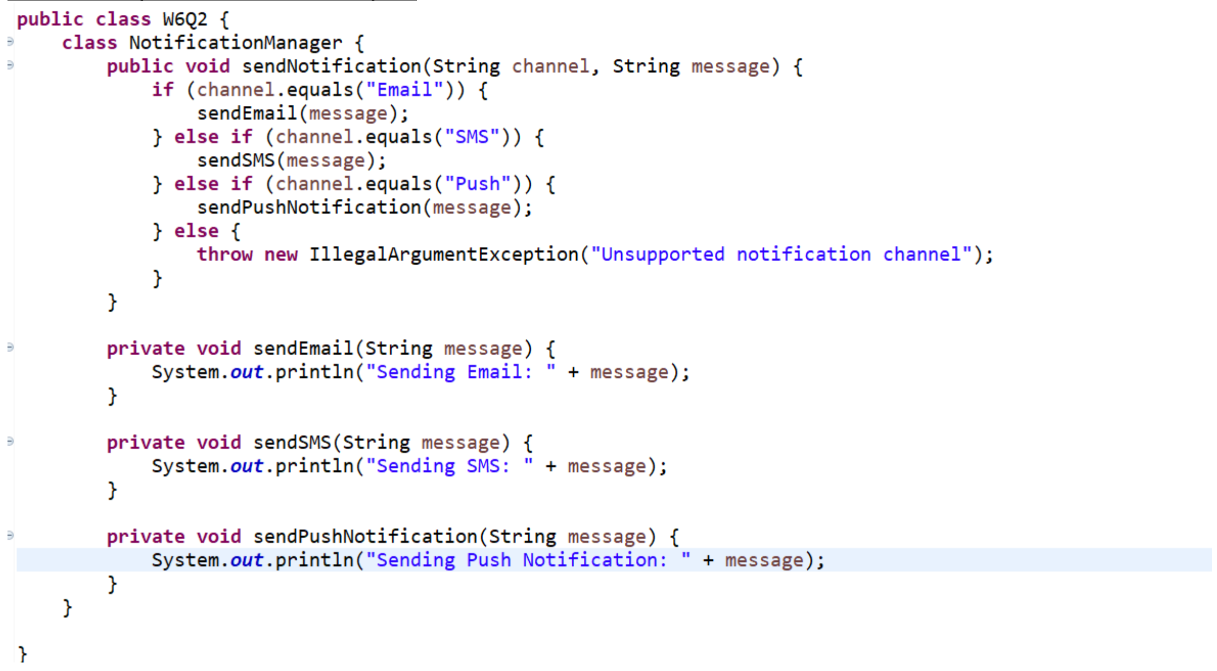
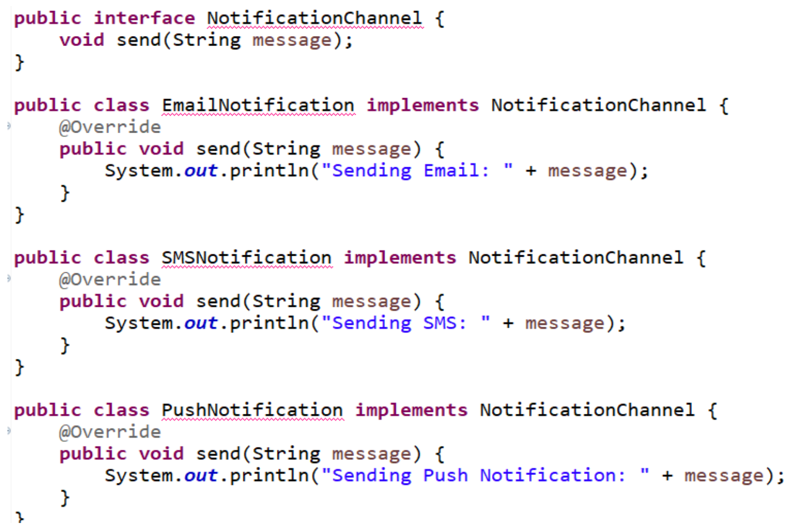
1. What is dependency inversion principle? Explain how it contributes to the more testable code.
   ANSWER: The dependency inversion principle emphasizes decoupling and abstraction. The principle consists of the core concepts; high-level modules should not depend on low level modules, and both should depend on abstractions. This inverted dependency relationship promotes flexibility, testability, and maintainability. Abstractions should not depend on details. Details should depend on abstractions. Dependency Inversion Principle advocates reducing the coupling between high level components and low-level components.

