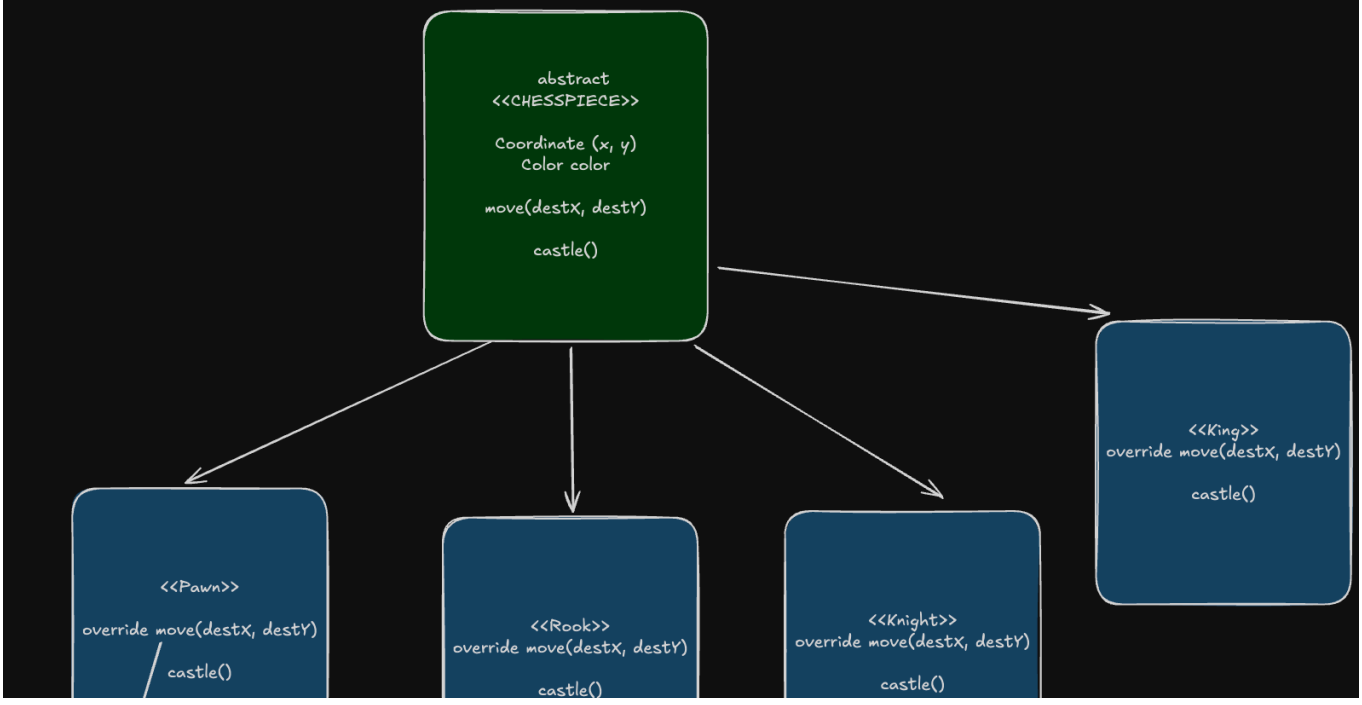
Chess



Write a class chesspiece !




```
if (this.name == 'R') {...}
else if(this.name == 'B') {...}
..
.
```



If someone gave us an
X coordinate which is
behind us,
it's an illegal argument

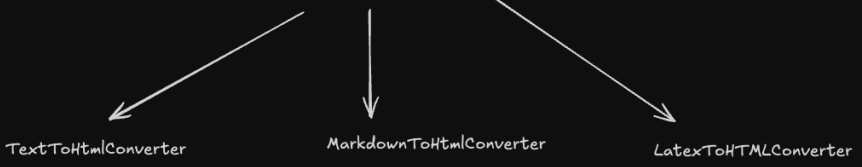
Liskov Substitution principle

Objects of a child class should be AS-IS substitutable in the variables of a parent class.

```
Chesspiece K = new King();
Chesspiece k = new Knight();
Chesspiece p1 = new Pawn();
....
```

