Advanced Software Engineering

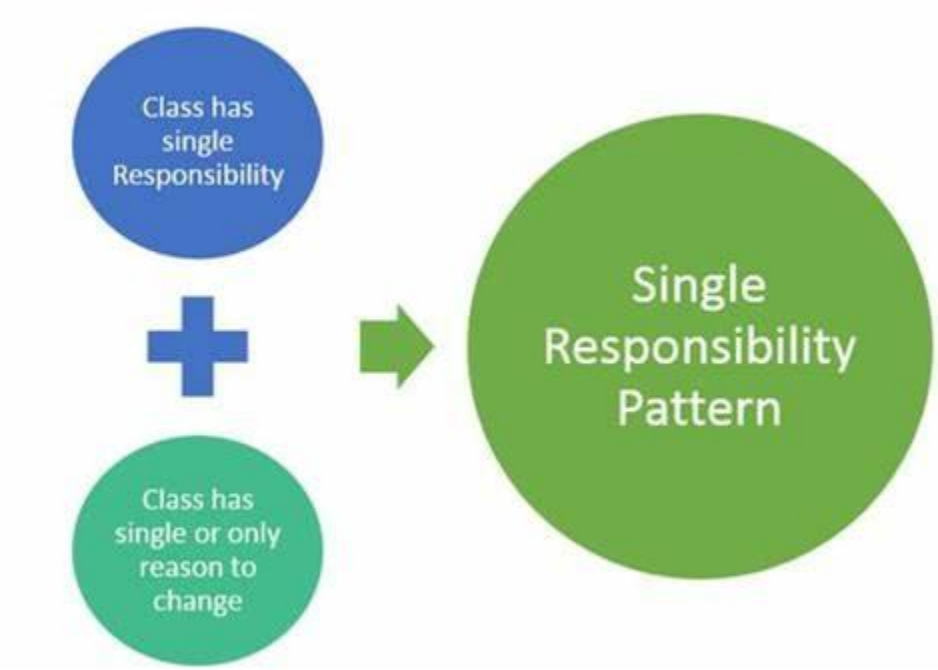
SOLID Principle

By:

Dr. Salwa Osama

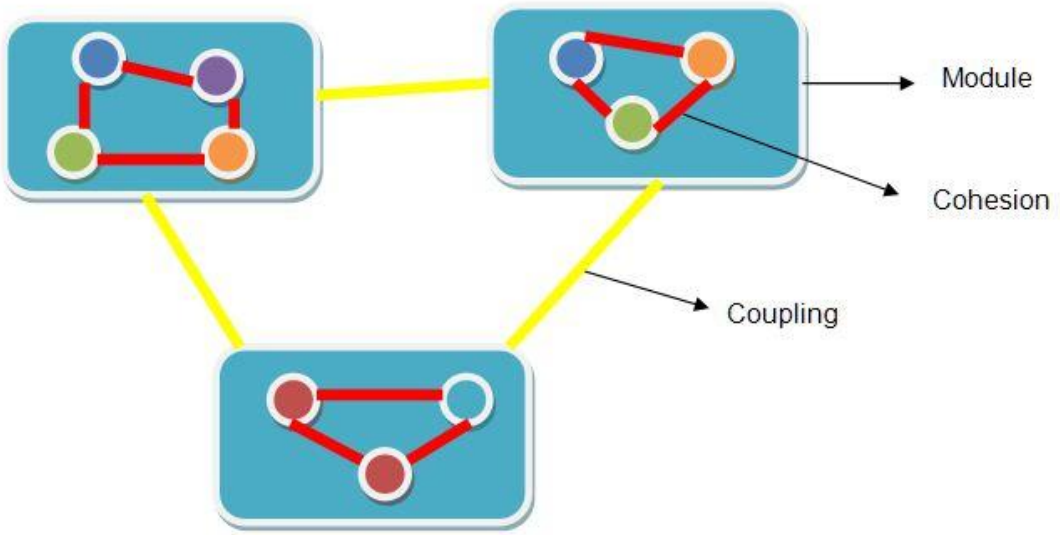
SRP - Single Responsibility Principle

- "Every software component should have one and only one responsibility.“
- *Component can be a class, a method, or a module.*



SRP - Single Responsibility Principle

- We work on two concepts - ***Cohesion and Coupling***.
- Single Responsibility Principle always advocates **higher cohesion** and always recommends **loose coupling**.



Cohesion

- Cohesion is the degree to witch the various parts of a software component are related.
- Stronger cohesion makes it easier to follow the Single Responsibility Principle.

Cohesion

- We could say that the contents of the unsegregated waste bin have a low cohesion, and the contents of each of the segregated waste bins have a high cohesion.
- Let's apply the same principle here. What do you make of the methods inside the Square class?

```
public class Square {
    private boolean highResolutionMonitor = true;
    private int side = 5;
    public int calculateArea() {
        return side * side; // side² - side ^ 2;
    }
    public int calculatePerimeter() {
        return side * 4;
    }
    public void draw() {
        if (highResolutionMonitor) {
            // Render a high-resolution image of a square
        } else {
            // Render a high normal image of a square
        }
    }
    public void rotate() {
        // Rotate the image of the square clockwise to
        // the required degree and re-render
    }
}
```

- Square class has Four methods: calculateArea(),calculatePerimeter(),draw(),and rotate()
- The calculateArea and calculatePerimeter functions do exactly what they are supposed to do
- They calculate the area and perimeter of a square given the length of its side.
- The draw() function renders the image of the square on the display.
- It has multiple code flow, depending on what type of display is being used.
- The rotate() function rotates the image of the square and re-renders it on the display.
- In the context of this code snippet, we will learn about a new concept termed Cohesion.
- Cohesion , in the software world. is defined as the degree to which the various parts of a software component are related.

```
public class Square {
    private boolean highResolutionMonitor = true;
    private int side = 5;
    public int calculateArea() {
        return side * side; // side² - side ^ 2;
    }
    public int calculatePerimeter() {
        return side * 4;
    }
    public void draw() {
        if (highResolutionMonitor) {
            // Render a high resolution image of a square
        } else {
            // Render a high normal image of a square
        }
    }
    public void rotate() {
        // Rotate the image of the square clockwise to
        // the required degree and re-render
    }
}
```

- The methods calculateArea and calculatePerimeter are closely related, in that they deal with the measurements of a square.
- So, there is a high level of cohesion between these two methods.
- The draw() method and the rotate() method deal with rendering the image of the square in a certain way on the display.
- So, there is a high level of cohesion between these two methods as well.
- But if you take all of the methods, the level of cohesion is low. For instance , the calculatePerimeter() method is not closely related to the draw() method.