2. Describe the scenario where applying the Open-Closed Principle leads to improved code
   quality.
   ANSWER: *Scenario: A Notification System*
   You are building a notification system for an application that needs to support multiple channels, such as Email, SMS, and Push Notifications. The system should also allow for the addition of future notification types, like Slack, WhatsApp, or others, without modifying the existing code.
   Without Open-Closed Principle:



```java
public class W6Q2 {
    class NotificationManager {
        public void sendNotification(String channel, String message) {
            if (channel.equals("Email")) {
                sendEmail(message);
            } else if (channel.equals("SMS")) {
                sendSMS(message);
            } else if (channel.equals("Push")) {
                sendPushNotification(message);
            } else {
                throw new IllegalArgumentException("Unsupported notification channel");
            }
        }

        private void sendEmail(String message) {
            System.out.println("Sending Email: " + message);
        }

        private void sendSMS(String message) {
            System.out.println("Sending SMS: " + message);
        }

        private void sendPushNotification(String message) {
            System.out.println("Sending Push Notification: " + message);
        }
    }

}
```

   With Open-Closed Principle:



```java
public interface NotificationChannel {
    void send(String message);
}

public class EmailNotification implements NotificationChannel {
    @Override
    public void send(String message) {
        System.out.println("Sending Email: " + message);
    }
}

public class SMSNotification implements NotificationChannel {
    @Override
    public void send(String message) {
        System.out.println("Sending SMS: " + message);
    }
}

public class PushNotification implements NotificationChannel {
    @Override
    public void send(String message) {
        System.out.println("Sending Push Notification: " + message);
    }
}
```

3. Explain the scenario where the Interface Segregation Principle was beneficial.
   ANSWER: Scenario: A Document Management System
   You are building a document management system that handles different types of documents, such as PDFs, Word files, and Spreadsheets. Each document type supports specific operations:

- PDF files: Support reading and printing.
- Word files: Support reading, writing, and printing.
- Spreadsheets: Support reading, writing, printing, and exporting.

   Benefits of ISP in This Scenario

1. Decoupled Implementations: Each document type only implements methods that are relevant to it. This reduces unnecessary dependencies and ensures clarity in implementation.
2. Improved Maintainability: If a method is added to or removed from a specific interface (e.g., Exportable), only the relevant classes are affected. This minimizes the impact of changes.
3. Cleaner Code: Avoiding unsupported operations like UnsupportedOperationException makes the codebase cleaner and more understandable.
4. Easier Testing: Smaller interfaces simplify unit testing, as you can focus on testing only the methods implemented by a specific document type.
5. Adherence to Single Responsibility Principle (SRP): The split interfaces ensure that each interface has a single responsibility, focused on a specific type of operation.

4. Examine the following code.
   public class Report {
   public void generateReport() {
   // generate report logic
   }
   public void exportToPDF() {
   // export report to PDF logic
   }
   public void exportToExcel() {

```
// export report to Excel logic
}
}
```

Which principle is violated in the code among Single Responsibility, Open Closed, Interface
Segregation, and Dependency Inversion Principles? Explain in detail.

ANSWER: The code given above violates the Single Responsibility Principle. SRP states that a class should have only one reason to change, meaning it should have only one responsibility. This principle ensures that a class is focused on a single functionality or purpose.

5. Can you provide an example of how to design an online payment processing system while adhering to the SOLID principles? Please explain how each principle can be applied in the context of this system and illustrate with code or a conceptual overview. Let's assume we have payment types like CreditCardPayment, PayPalPayment, Esewa,
and Khalti. Each of these payments should have a method of transferring the amount.

Single Responsibility Principle (SRP)

Concept: Each module/class should have one and only one responsibility or reason to change.

Application in Payment Processing System:

Separate the system into distinct components for different responsibilities:

- **Payment Processing**: Handles the logic of transferring money.
- **Transaction Logging**: Manages logging and recording payment transactions.
- **Validation**: Ensures payment details like account numbers or amounts are valid.
- **Notification**: Sends confirmation messages or alerts to the user after payment.

