Принципы чистой разработки

Грачев Д.Г.

Предисловие

Мир не стоит на месте и, как любое из направлений человеческой деятельности, программирование имеет циклическое эмпирическое развитие. То что было актуально 10 лет назад, сейчас может вызывать лишь улыбку, но не стоит недооценивать труды прошлых поколений - наши возможности держаться на результатах их работы. Но это не значит, что стоит жить прошлыми парадигмами, ибо идеал не достижим, а значит стоит экспериментировать вновь и вновь, подвергая догмы сомнениям. Ведь только в соревнованиях определяются победители.

Определения

Принцип — это основополагающая идея, правило или убеждение, служащее базой для теории, деятельности или поведения. Он определяет способы взаимодействия с реальностью и неизменную позицию в вопросах. Принципы помогают принимать решения, отличаясь устойчивостью и служа основой устройства механизмов.

Принципы чистой разработки (Clean Code) — это свод правил для создания понятного, поддерживаемого и масштабируемого программного обеспечения. Ключевые принципы включают: **читаемость** (осмысленные имена), **простоту** (KISS), **отсутствие дублирования** (DRY), **разделение ответственности** (SRP) и принцип минимализма (YAGNI), что помогает **снизить стоимость** сопровождения кода.

Пиши код так, будто его будет читать маньяк, который знает, где ты живешь!

Примеры плохого кода

```
// работает, но читать больно
int a = 5; int b = 10; int c = 0; for (int i = 0; i < b; i++) { c += a; } System.out.println(c);
```

Исправление

```java
int multiplier = 5;
int repeatCount = 10;
int result = 0;

for (int i = 0; i < repeatCount; i++) {
    result += multiplier;
}

System.out.println("Результат умножения: " + result);
```

SOLID

SOLID — это набор из пяти основных принципов объектно-ориентированного проектирования (ООП), сформулированных Робертом Мартином для создания гибкого, поддерживаемого и масштабируемого кода. Они помогают уменьшить сложность, улучшить читаемость и облегчить тестирование программного обеспечения.

- S (Single Responsibility Principle). Принцип единственной ответственности.
- O (Open/Closed Principle). Принцип открытости/закрытости.
- L (Liskov Substitution Principle). Принцип подстановки Барбары Лисков.
- I (Interface Segregation Principle). Принцип разделения интерфейсов.
- D (Dependency Inversion Principle). Принцип инверсии зависимостей.

SOLID (S - Single Responsibility Principle)
Принцип единственной ответственности

```java
public class UserService {

    public void registerUser(User user) {

        // Валидация

        if (user.getEmail() == null) throw new RuntimeException("Email нужен");

        // Сохранение в БД

        jdbcTemplate.execute("INSERT INTO users ...");

        // Отправка email

        sendEmail(user.getEmail(), "Добро пожаловать!");

          // Логирование

        log.info("Пользователь зарегистрирован");

    }

    private void sendEmail(String to, String text) {

        // SMTP код...

    }

}
```