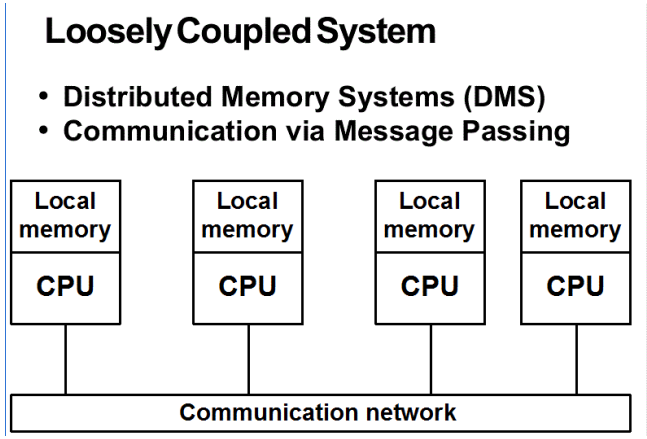
```
// Responsability: Measurements of squares
public class Square {
    int side = 5;
    public int calculateArea() {
        return side * side; // side² - side ^ 2;
    }
    public int calculatePerimeter() {
        return side * 4;
    }
}
```

```
// Responsability: Rendering images of
squares
public class SquareUI {
     public void draw() {
        if () {
            // Render a high resolution image of
a square
        } else {
            // Render a high normal image of a
square
        }
    }
    public void rotate() {
        // Rotate the image of the square
clockwise to
        // the required degree and re-render
    }
}
```

- We will name the class as SquareUI. By doing this, even though we have split the methods into two classes, we have increased the level of cohesion in each of the classes.
- All the two methods inside the Square class are now closely related, as both deal with the measurements of the square.
- All the two methods inside the SquareUI class are now closely related, as both deal with the graphic rendering of the square.
- So, one aspect of the Single Responsibility Principle is that we should always aim for high cohesion within a component, component means class in this case.

Coupling

- Coupling is defined as the **level of inter dependency between various software components**.
- Coupling is the degree of interdependence between software modules. A module could be a class or a package or even a microservice. Effectively, **the coupling is about how changing one thing required change in another.**



- What do you notice about the width of the tracks.
- So, each train cannot move on other track.
- In other words, a train is tightly coupled to its track.
- Tight coupling may be a necessity in railways, but in software, tight coupling is an undesirable feature.
- We'll see why.

Tightly coupling

```
public class Student {
    private String id;
    private String name;
    private String address;
    public void save() {
        Connection connection = null;
        Statement statement = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/MyDB", "root", "password");
            statement = connection.createStatement();
            statement.execute("INSERT INTO student VALUES (" + this.getId() + ", " + this.getAddress() + ", " + this.getBirth() + ")");
        } catch (Exception e) {
            e.printStackTrace();
        }   }
    public String getId() {
        return this.id;
    }
    public void setId(String id) {
        this.id = id;
    }
    // others getter and setters...
}
```

Coupling

- The 'save' method will convert the student class into a serialized form and persist it into a Database.
- You can see that this method deals with a lot of low-level details related to handling record insertion into a database.
- Let's assume the database you are using now is MySQL.
- Sometime in the future, if you decide to go with a NoSQL database like, say, MongoDB, most of this code will need to change.
- So, you can see that the Student class is tightly coupled with the database layer we use at the back end.
- The Student class should ideally deal with only basic student related functionalities like getting student id, date of birth, address etc.
- The Student class should NOT be made cognizant of the low-level details related to dealing with the back-end database.
- So tight coupling is bad in software.
- So how do we fix this.

```java
// Responsability: Handle core student profile data
public class Student {
    private String id;
    private String address;
    private String name;
    public void save() {
        new StudentRepository().save(this);
    }
    public String getId() {
        return this.id;
    }
    public void setId(String id) {
        this.id = id;
    }
    // others getter and setters...
}
```

```java
// Responsability: Handle Database operations for students

public class StudentRepository {
    public void save(Student student) {
        Connection connection = null;
        Statement statement = null;

        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/MyDB", "root", "password");

            statement = connection.createStatement();
            statement.execute("INSERT INTO student VALUES (" + student.getId() + ", '" + student.getName() + "', '" + student.getAddress() + "')");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Loosely coupling

Coupling - Example

- We'll take the database related code, and we'll move it into a new Repository class.
- Then we'll refer to this Repository method from inside the Student class.
- By doing so, we have removed the tight coupling and made it loose.
- So now if we change the underlying database, the Student class does NOT need to get changed and recompiled.
- You only need to change the Repository class.
- If you look at this in terms of responsibilities, the Student class has the responsibility of dealing with core student related data.
- And the Repository class has a single responsibility of dealing with database operations.
- So, by removing tight coupling, and making the coupling loose, we are again abiding by the Single Responsibility principle.

Uncle Bob Definition - SRP - Single Responsibility Principle

- *"Every software component should have one and only one reason to change."*
- So, what's this new phrase 'reason to change'?
- In the words of the Greek Philosopher - Heraclitus . "The only thing that is constant is change"
- This quote applies to the software world as well.
- Software is never dormant it always keeps changing

Example

- For Example, for Student class “before modification”.
- There could be multiple reasons for the Student class to change in future.
- A change in the student id format.
- A change in the student's name format.
- A change in the database back end.
- Okay, we have produced three reasons to change now.

