Язык Java

Глава IV. Основы объектно-ориентированного проектирования

Виталий Витальевич Перевощиков

Осенний семестр 2021

Содержание

Чистый код (Clean Code)

- **Рефакторинг** - перепроектирование, равносильное преобразование кода с целью:
  - облегчить понимание работы программы
  - избегать неочевидных и с трудом исправляемых ошибок
  - облегчить расширение функциональности программы
- Веб-страница, посвященная рефакторингу: refactoring.guru

“Запахи” кода (Code Smells)

- Распространенные примеры “запахов” кода:
  - refactoring.guru
  - Github-репозиторий

Принципы SOLID

- 5 основных принципов объектно-ориентированного программирования и проектирования:
  - **S**ingle responsibility - принцип единственной ответственности
  - **O**pen-closed - принцип открытости/закрытости
  - **L**iskov substitution - принцип подстановки Liskov
  - **I**nterface segregation - принцип разделения интерфейса
  - **D**ependency inversion - принцип инверсии зависимостей

Принцип единственной ответственности (S)

- “Класс должен иметь только одну причину для изменения”
- Каждый класс должен решать лишь одну задачу
- Пример нарушения принципа:

```
class Student {
    private String name;
    private MySQLDatabase database;

    Student(String name, MySQLDatabase database) {
        this.name = name;
        this.database = database;
    }

    String getName() { return name; }

    void saveInDatabase() {
        String sqlQueryString = "INSERT INTO ..." + name + "...";
        database.query(sqlQueryString);
    }
}
```

Принцип единственной ответственности (S)

- Улучшение кода:

```
class Student {
    private String name;

    Student(String name) { this.name = name; }

    String getName() { return this.name; }
}

class StudentDatabaseRepository {
    private MySQLDatabase database;

    StudentDatabaseRepository(MySQLDatabase database) { this.database = database; }

    void save(Student student) {
        String sqlQueryString = "INSERT INTO ... " + student.getName() + "...";
        database.query(sqlQueryString);
    }
}
```

Принцип открытости/закрытости (O)

- “Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для изменения”
- Пример нарушения принципа:

```
class Student {
    private String role;

    Student(String role) { this.role = role; }

    String getInfo() {
        switch (role) {
            case "Математика":
                return "Студент-математик";
            case "Программирование":
                return "Студент-программист";
            default:
                return "Студент";
        }
    }
}
```

```
void test() {
    Student mathStudent = new Student("Математика");
    String mathStudentInfo = mathStudent.getInfo(); // "Студент-математик"

    Student progStudent = new Student("Программирование");
    String progStudentInfo = progStudent.getInfo(); // "Студент-программист"
}
```

Принцип открытости/закрытости (O)

- Улучшение кода:

```
class Student {
    String getInfo() {
        return "Студент";
    }
}

class MathStudent extends Student {
    @Override
    String getInfo() {
        return "Студент-математик";
    }
}

class ProgrammingStudent extends Student {
    @Override
    String getInfo() {
        return "Студент-программист";
    }
}
```

```
void test() {
    Student mathStudent = new MathStudent();
    String mathStudentInfo = mathStudent.getInfo(); // "Студент-математик"

    Student progStudent = new ProgrammingStudent();
    String progStudentInfo = progStudent.getInfo(); // "Студент-программист"
}
```

Принцип подстановки Liskov (L)

- “Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом”
- Производный класс должен быть взаимозаменяем с родительским классом
- Пример нарушения принципа:

```
class Student {
    String name;
}

class MathStudent extends Student {
    String teachingInternship;
}

class ProgrammingStudent extends Student {
    String programmingInternship;
}
```

```
// сервис по студенческой практике
class StudentInternshipService {
    // нарушение принципа подстановки Liskov
    static void setInternship(Student student, String internship) {
        if (student instanceof MathStudent) {
            ((MathStudent) student).teachingInternship = internship;
        } else if (student instanceof ProgrammingStudent) {
            ((ProgrammingStudent) student).programmingInternship = internship;
        }
    }
}
```

Принцип подстановки Liskov (L)

- Улучшение кода:

```java
abstract class Student {
    String name;
    abstract void setInternship(String internship);
}

class MathStudent extends Student {
    String teachingInternship;

    @Override
    void setInternship(String internship) {
        teachingInternship = internship;
    }
}

class ProgrammingStudent extends Student {
    String programmingInternship;

    @Override
    void setInternship(String internship) {
        programmingInternship = internship;
    }
}
```

```java
class StudentInternshipService {
    static void setInternship(Student student, String internship) {
        student.setInternship(internship);
    }
}
```

