Course: Software Engineering

LECTURE 5: SOLID PRINCIPLES

Yutaka Watanobe

Topics covered

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

SOLID Principles

- **SOLID** is a set of five principles that should be followed when designing software in object-oriented programming.
- The principles are called the SOLID principles, from the initial letters of the following five principles.

  S : Single responsibility principle

  O : Open/closed principle

  L : Liskov substitution principle

  I : Interface segregation principle

  D : Dependency inversion principle

SOLID Principles as Guidelines

- The SOLID principles are software design principles that apply at the level above the code level.
- These five software development principles are guidelines that should be followed when building software.
- They are intended to improve the **scalability** and **maintainability** of software.

Purpose of the SOLID Principles

- The purpose of the SOLID principles is to bring the following properties to software structure.
  - ➢Improved **extensibility**: Easy to change, easy to reuse
  - ➢Improved **maintainability**: Resistant to change
  - ➢Improved **readability**: Easy to understand

- By following these principles in software design, you can improve the quality of your software.
- In particular, <u>as the scale of your software becomes larger</u>, it becomes more important to follow the SOLID principles.

The Significance of Learning the SOLID Principles

- The SOLID principles are fundamental programming principles, and are essential knowledge for professional developers.
- If software developers apply the SOLID principles appropriately, they can improve the quality of their code and enhance their design skills.
- In addition, understanding the SOLID principles will help you <u>communicate more effectively with other programmers.</u>

Caution

- The SOLID principles are not always the best solution in every situation.
- If you are too fixated on following the principles, it is possible that the readability of the code will decrease or development efficiency will fall. For this reason, it is important to apply the SOLID principles appropriately.
- Some of these principles may seem similar, but they do not necessarily have the same purpose. Even if they are similar, it is possible to satisfy one principle while breaking another.

Note

- In this lecture note, we mainly use the word “class” for explanation purposes, but please note that it also applies to functions, methods, and modules.

S: Single Responsibility Principle

S: Single Responsibility Principle

In all classes, it should have only one responsibility

- A responsibility is the *reason* for a change. In other words, the Single Responsibility Principle can also be expressed as “there should be only one reason to change a class/module”.
- It is easy to misunderstand, but this principle does not mean that a module should only do one thing. It is a principle that “there should be only one target for a responsibility” .

Purpose of the Single Responsibility Principle

- The purpose of this principle is to <u>separate the behavior </u>so that if a bug occurs as a result of a change, it will not affect other unrelated behavior.
- If a class has many responsibilities, the possibility of a bug occurring increases, because if you make a change to one of those responsibilities, it is possible that you will affect other responsibilities without noticing.

Example of Violation of the Single Responsibility Principle

```
class User {
    private String name;
    private String email;

    public User(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    public String getUserInfo() {
        return "Name: " + name + ", Email: " + email;
    }

    public void sendEmail(String message) {
        System.out.println("Send email to " + email + ": " + message);
    }
}
```

Example of Violation of the Single Responsibility Principle

```
class Application {
    public static void main(String[] args) {
        User user = new User("Alice", "alice@example.com");
        System.out.println(user.getUserInfo());
        user.sendEmail("Hello, this is a test message.");
    }
}
```

- In this example, the User class has two roles: to obtain user information and to send emails. In terms of the reason for the change, at least the following two possibilities can be considered.
    1. When the content of the user information changes: e.g. adding an address to the user information
    2. When the method of sending email changes: e.g. adding a CC when sending email
- Therefore, this violates the Single Responsibility Principle.

Example of Adhering to the Single Responsibility Principle

- The example User class that is not being followed is divided as follows.

```
class UserInfo {
    private String name;
    private String email;

    public UserInfo(String name, String email) {
        this.name = name;
        this.email = email;
    }

    public String getName() {
        return name;
    }

    public String getEmail() {
        return email;
    }

    public String getUserInfo() {
        return "Name: " + name + ",
                Email: " + email;
    }
}
```

```
class EmailSender {
    public void sendEmail(UserInfo user,
                          String message) {
          System.out.println("Send email to “
 + user.getEmail() + ": " + message);
    }
}
```

Example of adhering to the Single Responsibility Principle

```
public class Application {
    public static void main(String[] args) {
        UserInfo user = new UserInfo("Alice", "alice@example.com");
        EmailSender emailSender = new EmailSender();

        System.out.println(user.getUserInfo());
        emailSender.sendEmail(user, "Hello, this is a test message.");
    }
}
```

- By splitting the User class into the UserInfo class and the EmailSender class, **each class now has a single role**.
- This means that there is now a single reason for changing the User class, and the Single Responsibility Principle can be followed.