Abstract Class HtmlConverter

convertHTML();



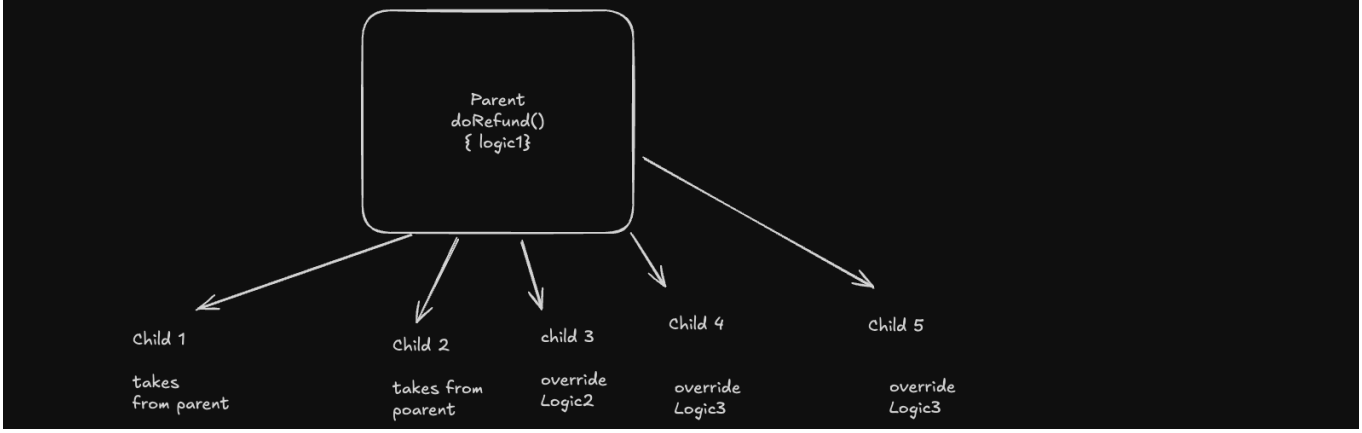
---

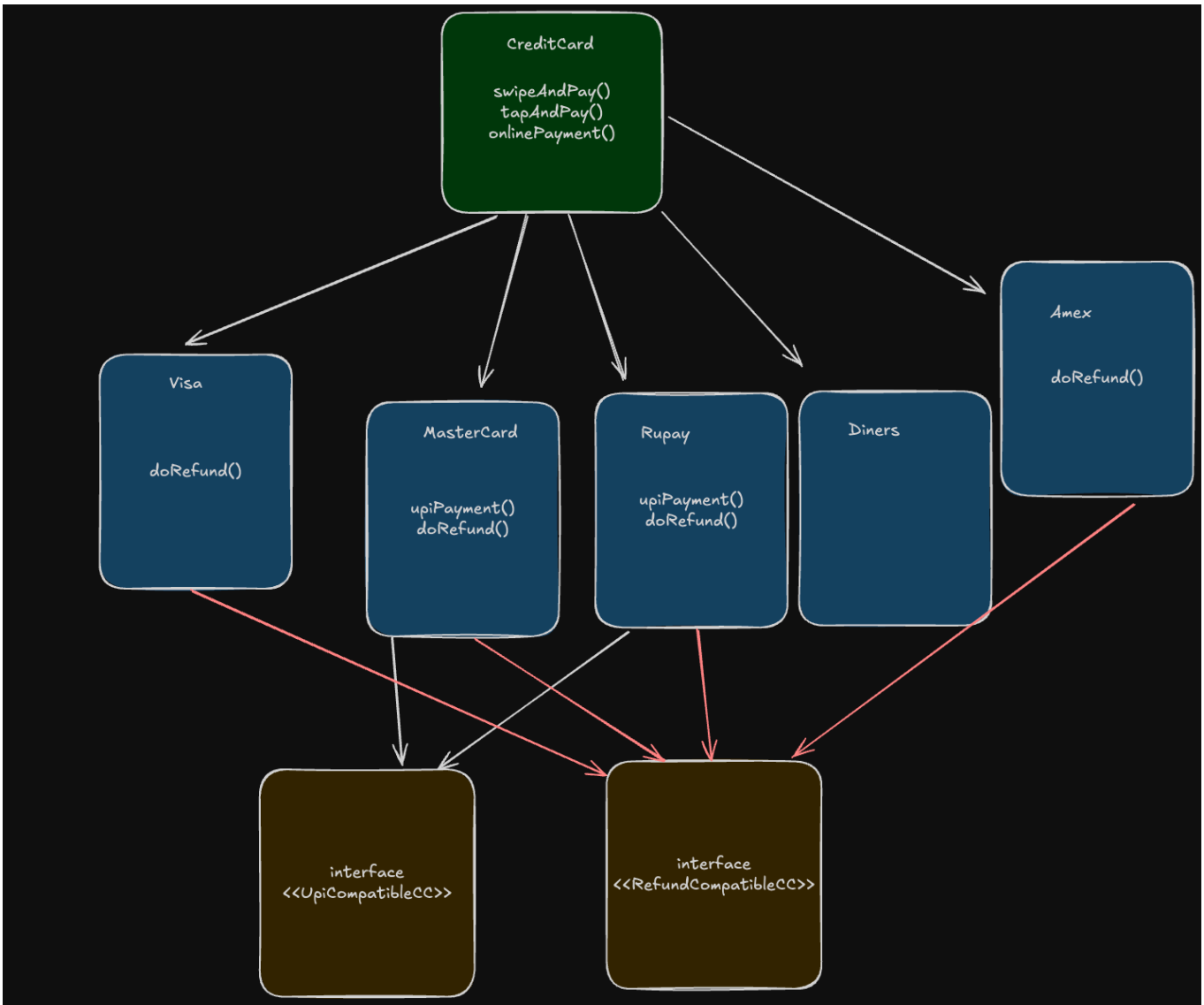
may be:

- Rook --> Rook + Knight
- Queen -> Queen + Knight
- Knight -> Bishop + king
- King - Pawn + queen + bishop + knight

The interface segregation principle (ISP) is a software engineering principle that states that code should not be forced to use methods it doesn't need.
It states that if any particular child isn't intended to be having a particular behaviour of the parent, then it should not

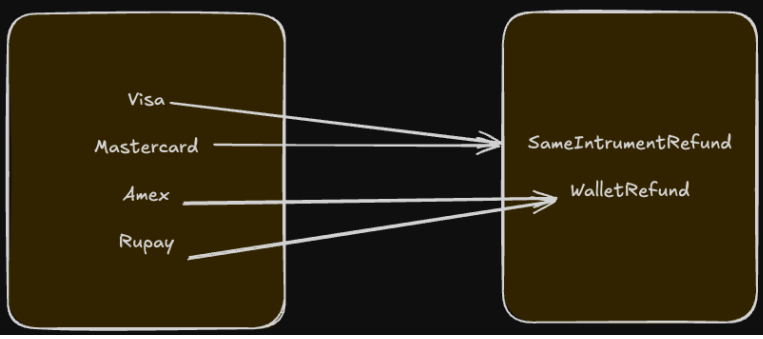
be forced to have that behaviour.



