S.O.L.I.D.
JAVA

Жданов Дмитрий
Черницын Иван
5030102/30201

S.O.L.I.D. - принципы

1. Принцип единственности ответственности (*S*RP: Single Responsibility Principle)

2. Принцип открытости/закрытости (OCP: Open/Closed Principle)

3. Принцип подстановки Лисков (LSP: Liskov Substitution Principle)

4. Принцип разделения интерфейса (ISP: Interface Segregation Principle)

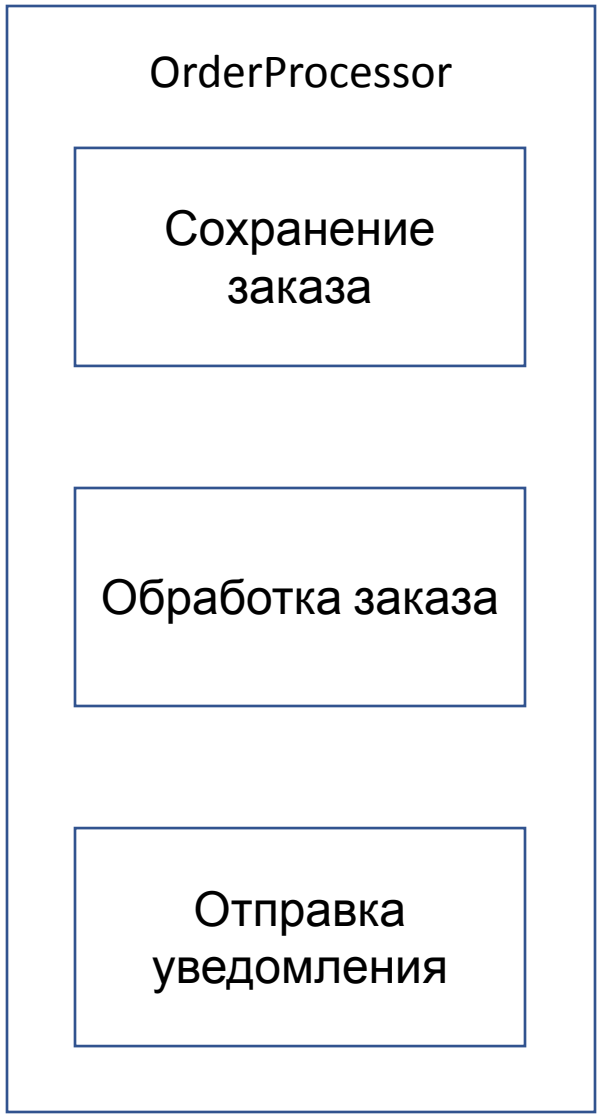
5. Принцип инверсии зависимостей (DIP: Dependency Inversion Principle)

1. Принцип единственности ответственности (SRP: Single Responsibility Principle)

Формулировка:

Не должно быть больше одной причины для изменения класса

Каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в пределах класса.




```
// Неправильное решение
public class OrderProcessor {

    public void process(Order order){
        if (order.isValid() && save(order)) {
            sendConfirmationEmail(order);
        }
    }

    private boolean save(Order order) {
        //подключаемся к базе данных
        // сохраняем заказ в базу данных
        //отключаемся от базы данных
        return true;
    }

    private void sendConfirmationEmail(Order order) {
        String name = order.getCustomerName();
        String email = order.getCustomerEmail();
        // Шлем письмо клиенту
    }
}
```