This ensures each component is focused, easier to maintain, and can evolve independently.

Open-Closed Principle (OCP)

Concept: A system should be open for extension but closed for modification.

Application in Payment Processing System:

- Use an abstraction (like an interface or a base class) for payment methods.
- Each payment type (e.g., CreditCard, PayPal, Esewa, Khalti) should be an implementation of the abstraction.
- If a new payment type is introduced (e.g., Google Pay), the system should allow its addition without modifying existing payment logic.

This ensures the system is extensible without introducing changes that could break existing functionality.

Liskov Substitution Principle (LSP)

Concept: Subtypes should be substitutable for their base types.

Application in Payment Processing System:

- All payment methods should be interchangeable and usable wherever the system expects a payment type.
- For example, a processPayment function should work equally well for a CreditCard payment, a PayPal payment, or a Khalti payment.
- The system should not rely on the specifics of any payment method, only on the common functionality defined by the abstraction.

This ensures consistency and prevents unexpected behaviour when swapping payment methods.

Interface Segregation Principle (ISP)

Concept: Clients should not be forced to depend on methods they do not use.

Application in Payment Processing System:

Split functionalities into smaller, focused interfaces.

- For example:
    - A **Payment interface** might include only methods to transfer money.
    - A **Refundable interface** might handle refunds for payment methods that support them.
    - A **BalanceCheckable interface** might check available balance for certain payment types.

- A payment type should implement only the interfaces relevant to its capabilities:
    - PayPal might support balance checks and refunds.
    - Credit cards might only support payments and refunds but not balance checks.

This avoids unnecessary implementation requirements and keeps the system modular.

Dependency Inversion Principle (DIP)

Concept: High-level modules should depend on abstractions, not concrete implementations.

Application in Payment Processing System:

The payment processing logic should depend on an abstraction for payments, not specific payment types.

For example:

- The system should expect a "Payment" abstraction and process payments generically, regardless of whether it is CreditCard, PayPal, Esewa, or Khalti.

This abstraction ensures flexibility. New payment types can be integrated into the system without changing its core logic.

By designing the system in this way, the high-level logic (e.g., "process payment") remains decoupled from low-level details (e.g., how PayPal handles its transactions).

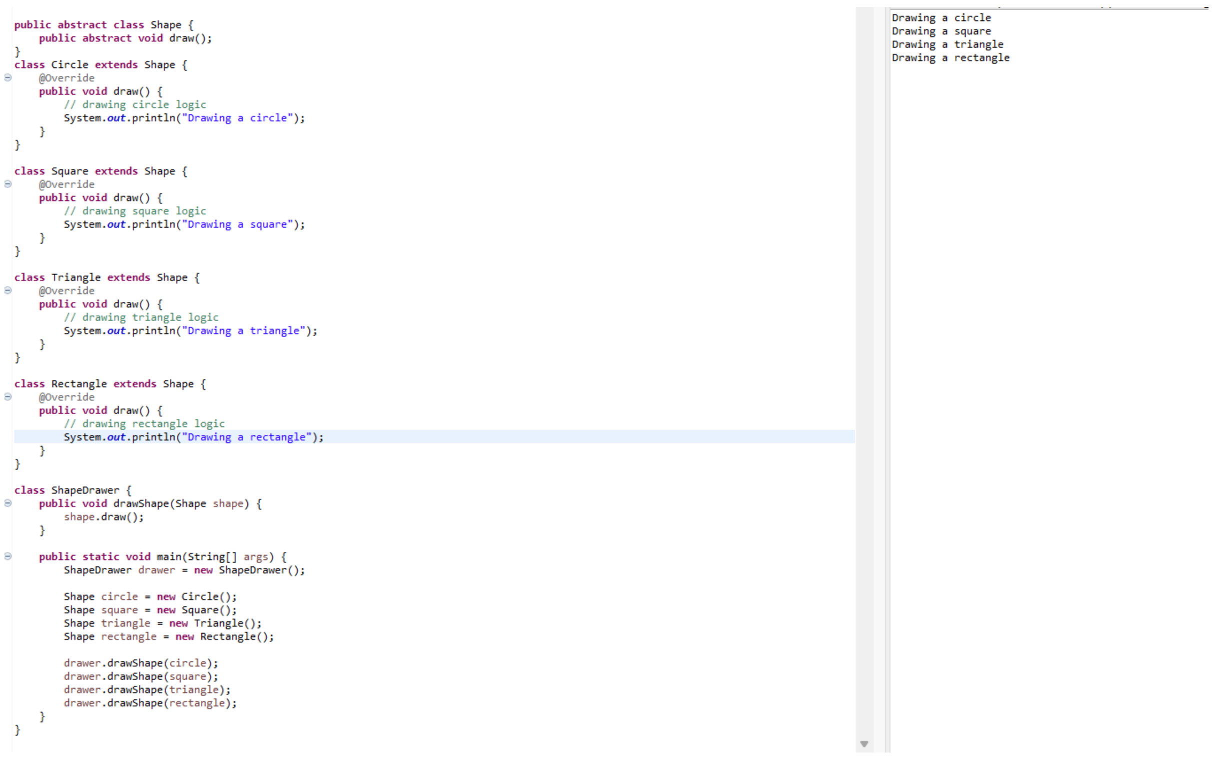
6. Examine the following code.

```
public class Shape {

public void drawCircle() {

// drawing circle logic

}

public void drawSquare() {

// drawing square logic

}s

}
```