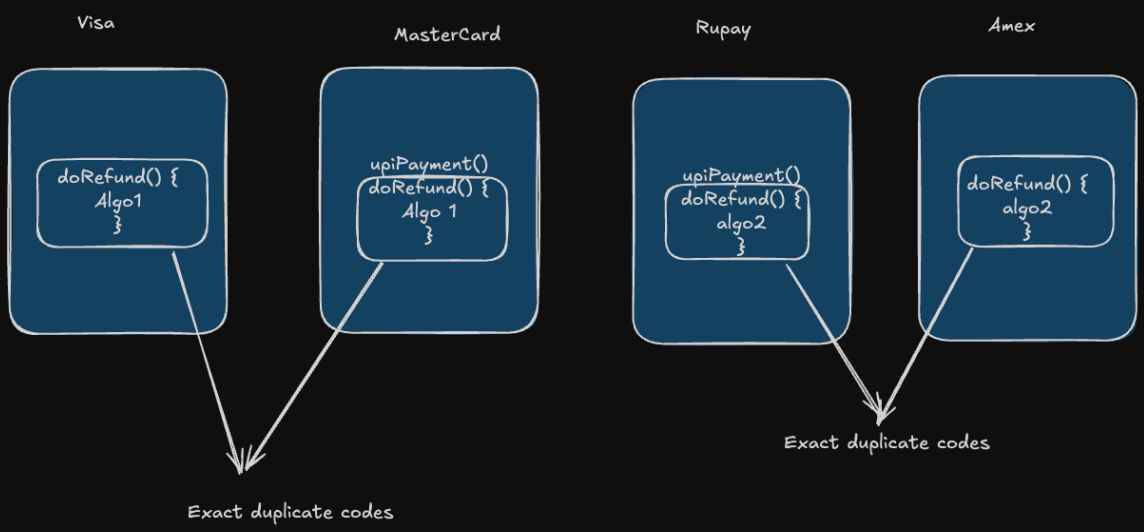


Let's the refund logic in a CC, can be custom and can be having different possible implementations.

But things become more interesting if we consider that there is a group of CC's which support one type of refund algorithm and then there is another group of CC's which support a different algorithm.



The `doRefund` method in visa and mastercard class, will be end to end same, and the `doRefund` method in rupay and amex class will be end to end same. Why ? because visa and mastercard supports exact same algorithm and rupay as well as amex support exact same algorithm.



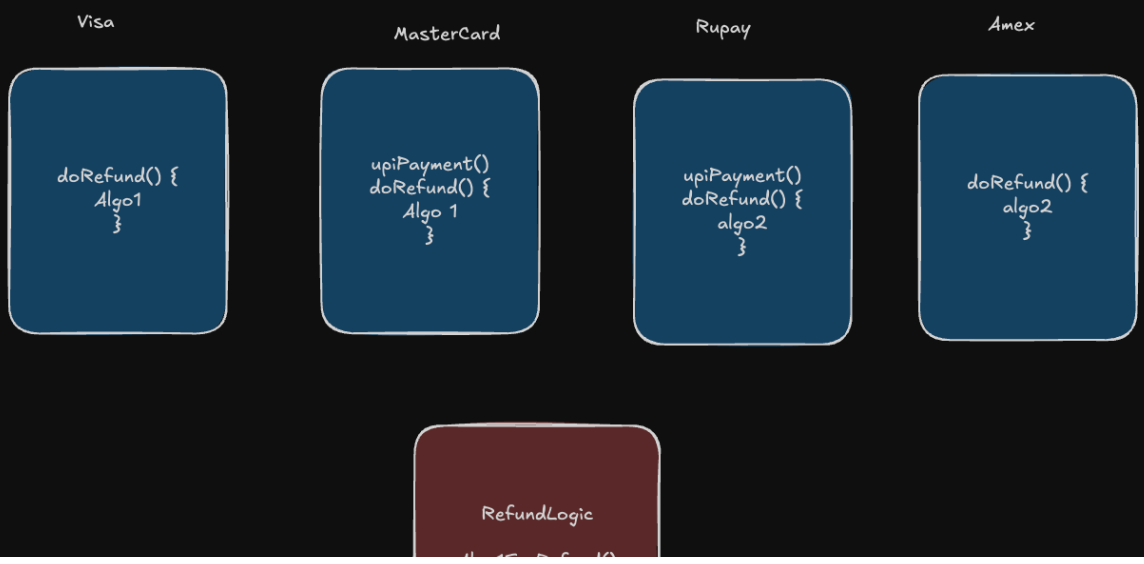
How to solve this code duplicacy problem ?

