SOLID PRINCIPLES ASSIGNMENT

Single Responsibility Principle (SRP):

Watch Class (Watch.java):

Responsibility: Representing a watch entity.

The Watch class is responsible for representing a watch entity with properties like id and name.

It does not more than one reason to change.

If you added the **TimeInterface** to the Watch class and then used the Watch class as part of the **Cart** class (which implements **WatchListInterface**), you might face several problems, primarily related to violating the **Single Responsibility Principle (SRP)** and potential confusion in the design.

Let's analyze the potential issues:

Violation of SRP :The **TimeInterface** introduces a new responsibility related to time display.

The **Watch** class, being part of the **Cart**, is now responsible not only for being a part of the shopping cart but also for potentially displaying time, violating the SRP.

Unexpected Behavior: A Watch in a shopping cart is not expected to display time; it is primarily there to represent a product in the cart.

Users of the **Cart** class might find it confusing if the **Watch** class suddenly introduces time-related behavior within the context of the shopping cart.

Open/Closed Principle (OCP) :

Scenario of Violation of OCP: In the scenario where the **Watch** class is not designed to be open for extension, if you wanted to introduce a new type of watch (let's say, a digital watch) without violating the **Open/Closed Principle**, you might need to modify the existing Watch class. Here's how it might look in a non-Open/Closed compliant scenario:

```
// Hypothetical Watch class without being open for extension

public class Watch {

    private int id ;
```

```
    private String name ;

    private String type ; // New field added

    // ... other properties, constructors, getters, and setters ...

    public void displayTime() {

        if ("Analog".equals(type)) {

            // Display analog time logic

        } else if ("Digital".equals(type)) {

            // Display digital time logic

        } else {

            // Handle other types

        }

    }

}
```

In this modified scenario, the **displayTime** method has been altered to include a conditional statement to handle different types of watches. This violates the **Open/Closed Principle** because when a new watch type is introduced, you must modify the existing **Watch** class to accommodate it. This kind of modification makes the code less maintainable and can lead to unintended consequences in the existing functionality.

Contrast this with the original code:

```
public abstract class Watch  {

    // ... properties, constructors, getters, and setters ...

    // Abstract method to be implemented by subclasses

}
```

TimeInterface for displayTime() :

```
public interface TimeInterface {
```

```
void displayTime();

}
```

And the subclasses:

```
public class AnalogWatch extends Watch implements TimeInterface {

   // ... implementation specific to AnalogWatch ...

}

public class SmartWatch extends Watch implements DateInterface, TimeInterface {

   // ... implementation specific to SmartWatch ...

}
```

In this design, each type of watch is encapsulated in its own class, extending the base **Watch** class. Each subclass provides its own implementation of the **displayTime** method. This adheres to the **Open/Closed Principle** because you can add new watch types by introducing new subclasses without modifying the existing **Watch** class or the behavior of other watch types.

Liskov Substitution Principle (LSP) :

If I did not follow the **Liskov Substitution Principle (LSP)** when implementing the **displayWatchTime** and **displayWatchDate** methods, I might encounter issues related to unexpected behavior or potential violations of the principle. Let's explore a hypothetical scenario where **LSP** is not followed:

Scenario without LSP: Suppose you did not adhere to **LSP** and, instead of using interfaces, you directly used specific watch classes in the **displayWatchTime** and **displayWatchDate** methods:

```
private static void displayWatchTime(AnalogWatch analogWatch) {

   analogWatch.displayTime();

}

private static void displayWatchDate(SmartWatch smartWatch) {

   smartWatch.displayDate();
```

```
}
```

Potential Problems:

1. Rigidity in Code:

•The methods now accept specific subclasses (**AnalogWatch** and **SmartWatch**) instead of interfaces (**TimeInterface** and **DateInterface**).

•This creates rigidity because these methods can only work with instances of **AnalogWatch** and **SmartWatch**. If a new type of watch is introduced, these methods need modification.

2. Violation of LSP:

•**LSP** violation occurs because these methods are now tied to specific subclasses, breaking the principle that objects of a superclass should be substitutable by objects of its subclasses without affecting the correctness of the program.

3. Limited Reusability:

•The methods become less reusable. If you want to use these methods with a different type of watch (e.g., a future **DigitalWatch** class), you'd need to modify the methods, violating the principle of substitutability.

4. Maintenance Challenges:

•As new watch types are introduced, the methods need continuous modification, making the codebase more challenging to maintain and extending the system more error prone.

Solution with LSP: Using interfaces (**TimeInterface** and **DateInterface**) allows the methods to accept any object that implements these interfaces, providing flexibility, reusability, and adhering to LSP:

```
private static void displayWatchTime(TimeInterface watch) {
    watch.displayTime();
}
private static void displayWatchDate(DateInterface watch) {
    watch.displayDate();
}
```

Benefits of LSP Adherence:

1.Flexibility: Methods can work with any class that implements the required interfaces, promoting flexibility in design.

2. Reusability: Methods can be reused with any future watch classes that implement the necessary interfaces, enhancing code reusability.

3.Extensibility: As new watch types are introduced, no modifications are needed in the existing methods, following the principle of open/closed systems.

By adhering to **LSP**, your code remains more adaptable, and future changes can be accommodated without the need for extensive modifications in existing methods.

Interface Segregation Principle (ISP) :

The Interface Segregation Principle (ISP) states that a class should not be forced to implement interfaces it does not use. It encourages creating small, specific interfaces rather than large, monolithic ones. Let's explore the importance of ISP in your program using the **DateInterface** and **TimeInterface**, consider a scenario where ISP is not followed, and how your code addresses it:

Scenario without ISP: Suppose initially you had a single, combined interface for both date and time functionality:

```
// A single interface for both date and time functionality

public interface DateTimeInterface {

   void displayTime();

   void displayDate();

}
```

And you forced your classes to implement this interface:

```
public class AnalogWatch implements DateTimeInterface {
```

```
    // Implementation of displayTime and displayDate for analog watch...
}

public class SmartWatch implements DateTimeInterface {
    // Implementation of displayTime and displayDate for smart watch...
}
```

Potential Issues without ISP:

1. Forced Implementations: If a class, like AnalogWatch or SmartWatch, does not need both date and time functionality, it is forced to implement methods it does not use. This violates ISP.

2. Potential Confusion: It can be confusing for clients of the class if they see methods they don't expect to be relevant for a specific type of watch.

Solution with ISP:

This issue is solved by breaking down the interfaces into smaller, more focused ones:

```
public interface TimeInterface {
    void displayTime();
}

public interface DateInterface {
    void displayDate();
}
```

And then, each class can implement only the interfaces it needs:

```
public class AnalogWatch implements TimeInterface {
    // Implementation of displayTime for analog watch...
}
```

```
public class SmartWatch implements TimeInterface, DateInterface {

   // Implementation of displayTime and displayDate for smart watch...

}
```

Benefits of Using ISP:

1. Focused Interfaces: Interfaces are more focused and cater to specific functionalities (time or date), making them clearer and more cohesive.

2. No Forced Implementations: Classes are not forced to implement methods they don't need, promoting adherence to ISP.

3. Better Client Understanding: Clients using these classes have a clearer understanding of the functionality each class provides without unnecessary methods.

By following ISP, interfaces are more modular, ensuring that classes only need to implement the specific functionalities they require, leading to a more maintainable and understandable design.

Dependancy Inversion Principle (DIP) :