What Problems can Occur if the Single Responsibility Principle is Violated?

- If the Single Responsibility Principle is violated, the dependencies become complex, and <u>a change in one role will affect another role</u>.
- In other words, a change in one place will affect places that are not directly related to that change.
- In that situation, every time something is changed, all the modules affected by that change must be investigated and tested after the change.
- These investigations and tests take a huge amount of time and effort compared to the actual changes themselves. In other words, a huge amount of time and effort is wasted on making even the smallest changes. Or to say it another way, maintainability declines and bugs are more likely to be created.

The Benefits of Following the Single Responsibility Principle

The amount of change can be minimized

- The impact of the change can be minimized, and because the impact of the change does not spread to other parts, it does not take long to test the changed parts

The changed parts become clear

- Because it is immediately clear where to make changes, it does not take long to find the changed parts

Readability improves

- It improves expandability and makes it easier to use from other modules.

Practices for Following the Single Responsibility Principle

Make modules small enough to function

- Make modules as small as possible, like screws or boards, so that they can no longer function at a smaller level. Then, combine these modules to create a larger system.

Separate things with different concepts

- Even if they have similar logic, separate things with different concepts into different modules. You should avoid repeating code, but you should not group things that have different concepts. If the actors are different, it is better to split the class for each actor.
  - For example, when implementing the User class, if you are also using user_id in other classes, you should create a separate class that only has user_id. This way, the User class will not be affected by changes to user_id that other classes may make.

O: Open-Closed Principle

O: Open-Closed Principle

Classes must be open to extension and closed to modification.

- Extension means adding functionality. Being open to extension means that you can add functionality by adding new code.
- Modification means modifying existing code. Being closed to modification means that existing code is not modified by adding functionality.
- This is a principle that should be followed when developing modules.
- As software changes as long as it exists, we must be aware of when, why and how it will change during development.

Purpose of the Open-Closed Principle

- The purpose of this principle is to extend the behavior of a class without changing the existing behavior of the class.
- This is to avoid causing bugs in places where the class is used.
- If you change the current behavior of a class, it will affect all systems that use that class.
- When you want to perform more functions in a class, the ideal method is to add to the existing functions and not to change them.

Example of a Violation of the Open-Closed Principle

- As an example, consider the getPointMultiplier function, which returns a point multiplier based on the user's rank.

```
enum Rank {
    GOLD,
    SILVER,
}
```

```
class User {
    private String name;
    private Rank rank;

    public User(String name, Rank rank) {
        this.name = name;
        this.rank = rank;
    }

    public String getName() {
        return name;
    }

    public Rank getRank() {
        return rank;
    }

    @Override
    public String toString() {
        return "Name: " + name + ", Rank: " + rank;
    }
}
```

Example of a Violation of the Open-Closed Principle

- Next, implement the getPointMultiplier function, which returns a point multiplier based on the user's rank.

```java
public class Application {
    public static int getPointMultiplier(User user) {
        if (user.getRank() == Rank.GOLD) {
            return 4;
        } else if (user.getRank() == Rank.SILVER) {
            return 2;
        } else {
            return 1;
        }
    }

    public static void main(String[] args) {
        User user1 = new User("Alice", Rank.GOLD);
        User user2 = new User("Bob", Rank.SILVER);

        System.out.println(user1 + " - Multiplier: " + getPointMultiplier(user1));
        System.out.println(user2 + " - Multiplier: " + getPointMultiplier(user2));
    }
}
```

Example of a Violation of the Open-Closed Principle

- In the getPointMultiplier function, if a new Rank is added, the getPointMultiplier function itself must be modified. For example, if PLATINUM is added to the Rank, the getPointMultiplier function would be modified to include a case where user.rank == Rank.PLATINUM.

```
enum Rank {
    GOLD,
    SILVER,
    PLATINUM
}
```

```
public static int getPointMultiplier(User user) {
        if (user.getRank() == Rank.GOLD) {
            return 4;
        } else if (user.getRank() == Rank.SILVER) {
            return 2;
        } else if (user.getRank() == Rank.PLATINUM) {
            return 8;
        } else {
            return 1;
        }
    }
```

Example of a Violation of the Open-Closed Principle

- It may seem like a simple fix to add a branch, but this fix will affect the existing getPointMultiplier function, which has been working fine until now.
- There is a possibility that the person making the modification might write the conditional branch incorrectly. For this reason, with this modification, it would be necessary to test the behavior of not only PLATINUM but also GOLD and SILVER.
- In other words, this getPointMultiplier function violates the open-closed principle. It can be said that a better design is one that adheres to the principle and can extend functionality without modifying existing code.