Принцип разделения интерфейса (I)

- “Много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения”
- Пример нарушения принципа:

```java
interface Student {
    String getFavoriteProgrammingLanguage();
    String getFavoriteTheorem();
    String getFavoriteBookAuthor();
}
```

```java
class ProgrammingStudent implements Student {
    @Override
    public String getFavoriteProgrammingLanguage() {
        return "Java";
    }

    @Override
    public String getFavoriteTheorem() {
        return "Теорема Вейерштрасса";
    }

    @Override
    public String getFavoriteBookAuthor() {
        return "Ф.М. Достоевский";
    }
}
```

```java
class MathStudent implements Student {
    @Override
    public String getFavoriteProgrammingLanguage() {
        return null; // oops...
    }

    @Override
    public String getFavoriteTheorem() {
        return "Теорема Лагранжа";
    }

    @Override
    public String getFavoriteBookAuthor() {
        return "А.С. Пушкин";
    }
}
```

```java
class LiteratureStudent implements Student {
    @Override
    public String getFavoriteProgrammingLanguage() {
        return null; // oops...
    }

    @Override
    public String getFavoriteTheorem() {
        return null; // oops...
    }

    @Override
    public String getFavoriteBookAuthor() {
        return "А.П. Чехов";
    }
}
```

Принцип разделения интерфейса (I)

- Улучшение кода: разбиение общего интерфейса на несколько

```java
interface Programmer {
    String getFavoriteProgrammingLanguage();
}

interface Mathematician {
    String getFavoriteTheorem();
}

interface Reader {
    String getFavoriteBookAuthor();
}
```

```java
class ProgrammingStudent implements Programmer,
        Mathematician, Reader {
    @Override
    public String getFavoriteProgrammingLanguage() {
        return "Java";
    }

    @Override
    public String getFavoriteTheorem() {
        return "Теорема Вейерштрасса";
    }

    @Override
    public String getFavoriteBookAuthor() {
        return "Ф.М. Достоевский";
    }
}
```

```java
class MathStudent implements Mathematician, Reader {
    @Override
    public String getFavoriteTheorem() {
        return "Теорема Лагранжа";
    }

    @Override
    public String getFavoriteBookAuthor() {
        return "А.С. Пушкин";
    }
}
```

```java
class LiteratureStudent implements Reader {
    @Override
    public String getFavoriteBookAuthor() {
        return "А.П. Чехов";
    }
}
```

Принцип инверсии зависимостей (D)

- “Объектом зависимости должна быть абстракция, а не что-то конкретное”
- Пример нарушения принципа:

```java
class MicrosoftAzureStorage {
    void uploadFile(String fileName, byte[] fileContent) {
        // загрузка файла в облачное хранилище данных Microsoft Azure
    }
}

class StudentService {
    private MicrosoftAzureStorage azureStorage;

    StudentService(MicrosoftAzureStorage azureStorage) {
        this.azureStorage = azureStorage;
    }

    // загрузить больничный лист в систему
    void uploadDoctorCertificate(byte[] document) {
        azureStorage.uploadFile("Больничный_Иванов.pdf", document);
    }
}
```

- Проблемы:
  - Как писать юнит-тесты?
  - Что делать, если нужно перейти на использование Amazon S3 или сервера БФУ?

Принцип инверсии зависимостей (D)

- Улучшение кода:

```java
interface FileStorage {
    void uploadFile(String fileName, byte[] fileContent);
}

class MicrosoftAzureStorage implements FileStorage {
    public void uploadFile(String fileName, byte[] fileContent) {
        // загрузка файла в облачное хранилище данных Microsoft Azure
    }
}

class AmazonS3Bucket implements FileStorage {
    public void uploadFile(String fileName, byte[] fileContent) {
        // загрузка файла в облачное хранилище данных Amazon S3
    }
}

class BFUStorage implements FileStorage {
    public void uploadFile(String fileName, byte[] fileContent) {
        // сохранение файла на сервере БФУ
    }
}
```