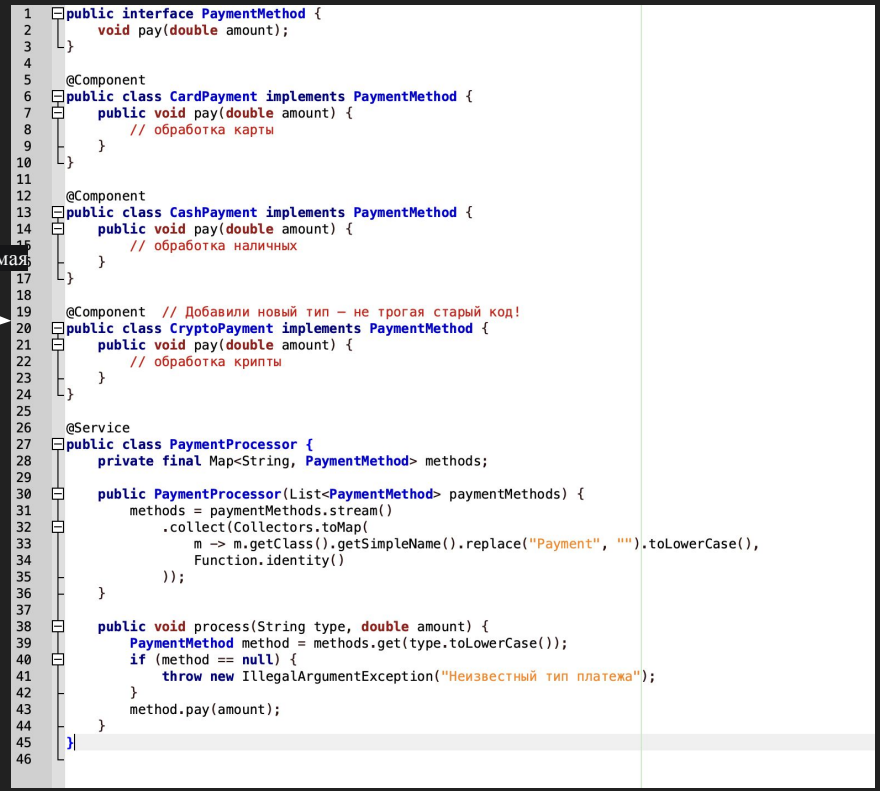
Один объект - одна зона ответственности

```java
@Service
public class UserRegistrationService {
    private final UserValidator validator;
    private final UserRepository repository;
    private final EmailService emailService;
    private final AuditService auditService;

    public User register(User user) {
        validator.validate(user);
        User saved = repository.save(user);
        emailService.sendWelcomeEmail(saved.getEmail());
        auditService.logRegistration(saved);
        return saved;
    }
}

@Component
public class UserValidator {
    public void validate(User user) {
        if (user.getEmail() == null || !user.getEmail().contains("@")) {
            throw new ValidationException("Некорректный email");
        }
        if (user.getPassword().length() < 8) {
            throw new ValidationException("Пароль слишком короткий");
        }
    }
}
```

SOLID (O — Open/Closed)
Открыт для расширения, закрыт для изменения

```java
public class PaymentProcessor {


    public void process(String paymentType, double amount) {

        if (paymentType.equals("CARD")) {

            // обработать карту

        } else if (paymentType.equals("CASH")) {

            // обработать наличные

        } else if (paymentType.equals("CRYPTO")) {

            // добавили новый тип — пришлось лезть в код!

        }

    }

}
```

Пиши код так, чтобы добавлять новое, не ломая старое



```java
public interface PaymentMethod {
    void pay(double amount);
}

@Component
public class CardPayment implements PaymentMethod {
    public void pay(double amount) {
        // обработка карты
    }
}

@Component
public class CashPayment implements PaymentMethod {
    public void pay(double amount) {
        // обработка наличных
    }
}

@Component  // Добавили новый тип — не трогая старый код!
public class CryptoPayment implements PaymentMethod {
    public void pay(double amount) {
        // обработка крипты
    }
}

@Service
public class PaymentProcessor {
    private final Map<String, PaymentMethod> methods;

    public PaymentProcessor(List<PaymentMethod> paymentMethods) {
        methods = paymentMethods.stream()
            .collect(Collectors.toMap(
                m -> m.getClass().getSimpleName().replace("Payment", "").toLowerCase(),
                Function.identity()
            ));
    }

    public void process(String type, double amount) {
        PaymentMethod method = methods.get(type.toLowerCase());
        if (method == null) {
            throw new IllegalArgumentException("Неизвестный тип платежа");
        }
        method.pay(amount);
    }
}
```

SOLID (L — Liskov Substitution)
Принцип подстановки Лисков

```java
class Bird {

    public void fly() {

        System.out.println("Летаю");

    }

}


class Penguin extends Bird {  // Пингвин не летает!

    @Override

    public void fly() {

        throw new UnsupportedOperationException("Пингвины не летают");

    }

}
```

Наследник должен уметь всё, что умеет родитель

```java
interface Bird {
    void move();
}

class FlyingBird implements Bird {
    public void move() {
        fly();
    }

    private void fly() {
        System.out.println("Летаю");
    }
}

class Penguin implements Bird {
    public void move() {
        swim();
    }

    private void swim() {
        System.out.println("Плаваю");
    }
}
```

SOLID (I — Interface Segregation)
Разделение интерфейсов

```
interface Worker {

    void work();

    void eat();

    void sleep();

    void code();

    void design();

    void test();

}

class Developer implements Worker {
    // Методы, которые не нужны разработчику, но
приходится реализовывать
    public void design() { /* не умею */ }
    public void test() { /* это не моя работа */ }
}
```