Examples of Following the Open-Closed Principle

- Implement the above example according to the open-closed principle. First, define the user class.

```java
class User {
    private String name;
    private Rank rank;

    public User(String name, Rank rank) {
        this.name = name;
        this.rank = rank;
    }

    public String getName() {
        return name;
    }

    public Rank getRank() {
        return rank;
    }

    @Override
    public String toString() {
        return "Name: " + name + ", Rank: " + rank;
    }
}
```

Examples of following the open-closed principle

- Next, we define the Rank interface, and then define the GoldRank and SilverRank classes that implement it.

```java
interface Rank {
    int getPointMultiplier();
    String toString();
}
```

```java
class GoldRank implements Rank {
    @Override
    public int getPointMultiplier() {
        return 4;
    }

    @Override
    public String toString() {
        return "GOLD";
    }
}
```

```java
class SilverRank implements Rank {
    @Override
    public int getPointMultiplier() {
        return 2;
    }

    @Override
    public String toString() {
        return "SILVER";
    }
}
```

Examples of Following the Open-Closed Principle

```java
public class Application {
    public static void main(String[] args) {
        User user1 = new User("Alice", new GoldRank());
        User user2 = new User("Bob", new SilverRank());
        User user3 = new User("Sakura", new Platinum()); // +

        System.out.println(user1 + " - Multiplier: " + user1.getRank().getPointMultiplier());
        System.out.println(user2 + " - Multiplier: " + user2.getRank().getPointMultiplier());
        System.out.println(user3 + " - Multiplier: " + user3.getRank().getPointMultiplier());
    }
}
```

- In this way, we divide the classes into ranks and implement the getPointMultiplier function for each.

Examples of Following the Open-Closed Principle

- Because of this, when adding a new rank, you only need to create a new class, and you don't need to touch the existing classes. For example, even if PLATINUM is added to Rank, you only need to add a class like the one below.

```java
class Platinum implements Rank {
    @Override
    public int getPointMultiplier() {
        return 8;
    }

    @Override
    public String toString() {
        return "PLATINUM";
    }
}
```

- This change does not affect existing programs when they are updated. Therefore, this code adheres to the open-closed principle.

What Problems can Occur if the Open/Closed Principle is not Followed?

- If the open/closed principle is not followed, when adding functions, you have to modify the existing code.
- **If you modify the existing program to add functions, you will affect the existing code that was working fine up until that point.**
- As a result, maintainability decreases and bugs become more likely to occur. Even if the changes are minor, careless mistakes cannot be prevented.
- Also, because you are modifying the existing code, **testing is required each time you modify the existing code**. In other words, the man-hours required for testing increase.

Benefits of Following the Open/Closed Principle

- Following the open/closed principle makes **extension easier**.
- Also, because extension does not affect existing code, **it is only necessary to test the extended functions**. In other words, maintainability is improved.

Practices for Following the Open/Closed Principle

Abstracting varied behavior

- In order to follow the open/closed principle, it is important to **abstract out behaviors that can be extended**.
- Abstracting means extracting common behaviors and defining them as abstract classes or interfaces.
- For example, if the behavior differs depending on the situation, such as the type (e.g. occupation, membership rank) or storage destination (e.g. CSV, MySQL), then separate each behavior into a different class. Then implement these classes so that they have a common interface or inherit a common abstract class.

Practices for Following the Open/Closed Principle

- Abstract out anything that has the potential to be extended, and leave the concrete implementation to the subclasses. This way, **when adding new behavior, you only need to create a new class,** and there is no need to modify the existing classes.

- However, there is a limit to everything. Basically, classes should be implemented in a way that they depend on interfaces, but it is excessive to create an interface for every class**. You should always be aware of whether or not the number of variations will increase, and if it will, you should abstract it using an interface or something similar.** Also, there is no need to abstract from the beginning, and you can abstract when the number of variations increases.

