

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ  
МИРЭА – РОССИЙСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

---

**Н.В. СТРОГАНКОВА, К.В. КАСЬЯНЕНКО, А.В. ХОЗЯИНОВ,  
О.В. СОБОЛЕВ**

## **ШАБЛОНЫ ПРОГРАММНЫХ ПЛАТФОРМ ЯЗЫКА JAVA**

Практикум для студентов, обучающихся по направлению подготовки  
09.03.04 «Программная инженерия»

Москва – 2021

УДК 532.78:548.5

ББК 22.317

И20

**Строганкова Н.В. Шаблоны программных платформ языка Java** [Электронный ресурс]: практикум / Строганкова Н.В., Касьяненко К.В., Хозяинов А.В., Соболев О.В. – М.: МИРЭА – Российский технологический университет (РТУ МИРЭА), 2021. – 1 электрон. опт. диск (CD-ROM).

Разработан в помощь студентам, выполняющим практические работы по дисциплине «Шаблоны программных платформ языка Java». В состав практикума входят: использование шаблонов проектирования в клиент-серверных приложениях.

Предназначено для студентов бакалавриата 2 курса направления подготовки 09.03.04 «Программная инженерия».

В состав практикума входят: цель работы; задачи для достижения поставленной цели; описание выполнения практической работы; ожидаемые результаты и форма их представления к защите.

Практикум издается в авторской редакции.

Авторский коллектив: Строганкова Наталья Владимировна, Касьяненко Константин Владимирович, Хозяинов Артем Владимирович, Соболев Олег Вадимович.

**Рецензент:**

**Петров Андрей Борисович**, доктор технических наук, профессор кафедры корпоративных информационных систем РТУ МИРЭА

Минимальные системные требования:

Поддерживаемые ОС: Windows 2000 и выше.

Память: ОЗУ 256МБ.

Жесткий диск: 120 Мб.

Устройства ввода: клавиатура, мышь, скрин.

Дополнительные программные средства: Программа Adobe Reader.

Подписано к использованию по решению Редакционно-издательского совета МИРЭА – Российского технологического университета от \_\_\_\_ 2021 г.

Объем: \_\_\_\_ Мб

**Тираж 10**

ISBN \_\_\_\_\_

© Н.В. Строганкова, К.В. Касьяненко,  
А.В. Хозяинов, О.В. Соболев, 2021

© МИРЭА – Российский технологический  
университет, 2021

## СОДЕРЖАНИЕ

ВВЕДЕНИЕ .....	4
1. ПРАКТИЧЕСКАЯ РАБОТА № 1 .....	5
2. ПРАКТИЧЕСКАЯ РАБОТА № 2 .....	10
3. ПРАКТИЧЕСКАЯ РАБОТА № 3 .....	15
4. ПРАКТИЧЕСКАЯ РАБОТА № 4 .....	20
5. ПРАКТИЧЕСКАЯ РАБОТА № 5 .....	24
6. ПРАКТИЧЕСКАЯ РАБОТА № 6 .....	26
7. ПРАКТИЧЕСКАЯ РАБОТА № 7 .....	29
8. ПРАКТИЧЕСКАЯ РАБОТА № 8 .....	33
9. ПРАКТИЧЕСКАЯ РАБОТА № 9 .....	39
10. ПРАКТИЧЕСКАЯ РАБОТА № 10 .....	44
11. ПРАКТИЧЕСКАЯ РАБОТА № 11 .....	52
12. ПРАКТИЧЕСКАЯ РАБОТА № 12 .....	53
13. ПРАКТИЧЕСКАЯ РАБОТА № 13 .....	55
14. ПРАКТИЧЕСКАЯ РАБОТА № 14 .....	57
15. ПРАКТИЧЕСКАЯ РАБОТА № 15 .....	61
16. ПРАКТИЧЕСКАЯ РАБОТА № 16 .....	67
17. ПРАКТИЧЕСКАЯ РАБОТА № 17 .....	71
18. ПРАКТИЧЕСКАЯ РАБОТА № 18 .....	73
19. ПРАКТИЧЕСКАЯ РАБОТА № 19 .....	77
20. ПРАКТИЧЕСКАЯ РАБОТА № 20 .....	80
21. ПРАКТИЧЕСКАЯ РАБОТА № 21 .....	83
22. ПРАКТИЧЕСКАЯ РАБОТА № 22 .....	85
23. ПРАКТИЧЕСКАЯ РАБОТА № 23 .....	86
24. ПРАКТИЧЕСКАЯ РАБОТА № 24 .....	90
25. ОПИСАНИЕ ВЫПОЛНЕНИЯ РАБОТ .....	94
26. ЗАЩИТА ПРАКТИЧЕСКИХ РАБОТ .....	95
ПЕРЕЧЕНЬ СОКРАЩЕНИЙ.....	96
БИБЛИОГРАФИЧЕСКИЙ СПИСОК .....	97