Сохранение заказа
OrderRepository

Обработка заказа
OrderProcessor

Отправка уведомления
ConfirmationEmailSender

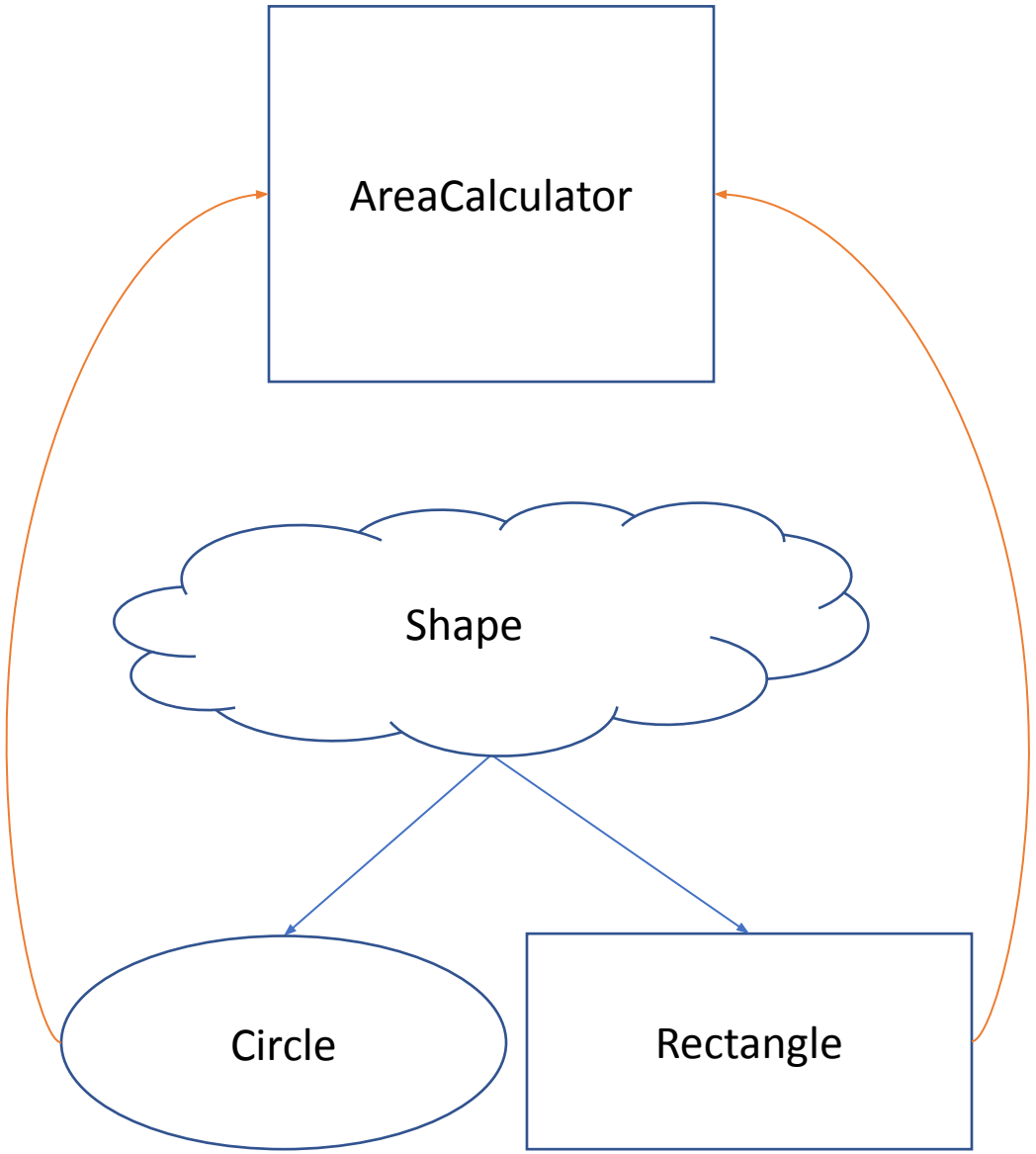
```
// Правильное решение
public class OrderRepository {
    public boolean save(Order order) {
        //подключаемся к базе данных
        // сохраняем заказ в базу данных
        //отключаемся от базы данных
        return true;
    }
}
public class ConfirmationEmailSender {
    public void sendConfirmationEmail(Order order) {
        String name = order.getCustomerName();
        String email = order.getCustomerEmail();
        // Шлем письмо клиенту
    }
}
public class OrderProcessor {
    public void process(Order order){
        OrderRepository repository = new OrderRepository();
        ConfirmationEmailSender mailSender = new ConfirmationEmailSender();
        if (order.isValid() && repository.save(order)) {
            mailSender.sendConfirmationEmail(order);
        }
    }
}
```

2. Принцип открытости/закрытости (OCP: Open/Closed Principle)