Application: Using Factory Pattern

- There is a method of applying the factory pattern in an object-oriented approach, and leaving the branching at the time of generation to the factory class.
- The factory class is a class that is used solely for object creation. This class contains conditional branching that selects the object to be created. If conditional branching is specialized and included in the factory class, when the conditional branching is changed, only the factory class needs to be modified, and readability is improved.

L: Liskov Substitution Principle

L: Liskov Substitution Principle

The subtype must be substitutable with its supertype.

- A super type is the thing that is inherited from. For example, this refers to interfaces, abstract classes and virtual functions.
- A subtype is something that inherits from a super type. For example, this refers to classes that implement interfaces, classes that inherit from abstract classes, and classes that override virtual functions.
- To put it another way, in terms of inheritance, “instances of subtypes should behave in the same way as instances of superclasses”. When inheriting, you must ensure that there are no problems with using a subtype instead of a superclass. If there are problems, then you are using inheritance incorrectly.

The Purpose of the Liskov Substitution Principle

- This is a guideline for the correct use of inheritance. If the Liskov Substitution Principle is not satisfied, inheritance should not be used.
- In general, inheritance is referred to as an is-a relationship. However, the Liskov Substitution Principle requires that not only the is-a relationship, but also the behavior, be the same.
- This principle aims to maintain consistency so that the parent class and its child classes can be used in the same way without errors.
- If the Liskov substitution principle is satisfied, the programmer can use the superclass without knowing the subtype's behavior, as long as they know the superclass's behavior.
- Conversely, if the Liskov substitution principle is not satisfied, the programmer must know the subtype's behavior, and the code will become more complex.

Example of a Violation of the Liskov Substitution Principle

- For example, suppose we define a superclass Rectangle and a subclass Square as follows.

```java
class Rectangle {
    private double width;
    private double height;

{   public Rectangle(double width, double height)
        this.width = width;
        this.height = height;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double calculateArea() {
        return width * height;
    }
}
```

```java
class Square extends Rectangle {
    public Square(double side) {
        super(side, side);
    }

    @Override
    public void setWidth(double side) {
        super.setWidth(side);
        super.setHeight(side);
    }

    @Override
    public void setHeight(double side) {
        super.setWidth(side);
        super.setHeight(side);
    }
}
```

Example of a Violation of the Liskov Substitution Principle

- In this example, the Square class inherits from the Rectangle class, but the Square class's width setter behaves differently from the Rectangle class's width setter. To be more specific, the Rectangle class's width setter does not change the height, but the Square class's width setter does change the height.
- In other words, Square class instances do not behave in the same way as Rectangle class instances.

- Therefore, this example does not satisfy the Liskov substitution principle.

Example of Following the Liskov Substitution Principle

- The following is an example of modifying the code to satisfy the Liskov substitution principle. The Square class no longer inherits from the Rectangle class, and both the Rectangle class and the Square class inherit from the superclass Shape class.

```java
iabstract class Shape {
    public abstract double calculateArea();
}
```

```java
class Rectangle extends Shape {
    private double width;
    private double height;

    public Rectangle(double width,
                     double height) {
        this.width = width;
        this.height = height;
    }

    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }
```

```java
public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    @Override
    public double calculateArea() {
        return width * height;
    }
}
```

```java
class Square extends Shape {
    private double side;

    public Square(double side) {
        this.side = side;
    }

    public double getSide() {
        return side;
    }

    public void setSide(double side) {
        this.side = side;
    }

    @Override
    public double calculateArea() {
        return side * side;
    }
}
```

Example of Following the Liskov Substitution Principle

- Both the Rectangle class and the Square class now have the same interface as the Shape class, which they inherit from.
- As a result, instances of the Rectangle class and instances of the Square class behave in the same way as Shape, and the Liskov substitution principle is satisfied.

What problems can occur if you don’t follow the Liskov Substitution Principle?

- Normally, actors expect that they can treat super types and subtypes in the same way. Therefore, if a subtype behaves differently to a super type, it can lead to unexpected results for the actor. This can cause bugs.

- In particular, if the subtype improperly modifies the behavior of the super type, existing code that uses the subtype object as a super type object will no longer function correctly.

- Furthermore, if the subtype behaves differently from the super type, **the programmer will have to understand all of the contents of the subtype.** For example, if a method is added only to a certain subclass, the programmer will have to be aware of its existence. This can cause bugs due to the programmer's failure to check.

