

CSE - 325
Information System Design
SOLID PRINCIPLES

NAME: MD. ZARZEES UDDIN SHAH CHOWDHURY

ID: 1805009

GROUP: 1

LECTURE: 1

DATE: 09/03/2023

The SOLID Principles are five principles of Object-Oriented class design. They are a set of rules and best practices to follow while designing a class structure.

These five principles help us understand the need for certain design patterns and software architecture in general.

Let's look at each principle one by one. Following the SOLID acronym, they are:

- The **S**ingle Responsibility Principle
- The **O**pen-Closed Principle
- The **L**iskov Substitution Principle
- The **I**nterface Segregation Principle
- The **D**ependency Inversion Principle

The Single Responsibility Principle

The Single Responsibility Principle states that **a class should do one thing and therefore it should have only a single reason to change.**

To state this principle more technically: Only one potential change (database logic, logging logic, and so on) in the software's specification should be able to affect the specification of the class.

This means that if a class is a data container, like a Book class or a Student class, and it has some fields regarding that entity, it should change only when we change the data model.

Following the Single Responsibility Principle is important. First of all, because many different teams can work on the same project and edit the same class for different reasons, this could lead to incompatible modules.

Second, it makes version control easier. For example, say we have a persistence class that handles database operations, and we see a change in that file in the GitHub commits. By following the SRP, we will know that it is related to storage or database-related stuff.

Merge conflicts are another example. They appear when different teams change the same file. But if the SRP is followed, fewer conflicts will appear – files will have a single reason to change, and conflicts that do exist will be easier to resolve.

Common Pitfalls and Anti-patterns

Here are some common mistakes that violate the Single Responsibility Principle. We will look at the code for a simple bookstore invoice program as an example. Let's start by defining a book class to use in our invoice.

```
class Book {
    String name;
    String authorName;
    int year;
    int price;
    String isbn;

    public Book(String name, String authorName, int year, int price, String isbn) {
        this.name = name;
        this.authorName = authorName;
        this.year = year;
        this.price = price;
        this.isbn = isbn;
    }
}
```

This is a simple book class with some fields.

Now let's create the invoice class which will contain the logic for creating the invoice and calculating the total price. For now, we assume that our bookstore only sells books and nothing else.

```

class Invoice {

    private Book book;
    private int quantity;
    private double discountRate;
    private double taxRate;
    private double total;

    public Invoice(Book book, int quantity, double discountRate, double taxRate) {
        this.book = book;
        this.quantity = quantity;
        this.discountRate = discountRate;
        this.taxRate = taxRate;
        this.total = this.calculateTotal();
    }

    public double calculateTotal() {
        double price = ((book.price - book.price * discountRate) * this.quantity);

        double priceWithTaxes = price * (1 + taxRate);

        return priceWithTaxes;
    }

    public void printInvoice() {
        System.out.println(quantity + "x " + book.name + " " + book.price + "$");
        System.out.println("Discount Rate: " + discountRate);
        System.out.println("Tax Rate: " + taxRate);
        System.out.println("Total: " + total);
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }
}

```

Here is our invoice class. It also contains some fields about invoicing and 3 methods:

- **calculateTotal** method, which calculates the total price,
- **printInvoice** method, that should print the invoice to console, and
- **saveToFile** method, responsible for writing the invoice to a file.

Our class violates the Single Responsibility Principle in multiple ways.

The first violation is the **printInvoice** method, which contains our printing logic. The SRP states that our class should only have a single reason to change, and that reason should be a change in the invoice calculation for our class.

But in this architecture, if we wanted to change the printing format, we would need to change the class. This is why we should not have printing logic mixed with business logic in the same class.

There is another method that violates the SRP in our class: the **saveToFile** method. It is also an extremely common mistake to mix persistence logic with business logic. We can create new classes for our printing and persistence logic so we will no longer need to modify the invoice class for those purposes.

We create 2 classes, **InvoicePrinter** and **InvoicePersistence**, and move the methods.

```
class InvoicePrinter {
    private Invoice invoice;

    public InvoicePrinter(Invoice invoice) {
        this.invoice = invoice;
    }

    public void print() {
        System.out.println(invoice.quantity + "x " + invoice.book.name + " " + invoice.book.price + " $");
        System.out.println("Discount Rate: " + invoice.discountRate);
        System.out.println("Tax Rate: " + invoice.taxRate);
        System.out.println("Total: " + invoice.total + " $");
    }
}
```

```
public class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }
}
```

Now our class structure obeys the Single Responsibility Principle and every class is responsible for one aspect of our application.

Open-Closed Principle

The Open-Closed Principle requires that **classes should be open for extension and closed to modification**. Modification means changing the code of an existing class, and extension means adding new functionality.

