SOLID Principles

S – Single Responsibility Principle:

A class should have only one reason to change.

Example:

```
class Ball {
        String color;
        int size;
        int price;
        // Parameterized Constructor.
        // Getters and Setters.
}
Class Invoice {
        Ball ball;
        int qty;
        // Parameterized Constructor.
        public int calculateTotal() {
                return ball.price * qty;
        }
        public void printInvoice() {
                // Print the invoice.
        }
        public void saveDB() {
                // Save the invoice into DB.
        }
}
```

Violation of SRP:

- The Invoice class is responsible for multiple concerns:
    1. Calculating the total price.
    2. Printing the invoice.
    3. Saving the invoice to the database.
- By combining different responsibilities, the class becomes less cohesive and harder to maintain or extend.

Refactored Example:

```
class Invoice {
        Ball ball;
        int qty;
        // Parameterized Constructor.
        public int calculateTotal() {
                return ball.price * qty;
        }
}
class PrintInvoice {
        public void printInvoice(Invoice invoice) {
                // Print the Invoice.
        }
}
class InvoiceDao {
        public void save(Invoice invoice) {
                // Save the Invoice into DB.
        }
}
```

Refactored design effectively adheres to the SRP by isolating different responsibilities into separate classes. This results in a more maintainable, testable, and scalable codebase.

O – Open/Closed Principle:

Software entities should be open for extension and closed for modification.

Example:

```
class InvoiceDao {
        public void saveDB(Invoice invoice) {

        }
}
```

In this example, if you need to add new functionalities (e.g saving to a file), you would need to modify the InvoiceDao class. This violates the Open/Closed Principle because the class is not open for extension but closed for modification.

Refactored Example:

```
interface InvoiceDao {
        void save(Invoice invoice);
}
class DatabaseInvoiceDao implements InvoiceDao {
        @Override
        public void save(Invoice invoice) {
                // Save the invoice inside the database.
        }
}
class FileInvoiceDao implements InvoiceDao {
        @Override
        public void save(Invoice invoice) {
                // Save the invoice inside the file.
        }
}
```

In the above program, you use the interface InvoiceDao that defines a contract for saving invoices. You have two implementations of this interface: DatabaseInvoiceDao and FileInvoiceDao. This approach adheres to the Open/Closed Principle because you can add new ways to save an invoice without modifying existing code.

L – Liskov Substitution Principle:

If class B is a sub type of class A, then we should be able to replace the object of A with the object of B, without breaking the program.

Example:

```
interface Bike {
        void turnOnEngine();
        void accelerate();
}
class MotorCycle implements Bike {
        boolean turnOnEngine = false;
        int speed = 0;
        @Override
        public void turnOnEngine() {
                turnOnEngine = true;
        }
        @Override
        public void accelerate() {
                // Increasing the speed.
                speed += 10;
        }
}
class Bicycle implements Bike {
        int speed = 0;
        @Override
        public void turnOnEngine() {
                throw new AssertionError(“there is no engine”);
        }
        @Override
        public void accelerate() {
                speed += 10;
        }
}
```

Bicycle class violates LSP because its turnOnEngine method throws an exception, while MotorCycle implements this method as expected for an engine-based vehicle. A user of a Bike interface expects all implementations to have a meaningful turnOnEngine method.

To adhere to LSP, the Bike interface and its implementations should be designed so that all subclasses can be used interchangeably without causing unexpected behaviour.

Refactored Example:

```
interface EngineBike {
        void turnOnEngine();
        void accelerate();
}
interface NonEngineBike {
        void accelerate();
}
class MotorCycle implements EngineBike {
        // All the implementations of EngineBike interface.
}
class Bicycle implements NonEngineBike {
        // All the implementations of NonEngineBike interface.
}
```

I – Interface Segmentation Principle:

A class should not force to implement the interfaces / functionalities in which it does not use.

Example:

```
interface RestaurantEmployee {
        void serveCustomer();
        void cookFood();
        void dishWashes();
}
class Waiter implements RestaurantEmployee {
        @Override
        public void serveCustomer() {

        }
        @Override
        public void cookFood() {

        }
        @Override
        public void dishWashes() {

        }
}
```

Waiter class is forced to give implementations for the functionalities in which it does not need or use. Hence the Waiter class is violating the Interface Segmentation principle.

Refactored Example:

```
interface WaiterInterface {
	void takeOrder();
	void serveCustomer();
}
interface ChefInterface {
	void decideMenu();
	void cookFood();
}
class Waiter implements WaiterInterface {
	@Override
	public void takeOrder() {

	}
	@Override
	public void serveCustomer() {

	}
}
```

By splitting the large interface into smaller, role-specific interfaces, you adhere to ISP and create a design that is more flexible, maintainable, and easier to understand. This approach not only improves code quality but also ensures that classes implement only the methods they need, avoiding unnecessary dependencies and responsibilities.

D – Dependency Inversion Principle:

Class should only depend on interfaces rather than concrete classes.

```
class WiredKeyboard {

}
class WiredMouse {

}
class MacBook {
        private final WiredKeyboard wiredKeyboard;
        private final WiredMouse wiredMouse;
        public MacBook() {
                wiredKeyboard = new WiredKeyboard();
                wiredMouse = new WiredMouse();
        }
}
```

MacBook class is tightly coupled to WiredKeyboard and WiredMouse classes. This means any change to the keyboard or mouse types requires modifying the MackBook class. Hence it's violates the Interface Segmentation principle.

Example:

```
interface KeyBoard {

}
class WiredKeyBoard implements KeyBoard {

}
Class BluetoothKeyBoard implements KeyBoard {

}
interface Mouse {

}
Class WiredMouse implements Mouse {

}
class BluetoothMouse implements Mouse {

}
class MacBook {
        private final KeyBoard keyboard;
        private final Mouse mouse;
```

```
    public MacBook (KeyBoard keyboard, Mouse mouse) {
        this.keyboard = keyboard;
        this.mouse = mouse;
    }
}
```

Refactored implementation follows the Dependency Inversion principle well. By depending on abstractions (interfaces) rather than concrete implementations, you enhance the flexibility and maintainability of your code.