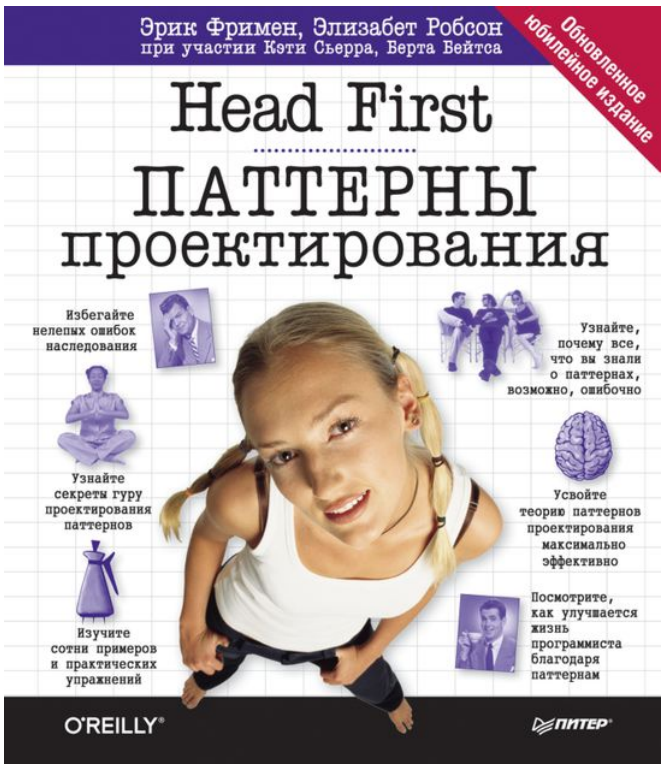
```java
class StudentService {
    private FileStorage fileStorage;

    StudentService(FileStorage fileStorage) { this.fileStorage = fileStorage; }

    // загрузить больничный лист в систему
    void uploadDoctorCertificate(byte[] document) {
        fileStorage.uploadFile("Больничный_Иванов.pdf", document);
    }
}
```

Паттерны (шаблоны) проектирования

- Архитектурная конструкция, являющаяся решением часто возникающей проблемы проектирования.
- Преимущества паттернов:
  - снижение сложности разработки за счет использования готовых архитектурных решений
  - облегчение коммуникации между разработчиками
  - снижение количества ошибок
- Недостатки паттернов:
  - Усложнение программы:
    - из-за “слепого“ следования паттерну проектирования
    - из-за неоправданного применения паттерна проектирования

Паттерны проектирования

- Дополнительные источники:
  - Веб-страница о паттернах проектирования: ссылка
  - Популярная книга о паттернах проектирования:



Паттерн “Фабричный метод” (Factory Method)

- Базовая идея:
  - Отказ от использования оператора `new` напрямую
  - Использование **фабричного метода** для создания объектов:

```java
class BachelorStudent {
    // поля, конструкторы, методы
}

class StudentService {
    // инкапсуляция вызова конструктора в "фабричный метод"
    BachelorStudent createStudent() {
        return new BachelorStudent();
    }
}
```

- Мотивация: создание объектов различных типов

```java
interface Student {
    // методы
}

class BachelorStudent implements Student {
    // поля, конструкторы, методы
}

class MasterStudent implements Student {
    // поля, конструкторы, методы
}

class StudentService {
    // "наивный" подход к созданию различных типов
    Student createStudent(String studentType) {
        if ("bachelor".equals(studentType)) {
            return new BachelorStudent();
        } else {
            return new MasterStudent();
        }
    }
}
```

Паттерн “Фабричный метод” (Factory Method)

- Упрощение реализации: каждый из типов объектов создается в классе-наследнике:

```
abstract class StudentService {
    abstract Student createStudent();
}

class BachelorService extends StudentService {
    @Override
    Student createStudent() {
        return new BachelorStudent();
    }
}

class MasterService extends StudentService {
    @Override
    Student createStudent() {
        return new MasterStudent();
    }
}
```

Паттерн “строитель” (Builder)

- Мотивация: создание сложных объектов

```
class Student {
    String name;
    String studyProgram;
    boolean isBachelor;
    boolean hasGithubAccount;
    boolean hasScholarship;
    int age;

    Student(String name, String studyProgram, boolean isBachelor,
            boolean hasGithubAccount, boolean hasScholarship, int age) {
        this.name = name;
        this.studyProgram = studyProgram;
        this.isBachelor = isBachelor;
        this.hasGithubAccount = hasGithubAccount;
        this.hasScholarship = hasScholarship;
        this.age = age;
    }
}
```

Удобно ли использовать такой конструктор?

```
Student student = new Student("Иванов И.И.", "Программирование", false, true, true, 17);
```

Паттерн “строитель” (Builder)