```
public class Student {
    private String id;
    private String name;
    private String address;
    public void save() {
        Connection connection = null;
        Statement statement = null;
        try {
            Class.forName("com.mysql.jdbc.Driver");
            connection = DriverManager.getConnection("jdbc:mysql://localhost:3306/MyDB", "root", "password");
            statement = connection.createStatement();
            statement.execute("INSERT INTO student VALUES (" + this.getId() + ", " + this.getAddress() + ", " + this.getBirth() + ")");
        } catch (Exception e) {
            e.printStackTrace();
        }   }
    public String getId() {
        return this.id;
    }
    public void setId(String id) {
        this.id = id;
    }
    // others getter and setters...
}
```

Student Example

- So back to Student class, there are three reasons to change, how to fix that?
- We'll take the database operations out and move it to a separate Repository class.
- Because we split the classes, lets split the 'reasons to change' as well So the Student class is left with 2 reasons to change. And the Repository class has one reason to change.
- 2 is still a problem, isn't it. We are supposed to have only one reason to change, right? Technically, yes. But if the reasons are closely related, you can go ahead and combine them.
- So, if we examine the 2 reasons to change for the Student class closely,
- one is related to student id, and another is related to student name. We could combine both these and say 'changes to student profile'.

OCP - Open Closed Principle

Software entities should be open to extension but closed to modification.

“Open to extension” means that you should design your classes so that new functionality can be added as new requirements are generated.

“Closed for modification” means that once you developed a class you should never modify it, except to correct bugs.

OCP - Open Closed Principle

- We want modules to be open to extension.
  - If requirements change and we get requests for new features, ➔ We want the ability to extend the behavior of our application so that we can adapt to our customer's needs.
- We want modules to be closed for modifications
  - If we create the right abstractions ➔ we can depend mostly on well-defined and stable behaviors.

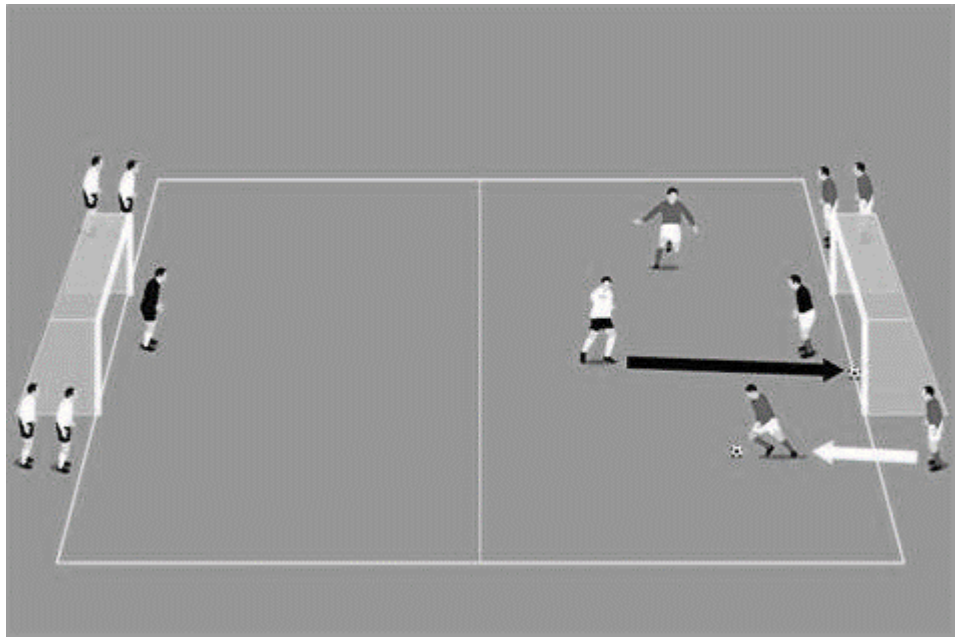
The code should be both flexible and stable

How can achieve flexibility and stability

- Generally, you achieve this by referring to abstractions for dependencies, such as interfaces or abstract classes, rather than using concrete classes.
- We can add the functionality by creating new classes that implement the interfaces.
- This reduces the risk of introducing new bugs to existing code, leading to more robust software.

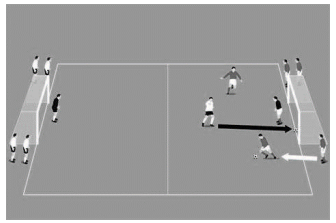
Example

- Football match
  - Al Ahly (Attack strategy)
  - Al Zamalk (Defend strategy)
  - bonus attack +2
  - bonus Defend +1
  - Welcome attack (Attacker)
  - Welcome Defend (Defensive)
  - Match consists of two teams
  - Match has score



Class Match
-Score
-Team1
-Team2
+SetTeam1()
+SetTeam2()
+GetScore()
+Play()
{
    If (Team1 is Ahly)
        Ahly is attacker
    Else if (Team1 is Zamalk)
        Zamalk is defend
}
+ Welcome()
{
    If (Team1 is Ahly)
        Welcome (Ahly attacker)
    Else if (Team1 is Zamalk)
        Welcome (Zamalk is defensive)

}



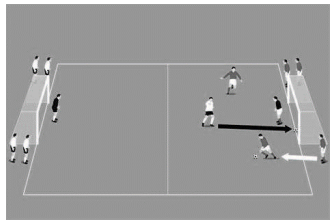
Example

Class Team
-name
-bonus
+set/get Name()
+set/get Bonus()

Class Attack
-name
+set/get Name()
+play()

Class defend
-name
+set/get Name()
+play()

```
Class Match
-Score, -Team1, -Team2
+SetTeam1()
+SetTeam2()
+Play()
{
        If (Team1 is Ahly)
                Ahly is attacker
        Else if (Team1 is Zamalk)
                Zamalk is defend
}
+ Welcome()
{
        If (Team1 is Ahly)
                Welcome (Ahly attacker)
        Else if (Team1 is Zamalk)
                Welcome (Zamalk is defensive)