## **ВВЕДЕНИЕ**

Данный практикум разработан к практическим работам в рамках дисциплины «Шаблоны программных платформ языка Java» для студентов-бакалавров направления 09.03.04 «Программная инженерия».

Цель курса:

- ознакомление студентов с основными технологиями, необходимыми для создания клиент-серверных приложений;
- изучение фреймворка Spring и работы с ним;
- предоставление теоретической базы для выполнения практических работ.

Данный практикум позволяет на практике разобраться с шаблонами проектирования, использованием шаблонов проектирования в клиент-серверных приложениях. В качестве основного языка программирования выбран язык Java. Также в практических работах будет использоваться фреймворк Spring, ORM библиотека Hibernate.

Практикум разработан к 24 практическим работам, в результате выполнения которых студенты смогут овладеть навыками использования шаблонов проектирования на языке Java. Каждая практическая работа рассчитана на выполнение одним студентом. Некоторые практические работы содержат варианты заданий, которые нужно выбрать в соответствии с порядковым номером студента в группе.

# 1. ПРАКТИЧЕСКАЯ РАБОТА № 1

## Цель работы

Знакомство со встроенными функциональными интерфейсами Java. Возможности Java 8. Лямбда-выражения. Области действия, замыкания. Предикаты. Функции. Компараторы.

### 1.1. Теоретическая часть

Версия Java 8 привнесла огромные изменения в язык, и самым важным является появление функциональной парадигмы в Java. Начнем с того, что же такое функциональное программирование.

Наиболее привычным является *императивное программирование*, в котором программирование происходит путем описания последовательности инструкций, идущих друг за другом, которые требуется выполнить для достижения требуемого результата. Но также существует *декларативное программирование*, в котором описываются не действия для получения результата, а сам результат – программист описывает только требования, которым должен удовлетворять результат. Самым простым примером декларативного программирования является использование языка SQL:

```
select * from users where age > 20;
```

*Функциональное программирование* является подтипом декларативного программирования. Оно основано на использовании функций как основных строительных блоков построения приложения. В Java функциональное программирование, в первую очередь, реализовано с использованием функциональных интерфейсов. *Функциональный интерфейс* – интерфейс с одним и только одним абстрактным методом. Они помечаются аннотацией `FunctionalInterface`, при помощи которой компилятор не допустит интерфейсы с более чем одним абстрактным методом.

```
@FunctionalInterface
public interface Summator<T> {
    T sum(T elem1, T elem2);
}
```

```
}
```

Если в каком-то методе функциональный интерфейс передан как параметр, то вместо его реализации можно использовать лямбда-функцию, что заметно упрощает код, делает его более читаемым.

*Лямбда функция* – блок кода, описывающий функцию интерфейса, некий аналог анонимного класса, только для функциональных интерфейсов.

Ниже приведен пример реализации интерфейса Summator с использованием лямбды:

```
Summator<Integer> summator = (a, b) -> a + b;
```

Однако, если будет использоваться внешний объект в лямбда выражении, то в таком случае ссылка должна быть или `final` или `effectively final` (может не помечаться как `final`, но обязана не изменяться (не должно нигде происходить присвоения ссылки новому значению)):

```
Integer c = 40;  
Summator<Integer> summator = (a, b) -> a + b + c;  
System.out.println(summator.sum(1, 3));
```

При этом, если попытаться изменить значение переменной `c`, то получим ошибку:

```
Integer c = 40;  
Summator<Integer> summator = (a, b) -> a + b + c;  
c = 11;  
System.out.println(summator.sum(1, 3));
```

**Ошибка:**

```
java: local variables referenced from a lambda expression  
must be final or effectively final
```

В Java 8 было добавлено большое количество стандартных функциональных интерфейсов. Например, `Predicate`, получающий на вход какой-то объект и возвращающий `boolean`:

```
@FunctionalInterface  
public interface Predicate<T> {  
    boolean test(T t);  
}
```

*Predicate* можно передавать как параметр в какие-то методы, например, для фильтрации объектов. Еще одним стандартным интерфейсом является *Function*:

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

Отлично подойдет для преобразования одного объекта в другой.