The Benefits of Following the Liskov Substitution Principle

- By following the Liskov substitution principle, you can add new functions and behaviors without changing the existing code.
- In particular, if the class hierarchy is designed according to this principle, you can freely replace objects of subclasses with objects of superclasses. In other words, this improves maintainability and extensibility.
- Furthermore, because there is no special behavior for subtypes alone, if you understand the super type's specifications, you can use it without checking the contents of the subtypes that inherit it.

Practices for Following the Liskov Substitution Principle

Do not add methods that are only available in subtypes

- If you add methods that only exist in a subtype, the instances of that subtype will no longer behave in the same way as instances of the super type. In other words, this violates the Liskov substitution principle.
- In particular, if methods are added only to a specific subtype, the programmer will have to understand and be aware of the contents of that subtype. This can cause implementation omissions and bugs.
- Also, if methods are only added to a specific subtype, the actor will have to use conditional branching such as if statements to treat that subtype specially. This would also break the Open/Closed principle.

Practices for Following the Liskov Substitution Principle

If no methods are overridden, don't inherit

- If a subtype does not override any of the superclass's methods, it is using inheritance to access functionality.
- If a subtype only needs to use the functionality of a superclass, it should stop using inheritance and use composition or some other method to use an instance of the superclass from within the subtype.
- Inheritance is not for sharing variables or methods between superclasses and subclasses.

Practices for Following the Liskov Substitution Principle

Avoid strengthening preconditions

- In an inheritance relationship, check that the subtype also permits all of the preconditions that the super type permits. The preconditions of the super type must not be strengthened by the subtype.
- Preconditions are conditions that must be met before a method or function is called. For example, this would include conditions for method arguments and the state of objects at the start of a method.
- For example, if the precondition of the super type is that x is any real number, but the precondition of the subtype is that x is a positive real number, then the precondition of the subtype is stronger than that of the super type. In this case, the Liskov substitution principle is not satisfied.

Practices for Following the Liskov Substitution Principle

Avoiding weakening of postconditions

- In an inheritance relationship, check that the subtype also guarantees all of the postconditions guaranteed by the supertype. The postconditions of the supertype must not be weakened by the subtype.
- A postcondition is a condition that must be satisfied after a method or function has been called. For example, this could be a condition for the return value of a method, or the state of an object when a method finishes.
- For example, let's say the postcondition of the super type is “The attribute ‘width’ is equal to the argument ‘width’, and the attribute ‘height’ is not changed”. In contrast, the postcondition of the subtype is “The attribute width is equal to the argument width, and the attribute height is also equal to the argument width”. In this case, the postcondition of the subtype does not include the postcondition of the super type “The attribute height is not changed”. In other words, the postcondition of the subtype has been weakened. Therefore, in this case, the Liskov substitution principle is not satisfied.

Practices for Following the Liskov Substitution Principle

Maintain interface consistency

- Ensure that the subtype accurately implements the super type's interface and does not add any unexpected behavior. For example, if the super type's method throws an exception, the subtype's method should also throw an exception.

Write unit tests

- Write a unit test that can confirm that the behavior of the subtype is the same as that of the super type. In this way, you can check whether the subtype has changed the behavior of the super type, i.e., whether it satisfies the Liskov substitution principle.

About Inheritance

- Inheritance is for subclasses to override the behavior of superclasses. It is <u>not for adding new functions to superclasses. It is not for sharing variables or methods between superclasses and subclasses.</u> Inheritance is not for getting the functions of superclasses, but for overriding the behavior of superclasses.
- Using inheritance because of similarities is a big mistake. If it does not satisfy the Liskov substitution principle, then inheritance should not be used. It is often the case that things that look similar are seen as the same, but this is just an illusion. “Similar” cannot become “the same”. Therefore, sharing things just because they are similar is the root cause of spaghetti code. Things that are not the same should not be shared.
- Also, when implementing, it should be implemented for the super type. If you do this, even if the number of subtypes increases, it will not affect the existing implementation. In particular, if you implement the subtype so that it cannot be used from the outside, and only expose the super type and the factory for creation to the outside, it will be easier to follow the Liskov substitution principle.

I: Interface Segregation Principle

I: Interface Segregation Principle

The interface client must not be forced to depend on properties/methods that the client does not use.

- This principle is about making interfaces easy for clients to use.
- Interfaces should have the fewest possible rules from the user's point of view.
- Clients should only be able to see the methods they need, not the ones they don’t.
- If some methods are not implemented in the client, you should consider separating the interface.
- If the interface is properly separated, the client does not have to implement unnecessary properties or methods.