Лучше несколько маленьких интерфейсов, чем один большой

```
interface Workable {
    void work();
}

interface Eatable {
    void eat();
}

interface Coder {
    void code();
}

interface Tester {
    void test();
}

class Developer implements Workable, Eatable, Coder {
    public void work() { /* работаю */ }
    public void eat() { /* ем */ }
    public void code() { /* пишу код */ }
}
```

SOLID(D - Dependency Inversion)
Инверсия зависимостей

```
public class EmailService {

    private SmtpServer smtp = new
SmtpServer("smtp.gmail.com");

        public void send(String to, String text) {

        smtp.send(to, text);

    }

}
```

SOLID(D - Dependency Inversion)
Инверсия зависимостей

```java
public interface MailServer {
    void send(String to, String text);
}

@Component
public class SmtpMailServer implements MailServer {
    private final String host;

    public SmtpMailServer(@Value("${mail.host}") String host) {
        this.host = host;
    }

    public void send(String to, String text) {
        // отправка через SMTP
    }
}

@Service
public class EmailService {
    private final MailServer mailServer;  // зависимость от интерфейса!

    public EmailService(MailServer mailServer) {  // DI через конструктор
        this.mailServer = mailServer;
    }

    public void send(String to, String text) {
        mailServer.send(to, text);
    }
}
```

DRY — Don't Repeat Yourself

Не повторяйся!

```java
public class OrderService {

    public void createOrder(Order order) {
        // Валидация
        if (order.getTotal() < 0) throw new IllegalArgumentException("Сумма < 0");
        if (order.getItems().isEmpty()) throw new IllegalArgumentException("Нет товаров");
        if (order.getCustomer() == null) throw new IllegalArgumentException("Нет клиента");

        // Сохранение
        orderRepository.save(order);
    }

    public void updateOrder(Order order) {
        // Та же валидация снова!
        if (order.getTotal() < 0) throw new IllegalArgumentException("Сумма < 0");
        if (order.getItems().isEmpty()) throw new IllegalArgumentException("Нет товаров");
        if (order.getCustomer() == null) throw new IllegalArgumentException("Нет клиента");

        // Обновление
        orderRepository.update(order);
    }
}
```

```java
@Service
public class OrderService {
    private final OrderValidator validator;
    private final OrderRepository repository;

    public void createOrder(Order order) {
        validator.validate(order);
        repository.save(order);
    }

    public void updateOrder(Order order) {
        validator.validate(order);
        repository.update(order);
    }
}

@Component
public class OrderValidator {
    public void validate(Order order) {
        if (order.getTotal() < 0) throw new ValidationException("Сумма не может быть отрицательной");
        if (order.getItems().isEmpty()) throw new ValidationException("Заказ пуст");
        if (order.getCustomer() == null) throw new ValidationException("Клиент не указан");
    }
}
```

KISS — Keep It Simple, Stupid!

Плохие примеры

```
public class StringUtils {

    public static String reverse(String input) {

        return new StringBuilder(input).reverse().toString();

    }

        public static String reverseSmart(String input) {

        // Кто-то хотел показать, что знает потоки и лямбды

        return input.chars()

            .mapToObj(c -> (char) c)

            .reduce("", (s, c) -> c + s, (s1, s2) -> s2 + s1);

        // Это работает, но зачем?!

    }

}
```

```
public class UserFilter {

    public List<User> filterAdultsComplex(List<User> users) {
        return users.stream()
            .filter(u -> u.getAge() >= 18)
            .filter(u -> u.getAge() <= 100)
            .filter(u -> u.getStatus() != UserStatus.BANNED)
            .filter(u -> u.getEmail() != null)
            .filter(u -> u.getEmail().contains("@"))
            .collect(Collectors.toList());
        }

}
```

KISS — Keep It Simple, Stupid!

Пример исправления

```
public class UserFilter {

    // ✅ Просто и понятно
    public List<User> filterAdultsSimple(List<User> users) {
        List<User> result = new ArrayList<>();

        for (User user : users) {
            if (isValidAdult(user)) {
                result.add(user);
            }
        }

        return result;
    }

    private boolean isValidAdult(User user) {
        if (user.getAge() < 18 || user.getAge() > 100) return false;
        if (user.getStatus() == UserStatus.BANNED) return false;
        if (user.getEmail() == null || !user.getEmail().contains("@")) return false;

        return true;
    }
}
```