}
+SetScore(winner)
{
        If (winner is Ahly)
                team1.bonus+2
        Else if (winner is Zamalk)
                team1.bonus+1
}
```



Example

Class Team
-name
-bonus
+set/get Name()
+set/get Bonus()

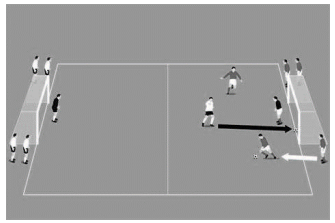
Abstract class
strategy
-name
+set/get Name()
+play()

Class Attack
extend strategy
+play()

Class defend
extend strategy
+play()

Violate OCP

Class Match
-Score, -Team1, -Team2
+SetTeam1(Strategy)
+SetTeam2(Strategy)
+Play()
{
Team1.play()
Team2.play()
}
+ Welcome()
{
Team1.welcome()
Team2.welcome()
}
+SetScore(winner)
{
Team.SetBonus()
Team2.SetBonus()
}



Class Team
-name
-bonus
-strategy
+set/get Name()
+set/get Bonus()
+set/get Strategy()
+Welcome()
{"hello"
+this.name
+strategy.name()
}
+SetBonus()
{strategy.SetScorePoint()}

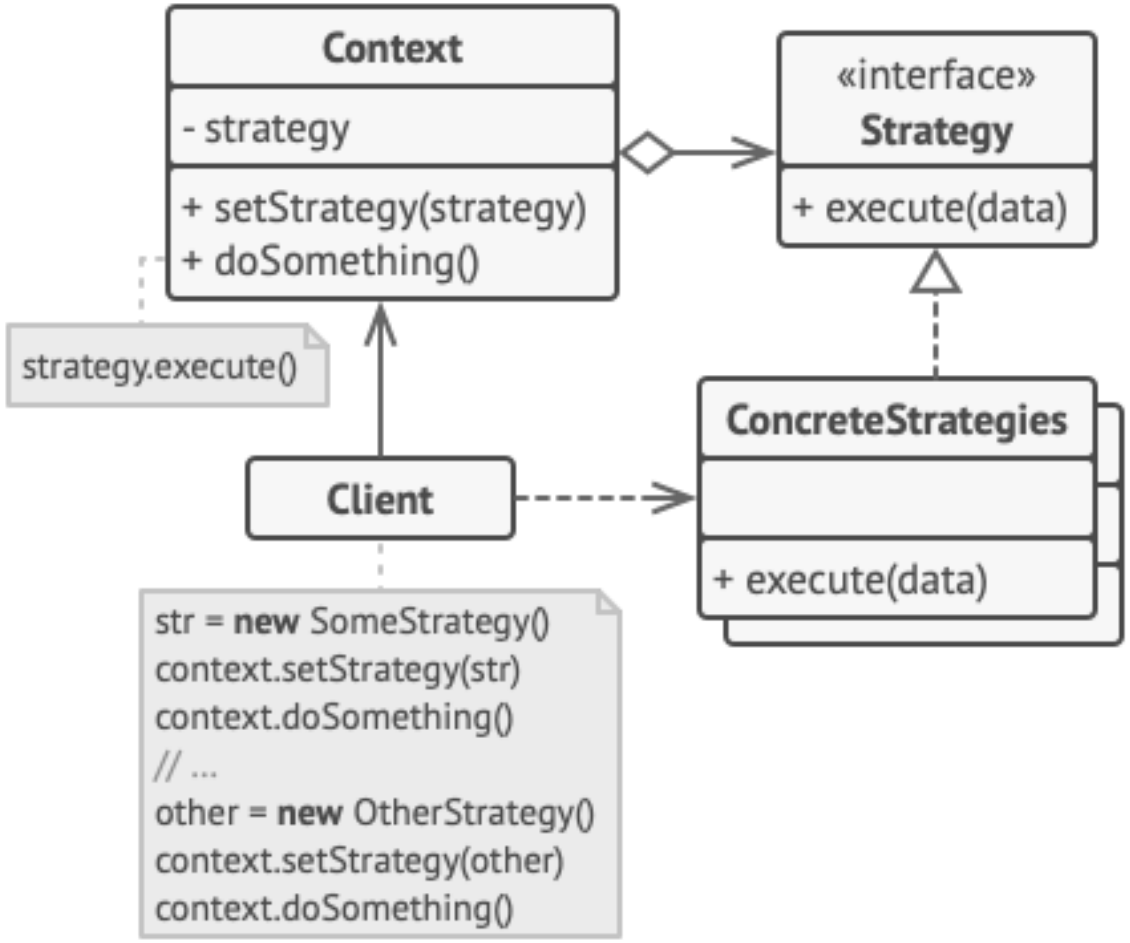
Solution

Abstract class
strategy
-name
-ScorePoint
+set/get Name()
+set/get
ScorePoint()
+play()

Class Attack
extend strategy
+play(){Attack}
SetScorePoint(){+2}

Class defend
extend strategy
+play() {defend}
SetScorePoint(){+1}

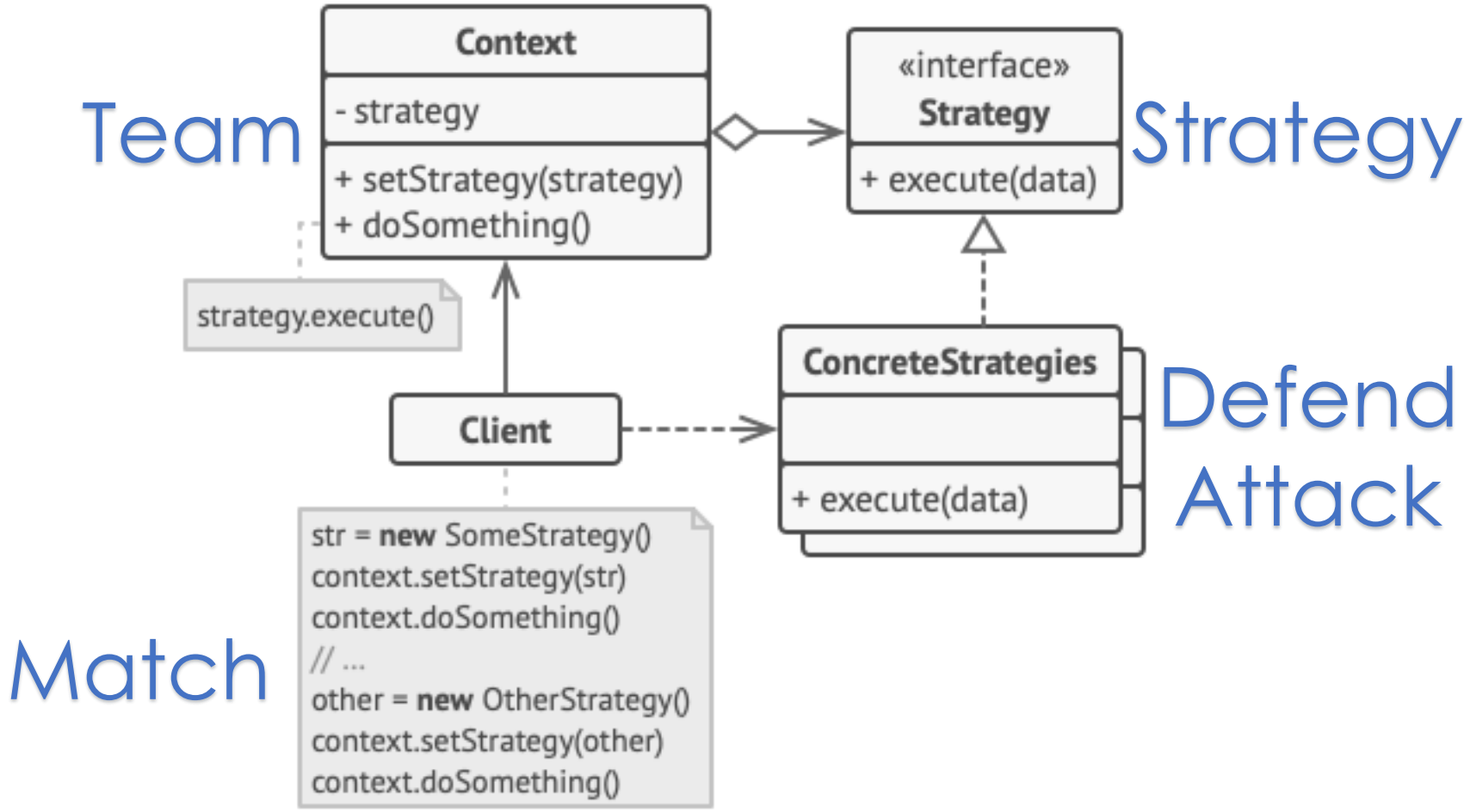
Solution Strategy Design pattern



Strategy Design pattern

- The **Context** maintains a reference to one of the concrete strategies and communicates with this object only via the strategy interface.
- The **Strategy** interface is common to all concrete strategies. It declares a method the context uses to execute a strategy.
- **Concrete Strategies** implement different variations of an algorithm the context uses.

Solution Strategy Design pattern



Liskov Substitution Principle (LSP)

- Substitutability is a principle in object-oriented programming introduced by Barbara Liskov in a 1987 conference keynote stating that, if class B is a subclass of class A, then wherever A is expected, B can be used instead:
- Objects of a superclass should be replaceable with objects of its subclasses without breaking the system.
- LSP implies that a subclass should not be more restrictive than the behavior specified by the superclass.

Liskov Substitution Principle (LSP)

- This principle ensures that inheritance (one of the OOP principles) is used correctly.
- If an override method does nothing or just throws an exception, then you're probably violating the LSP.
- It can apply to inheritance or interface.

The four conditions for abiding by the Liskov Substitution principle are as follows:

- **Condition 1 Contravariance:** Method signatures must match: Methods must take the same parameters; the parameters of the overriding methods should be the same or more generic than the types of the parent's method parameters (it is not allowed in java but allowed in PHP to generalize the parameters for overloaded method).
- **Condition 2:** **Preconditions cannot be strengthened in the subtype** The preconditions for any method can't be greater than that of its parent; Any inherited method should not have more conditionals that change the return of that method, such as throwing an Exception. For example, if the parent method in the parent class accepts an integer number while in an overridden method in the child class, you added a condition to accept the only positive integer number, that change in the child class violates LSP.

The four conditions for abiding by the Liskov Substitution principle are as follows:

- **Condition 3:** **Postconditions cannot be weakened in the subtype** Postconditions must be at least equal to that of its parent; Inherited methods should return the same type as that of their parent. For example, if the method in the parent class creates a connection to DB and at the end of the method, you close the connection, while in the child you keep the connection open, that changes in the child class violate LSP.
- **Condition 4:** **Exception types must match;** If a method is designed to return a FileNotFoundException in the event of an error, the same condition in the inherited method must return a FileNotFoundException too.

Vehicle Example: Violate LSP

```
class Vehicle
{
   String name;
   String getName() { ... }
   void setName(String n) {
... }
   double speed;
   double getSpeed() { ... }
   void setSpeed(double d)
{ ... }

   Engine engine;
   Engine getEngine() { ... }
   void setEngine(Engine e)
{ ... }
   void startEngine() { ... }
}
```

```
class Car extends
Vehicle
{
@Override
void startEngine() { ... }
}
```

```
class Bicycle extends Vehicle
{
@Override
void startEngine(){
throw new Expcetion("Not supported");
}
 /*problem!*/
}
```

- There is no problem here, right? A car **is a** Vehicle, and here we can see that it overrides the startEngine() method of its superclass.

- a bicycle **is a** Vehicle, however, it does not have an engine and hence, startEngine() method cannot be implemented. So, it violates LSP

Vehicle Example: Violate LSP

- **Now we create a list of objects of Car and** Bicycle in store it in list vehicleList
- **When the children call the method** startEngine(), the car object works fine while for the Bicycle object it shows an exception.