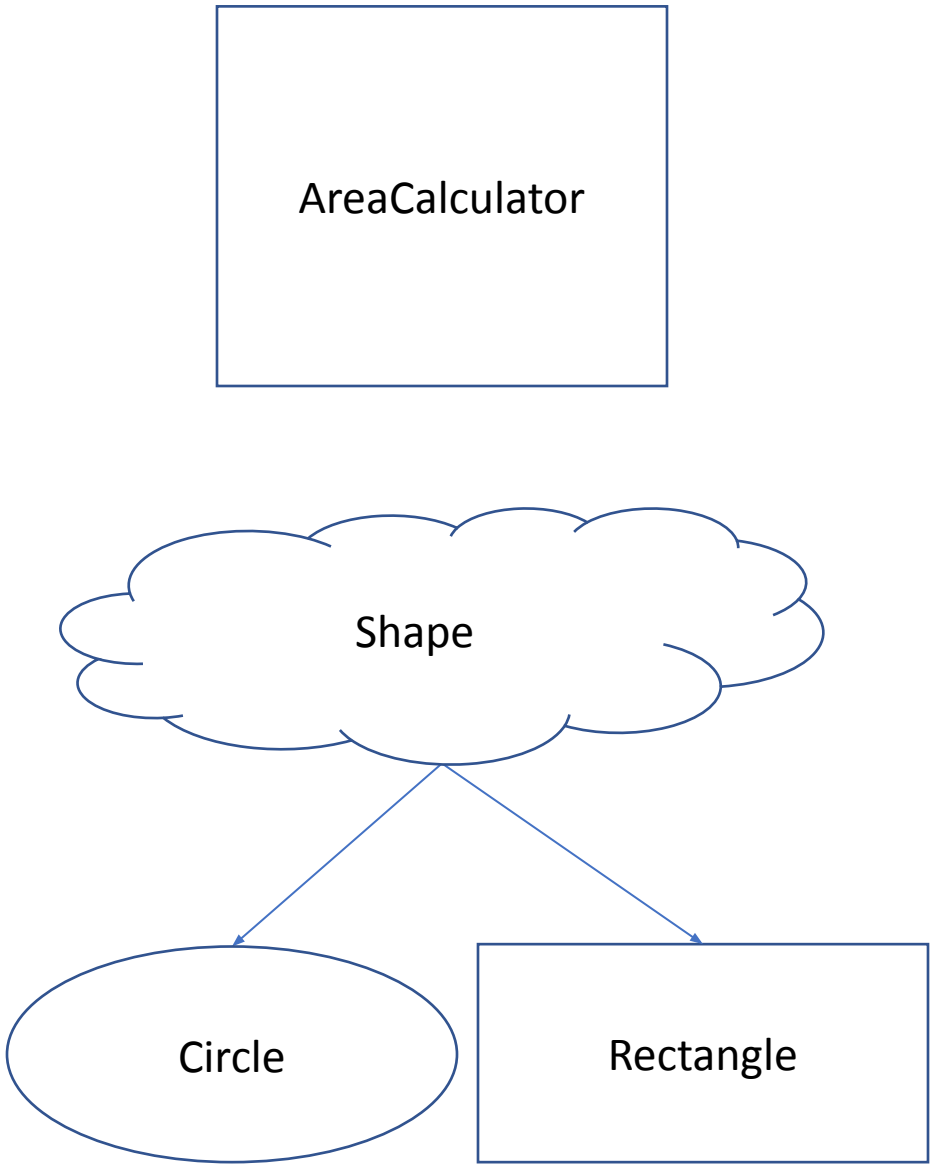
Формулировка:

программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для модификации

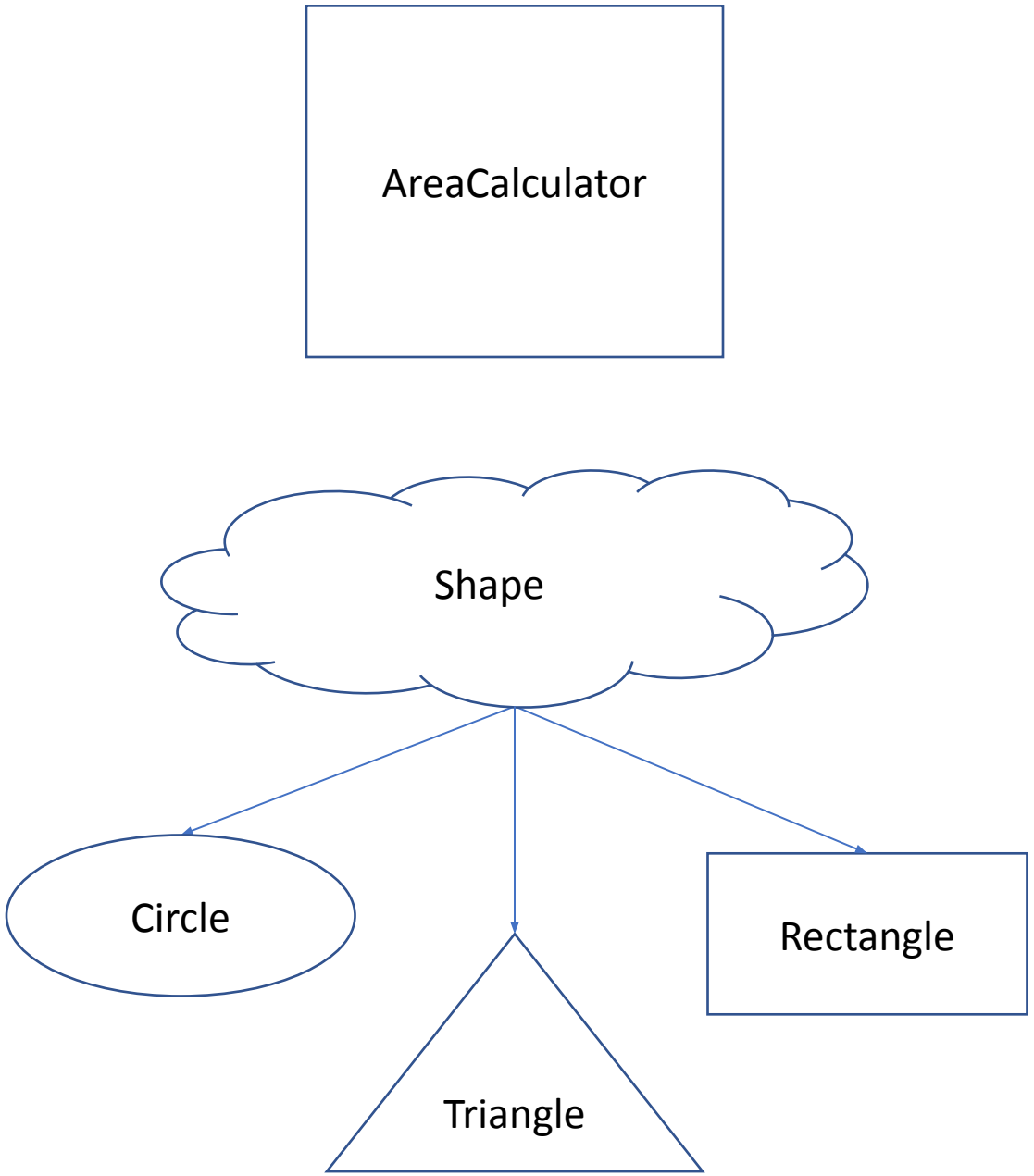
Это означает, что при добавлении новой функциональности не следует изменять существующий код, а лучше расширять его. Новая функциональность внедряется либо через наследование, либо через использование абстрактных интерфейсов.




```
class AreaCalculator {
    public double calculateArea(Shape shape) {
        if (shape instanceof Circle) {
            Circle c = (Circle) shape;
            return Math.PI * c.radius * c.radius;
        } else if (shape instanceof Rectangle) {
            Rectangle r = (Rectangle) shape;
            return r.width * r.height;
        }
        return 0;
    }
}

class Shape {}

class Circle extends Shape {
    double radius;
    Circle(double radius) {
        this.radius = radius;
    }
}

class Rectangle extends Shape {
    double width, height;
    Rectangle(double width, double height) {
        this.width = width;
        this.height = height;
    }
}
```




```
abstract class Shape {
   abstract double area();
}

class Circle extends Shape {
   double radius;
   Circle(double radius) { this.radius = radius; }
   @Override
   double area() { return Math.PI * radius * radius; }
}

class Rectangle extends Shape {
   double width, height;
   Rectangle(double width, double height) {
      this.width = width;
      this.height = height;
   }
   @Override
   double area() { return width * height; }
}

class AreaCalculator {
   public double calculateArea(Shape shape) {
      return shape.area();
   }
}
```




```
abstract class Shape {
   abstract double area();
}

class Circle extends Shape {
   double radius;
   Circle(double radius) { this.radius = radius; }
   @Override
   double area() { return Math.PI * radius * radius; }
}

class Rectangle extends Shape {
   double width, height;
   Rectangle(double width, double height) {
      this.width = width;
      this.height = height;
   }
   @Override
   double area() { return width * height; }
}

class Triangle extends Shape {
   double base, height;
   Triangle(double base, double height) {
      this.base = base;
      this.height = height;
   }
   @Override
   double area() { return 0.5 * base * height; }
}

class AreaCalculator {
   public double calculateArea(Shape shape) {
      return shape.area();
   }
}
```