*Comparator* используется для сравнения двух объектов одного типа:

```
@FunctionalInterface
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

Еще одно удобство использования функциональных интерфейсов – возможность передачи метода или конструктора в качестве аргумента:

```
Summator<Integer> sum = Integer::sum;
Integer result = sum.sum(1,2);
System.out.println(result);
```

Отдельно в вакууме стандартные функциональные интерфейсы не кажутся очень полезными, но их использование для создания каких-то других классов с дополнительной логикой значительно упрощает код. Удобство использования стандартных функциональных интерфейсов вы увидите в следующей практической работе – «Работа со Stream API в Java 8».

## 1.2. Варианты индивидуального задания

1) Имплементировать интерфейс *Function*, получающий на вход массив чисел (каждое число может быть значением от 0 до 9) и возвращающий строку – наименьшее число, которое возможно собрать из данных чисел, используя цифры только один раз (игнорируя дубликаты). Пример:

```
minValue ({1, 3, 1}) ==> return (13)
```

2) Имплементировать интерфейс `Function`, получающий на вход массив студентов и возвращающий сгруппированных по группе студентов (`Map<String, List<Student>>`).

3) Имплементировать интерфейс `Function`, получающий на вход массив строк и возвращающий массив отзеркаленных строк.

4) Имплементировать интерфейс `Function`, получающий на вход пару чисел и возвращающий наибольший общий делитель.

5) Имплементировать интерфейс `Comparator`, сравнивающий две строки по сумме всех чисел, представленных в строке.

6) Имплементировать интерфейс `Comparator`, сравнивающий двух студентов по набранным за семестр баллов.

7) Имплементировать интерфейс `Comparator`, сравнивающий два числа по модулю.

8) Имплементировать интерфейс `Comparator`, сравнивающий два массива с одинаковыми типами элементов по количеству элементов в данных массивах.

9) Имплементировать интерфейс `Predicate`, определяющий, является ли данная строка PIN-кодом (содержит ровно 4 цифры или 6 цифр).

10) Имплементировать интерфейс `Predicate`, определяющий, является ли данная строка email-адресом, используя регулярное выражение.

11) Имплементировать интерфейс `Predicate`, определяющий, является ли число степенью двойки.

12) Имплементировать интерфейс `Predicate`, определяющий, содержит ли массив студентов студента с максимальным количеством баллов (максимальное значение – 100).

13) Имплементировать интерфейс `Consumer`, принимающий на вход строку, заменяющий каждый третий символ строки на такой же символ в верхнем регистре и выводящий в консоль результат.



14) Имплементировать интерфейс Consumer, принимающий на вход массив строк и выводящий в консоль строку с наибольшим количеством уникальных символов.

15) Имплементировать интерфейс Consumer, принимающий на вход массив чисел и выводящий в консоль в порядке возрастания.

## 2. ПРАКТИЧЕСКАЯ РАБОТА № 2

### Цель работы

Работа со Stream API в Java 8.

### 2.1. Теоретическая часть

В Java 8 был добавлен новый способ взаимодействия с некими коллекциями объектов – Stream API. В первую очередь, Stream стоит воспринимать именно как поток неких объектов, который можно так или иначе изменять. Особенности Stream:

- потоки не являются структурой данных, не хранит элементы;
- потоки не изменяют начальную структуру данных, а лишь возвращают результат в виде потока новых данных;
- нетерминальные операции в потоках являются «ленивыми», то есть запускаются только по требованию при запуске терминальных операций.

*Нетерминальные операции*, или промежуточные – те операции, которые возвращают трансформированный поток данных, *терминальные операции* возвращают конечный результат и фактически завершают поток. Пример создания потока:

```
Stream<Integer> stream = Stream.of(1, 2, 3);
```

Ниже перечислены некоторые виды нетерминальных и терминальных операций с их описанием.

#### **Нетерминальные операции:**

- sorted() – сортирует поток. Может быть передана реализация интерфейса Comparator для сортировки;
- filter(Predicate<? super T> predicate ) – фильтрует элементы потока в соответствии с реализацией Predicate;
- map(Function<? super T, ? extends R> mapper) – трансформирует объекты потока;
- distinct() – убирает дубликаты;
- skip(long n) – пропускает первые n элементов;

– `<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)` – превращает один объект в поток, затем все потоки конкатенирует.

### **Терминальные операции:**

– `forEach(Consumer<? super T> action)` – выполняет какую-то операцию для каждого элемента;

– `reduce(BinaryOperator<T> accumulator)` – аккумулирует все объекты в один, по очереди применяя ее к парам из элементов, затем также к результату операции и следующему элементу, и так далее по всем элементам, получая в итоге один результат. Самый простой пример для понимания – нахождение суммы:

```
Optional<Integer> sum = Stream.of(1, 2, 3).reduce((a, b) -> a + b);
```

– `min(Comparator<? super T> comparator)` – находит минимальный элемент с использованием `Comparator`;

– `count()` – возвращает количество элементов;

– `findFirst()` – возвращает первый элемент.

Конечно же, `Stream` имеет много больше различных операций, здесь перечислены только самые базовые операции.

В Java 8 потоки не могут быть использованы повторно. После вызова любого терминального метода поток завершается:

```
Stream<Integer> stream = Stream.of(1, 2, 3);
stream.forEach((a) -> {});
stream.findFirst();
```

### **Ошибка:**

```
Exception in thread "main" java.lang.IllegalStateException:
stream has already been operated upon or closed
```

### **Пример использования потоков**

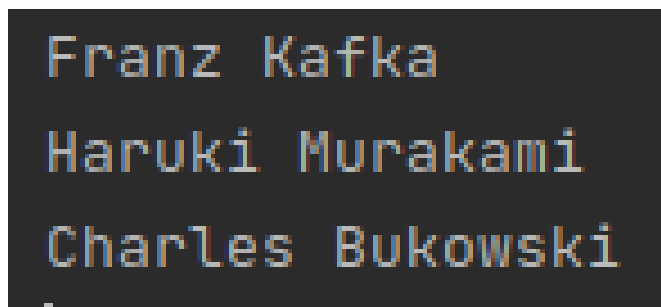
Например, дан список строк, нужно отфильтровать все строки длиной больше 10 и отсортировать по длине строки. Для начала следует создать список, заполнить его данными и получить из него поток:

```
List<String> input = new ArrayList<>();  
input.add("My");  
input.add("Authors");  
input.add("Haruki Murakami");  
input.add("Franz Kafka");  
input.add("Charles Bukowski");  
Stream<String> stream = input.stream();
```

Затем отфильтровать, отсортировать по длине и вывести результат в консоль:

```
stream.filter(str -> str.length() > 10)  
.sorted(Comparator.comparingInt(String::length))  
.forEach(System.out::println);
```

В итоге получим такой результат (рисунок 1):