So what this principle wants to say is: We should be able to add new functionality without touching the existing code for the class. This is because whenever we modify the existing code, we are taking the risk of creating potential bugs. So we should avoid touching the tested and reliable (mostly) production code if possible.

But how are we going to add new functionality without touching the class? It is usually done with the help of interfaces and abstract classes. Let's say we want invoices to be saved to a database so that we can search them easily.

We create the database, connect to it, and we add a save method to our **InvoicePersistence** class:

```
class InvoicePersistence {
    Invoice invoice;

    public InvoicePersistence(Invoice invoice) {
        this.invoice = invoice;
    }

    public void saveToFile(String filename) {
        // Creates a file with given name and writes the invoice
    }

    public void saveToDatabase() {
        // Saves the invoice to database
    }
}
```

Unfortunately we did not design the classes to be easily extendable in the future. So in order to add this feature, we have modified the **InvoicePersistence** class.

If our class design obeyed the Open-Closed principle we would not need to change this class.

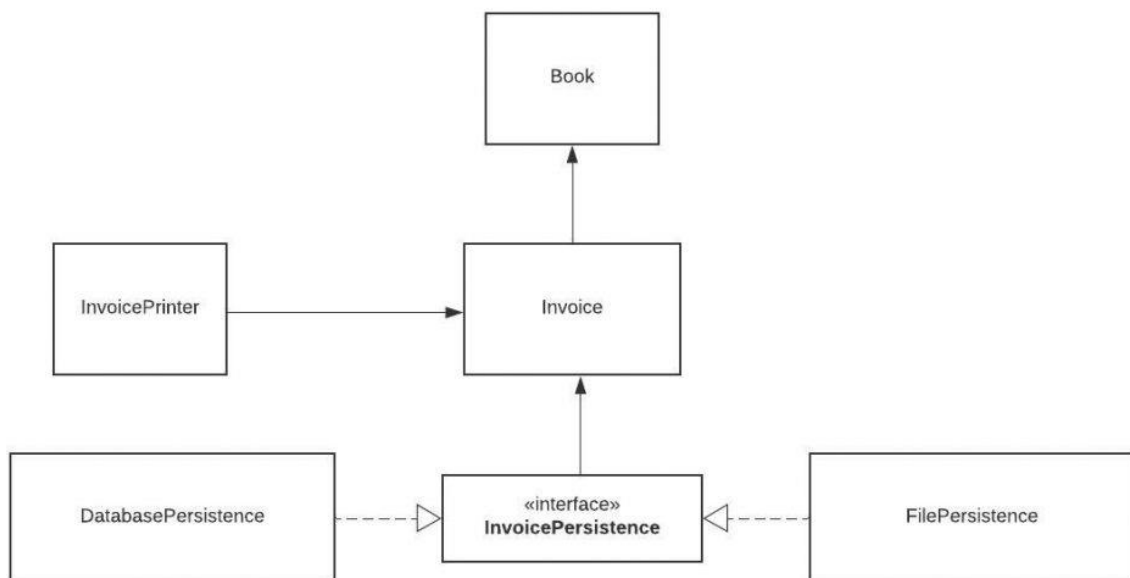
So, we see the design problem and decide to refactor the code to obey the principle.

```
interface InvoicePersistence {
    public void save(Invoice invoice);
}
```

We change the type of **InvoicePersistence** to Interface and add a save method. Each persistence class will implement this save method.

```
public class DatabasePersistence implements InvoicePersistence {
    @Override
    public void save(Invoice invoice) {
        // Save to DB
    }
}
public class FilePersistence implements InvoicePersistence {
    @Override
    public void save(Invoice invoice) {
        // Save to file
    }
}
```

So our class structure now looks like this:



Now our persistence logic is easily extendable. If we are asked to add another database and have 2 different types of databases like MySQL and MongoDB, we can easily do that.

Liskov Substitution Principle

The Liskov Substitution Principle states that subclasses should be substitutable for their base classes.

This means that, given that class B is a subclass of class A, we should be able to pass an object of class B to any method that expects an object of class A and the method should not give any weird output in that case. This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has. The child class extends the behavior but never narrows it down. Therefore, when a class does not obey this principle, it leads to some nasty bugs that are hard to detect.

Liskov's principle is easy to understand but hard to detect in code. So let's look at an example.

```
class Rectangle {
    protected int width, height;

    public Rectangle() {
    }

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

    public int getWidth() {
        return width;
    }

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

    public int getHeight() {
        return height;
    }

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

    public int getArea() {
        return width * height;
    }
}
```


We have a simple Rectangle class, and a **getArea** function which returns the area of the rectangle.

Now we decide to create another class for Squares.

```
class Square extends Rectangle {
    public Square() {}

    public Square(int size) {
        width = height = size;
    }

    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setHeight(height);
        super.setWidth(height);
    }
}
```

Our Square class extends the Rectangle class. We set height and width to the same value in the constructor, but we do not want any client (someone who uses our class in their code) to change height or weight in a way that can violate the square property.