```
public class Befor_LiskovSubstitutionPrinciple
{
public static void main(String[] args) {
     ArrayList< Vehicle > vehicleList =new
ArrayList();
     Vehicle BMW =new Car();
     vehicleList.add(BMW);
     Vehicle bike =new Bicycle();
     vehicleList.add(bike);
     for(Vehicle v : vehicleList){
      v. startEngine();
}
}
}
```

Vehicle Example: Violate LSP

- These are the kinds of problems that violation of the Liskov Substitution Principle leads to, and they can most usually be recognized by a method that does nothing or even can’t be implemented.
- The solution to these problems is a correct inheritance hierarchy, and in our case, we would solve the problem by differentiating classes of vehicles with and without engines.
- Even though a bicycle is a vehicle, it doesn’t have an engine.

Vehicle Example: Following LSP

```
class Vehicle
{
   String name;
   String getName() { ... }
   void setName(String n) {
... }
   double speed;
   double getSpeed() { ... }
   void setSpeed(double d)
{ ... }

}
```

```
class VehicleWithEngines extends
Vehicle
{
Engine engine;
Engine getEngine() { ... }
void setEngine(Engine e) { ... }
void startEngine() { ... }
}
```

```
class Car extends VehicleWithEngines
{
@Override
void startEngine() { ... }
}
```

```
class VehicleWithoutEngines extends
Vehicle
{
void startMoving() { ... }
}
```

```
class Bicycle extends
VehicleWithoutEngines{
@Override
void startMoving() { ... }
}
```

Object-Oriented Design can violate the LSP in the following situations

- If a subclass returns an object that is completely different from what the superclass returns.
- If a subclass throws an exception that is not defined in the superclass.
- There are any side effects in subclass methods that were not part of the superclass definition.

Interface Segregation Principle (ISP)

- The Interface Segregation Principle was defined by Robert C. Martin

- While consulting for Xerox to help them build the software for their new printer systems. He defined it as:
  - “Clients should not be forced to depend upon interfaces that they do not use.”

- The goal of this principle is to **reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones**.
  - It's like the Single Responsibility Principle, where each class or interface serves a single purpose.

Interface Segregation Principle (ISP)

- First, no class should be forced to implement any method(s) of an interface they don’t use.
- Second, instead of creating large (fat interfaces), create multiple smaller interfaces with the aim that the clients should only think about the methods that are of interest to them.

Example Printer : Violate Interface Segregation Principle (ISP)

```
public interface PrinterTasks {
    void Print(String PrintContent);
    void Scan(String ScanContent);
    void Fax(String FaxContent);
    void PrintDuplex(String PrintDuplexContent);}
```

- We have an interface i.e. PrinterTasks declared with four methods.
- Now if any class wants to implement this interface, then that class should have to provide the implementation to all the four methods of the PrinterTasks interface.
- **HPLaserJetPrinterImp** wants all the services provided by the PrinterTasks.
- **LiquidInkjetPrinterImp** class the Fax and PrintDuplex methods are not required by the class but, still, it is implementing these two methods.

```
public class HPLaserJetPrinter implements
PrinterTasks{

    @Override
    public void Print(String PrintContent) {
            System.out.println(PrintContent);
    }

    @Override
    public void Scan(String ScanContent) {
            System.out.println(ScanContent);
    }

    @Override
    public void Fax(String FaxContent) {
            System.out.println(FaxContent);
    }

    @Override
    public void PrintDuplex(String
PrintDuplexContent) {
            System.out.println(PrintDuplexContent);
    }
```

```
public class LiquidInkjetPrinter implements
PrinterTasks{

    @Override
    public void Print(String PrintContent) {
            System.out.println(PrintContent);
    }

    @Override
    public void Scan(String ScanContent) {
            System.out.println(ScanContent);
    }

    @Override
    public void Fax(String FaxContent) {
        throw new
UnsupportedOperationException("Not
supported yet.");
    }
    @Override
    public void PrintDuplex(String
PrintDuplexContent) {
        throw new
UnsupportedOperationException("Not
```

Example Printer : Follow Interface Segregation Principle (ISP)

- we have split that big interface into four small interfaces. Each interface now has some specific purpose.
- Now if any class wants all the services, then that class needs to implement all the four interfaces as shown below.

```
public interface PrintingTasks {
        public void Print(String PrintContent);
}
```

```
public interface ScanTasks {
    void Scan(String ScanContent);
}
```

```
public interface FaxTasks {
    void Fax(String FaxContent);
}
```

```
public interface DuplexTasks {
        void PrintDuplex(String
PrintDuplexContent);
}
```

```
public class HPLaserJetPrinter implements
PrintingTasks, ScanTasks, FaxTasks, DuplexTasks {

    @Override
    public void Print(String PrintContent) {
        System.out.println(PrintContent);
    }

    @Override
    public void Scan(String ScanContent) {
        System.out.println(ScanContent);
    }

    @Override
    public void Fax(String FaxContent) {
        System.out.println(FaxContent);
    }

    @Override
    public void PrintDuplex(String
PrintDuplexContent) {
        System.out.println(PrintDuplexContent);
    }
}
```

```
public class LiquidInkjetPrinter implements
PrintingTasks, ScanTasks {

    @Override
    public void Print(String PrintContent) {
        System.out.println(PrintContent);
    }

    @Override
    public void Scan(String ScanContent) {
        System.out.println(ScanContent);
    }