```
Franz Kafka  
Haruki Murakami  
Charles Bukowski
```

Рисунок 1 – Результат выполнения примера использования потоков

Для лучшего изучения потоков и функционального программирования в Java рекомендуется прочитать книгу «Functional Interfaces in Java» Ralph Lecessi (<https://www.amazon.com/Functional-Interfaces-Java-Fundamentals-Examples-ebook/dp/B07NRHQSCW>).

## 2.2. Задание

В ходе выполнения практической работы должно быть реализовано:

1) класс Human (int age, String firstName, String lastName, LocalDate birthDate, int weight);

2) приложение, которое создает список из объектов класса Human, а затем производит действия в соответствии с вариантом индивидуального задания (список после каждого этапа должен выводиться в консоль).

Все действия должны производиться только с использованием Stream API.

Индивидуальное задание должно быть оформлено в отдельном проекте.

Для проверки работоспособности выполненного индивидуального задания следует использовать отдельный класс с методом main.

### **2.3. Варианты индивидуального задания**

1) Сортировка по имени, фильтрация по дате рождения большей, чем 24 июня 2000, сортировка по фамилии, нахождение суммы всех возрастов.

2) Сортировка по дате рождения, фильтрация по возрасту меньше, чем 50, сортировка по весу, конкатенация всех имен в одну большую строку через пробел.

3) Сортировка по весу в обратном порядке, фильтрация по фамилии не Иванов, сортировка по возрасту, произведение всех возрастов.

4) Сортировка по второй букве имени, фильтрация по весу кратно 10, сортировка по произведению веса на возраст, произведение всех весов.

5) Сортировка по возрасту в обратном порядке, фильтрация по имени «начинается с А», сортировка по дате рождения, расчет среднего веса.

6) Уменьшение веса каждого объекта на 5, фильтрация по дате рождения меньшей, чем 3 февраля 1999, конкатенация фамилий в строку через пробел.

7) Выбор первых 5 элементов списка, сортировка по дате рождения от старых к новым, фильтрация по весу меньше, чем 60, вывод имени и фамилии через пробел.

8) Фильтрация по возрасту больше чем 20, сортировка по последней букве имени, увеличение возраста каждого на 3, вычисление среднего возраста всех элементов.

9) Фильтрация по признаку «вес больше, чем возраст», сортировка по фамилии в обратном порядке, сумма всех весов.

10) Сортировка по второй букве имени в обратном порядке, фильтрация по весу больше, чем 60, сортировка по возрасту, произведение всех возрастов

11) Сортировка по имени в обратном порядке, фильтрация по возрасту больше, чем 20, выбор первых 3 элементов списка, конкатенация имен в строку через пробел.

12) Сортировка по последней букве фамилии, фильтрация по признаку «возраст больше, чем вес», сортировка по дате рождения, произведение всех возрастов.

13) Сортировка по возрасту, фильтрация по возрасту меньше, чем 20, фильтрация по имени «содержит 'е'», конкатенация первых букв имен.

14) Сортировка по сумме веса и возраста, фильтрация по весу кратно 5, выбор первых четырёх элементов, конкатенация имён через пробел.

15) Увеличение веса каждого объекта на 3, сортировка по весу в обратном порядке, фильтрация по дате рождения меньшей, чем 01.01.2000, сумма всех весов.

### 3. ПРАКТИЧЕСКАЯ РАБОТА № 3

#### Цель работы

Знакомство с конкурентным программированием в Java. Потокобезопасность, ключевое слово `synchronized`, мьютексы, семафоры, мониторы, барьеры.

#### 3.1. Теоретическая часть

В Java для реализации многопоточности используются нативные потоки (но в скором времени может появиться Project Loom, который предоставит возможность использовать «зеленые» потоки в Java). Для работы с потоком используется класс `Thread`. Но часто может потребоваться, чтобы разные потоки обращались к одним и тем же данным, и это может привести к отсутствию консистентности данных, так как фактически ни одна команда не является идеально атомарной. Даже инкремент целочисленной переменной внутри выполняется не как одна команда, и, если несколько потоков будут инкрементировать одну переменную, будут возникать странные результаты.

```
static volatile int buf;
static void increment() {
    buf++;
}
public static void main(String[] args) throws Exception {
    buf = 0;
    Thread one = new Thread(()->{
        for (int i = 0; i < 5000; i++) {
            increment();
        }
    });
    Thread two = new Thread(()->{
        for (int i = 0; i < 5000; i++) {
            increment();
        }
    });
}
```

```
one.start();  
two.start();  
Thread.sleep(3000);  
System.out.println(buf);  
}
```

Каждый раз будет выводиться разное значение переменной `buf`, и причина будет не в том, что `Thread` не успевает доработать (попробуйте увеличить значение `sleep`). Проблема будет в том, что инкремент не атомарный, и при изменении непосредственно значения в памяти оно уже могло измениться, и эти изменения просто пропадут. Поэтому требуется как-то добиться атомарности метода `increment`.

### **Потокобезопасность**

Для потокобезопасности существует такое понятие, как мьютекс.

*Мьютекс* – специальный объект для синхронизации потоков. У каждого объекта и класса существует мьютекс. Управлять мьютексом напрямую невозможно, им полностью управляет Java Virtual Machine (JVM).

*Монитор* – надстройка над мьютексом, позволяющая обеспечить синхронизацию. Для работы с монитором используется несколько технологий.

### **Ключевое слово `synchronized`**

При использовании слова *synchronized* происходит захват монитором определенного объекта. Попробуем добавить ключевое слово `synchronized`:

```
synchronized static void increment() {  
    buf++;  
}
```

И при запуске программы она возвращает нужное нам число – 10000. Что же произошло? При входе в метод `increment` активируется блок на класс, поэтому другой поток не может войти в данный метод и ждет завершения предыдущего.



Слово `synchronized` можно применять к определенному методу (тогда блокировка производится на класс или объект, чей метод вызывается) или на определенный объект (например, `synchronized(this)`).

### **Имплементация класса `Lock`**

Добавим новое статическое поле `Lock`:

```
private static final Lock lock = new ReentrantLock();
```

И изменим код метода `increment`:

```
lock.lock();
```

```
buf++;
```

```
lock.unlock();
```

При запуске программы она покажет 10000. Блокировка работает.

Использование `Lock` может понадобиться для более точечной блокировки.

Также можно использовать отдельно `ReadLock` и `WriteLock` для того, чтобы не блокировать `Thread` чтения, если нет записи.

### **Использование `Semaphore`**

*Semaphore* – класс, который принимает количество возможных разрешений. Когда количество разрешений заканчивается, следующий `Thread`, пытающийся его получить, блокируется.