3. Принцип подстановки Лисков (LSP: Liskov Substitution Principle)

Формулировка:

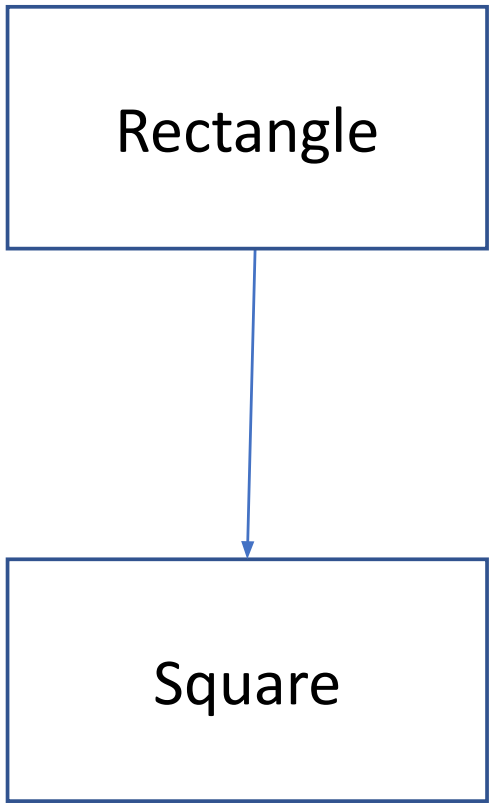
Производные классы должны быть заменяемы их базовыми

классами.

Это означает:

- Если `S` — подтип `T`, то объекты типа `T` могут быть заменены объектами типа `S` без нарушения логики программы.
- Нарушение LSP приводит к хрупкому коду и затрудняет повторное использование.