Therefore we override the setters to set both properties whenever one of them is changed. But by doing that we have just violated the Liskov substitution principle.

Let's create a main class to perform tests on the **getArea** function.

```

class Test {

    static void getAreaTest(Rectangle r) {
        int width = r.getWidth();
        r.setHeight(height: 10);
        System.out.println("Expected area of " + (width * 10) + ", got " + r.getArea());
    }

    public static void main(String[] args) {
        Rectangle rc = new Rectangle(width: 2, height: 3);
        getAreaTest(rc);

        Rectangle sq = new Square();
        sq.setWidth(width: 5);
        getAreaTest(sq);
    }
}

```

A tester can come up with the testing function **getAreaTest** and can tell that our **getArea** function fails to pass the test for square objects.

In the first test, we create a rectangle where the width is 2 and the height is 3 and call **getAreaTest**. The output is 20 as expected, but things go wrong when we pass in the square. This is because the call to **setHeight** function in the test is setting the width as well and results in an unexpected output.

Interface Segregation Principle

Segregation means keeping things separated, and the Interface Segregation Principle is about separating the interfaces. The principle states that many client-specific interfaces are better than one general-purpose interface. Clients should not be forced to implement a function they do not need.

This is a simple principle to understand and apply, so let's see an example.

```

public interface ParkingLot {

    void parkCar(); // Decrease empty spot count by 1
    void unparkCar(); // Increase empty spots by 1
    void getCapacity(); // Returns car capacity
    double calculateFee(Car car); // Returns the price based on number of hours
    void doPayment(Car car);
}

class Car {

}

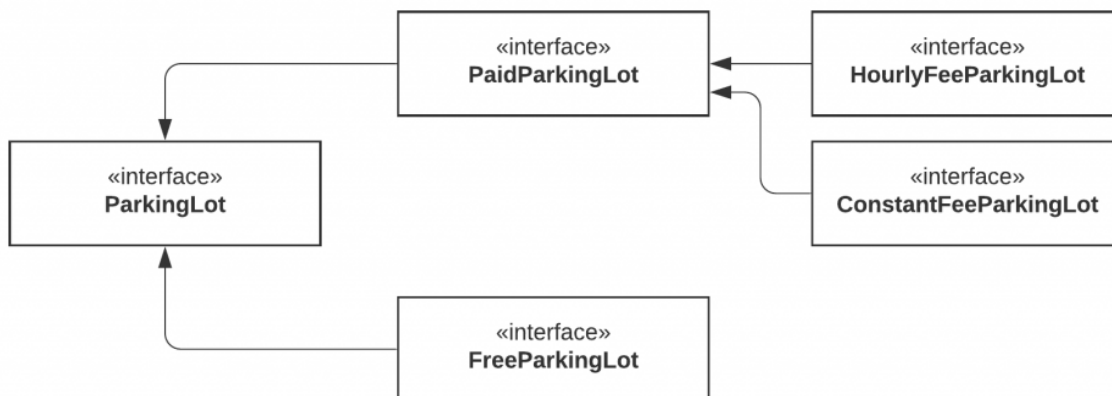
```

We modeled a very simplified parking lot. Now if we want to implement a parking lot that is free,

```
public class FreeParking implements ParkingLot {  
  
    @Override  
    public void parkCar() {  
  
    }  
  
    @Override  
    public void unparkCar() {  
  
    }  
  
    @Override  
    public void getCapacity() {  
  
    }  
  
    @Override  
    public double calculateFee(Car car) {  
        return 0;  
    }  
  
    @Override  
    public void doPayment(Car car) {  
        throw new Exception(message: "Parking lot is free");  
    }  
}
```

Our parking lot interface was composed of 2 things: Parking related logic (park car, unpark car, get capacity) and payment related logic.

But it is too specific. Because of that, our FreeParking class was forced to implement payment-related methods that are irrelevant. Let's separate or segregate the interfaces.



We've now separated the parking lot. With this new model, we can even go further and split the **PaidParkingLot** to support different types of payment.

Now our model is much more flexible, extendable, and the clients do not need to implement any irrelevant logic because we provide only parking-related functionality in the parking lot interface.

Dependency Inversion Principle

The Dependency Inversion principle states that our classes should depend upon interfaces or abstract classes instead of concrete classes and functions. If the OCP states the goal of OO architecture, the DIP states the primary mechanism. These two principles are indeed related and we have applied this pattern before while we were discussing the Open-Closed Principle.

Conclusion

It is not a surprise that all the concepts of clean coding, object-oriented architecture, design patterns and SOLID principles are somehow connected and complementary to each other

They all serve the same purpose:

"To create understandable, readable, and testable code that many developers can collaboratively work on."