```
private static final Semaphore semaphore = new Semaphore(1);  
static void increment() {  
    try {  
        semaphore.acquire();  
        buf++;  
        semaphore.release();  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
}
```

А если мы хотим воспользоваться коллекцией? Нам нужно вручную делать ее потокобезопасной? Нет, в Java есть даже 2 типа потокобезопасных коллекций:

- synchronized коллекции (Collections.synchronizedList());
- конкурентные коллекции (ConcurrentHashMap).

Они имеют свои плюсы и минусы, и для каждой ситуации нужно выбирать подходящую коллекцию.

### **3.2. Задание**

Создать свои потокобезопасные имплементации интерфейсов в соответствии с вариантом индивидуального задания.

### **3.3. Варианты индивидуального задания**

- 1) Map с использованием ключевого слова synchronized, List с использованием Semaphore.
- 2) Map с использованием Semaphore, List с использованием Lock.
- 3) Map с использованием Lock, Set с использованием ключевого слова synchronized.
- 4) Set с использованием ключевого слова synchronized, Map с использованием Lock.
- 5) Set с использованием Semaphore, List с использованием ключевого слова synchronized.
- 6) Set с использованием Lock, Map с использованием Semaphore.
- 7) List с использованием ключевого слова synchronized, Set с использованием Semaphore.
- 8) List с использованием Semaphore, Map с использованием ключевого слова synchronized.
- 9) List с использованием Lock, Map с использованием Semaphore.
- 10) Map с использованием ключевого слова synchronized, Set с использованием Semaphore.
- 11) Set с использованием ключевого слова synchronized, List с использованием Lock.
- 12) Map с использованием Semaphore, Set с использованием ключевого слова synchronized.

13) Set с использованием Semaphore, List с использованием Lock.

14) List с использованием Semaphore, Set с использованием ключевого слова synchronized.

15) Map с использованием Lock, Set с использованием Semaphore.

## 4. ПРАКТИЧЕСКАЯ РАБОТА № 4

### Цель работы

Работа с `ExecutorService`, `CompletableFuture`.

### 4.1. Теоретическая часть

Работать с потоками напрямую является достаточно неудобным занятием. Мы все любим удобные абстракции, и *ExecutorsService* с *CompletableFuture* являются прекрасными абстракциями, которыми можно не бояться пользоваться.

Также для реализации многопоточного программирования часто используется асинхронность в том или ином виде.

*Асинхронность* – возможность выполнения блока программы в неблокирующем виде системного вызова, что позволяет потоку программы продолжить обработку.

Ниже перечислены некоторые реализации асинхронности.

### Использование callback-функций

Когда нужно что-то выполнить асинхронно, дополнительно в параметр передается некая функция – `callback`, которая вызывается при завершении асинхронного блока, что позволяет выполнить некую логику по завершении асинхронного блока. У данной реализации есть один минус, он называется `callback hell` – усложнение читаемости кода. На рисунке 2 приведен пример использования `callback-функций`.