Purpose of the Interface Segregation Principle

- The purpose of this principle is to divide the set of operations into smaller parts, and to execute only those that the class needs.
- Attempting to make a class execute operations that it does not use is a waste of resources, and if the class does not have the functionality to execute those operations, it may cause unexpected bugs.
- Classes should only perform the actions necessary to fulfill their role. Any other actions should either be removed completely or moved to a different location if there is a possibility that they may be used by other classes in the future.

Example of a Violation of the Interface Segregation Principle

- For example, consider the superclass Bird and its subclasses Duck and Penguin, as shown below.

```java
interface Bird{
    public void fly();
    public void swim();
}
```

```java
class Duck implements Bird{
    public void fly(){
        System.out.println("Duck is flying");
    }
    public void swim(){
        System.out.println("Duck is swiming");
    }
}
```

```java
class Application {
    public static void main(String[] args){
        new Duck().fly();
        new Duck().swim();
        new Penguin().fly();
        new Penguin().swim();
    }
}
```

```java
class Penguin implements Bird{
    public void fly(){
        throw new RuntimeException("Penguin cannot fly");
    }

    public void swim(){
        System.out.println("Penguin is swimming");
    }
}
```

Example of a Violation of the Interface Segregation Principle

- In the example above, the Bird class has a fly method and a swim method. The Duck class has no problem because it implements both methods. However, the Penguin cannot fly, so the Penguin does not need to implement the fly method.

- However, in this implementation, the Bird class has a fly method. Therefore, the Penguin class also has to implement the fly method. In other words, the Bird class, which is the super type, is forcing the Penguin class, which is the sub type, to implement the unnecessary fly method. This violates the principle of interface separation.

Example of Following the Interface Segregation Principle

- In order to follow the principle of the Interface segregation principle, it is desirable to separate the methods of the Bird class into the Flyable class and the Swimmable class.

```
interface Flyable{
    public void fly();
}
```

```
interface Swimmable{
    public void swim();
}
```

```
class Application{
    public static void main(String[] args){
        new Duck().fly();
        new Duck().swim();
        new Penguin().swim();
    }
}
```

```
class Duck implements Flyable, Swimmable{
    public void fly(){
        System.out.println("Duck is flying");
    }
    public void swim(){
        System.out.println("Duck is swiming");
    }
}
```

```
class Penguin implements Swimmable{
    public void swim(){
        System.out.println("Penguin is swimming");
    }
}
```

Example of Following the Interface Segregation Principle

- In the example above, the methods of the Bird class are separated into the Flyable interface and the Swimmable interface.
- The Duck class implements the Flyable class and the Swimmable class.
- The Penguin class implements the Swimmable class.
- By doing this, the Penguin class does not need to implement the unnecessary fly method, and the principle of interface separation is upheld.

What problems can occur if the Interface Segregation Principle is not followed?

- If the principle of interface separation is not followed, when changes are made to the interface, even if the method is not being used, modifications will be required in the subtype. This forced modification can cause bugs.
- In addition, if a subtype causes an error in a method that is not needed, or if it is implemented to do nothing, it violates the Liskov Substitution Principle.
- Furthermore, if an interface is used by multiple actors, the code becomes too complex because there are too many methods in the interface. In addition, it violates the Single Responsibility Principle.

The Benefits of Following the Interface Segregation Principle

- By properly separating interfaces, you can avoid implementing unnecessary/useless code in subtypes. This will help you to follow the Liskov substitution principle and make your code more robust against interface changes.
- In addition, it will make your code simpler, and improve its extensibility, readability and maintainability.

Practices for Following the Interface Segregation Principle

Separate interfaces appropriately

- Separate interfaces appropriately, and minimize the methods they contain. For example, separate interfaces by role. Avoid mixing multiple roles within an interface, and make it possible to add and remove roles and permissions.

Avoid implementing empty methods in subtypes

- If there are methods in the client of an interface that only throw errors or are empty, there is a high possibility of a violation of the principle. If you see these, consider separating the interface.

Avoid creating fat interfaces and interface soup

- An interface that contains everything is called a fat interface and is considered an anti-pattern.In particular, combining multiple separate interfaces into a single interface is an anti-pattern called interface soup. If you create a soup with interfaces as its ingredients, you will lose the benefits of interface separation.