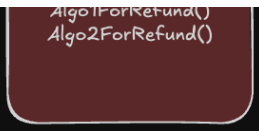
The most fundamental way to resolve the problem of code duplication is by writing function. Why ?

Because functions help us to implement DRY (Don't repeat yourself) principle.

May be for our usecase we can write, the algorithm1 and algorithm2 in separate functions. But where these functions should be placed ?

We should place these functions in a separate class.



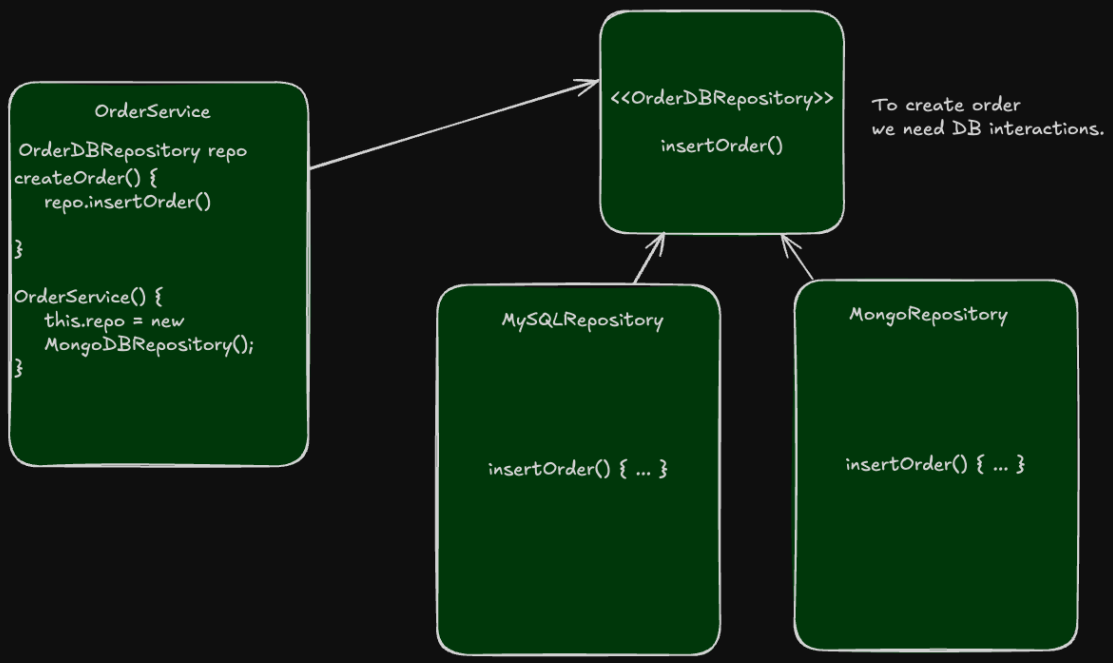


Dependency Inversion Principle (DIP)

This is NOT dependency injection

Higher level objects/module/class should never depend on concrete implementations of a lower level object/module/class.

In simple word, one class should try to avoid as much as possible any dependency on other concrete classes.



Note: Service classes are those which contain core business logic, they are the main algorithm driven classes of our apps. To implement those algorithms if they need some data, some network interaction or anything else they depend on other classes.

Dependency Injection

It involved techniques to supply dependencies of a class with their value at runtime.

Dependency injection can be done in multiple ways: