**SOLID Principles Assignment**

In Rolex Clone I did Store Page, so I Created a java project to depict Store Flow

I decided to create 2 projects one without SOLID Principles and one with all Principles

The first Project without SOLID Principles includes 2 Entity classes (Order, Watch), 3 Service classes (Cart, Payment, RolexStore ) and 1 Utility class (StoreFlow).

The Java Code is in the files present with this

### There Are Many Situations Where the Code Will Break:

* **Single Responsibility Principle (SRP):**
  + The RolexStore class is responsible for both managing the shopping cart and processing payments and handling the checkout process violating SRP.
* **Open/Closed Principle (OCP):**
  + If we want to introduce a new payment method, we might need to modify the Payment class, violating OCP.
  + The code might not be open for extension but closed for modification, especially in the RolexStore class.
* **Liskov Substitution Principle (LSP):**
  + The Order class does not support easy substitution of different order types (e.g., online orders vs. in-store orders).
  + There is no clear indication that the classes follow the Liskov Substitution Principle.
* **Interface Segregation Principle (ISP):**
  + There may be methods in the Payment that are not needed for all payment methods.
  + If there are future requirements that necessitate different behaviors for carts or payments, interface might lead to implementing unnecessary methods.
* **Dependency Inversion Principle (DIP):**
  + The RolexStore class is directly dependent on concrete implementations of the Cart and Payment classes.

To Fix these Issues I codded again from scratch to create a store flow with SOLID methods

The New Project Includes 3 Entity classes (Cart, RolexStore, Watch), 3 Service Classes (CreditCardPayment, InStoreOrder, OnlineOrder), 4 Interfaces (CartOperations, Order, Payment, StoreOperations) and 1 Utility Class (StoreFlow).

The Java Code is in the files present with this

**Here is an Explanation of what each principle is, how it is used, and its benefits.**

### **1. Single Responsibility Principle (SRP):**

**Principle:** A class should have only one reason to change, meaning it should have only one responsibility.

**Usage in Code:**

* Cart class is responsible for managing items in the shopping cart (addProduct, removeProduct, calculateTotalAmount).
* RolexStore class is responsible for store operations (addItemToCart, removeItemFromCart, viewCart, checkout).
* Each class has a single responsibility.

**Benefits:**

* Code is easier to understand, maintain, and extend because each class has a clear and focused responsibility.
* Changes related to one responsibility do not affect the other, reducing the risk of introducing bugs.

### **2. Open/Closed Principle (OCP):**

**Principle:** Software entities (classes, modules, functions) should be open for extension but closed for modification.

**Usage in Code:**

* The RolexStore class is open for extension (you can create new classes implementing Order without modifying RolexStore).
* New order types (OnlineOrder, InStoreOrder) can be added without modifying existing code. We can even add more like MobileOrder.

**Benefits:**

* Promotes the creation of reusable and extensible code.
* Allows for the addition of new features without changing existing, well-tested code.

### **3. Liskov Substitution Principle (LSP):**

**Principle:** Subtypes must be substitutable for their base types without altering the correctness of the program.

**Usage in Code:**

* OnlineOrder and InStoreOrder implement the Order interface.
* They can be used interchangeably in the checkout method of RolexStore.

**Benefits:**

* Promotes polymorphism and the use of interfaces or abstract classes.
* Simplifies code by allowing the use of different implementations without worrying about breaking existing functionality.

### **4. Interface Segregation Principle (ISP):**

**Principle:** A class should not be forced to implement interfaces it does not use.

**Usage in Code:**

* Each interface (CartOperations, Order, Payment, StoreOperations) is specific and focused on a particular set of related operations.
* No class is forced to implement methods it does not use.

**Benefits:**

* Classes only implement methods relevant to their functionality.
* Avoids the problem of "fat interfaces" where implementing classes are forced to provide unnecessary methods.

### **5. Dependency Inversion Principle (DIP):**

**Principle:** High-level modules should not depend on low-level modules. Both should depend on abstractions.

**Usage in Code:**

* RolexStore depends on abstractions (CartOperations, Payment) rather than concrete implementations.
* Allows for flexibility and easy substitution of different implementations for cart operations and payment.

**Benefits:**

* Reduces coupling between high-level and low-level modules.
* Enables easy changes in implementations without affecting higher-level modules.

In summary, following the SOLID principles results in code that is modular, maintainable, and flexible. It encourages the creation of code that is easy to understand, extend, and adapt to changing requirements over time.