D: Dependency Inversion Principle

D: Dependency Inversion Principle

- The upper-level module must not depend on the lower-level module. Both should depend on the abstraction.
- The abstraction must not depend on the details. The details should depend on the abstraction.

- **Upper-level module (or class)**: Class that uses a tool to perform an action
- **Lower-level module (or class)**: Tool required to perform an action
- **Abstraction**: Interface that connects the two classes
- **Details**: How the tool works

- In this principle, classes should not be merged with the tools used to perform operations. Rather, they should be merged with the interface that allows the tool to connect to the class.
- Neither the class nor the interface should know how the tool operates. However, the tool must meet the interface specifications.

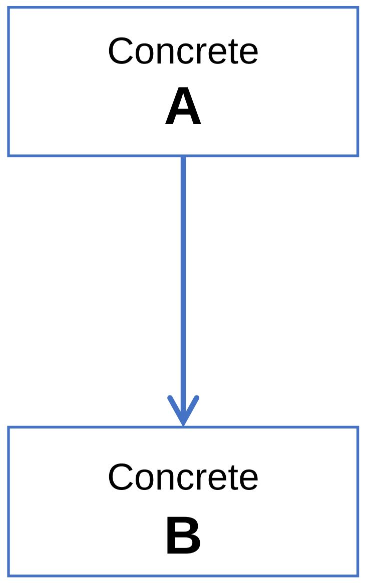
The Purpose of the Dependency Inversion Principle

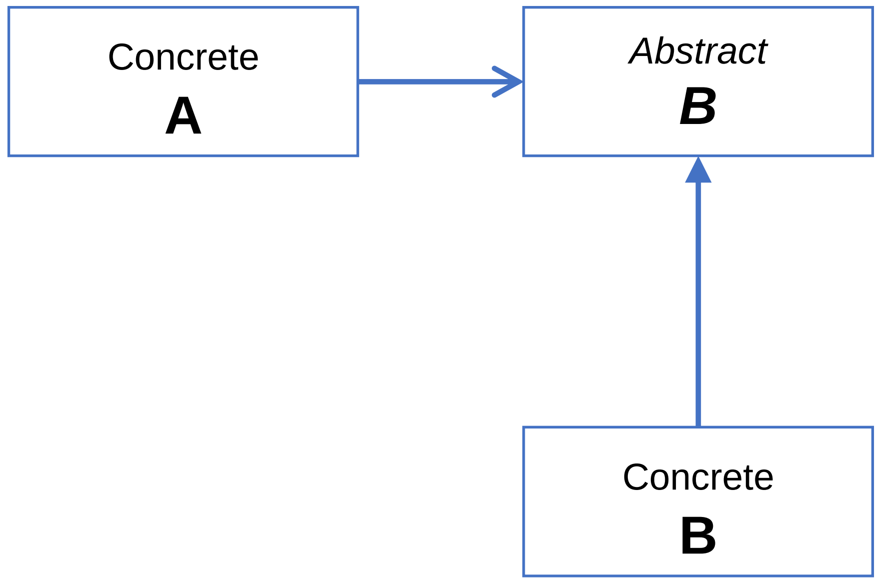
- This principle aims to reduce the dependency of higher-level classes on lower-level classes by introducing interfaces.
- Dependency refers to a module using the functionality of another module.
  - Higher-level module: a module that uses the functionality
  - Lower-level module: a module that is used
- The Dependency Inversion Principle is a principle for separating the dependencies between modules. **The dependencies of a module should be limited to abstractions, and it should not depend on concrete objects**.

The Process of Inversion of Dependency

- The word “abstract” refers to abstract classes and interfaces. For example, “save” is abstract. The details, such as how to save and where to save, are specific.
- If you write code without thinking about it, you will naturally end up with an implementation where Module A directly uses Module B. The **principle of inversion of dependency states that we should avoid this direct use, insert an abstraction in between, and implement so that both A and B depend on the abstraction.** *A* should depend on the abstraction of *B*, and *B* should be implemented so that it depends on the abstraction of *B*.
- This reverses the dependency relationship arrow, which was one-way from *A* to *B*, between the abstraction and *B*. This is why it is called inversion of dependency.