- Идея: доверить создание объектов новому классу-строителю:

```java
class StudentBuilder {
    // поля класса Student
    private String name;
    private boolean isBachelor;
    // ...

    // сеттеры для полей
    StudentBuilder setName(String name) {
        this.name = name;
        return this;
    }
    StudentBuilder setIsBachelor(boolean isBachelor) {
        this.isBachelor = isBachelor;
        return this;
    }
    // ...

    // создание экземпляра класса Student
    Student build() {
        Student student = new Student();
        student.name = name;
        student.isBachelor = isBachelor;
        // ...
        return student;
    }
}
```

```java
StudentBuilder studentBuilder = new StudentBuilder();
Student student = studentBuilder
        .setName("Иванов И.И.")
        .setIsBachelor(true)
        // ...
        .build();
```

Паттерн “декоратор” (Decorator)

- Позволяет динамически расширить функциональность объектов
- “Плохой” пример:

```
class Student {
    String getName() {
        return "Иванов И.И.";
    }
}

class MasterStudent extends Student {
    String getRole() {
        return "студент-магистр";
    }
}

class ProgrammingMasterStudent extends MasterStudent {
    String getGithubLink() {
        return "github.com/ivanov";
    }
}

class StudentInfo {
    private Student student;

    StudentInfo(Student student) { this.student = student; }

    // Нарушение принципа подстановки Liskov
    String getInfo() {
        String info = "имя: " + student.getName() + ";";
        if (student instanceof MasterStudent) {
            info += " роль: " + ((MasterStudent) student).getRole() + ";";
        }
        if (student instanceof ProgrammingMasterStudent) {
            info += " профиль на Github: " + ((ProgrammingMasterStudent) student).getGithubLink() + ";";
        }
        return info;
    }
}
```

Паттерн “декоратор” (Decorator)

- Определение общего интерфейса для базового класса и оберток

```java
interface StudentInfo {
    String getInfo();
}

// базовый класс для вывода базовой информации о студенте
class BaseStudentInfo implements StudentInfo {
    private Student student;

    BaseStudentInfo(Student student) { this.student = student; }

    public String getInfo() {
        return "имя: " + student.getName() + ";";
    }
}
```

Паттерн “декоратор” (Decorator)

```java
// класс-декоратор для вывода специфической информации для студентов-магистров
class MasterInfoDecorator implements StudentInfo {
    private StudentInfo baseStudentInfo;
    private MasterStudent masterStudent;

    public MasterInfoDecorator(MasterStudent masterStudent, StudentInfo baseStudentInfo) {
        this.baseStudentInfo = baseStudentInfo;
        this.masterStudent = masterStudent;
    }

    public String getInfo() {
        final String roleInfo = " роль: " + masterStudent.getRole() + ";";
        return baseStudentInfo.getInfo() + roleInfo;
    }
}
```

```java
// класс-декоратор для вывода специфической информации для программирующих студентов-магистров
class ProgrammingMasterInfoDecorator implements StudentInfo {
    private StudentInfo baseStudentInfo;
    private ProgrammingMasterStudent student;

    public ProgrammingMasterInfoDecorator(ProgrammingMasterStudent student, StudentInfo baseStudentInfo) {
        this.baseStudentInfo = baseStudentInfo;
        this.student = student;
    }

    public String getInfo() {
        final String githubInfo = " профиль на Github: " + student.getGithubLink() + ";";
        return baseStudentInfo.getInfo() + githubInfo;
    }
}
```

Паттерн “декоратор” (Decorator)

```
ProgrammingMasterStudent student = new ProgrammingMasterStudent();
BaseStudentInfo baseInfo = new BaseStudentInfo(student);
MasterInfoDecorator masterInfo = new MasterInfoDecorator(student, baseInfo);
ProgrammingMasterInfoDecorator progMasterInfo = new ProgrammingMasterInfoDecorator(student, masterInfo);

System.out.println(baseInfo.getInfo());       // "имя: Иванов И.И."
System.out.println(masterInfo.getInfo());     // "имя: Иванов И.И.; роль: студент-магистр;"
System.out.println(progMasterInfo.getInfo()); // "имя: Иванов И.И.; роль: студент-магистр; профиль на Github: github.com/ivanov;"
```

Паттерн “одиночка“ (Singleton)

- Гарантирует, что у класса существует только один экземпляр
- Представляет глобальную точку доступа к этому экземпляру

```java
class Database {
    private static Database instance;

    // можно создать объект только внутри класса
    private Database() {}

    // возвращает единственный экземпляр класса
    public static Database getInstance() {
        // острожно! возможно "состояние гонки" при многопоточности,
        // из-за чего создастся несколько экземпляров класса
        if (instance == null) {
            instance = new Database();
        }
        return instance;
    }
}
```

```java
Database database = Database.getInstance();
Database anotherDatabase = Database.getInstance();

System.out.println(database == anotherDatabase); // true
```