}
```

Dependency Inversion Principle (DIP)

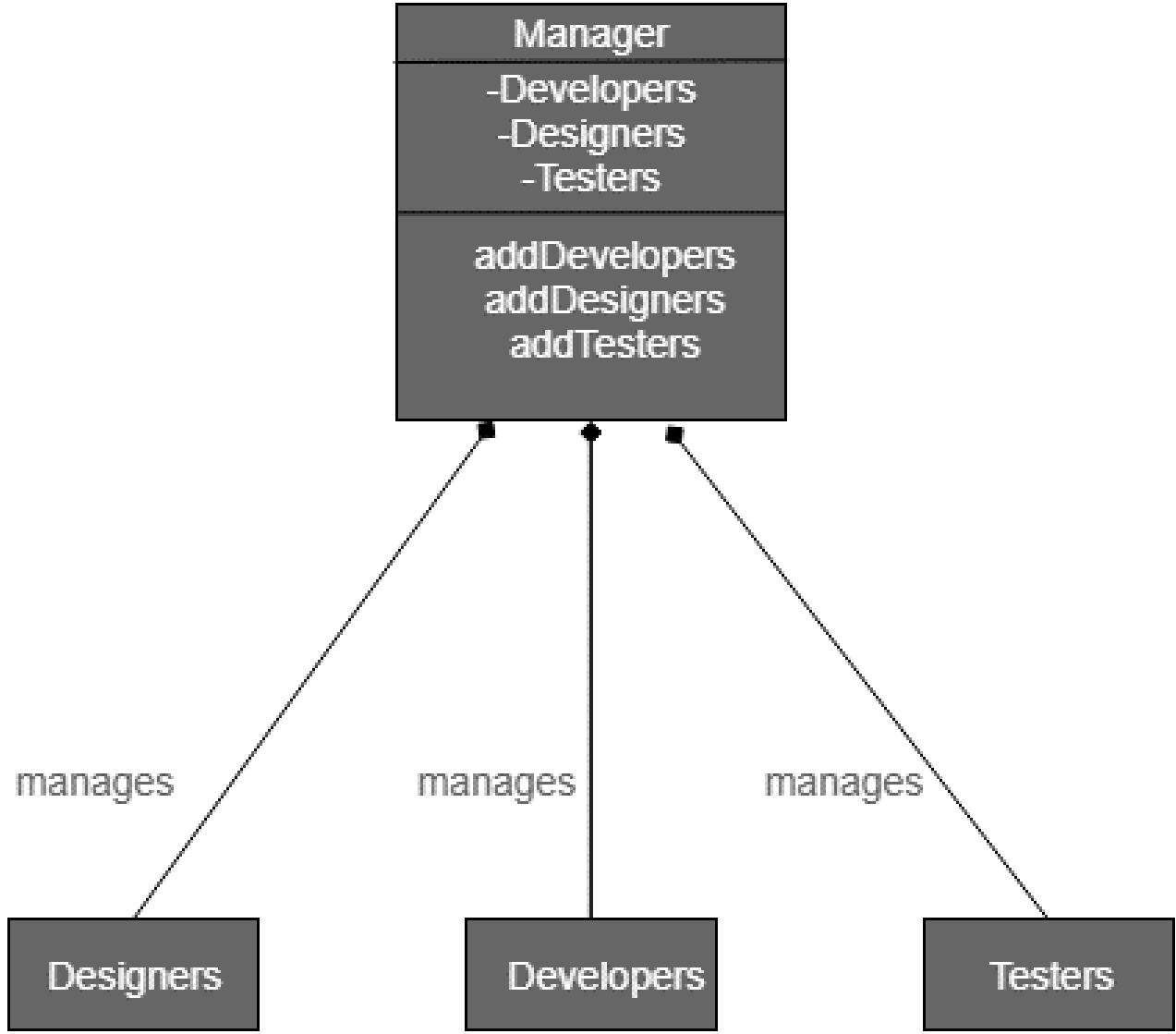
- The Dependency Inversion Principle (DIP) states that high-level modules should not depend on low-level modules; both should depend on abstractions. Abstractions should not depend on details. Details should depend upon abstractions.
- It means that there is an inversion of dependencies! Hence the name of the principle.

Why should I follow this principle?

- Because code that violates this principle may become too coupled together, and that makes the code hard to maintain, unreadable, and prone to side effects.

Example Violate Dependency Inversion Principle (DIP)

- A manager have some persons as an employee of which some are developers and some are graphic designers and rest are testers.



```
class Manager():
{
developers=[]
designers=[]
testers=[]

     addDeveloper(self,dev):
       developers.append(dev)

      addDesigners(self,design):
       designers.append(design)

      addTesters(self,testers):
       testers.append(testers)
}
```

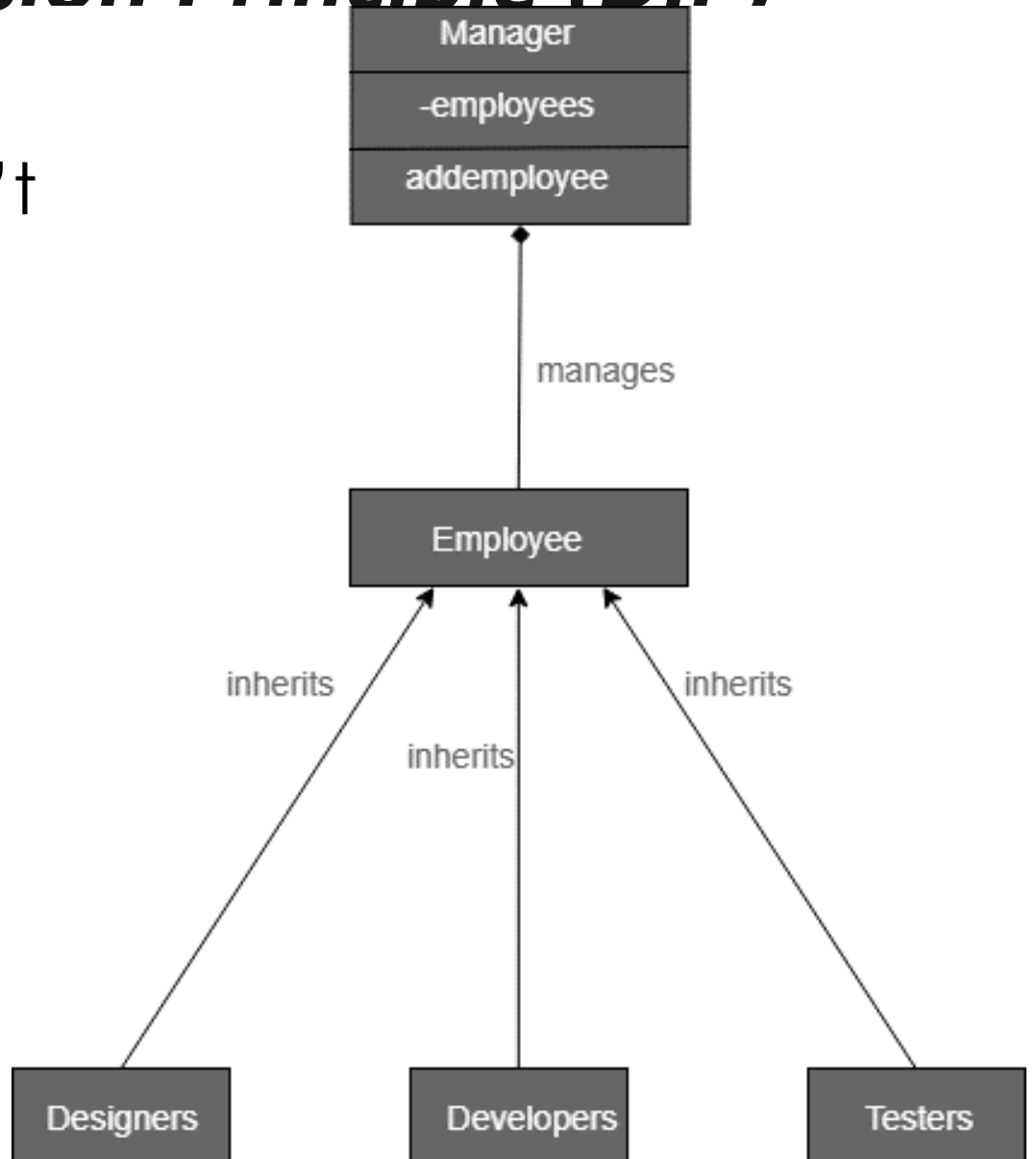
```
class Developer(){
        Developer(){
                print "developer added“}}