The Process of Inversion of Dependency





Example of a Violation of the Dependency Inversion Principle

- For example, consider the process of “a service storing specified data at a specified destination”. If the design is implemented carelessly, it will look like this.

```java
class File{
    public void save(String data, String path){
        System.out.println("Save " + data + " to File " + path);
    }
}
```

```java
class Service{
    private File file;
    Service(){
        file = new File();
    }

    public void execute(String data, String path){
        file.save(data, path);
    }
}
```

```java
class Application {
    public static void main(String[] args){
        new Service().execute("100", "A");
    }
}
```

Example of a Violation of the Dependency Inversion Principle

- In the example above, the Service class directly depends on the File class.
- Therefore, for example, if the save destination changes from File to Database, the implementation of the Service class itself must be changed.

Example of Following the Dependency Inversion Principle

- We abstract the “save” process and create a Repository class. Then, we create a File class and a Database class as details.

```
interface Repository{
    void save(String data, String path);
}
```

```
class File implements Repository{
    public void save(String data, String path){
        System.out.println("Save " + data + " to File " + path);
    }
}
```

```
class Database implements Repository{
    public void save(String data, String path){
        System.out.println("Save " + data + " to Database " + path);
    }
}
```

Example of Following the Dependency Inversion Principle

- The Service class does not depend on the File class or the Database class, but on the abstract Repository class.

```
class Service{
    private Repository repository;
    Service(Repository repository){
        this.repository = repository;
    }

    public void execute(String data, String path){
        repository.save(data, path);
    }
}
```

- In this way, the Service class now depends on the abstract Repository class, and the File class also depends on the abstract Repository class. Therefore, even if the storage destination changes from a file to a database, it is only necessary to change the instance passed to the Service class, and there is no need to change the implementation of the Service class.

What Problems can Occur if the Dependency Inversion Principle is not Followed?

- If a higher-level module directly depends on a lower-level module, the following problems can occur.
- **Changes to the lower-level module will affect the higher-level module**
- The higher-level module cannot be developed without developing the lower-level module
- **Processes cannot be changed flexibly**
- Unit testing becomes difficult because dummy objects for testing cannot be used.

Benefits of Following the Dependency Inversion Principle

- When you follow the Dependency Inversion Principle, the dependency between classes is weakened. Therefore, **even if a lower-level module is changed, the upper-level module will not be affected unless the abstraction is changed.** This reduces the amount of work required for testing.
- Also, because the lower-level modules depend on the abstractions, **it is easy to switch the modules used by the upper-level modules**. This improves the flexibility of the entire system and makes testing easier.
- In short, by separating the abstractions from the details, you become more resistant to changes in the details.

Practices for Following the Dependency Inversion Principle

Do not refer to *mutable* concrete classes

- Do not directly refer to concrete classes using variables, etc.
- Refer to abstract classes and interfaces instead of concrete classes.
- However, it is not necessary to follow all of these principles. What you don't want to depend on are the concrete elements in the system that are prone to change. If the concrete elements are unlikely to change, it is unlikely to cause a problem even if you depend on them. Therefore, it is acceptable to hold concrete elements that do not change or are unlikely to change.

Do not override instance methods/functions

- Overriding functions does not eliminate dependencies. Instead of overriding concrete functions, make the original functions abstract functions and override them.

Injecting Dependencies

- One technique for implementing the principle of dependency inversion is **Dependency Injection** (DI).
- DI refers to the practice of not creating dependent objects yourself, but rather receiving them from the outside via constructors, setters, etc.
- By receiving instances from the outside rather than creating them directly internally, the degree of dependency can be reduced.

Injecting Dependencies

- This is another example of DI.

```
class Application {
    public static void main(String[] args){
        new Service(new Database()).execute("100", "A");
        new Service(new File()).execute("200", "B");
    }
}
```

- By using DI, we can realize the principle of inversion of dependency and achieve flexible design.

DI Container

- However, DI has its disadvantages. One of these is that it is difficult to create instances passed from the outside. In particular, if the creation method is complex, the code becomes more complex when using DI.
- A **DI container** is a system that automatically generates instances for DI. By registering the class you want to instantiate and the generation method in the DI container, and then instantiating it via the DI container, you can obtain an instance in the state you registered in advance. This saves you the trouble of generating instances from outside.

Factory Pattern

- In effect, in order to create an object, it is necessary to create a concrete class somewhere. In addition to DI containers, the **Factory Pattern** is also useful for this.
- The Factory Pattern is a method of creating an instance by preparing a class that is used only for creating instances. By using the Factory Pattern, you can delegate the creation of instances to an external source without using a DI container.

Reference

- Robert C.Martin, Clean Architecture