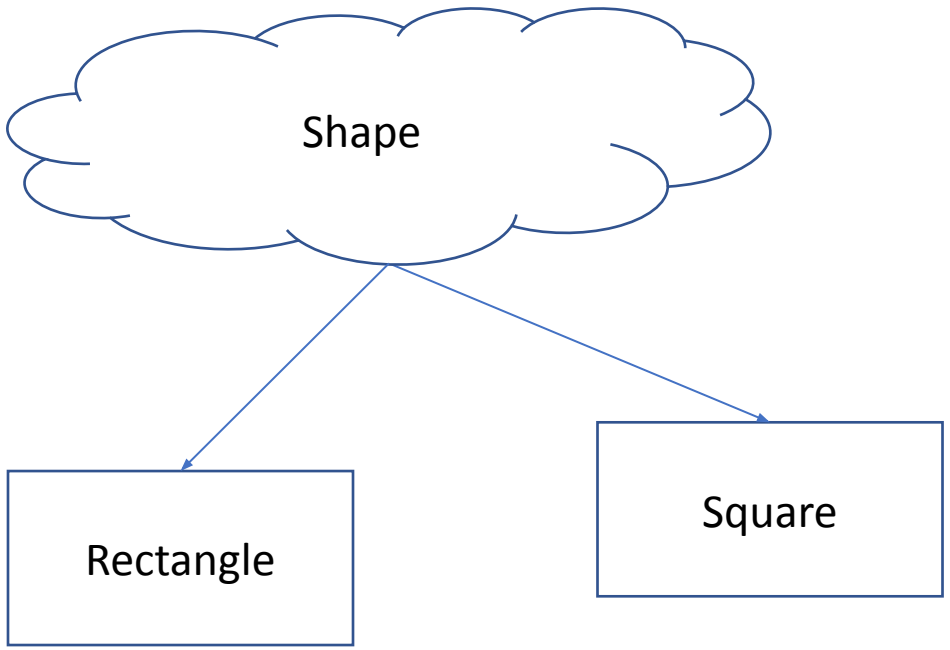
3. Принцип подстановки Лисков (LSP: Liskov Substitution Principle)




```
// Базовый класс
public class Rectangle {
  protected int width;
  protected int height;

  public void setWidth(int w) { this.width = w; }
  public void setHeight(int h) { this.height = h; }

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

// Подкласс, который выглядит как прямоугольник, но ведёт себя иначе
public class Square extends Rectangle {
  @Override
  public void setWidth(int w) {
    this.width = w;
    this.height = w; // поддерживаем квадрат
  }

  @Override
  public void setHeight(int h) {
    this.height = h;
    this.width = h; // поддерживаем квадрат
  }
}
```

```
public class ShapeDemo {
    // Клиент ожидает, что поведение прямоугольника
— независимое изменение сторон
    public static void resizeRectangle(Rectangle rect) {
        rect.setWidth(5);
        rect.setHeight(10);

        // Ожидаем: ширина = 5, высота = 10, площадь =
50
        System.out.println("Expected area = 50, actual = " +
rect.getArea());
    }

    public static void main(String[] args) {
        Rectangle rectangle = new Rectangle();
        Square square = new Square();

        System.out.println("Using Rectangle:");
        resizeRectangle(rectangle);

        System.out.println("Using Square:");
        resizeRectangle(square);
    }
}
```

3. Принцип подстановки Лисков (LSP: Liskov Substitution Principle)




```
public interface Shape {
  int getArea();
}

public class Rectangle implements Shape {
  private final int width;
  private final int height;

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

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

public class Square implements Shape {
  private final int side;

  public Square(int side) { this.side = side; }

  @Override
  public int getArea() { return side * side; }
}

// Клиент
public static void printArea(Shape s) {
  System.out.println(s.getArea());
}
```

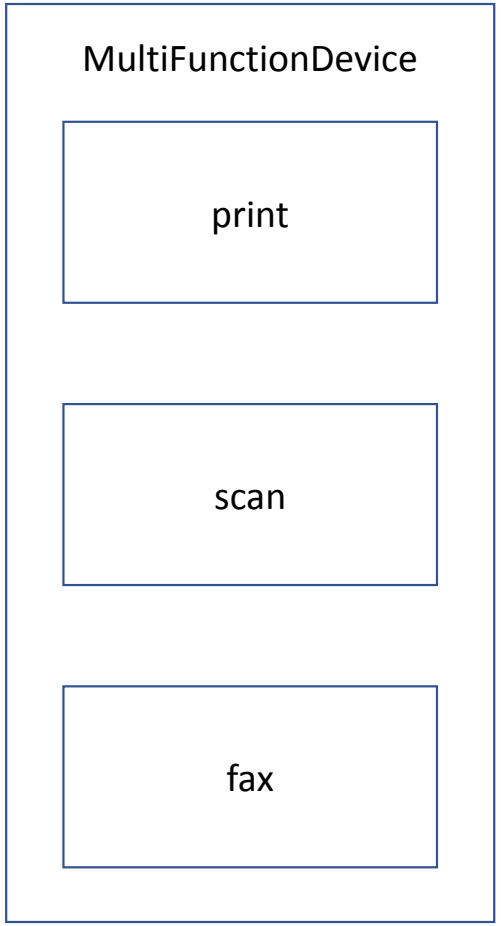
4. Принцип разделения интерфейса (ISP: Interface Segregation Principle)

Формулировка:

Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента.

