10. Разработка ПО: SOLID, Паттерны, Практики

1. SOLID

Принципы объектно-ориентированного проектирования для создания гибкого и поддерживаемого кода.

① Single Responsibility (Принцип единственной ответственности)

- Класс должен иметь **только одну причину для изменения** (решать одну задачу).
- ❌ Плохо:

```
class User {
    void saveToDatabase() { ... }  // Работа с БД
    void sendEmail() { ... }       // Отправка email
}
```

- ✅ Хорошо:

```
class User { ... }
class UserRepository { void save(User user) { ... } }
class EmailService { void sendEmail(User user) { ... } }
```

---

② Open-Closed (Принцип открытости/закрытости)

- Классы должны быть **открыты для расширения, но закрыты для изменения**.
- Используйте **абстракции (интерфейсы)** вместо прямого изменения кода.
- ❌ Плохо:

```
class PaymentProcessor {
    void process(String paymentType) {
        if (paymentType.equals("credit")) { ... }
        else if (paymentType.equals("paypal")) { ... } // При добавлении
нового типа нужно менять класс
    }
}
```

- ✅ Хорошо:

```
interface PaymentMethod { void process(); }
class CreditCard implements PaymentMethod { ... }
class PayPal implements PaymentMethod { ... }
// Новый платежный метод — новый класс, а не изменение PaymentProcessor.
```

---

③ Liskov Substitution (Принцип подстановки Барбары Лисков)

- Подтипы должны **быть заменяемыми** на свои базовые типы.
- Наследник не должен ужесточать условия (например, бросать новые исключения).

- ❌ Плохо:

```
class Rectangle {
    void setWidth(int w) { ... }
    void setHeight(int h) { ... }
}
class Square extends Rectangle {
    void setWidth(int w) { width = height = w; } // Нарушает логику Rectangle!
}
```

- ✅ Решение: Не наследовать `Square` от `Rectangle`, если их поведение отличается.

---

④ Interface Segregation (Принцип разделения интерфейсов)

- Клиенты не должны зависеть от методов, которые они не используют.
- Лучше много маленьких интерфейсов, чем один "жирный".
- ❌ Плохо:

```
interface Worker {
    void work();
    void eat();  // Офисный работник ест, а робот — нет, но должен реализовать
метод!
}
```

- ✅ Хорошо:

```
interface Workable { void work(); }
interface Eatable { void eat(); }
class Human implements Workable, Eatable { ... }
class Robot implements Workable { ... }
```

---

⑤ Dependency Inversion (Принцип инверсии зависимостей)

- Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба должны зависеть от абстракций.
- Внедрение зависимостей (DI) через конструктор/сеттер.
- ❌ Плохо:

```
class UserService {
    private MySQLDatabase db = new MySQLDatabase();  // Жесткая зависимость
}
```

- ✅ Хорошо:

```
interface Database { void save(); }
class UserService {
    private Database db;
    UserService(Database db) { this.db = db; }  // DI
}
// Теперь можно подменить MySQL на PostgreSQL, не меняя UserService.
```

2. Паттерны проектирования

Creational (Порождающие)

- **Singleton** – гарантирует один экземпляр класса:

```
class Logger {
    private static Logger instance;
    private Logger() {}
    public static Logger getInstance() {
        if (instance == null) instance = new Logger();
        return instance;
    }
}
```

- **Factory** – создает объекты без указания конкретного класса:

```
interface Car { ... }
class Sedan implements Car { ... }
class SUV implements Car { ... }

class CarFactory {
    Car createCar(String type) {
        return switch (type) {
            case "sedan" -> new Sedan();
            case "suv" -> new SUV();
            default -> throw new IllegalArgumentException();
        };
    }
}
```

- **Builder** – пошаговое создание сложных объектов:

```
User user = new User.Builder().name("Alice").age(25).build();
```

Structural (Структурные)

- **Adapter** – адаптирует несовместимые интерфейсы:

```
class OldSystem { void oldRequest() { ... } }
interface NewSystem { void newRequest(); }

class Adapter implements NewSystem {
    private OldSystem old;
    Adapter(OldSystem old) { this.old = old; }
    void newRequest() { old.oldRequest(); }
}
```

- **Proxy** – контролирует доступ к объекту (логирование, кэширование):

```
interface Image { void display(); }
class RealImage implements Image { ... }
```

```
class ProxyImage implements Image {
    private RealImage realImage;
    void display() {
        if (realImage == null) realImage = new RealImage();
        realImage.display();
    }
}
```

- **Decorator** – динамически добавляет поведение:

```
interface Coffee { double getCost(); }
class SimpleCoffee implements Coffee { ... }

class MilkDecorator implements Coffee {
    private Coffee coffee;
    MilkDecorator(Coffee coffee) { this.coffee = coffee; }
    double getCost() { return coffee.getCost() + 0.5; }
}
```

Behavioral (Поведенческие)

- **Observer** – уведомляет подписчиков об изменениях:

```
interface Observer { void update(String news); }
class NewsAgency {
    private List<Observer> observers = new ArrayList<>();
    void addObserver(Observer o) { observers.add(o); }
    void notifyObservers(String news) {
        for (Observer o : observers) o.update(news);
    }
}
```

- **Strategy** – инкапсулирует алгоритмы:

```
interface PaymentStrategy { void pay(int amount); }
class CreditCardStrategy implements PaymentStrategy { ... }
class PayPalStrategy implements PaymentStrategy { ... }

class ShoppingCart {
    private PaymentStrategy strategy;
    void setStrategy(PaymentStrategy s) { this.strategy = s; }
    void checkout(int amount) { strategy.pay(amount); }
}
```

- **Command** – инкапсулирует запросы в объекты:

```
interface Command { void execute(); }
class LightOnCommand implements Command {
    private Light light;
    void execute() { light.turnOn(); }
}
class RemoteControl {
    void submit(Command cmd) { cmd.execute(); }
}
```

3. Практики разработки

① DRY (Don’t Repeat Yourself)

- Избегайте дублирования кода. Выносите общую логику в методы/классы.
- ❌ Плохо:

```
void processOrderA() { validate(); saveToDB(); log(); }
void processOrderB() { validate(); saveToDB(); log(); }  // Та же логика
```

- ✅ Хорошо:

```
void processOrder(Order order) { validate(); saveToDB(); log(); }
```

② KISS (Keep It Simple, Stupid)

- Делайте код максимально простым и понятным.
- ❌ Плохо:

```
String result = list.stream().filter(x -> x % 2 == 0).map(x -> x *
2).collect(Collectors.joining(", "));
```

- ✅ Хорошо (если не нужна сложная логика):

```
for (int x : list) if (x % 2 == 0) System.out.print(x * 2 + ", ");
```

③ YAGNI (You Aren’t Gonna Need It)

- Не добавляйте функциональность "на будущее".
- ❌ Плохо:

```
class User {
    private String name;
    private String futureField1; // А вдруг пригодится?
    private String futureField2;
}
```

- ✅ Хорошо:

```
class User { private String name; }  // Добавим поля, когда они реально
понадобятся.
```

Итог

- **SOLID** – основа ООП, делает код гибким и расширяемым.
- **Паттерны** – проверенные решения для типовых задач.
- **DRY/KISS/YAGNI** – практики для поддержки чистоты кода.

Следование этим принципам упрощает поддержку и масштабирование проектов! 🚀