class Designer(){
    Designer(){
        print "designer added“}}

class Testers(){
    Testers(){
        print "tester added“}}

Main(){
    a=Manager()
    a.addDeveloper(Developer())
    a.addDesigners(Designer())}
```

Example, Dependency Inversion Principle (DIP)

- First, you have exposed everything about the lower layer to the upper layer, thus abstraction is not mentioned.
- That means Manager must already know about the type of the workers that he can supervise.

Example, Dependency Inversion Principle (DIP)

- In this code, the manager doesn't have an idea beforehand about all the type of workers that may come under him/her making the code truly decoupled.



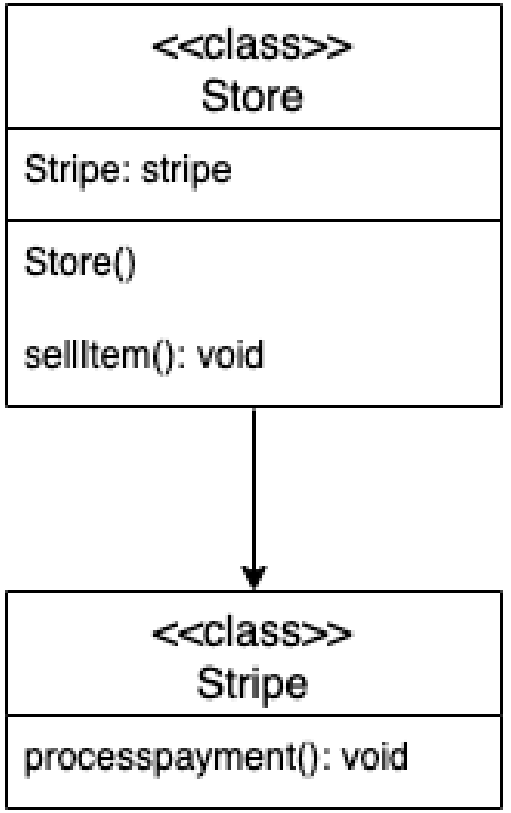
Example Store App : Violate The Dependency Inversion Principle (DIP)

- Let's say that you are creating an e-commerce application.
- On your e-commerce application, you must install a payment gateway
- There are multiple options such as Stripe and PayPal.
- So, let's say that you decide to use Stripe’s API first to implement the payment gateway on your application!

Example Store App : Violate The Dependency Inversion Principle (DIP)

```
public class Store {
    Stripe stripe;

   Store() {
        this.stripe = new Stripe();
    }
    public void sellItem() {

this.stripe.processpayment();
    }
}
```

```
Public class Stripe {
    public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
}
```



Example Store App : Violate The Dependency Inversion Principle (DIP)

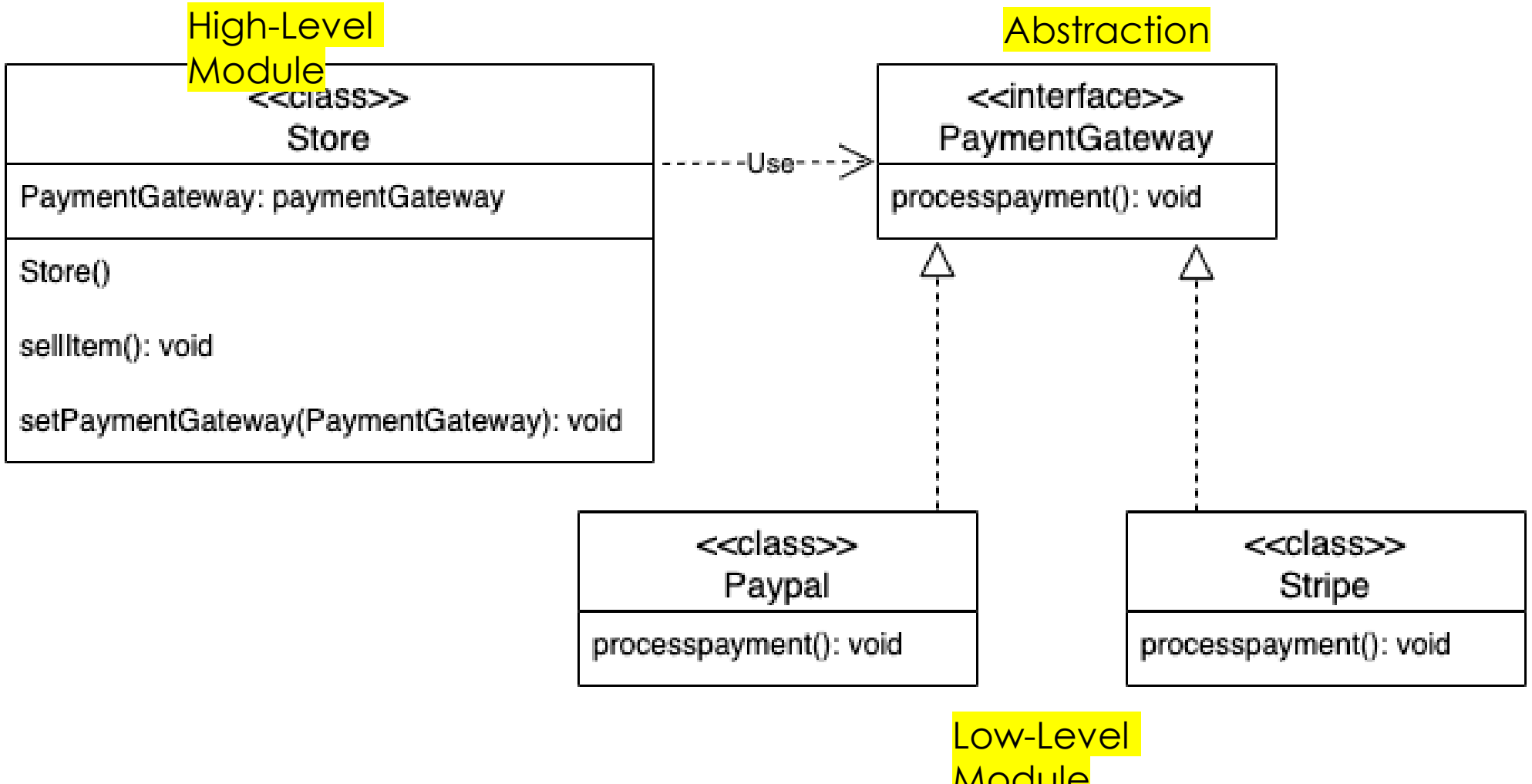
- In this case, you are directly calling methods from the Stripe API itself. You may ask ***well but it works right?***
- There's an issue here — do you realise that the higher-level module (Store) is depending on the lower-level module (Stripe payment gateway)?
- What if Stripe suddenly goes under maintenance but your store is still online! There are still many people who want your product and so you have to implement a new payment gateway (PayPal).
- However, what is happening now? There is an issue — we have to change the code from our Store class which is a higher-level module.

Example Store App : Violate The Dependency Inversion Principle (DIP)

- While this looks alright since it is a small class in this example, in a bigger application, there are going to be multiple instances where our ‘stripe’ variable is used, and the API is called.
- So that means we must switch **every single one** of them. This could mean **LOTS** of time wastage, convoluted code and a huge number of potential errors.

Example Store App : Follow The Dependency Inversion Principle (DIP)

- So how do we avoid this problem? As mentioned above and to adhere to DIP, we should set up an abstraction layer!



Example Store App : Follow The Dependency Inversion Principle (DIP)