YAGNI — You Ain't Gonna Need It

```
public class ProductService {

    // Зачем-то методы для работы с XML, хотя используем JSON
    public String toXml(Product product) { /* ... */ }
    public Product fromXml(String xml) { /* ... */ }

    // Методы для 10 разных форматов экспорта
    public byte[] toCsv(List<Product> products) { /* ... */ }
    public byte[] toExcel(List<Product> products) { /* ... */ }
    public byte[] toPdf(List<Product> products) { /* ... */ }
    public byte[] toWord(List<Product> products) { /* ... */ }

    // На всякий случай поддержка устаревших версий
    public void saveLegacyV1(Product product) { /* ... */ }
    public void saveLegacyV2(Product product) { /* ... */ }

    // Потенциально полезные методы
    public Product mergeProducts(Product p1, Product p2) { /* ... */ }
    public Product cloneProduct(Product product) { /* ... */ }

}
```

Не пиши код, который может пригодиться когда-нибудь потом

```
public class ProductService {
    private final ProductRepository repository;

    // Только то, что реально используется
    public Product findById(Long id) {
        return
repository.findById(id).orElseThrow();
    }

    public Product save(Product product) {
        return repository.save(product);
    }

    public void delete(Long id) {
        repository.deleteById(id);
    }

    // Если понадобится JSON — добавим потом
    // Если понадобится экспорт — добавим когда
придет задача
  }
```

Сводная таблица

Принцип	Что значит	Как понять, что нарушили
Single Responsibility	Один класс — одна причина для изменения	Класс слишком большой, делает много всего
Open/Closed	Открыт для расширения, закрыт для изменения	При добавлении фичи правим существующий код
Liskov Substitution	Наследники не ломают поведение	Переопределенные методы кидают исключения
Interface Segregation	Много маленьких интерфейсов	Класс реализует методы, которые не использует
Dependency Inversion	Зависи от абстракций	Создаем объекты через new, а не через DI
DRY	Не повторяйся	Копипаст одинаковой логики
KISS	Не усложняй	Есть решение проще, но мы сделали сложно
YAGNI	Не пиши на будущее	Есть код, который никто не использует

Именование — говори на человеческом языке

```java
// Плохо
int d;  // что это?
String s;  // зачем?
List<User> ul;  // user list?
void proc()  // что делает?
```

```java
// Понятное назначение
int daysUntilExpiration;
String userName;
List<User> activeUsers;
void sendWelcomeEmail()
```

```java
// Для boolean — вопросы
boolean isActive;
boolean hasAccess;
boolean canDelete;
```

```java
// Методы-глаголы
User getUser();
void saveUser();
void deleteById();
```

```java
// Наименование сервисом и компонентов
UserService  // сервис для работы с пользователями
EmailSender  // отправитель писем
OrderProcessor  // обработчик заказов
```

Магические числа и константы

```java
// Плохо
public class DiscountService {

    public double calculateDiscount(double price) {
        if (price > 10000) {
            return price * 0.3;  // что за 0.3?
        } else if (price > 5000) {
            return price * 0.2;  // а это?
        } else {
            return price * 0.1;  // ?
        }
    }
}
```

```java
public class DiscountService {
    private static final double PREMIUM_DISCOUNT_RATE = 0.3;
    private static final double STANDARD_DISCOUNT_RATE = 0.2;
    private static final double BASIC_DISCOUNT_RATE = 0.1;

    private static final double PREMIUM_THRESHOLD = 10000;
    private static final double STANDARD_THRESHOLD = 5000;

    public double calculateDiscount(double price) {
        if (price > PREMIUM_THRESHOLD) {
            return price * PREMIUM_DISCOUNT_RATE;
        } else if (price > STANDARD_THRESHOLD) {
            return price * STANDARD_DISCOUNT_RATE;
        } else {
            return price * BASIC_DISCOUNT_RATE;
        }
    }
}
```

Магические числа и константы

```
public interface DiscountServiceDict {
    public static final double PREMIUM_DISCOUNT_RATE = 0.3;
    public static final double STANDARD_DISCOUNT_RATE = 0.2;
    public static final double BASIC_DISCOUNT_RATE = 0.1;
    public static final double PREMIUM_THRESHOLD = 10000;
    public static final double STANDARD_THRESHOLD = 5000;
}

public class DiscountService implements DiscountServiceDict {

    public double calculateDiscount(double price) {
        if (price > PREMIUM_THRESHOLD) {
            return price * PREMIUM_DISCOUNT_RATE;
        } else if (price > STANDARD_THRESHOLD) {
            return price * STANDARD_DISCOUNT_RATE;
        } else {
            return price * BASIC_DISCOUNT_RATE;
        }
    }
}
```