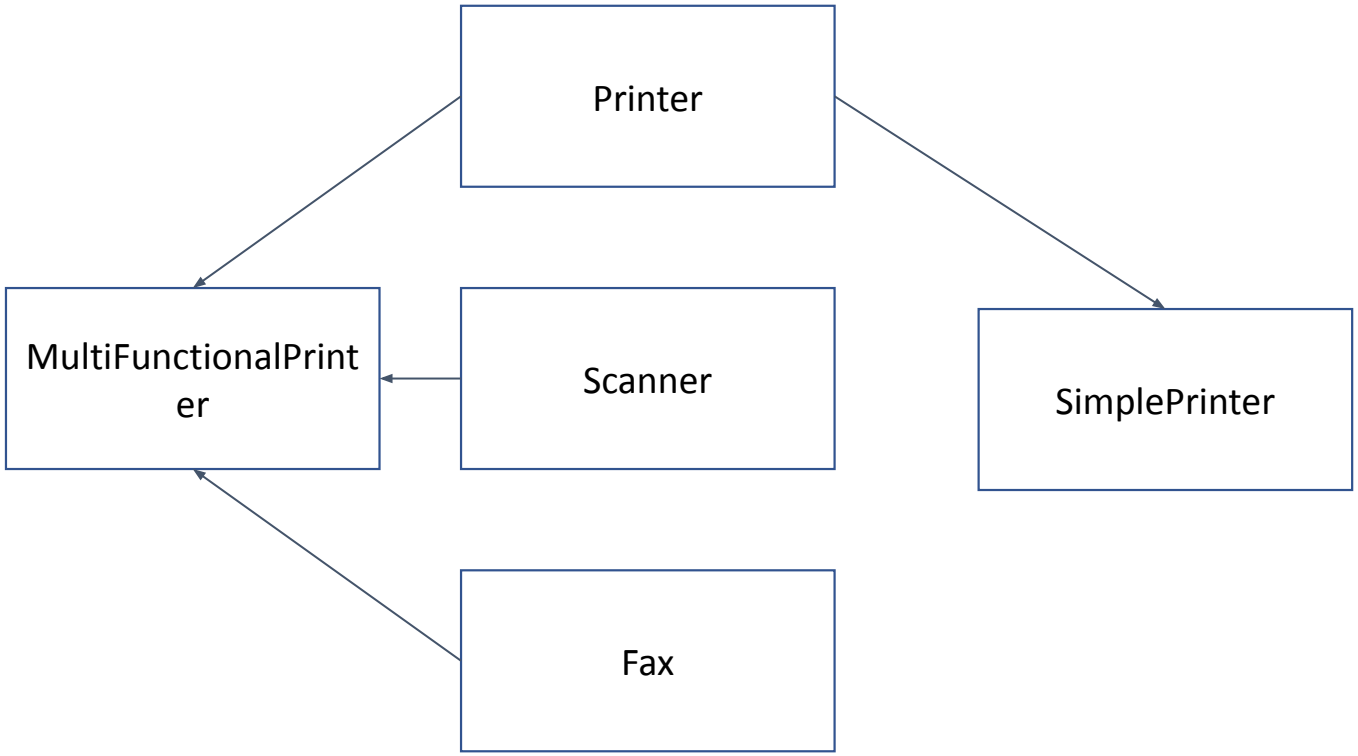
Интерфейсы должны быть узкими и ориентированными на конкретного клиента

4. Принцип разделения интерфейса (ISP: Interface Segregation Principle)




```
public interface MultiFunctionDevice {
  void print(Document d);
  void scan(Document d);
  void fax(Document d);
}

// Класс-принтер, который не умеет отправлять факсы, но вынужден реализовать метод fax()
public class SimplePrinter implements MultiFunctionDevice {
  @Override
  public void print(Document d) { /* печать */ }

  @Override
  public void scan(Document d) { throw new UnsupportedOperationException(); }

  @Override
  public void fax(Document d) { throw new UnsupportedOperationException(); }
}
```

4. Принцип разделения интерфейса (ISP: Interface Segregation Principle)




```
public interface Printer {
  void print(Document d);
}

public interface Scanner {
  void scan(Document d);
}

public interface Fax {
  void fax(Document d);
}

// Композиция для многофункционального устройства
public class MultiFunctionPrinter implements Printer, Scanner, Fax {
  @Override
  public void print(Document d) { /* печать */ }
  @Override
  public void scan(Document d) { /* сканирование */ }
  @Override
  public void fax(Document d) { /* факс */ }
}

// Простой принтер реализует только то, что нужно
public class SimplePrinter implements Printer {
  @Override
  public void print(Document d) { /* печать */ }
}
```

5. Принцип инверсии зависимостей (DIP: Dependency Inversion Principle)

Формулировка:

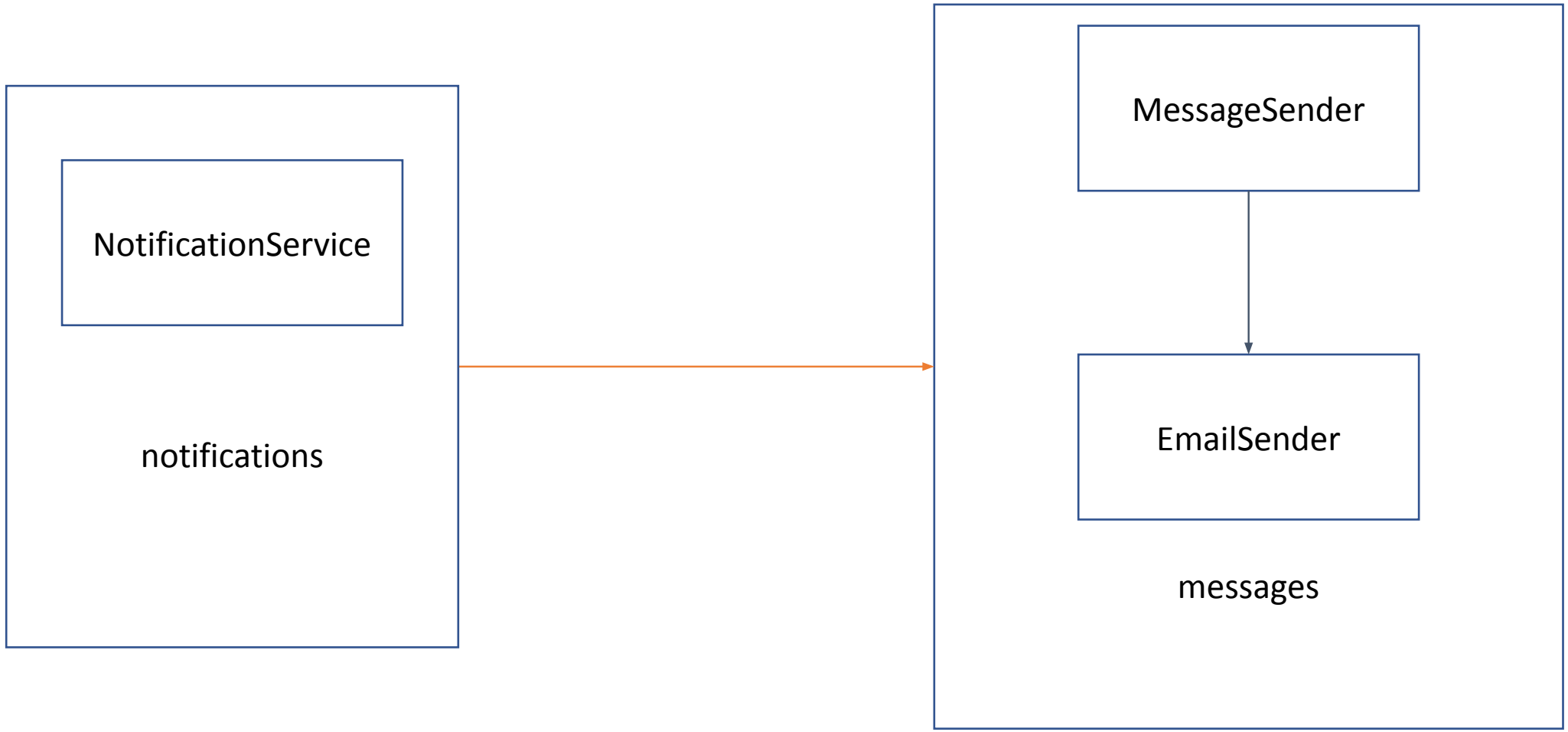
Высокоуровневые модули не должны зависеть от низкоуровневых модулей.

5. Принцип инверсии зависимостей (DIP: Dependency Inversion Principle)

```
// MessageSender.java

package com.example.messages;

public interface MessageSender {
  void send(String to, String body);
}

// EmailSender.java

package com.example.messages;

public class EmailSender implements MessageSender {
  @Override
  public void send(String to, String body) {
    // отправка email
  }
}
```

```
// NotificationService

package com.example.notifications;

import com.example.messages.MessageSender;

public class NotificationService {
  private final MessageSender sender;

  public NotificationService(MessageSender sender) {
    this.sender = sender;
  }

  public void notifyUser(String user, String message) {
    sender.send(user, message);
  }
}
```

5. Принцип инверсии зависимостей (DIP: Dependency Inversion Principle)