```
public class Store {
    PaymentGateway paymentGateway;
    Store() {
    }
    public void
setPaymentGateway(PaymentGateway pg) {
        this.paymentGateway = pg;
    }
    public void sellItem() {
        this.paymentGateway.processpayment();
    }
}
```

```
interface PaymentGateway {
    public void
processpayment();
}
```

```
public class Stripe implements
PaymentGateway {
@Override
public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
}
```

```
public  class Paypal implements
PaymentGateway {
@Override
    public void processpayment() {
        System.out.println("Earned $10 through
Paypal");
    }
}
```

Key points

- One of the ways to achieve the Open-Close Principle is to use the Dependency Inversion Principle.
- Code that doesn't follow the Dependency Inversion Principle can be too coupled, which means you will have a hard time managing the project.

Dependency Injection

- When a class ClassA uses any method of another class ClassB, we can say that ClassB is a dependency of ClassA.
- If we needed to change/replace ClassB with ClassC because ClassC has an optimized version of the method used by ClassA, we need to recompile ClassA because we don't have a way to change that dependency, it's hardcoded inside of ClassA.
- The Dependency Injection is nothing but being able to pass (inject) the dependencies when required instead of initializing the dependencies inside of the recipient class.

Dependency Injection

- Dependencies among modules can lead to code that's tightly coupled and less maintainable.
- Dependency Injection (DI) is therefore used to resolve dependencies at runtime rather than at compile time.
- Dependency Injection is considered a design pattern and not a framework. It's one way of implementing a more general software concept called Inversion of Control (IoC).
- **Inversion of Control (IoC)** is basically used to invert different kinds of additional responsibilities of a class rather than the main responsibility.

Example Store App : Violate The Dependency Injection (DI)

```
public class Store {
    Stripe stripe;

    class Store() {
        this.stripe = new Stripe();
    }
    public void sellItem() {

this.stripe.processpayment();
    }
}
```

```
Public class Stripe {
    public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
}
```

- Here class **Store** uses class **Stripe** The class **Store** is said to be dependent on the class **Stripe**, and **Stripe** is called a **dependency** of **Store**.
- This kind of dependency is very trivial in programming. However, when the application's code gets bigger and more complex, the hard-coded dependency among classes introduces some drawbacks:

Example Store App : Violate The Dependency Injection (DI)

```
public class Store {
    Stripe stripe;

    class Store() {
        this.stripe = new Stripe();
    }
    public void sellItem() {

this.stripe.processpayment();
    }
}
```

```
Public class Stripe {
    public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
}
```

- The code is inflexible - it's hard to maintain and extend as when a class permanently depends on another class, a change to the depending class may require a change to the dependent class. And it's impossible to change the depending class later without updating and recompiling the code.
- The code is hard for unit testing because when you want to test only the functionalities of a class, you must test other depending classes as well.
- The code is hard to reuse because the classes are tightly coupled.

Example Store App : Violate The Dependency Injection (DI)

```
public class Store {
    Stripe stripe;

    class Store() {
        this.stripe = new Stripe();
    }
    public void sellItem() {

this.stripe.processpayment();
    }
}
```

```
Public class Stripe {
    public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
}
```

Therefore, dependency injection comes to address these drawbacks, making the code more flexible to changes, easy for unit testing, and truly reusable.

What are the different types of dependency injection?

- Developer can apply Dependency injection via any of the following three types.
- Constructor Injection, Interface injection, or Setter injection.

What are the different types of dependency injection?

- **Constructor Injection:** Dependency is passed to the object via its constructor that accepts an interface as an argument. A concrete class object is bound to the interface handle. This is typically used if the dependent object must use the same concrete class for its lifetime.
- **Interface injection:** An interface provides an injector method that is responsible for injecting the dependency to any class that may require it. The client class must implement the interface and override the injector method. The clients must implement an interface that will expose a setter method that accepts the dependency.
- **Setter injection:** The dependent class has a public setter method through which the dependency is injected.

Example Store App : Follow The Dependency Injection – Using Setter Injection

```
public class Store {
    PaymentGateway paymentGateway;
    Store() {
    }
    public void
setPaymentGateway(PaymentGateway pg) {
    this.paymentGateway = pg;
    }
    public void sellItem() {
        this.paymentGateway.processpayment();
    }
}
```

```
interface PaymentGateway {
    public void
processpayment();
}
```

```
public class Stripe implements
PaymentGateway {
@Override
public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
```

```
public  class Paypal implements
PaymentGateway {
@Override
    public void processpayment() {
        System.out.println("Earned $10 through
Paypal");
    }
}
```

Example Store App : Follow The Dependency Injection – Using Setter Injection

- We inject(pass) the type of object **PaymentGateway** to class **Store** via the setter method **setPaymentGateway()**

Example Store App : Follow The Dependency Injection – Using Constructor Injection

```
public class Store {
    PaymentGateway paymentGateway;

    Store(PaymentGateway pg) {
    this.paymentGateway = pg;
    }
public void sellItem() {
        this.paymentGateway.processpayment();
    }
}
```

```
interface PaymentGateway {
    public void
processpayment();
}
```

```
public class Stripe implements
PaymentGateway {
@Override
public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
```

```
public  class Paypal implements
PaymentGateway {
@Override
    public void processpayment() {
        System.out.println("Earned $10 through
Paypal");
    }
}
```

Example Store App : Follow The Dependency Injection – Using Constructor Injection

- We inject(pass) the type of object **PaymentGateway** to class **Store** via the constructor **Store()**

Example Store App : Follow The Dependency Injection – Using Interface Injection

```
interface PaymentGatewayInjector {
    public void setPaymentGateway
(PaymentGateway pg);
}
public class Store implements
PaymentGatewayInjector{
    PaymentGateway paymentGateway;
    Store() {
    }
    @Override
    public void
setPaymentGateway(PaymentGateway pg) {
        this.paymentGateway = pg;
    }
    public void sellItem() {
        this.paymentGateway.processpayment();
    }
}
```

```
interface PaymentGateway {
    public void
processpayment();
}
public class Stripe implements
PaymentGateway {
@Override
public void processpayment() {
        System.out.println("Earned $10 through
Stripe");
    }
public  class Paypal implements
PaymentGateway {
@Override
    public void processpayment() {
        System.out.println("Earned $10 through
Paypal");
    }
}
```

Example Store App : Follow The Dependency Injection – Using Interface Injection

- We inject(pass) the type of object **PaymentGateway** to class **Store** via the implement the method **setPaymentGateway()** from interface **PaymentGatewayInjector**

Advantages of dependency injection

- Makes testing easier.
- Reduces coupling between client and dependency classes.
- The code is easier to maintain, reuse and extend.

Key points:

- Dependency injection is a technique that allows the client code to be independent of the services it is relying on. The client does not control how objects of the services are created - it works with an implementation of the service through an interface. This is somewhat inverse to trivial programming, so dependency injection is also called ***inversion of control***.
- We used the **Dependency Injection (DI)** technique to implement the **Dependency Inversion Principle (DIP)**, to apply the **Open-Close Principle (OCP)**.