Размер методов — не больше экрана!

```
override fun onResponse(
    call: Call<List<Destination>>, response:
Response<List<Destination>>
) {
    when {
        response.isSuccessful -> {  // status code: 200 - 299
            // get destinationList from responseBody
            // retrofit Gson converts JSON to Java Object
            val destinationList = response.body()!!
// set destiny_recycler_view.adapter to DestinationAdapter using
destinationList
            destiny_recycler_view.adapter =
DestinationAdapter(destinationList)
            Log.d(TAG, "onResponse: Successfull!")
        }
        response.code() == 401 -> {
            toastMessage("Your session has expired",
this@DestinationListActivity)
        }

        else -> {
            /**
             * Response code was not in 200s range
             * Example: Server Error 422: Unprocessable Entity:
             * the server understands the content type of the
request entity,
             * and the syntax of the request entity is correct,
             * but it was unable to process the contained
instructions.
             */
            Log.e(TAG, "onResponse: ${response.errorBody()}")
            toastMessage("Failed to retrieve items.",
this@DestinationListActivity)
        }
    }
```

```
187
188  public void processOrder(Order order) {
189      validateOrder(order);
190      checkInventory(order);
191      applyDiscounts(order);
192      processPayment(order);
193      updateOrderStatus(order);
194      sendConfirmationEmail(order);
195  }
196
197  private void validateOrder(Order order) {
198      // валидация
199  }
200
201  private void checkInventory(Order order) {
202      // проверка склада
203  }
204
205
206
```

Комментарии — только там, где нужно

```
// ❌ ПЛОХО: Очевидные комментарии
// увеличиваем счетчик на 1
counter++;

// получаем пользователя по id
User user = userRepository.findById(id);

// ✅ ХОРОШО: Комментарии объясняют ПОЧЕМУ, а не ЧТО

// Используем кэш, потому что этот запрос очень медленный
User user = cache.get(id);

// Округляем вниз, чтобы не переплатить (условия контракта)
double payment = Math.floor(total * 0.1);

// Костыль: старый клиент шлет id как строку, парсим
try {
    userId = Long.parseLong(request.getUserId());
} catch (NumberFormatException e) {
    // Если не спарсилось — ищем по логину
    userId = userRepository.findByLogin(request.getUserId()).getId();
  }
```

Правило бойскаута
Оставь код лучше, чем ты его нашел

```java
// Вы пришли в проект и увидели:

public class BadCode {

    public void d() {  // что за d?

        int x = 5;

        int y = 10;

        int z = x * y;  // зачем?

        System.out.println(z);

        // 50 строк спагетти-кода...

    }

  }
```

```java
// Вы уходите:
public class Calculator {
    private static final int DEFAULT_MULTIPLIER = 5;
    private static final int DEFAULT_MULTIPLICAND = 10;

    public void demonstrateMultiplication() {
        int result = multiply(DEFAULT_MULTIPLIER,
DEFAULT_MULTIPLICAND);
        System.out.println("Результат умножения: " +
result);
    }

    private int multiply(int a, int b) {
        return a * b;
    }
  }
```

Подводя итоги

Читабельность важнее "умности"

Имена должны говорить сами за себя

Методы должны делать ОДНО дело

Классы должны быть СФОКУСИРОВАННЫ

Дублирование — зло (DRY)

Простота — это круто (KISS)

Не пиши код "на всякий случай" (YAGNI)

Оставляй код чище, чем нашел

Комментируй ПОЧЕМУ, а не ЧТО

Код читают люди (в том числе ты через месяц)

Ресурсы

- https://timeweb.cloud/blog/solid-principy-i-ih-rol-v-razrabotke-po
- https://en.wikipedia.org/wiki/Don%27t_repeat_yourself
- https://dev.to/juniourrau/clean-code-essentials-yagni-kiss-and-dry-in-software-engineering-4i3j
- https://www.in-com.com/ru/blog/the-boy-scout-rule-the-secret-to-effortless-refactoring-that-scales/
- https://www.ozon.ru/category/chistyy-kod-robert-martin/ - Книга “Чистый код” Мартина
- https://www.ozon.ru/product/sovershennyy-kod-makkonnell-stiv-221777342/ - Соверщеный код (**Макконнелл Стив**)