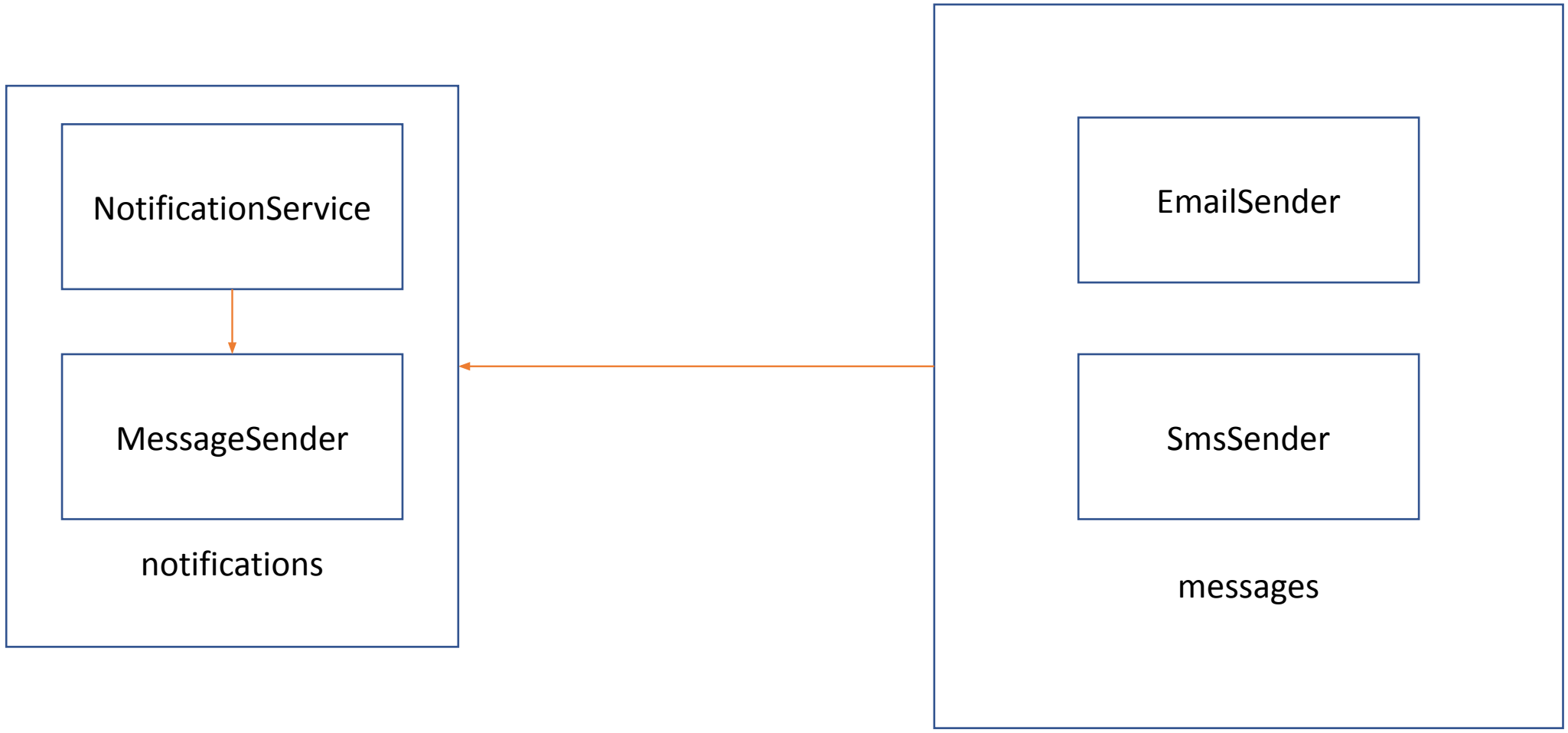


5. Принцип инверсии зависимостей (DIP: Dependency Inversion Principle)

```
// EmailSender.java

package com.example.messages;

import com.example.notifications.MessageSender;

public class EmailSender implements MessageSender {
  @Override
  public void send(String to, String body) {
    // отправка email
  }
}

// SmsSender.java

package com.example.messages;

import com.example.notifications.MessageSender;

public class SmsSender implements MessageSender {
  @Override
  public void send(String to, String body) {
    // отправка sms
  }
}
```

```
// MessageSender.java

package com.example.notifications;

public interface MessageSender {
  void send(String to, String body);
}

// NotificationService

package com.example.notifications;

public class NotificationService {
  private final MessageSender sender;

  public NotificationService(MessageSender sender) {
    this.sender = sender;
  }

  public void notifyUser(String user, String message) {
    sender.send(user, message);
  }
}
```

5. Принцип инверсии зависимостей (DIP: Dependency Inversion Principle)



Заключение

Принципы **SOLID** — это фундамент качественного проектирования:

- **SRP** упрощает поддержку кода.
- **OCP** позволяет расширять функциональность без изменения существующего кода.
- **LSP** предотвращает ошибки наследования.
- **ISP** избавляет от лишних методов.
- **DIP** снижает связанность и упрощает тестирование.

Следуя этим принципам, мы получаем более гибкие и масштабируемые приложения. В Java эти идеи особенно актуальны, так как язык активно используется в больших корпоративных системах, где поддерживаемость и расширяемость кода критически важны.

Список используемой литературы

1) Маттиас Нобак. Принципы разработки програмных пакетов. Проектирование повторно используемых компонентов.

2) "Чистый код" Роберт Мартин

3) javarush.com/groups/posts/68569-principih-solid-prostihmi-slovami

4) habr.com/ru/companies/tbank/articles/472186/

5) makedev.org/principles/solid/

Благодарим за внимание!