You want to add more shapes (e.g., triangles, rectangles) without modifying the existing Shape class. Which design change would adhere to the Open-Closed Principle?

<u>Answer:</u>

This can be achieved by using polymorphism and inheritance. Here's the refined code:



```
public abstract class Shape {
    public abstract void draw();
}
class Circle extends Shape {
    @Override
    public void draw() {
        // drawing circle logic
        System.out.println("Drawing a circle");
    }
}

class Square extends Shape {
    @Override
    public void draw() {
        // drawing square logic
        System.out.println("Drawing a square");
    }
}

class Triangle extends Shape {
    @Override
    public void draw() {
        // drawing triangle logic
        System.out.println("Drawing a triangle");
    }
}

class Rectangle extends Shape {
    @Override
    public void draw() {
        // drawing rectangle logic
        System.out.println("Drawing a rectangle");
    }
}

class ShapeDrawer {
    public void drawShape(Shape shape) {
        shape.draw();
    }

    public static void main(String[] args) {
        ShapeDrawer drawer = new ShapeDrawer();

        Shape circle = new Circle();
        Shape square = new Square();
        Shape triangle = new Triangle();
        Shape rectangle = new Rectangle();

        drawer.drawShape(circle);
        drawer.drawShape(square);
        drawer.drawShape(triangle);
        drawer.drawShape(rectangle);
    }
}
```

```
Drawing a circle
Drawing a square
Drawing a triangle
Drawing a rectangle
```

7. Examine the following code.

```
public class Duck {
public void swim() {
System.out.println("Swimming");
}
public void quack() {
System.out.println("Quacking");
}
}
public class WoodenDuck extends Duck {
@Override
public void quack() {
throw new UnsupportedOperationException("Wooden ducks don't quack");
}
}
```

Which principle is violated in the above code among Open Closed, Single Responsibility, Liskov, and Interface Segregation Principle? Explain in detail. Also, update the above code base to solve the issue.

Answer:

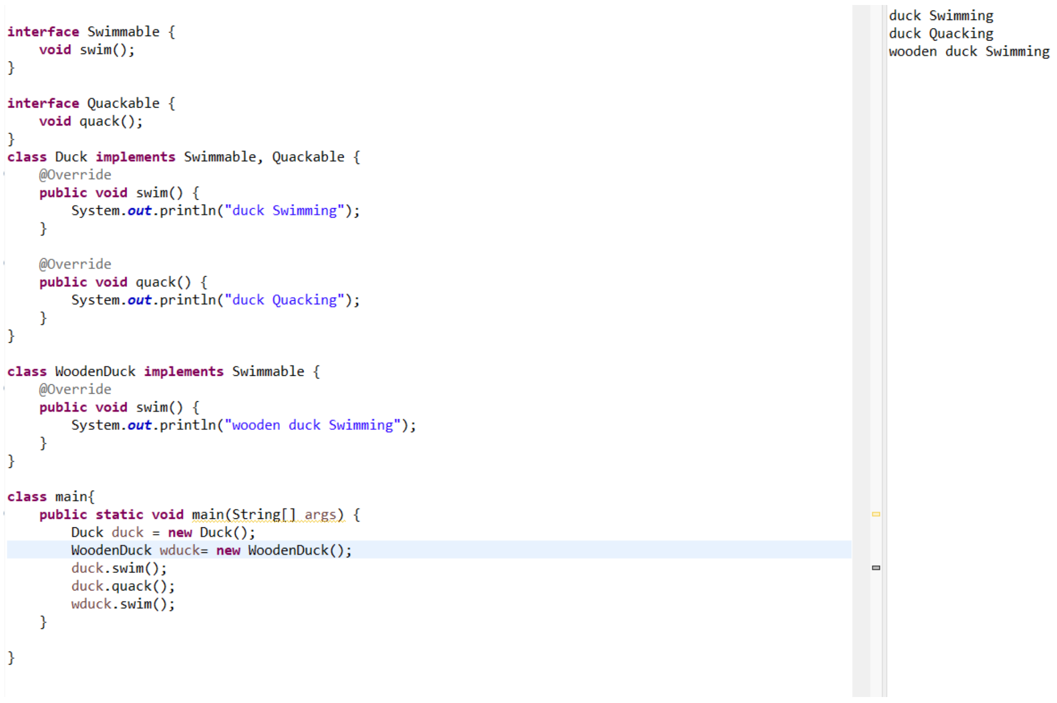
The code violates the Liskov Substitution Principle (LSP). The Liskov Substitution Principle states that objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program. In other words, subclasses should be able to stand in for their parent classes without causing unexpected behavior.

In the given code, the WoodenDuck class extends the Duck class but changes the quack method to throw an UnsupportedOperationException. This breaks the Liskov Substitution Principle (LSP) because a WoodenDuck cannot be used as a Duck without causing an error. According to LSP, any subclass of Duck should be able to do everything a Duck can do without problems.

Solution

To follow the Liskov Substitution Principle, we should change the code so that WoodenDuck does not inherit behaviors it cannot perform. One way to do this is by using interfaces to define the behaviors and then implementing these interfaces in the appropriate classes.

Here’s the updated code:



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

interface Quackable {
    void quack();
}
class Duck implements Swimmable, Quackable {
    @Override
    public void swim() {
        System.out.println("duck Swimming");
    }

    @Override
    public void quack() {
        System.out.println("duck Quacking");
    }
}

class WoodenDuck implements Swimmable {
    @Override
    public void swim() {
        System.out.println("wooden duck Swimming");
    }
}

class main{
    public static void main(String[] args) {
        Duck duck = new Duck();
        WoodenDuck wduck= new WoodenDuck();
        duck.swim();
        duck.quack();
        wduck.swim();
    }

}
```

```
duck Swimming
duck Quacking
wooden duck Swimming
```

WoodenDuck no longer inherits the quack method it cannot perform, ensuring it can be used interchangeably with other Swimmable objects without causing exceptions.

8. Examine the following code.

```
public interface PaymentMethod {
void processPayment();
}
public class PaypalPayment implements PaymentMethod {
@Override
public void processPayment() {
System.out.println("Processing PayPal payment");
}
}
public class OrderService {
private PaymentMethod paymentMethod;
public OrderService(PaymentMethod paymentMethod) {
this.paymentMethod = paymentMethod;
}
public void makePayment()
{
paymentMethod.processPayment();
}
}
```

Which solid principle is being followed above? Explain in detail.

<u>Answer</u>:

The code implements the Dependency Inversion Principle.

The Dependency Inversion Principle states that:

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Explanation

In the given code:

- Abstraction: The PaymentMethod interface serves as an abstraction for different payment methods.
- High-Level Module: The OrderService class is a high-level module that depends on the PaymentMethod interface, not on any specific implementation.
- Low-Level Module: The PaypalPayment class is a low-level module that implements the PaymentMethod interface.

By depending on the PaymentMethod interface, the OrderService class does not need to know the details of the specific payment method being used. This makes the system more flexible and easier to extend. For example, you can add new payment methods (like CreditCardPayment or BankTransferPayment) without modifying the OrderService class.