```

1
2  var async = require("async");
3
4  User.find(userId, function(err, user){
5    if (err) return errorHandler(err);
6    User.all({where: {id: {$in: user.friends}}}, function(err, friends) {
7      if (err) return errorHandler(err);
8      async.each(friends, function(friend, done){
9        friend.posts = [];
10       Post.all({where: {userId: {$in: friend.id}}}, function(err, posts) {
11         if (err) return errorHandler(err);
12         async.each(posts, function(post, donePosts){
13           friend.push(post);
14           Comments.all({where: post.id}, function(err, comments) {
15             if (err) donePosts(err);
16             post.comments = comments;
17             donePosts();
18           });
19         }, function(err) {
20           if (err) return errorHandler(err);
21           done();
22         });
23       });
24     }, function(err) {
25       if (err) return errorHandler(err);
26       render(user, friends);
27     });
28   });
29 });
30

```

Рисунок 2 – Пример использования callback-функций

### Async/await

Async/await – асинхронные функции помечаются как async, что позволяет их выполнить параллельно другой логике. Если требуется получить что-то из асинхронной функции, нужно использовать await, но await можно применять только в асинхронной функции, что не даст выполнить асинхронную тяжелую функцию с блокированием. Минус – малая вариативность, почти ничего невозможно настроить, или, в случае чего, остановить асинхронный код.

### Корутины

Корутины – чаще всего так называют использование облегченных зеленых потоков, которые не являются нативным потоком, а стек вызова хранят в памяти вместо стека. Соответственно не происходит переключения контекста, но при этом корутины могут быть чрезвычайно вариативны, настраиваемы и читаемы.

## Реактивность

*Реактивность* – в некотором роде полный отказ от блокирующего кода.

Под реактивным программированием фактически понимается целая парадигма, ориентированная на представление всей информации в приложении как потоки данных, а также на распространение изменений. Лучше всего для изучения данной темы посмотреть *Реактивный Манифест* (<https://www.reactivemanifesto.org/>).

## ExecutorService

*ExecutorService* – абстракция, представляющая собой некое множество потоков, которым можно передавать определенные задачи на выполнение. Данные задачи могут быть имплементацией интерфейсов Runnable и Callable. Возвращает Future для каждой задачи. *Future* является интерфейсом и представляет собой некое обещание, что по выполнению вернется некий объект. Также при помощи Future можно проверить, выполнялась ли задача, а также отменить ее.

### Примеры использования ExecutorService:

```
ExecutorService executorService =  
Executors.newSingleThreadExecutor();  
executorService.submit(() -> {  
    try {  
        Thread.sleep(200);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
    System.out.println("We run it");  
});  
executorService.submit(() -> System.out.println("Start"));
```

Сначала выведется «We run it», а затем «Start».

```
ExecutorService executorService =  
Executors.newFixedThreadPool(3);  
executorService.submit(() -> {  
    try {
```

```

        Thread.sleep(200);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("We run it");
});
executorService.submit(() -> System.out.println("Start"));

```

Сначала выведется «Start», а затем «We run it».

## CompletableFuture

CompletableFuture – иной способ для использования асинхронности, предоставляет удобный интерфейс для запуска асинхронных задач, например, `runAsync`. Позволяет легко строить цепочки из задач (Изучите код `CompletableFuture`, и определите, создается ли для каждой задачи отдельный поток). Примеры кода с `CompletableFuture`:

```

CompletableFuture<String> future =
CompletableFuture.supplyAsync(() -> "Hello");

CompletableFuture<String> anotherFuture =
future.thenApply((s) -> {
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    return s + " world";
});

System.out.println(anotherFuture.get(3, TimeUnit.SECONDS));

```

## 4.2. Задание

Реализовать собственную имплементацию `ExecutorService` с единственным параметром конструктора – количеством потоков.

## 5. ПРАКТИЧЕСКАЯ РАБОТА № 5

### Цель работы

Познакомиться с паттернами проектирования, их определением и классификацией. Обзор паттернов GoF. Паттерн Синглтон.

### 5.1. Теоретическая часть

#### Паттерны проектирования

В процессе разработки мы сталкиваемся часто с достаточно похожими проблемами, встречаемся с тяжело поддерживаемым кодом. И хотелось бы иметь какие-то общепринятые способы решения для стандартных проблем. И такими решениями являются *паттерны проектирования*. Следует уточнить несколько моментов:

- 1) паттерн проектирования не решение всех проблем. Он лишь позволяет помочь решить какие-то конкретные случаи;
- 2) можно создавать хороший поддерживаемый код и без паттернов проектирования. Они лишь подспорье к разработке;
- 3) стоит знать паттерны проектирования, так как они используются во многих фреймворках и библиотеках, они облегчают дальнейшее понимание той или иной технологии.

Паттернов огромное количество, но мы остановимся на важнейших – паттернах GoF.

#### Паттерны GoF

Паттерны GoF делятся на 3 вида:

- 1) порождающие;
- 2) структурные;
- 3) поведенческие.

#### Паттерн Синглтон (Singleton)

*Синглтон* – порождающий паттерн проектирования. Он позволяет гарантировать, что будет существовать ровно один объект существующего класса. Этот паттерн используется практически во всех возможных



приложениях, является невероятно полезным. В первую очередь, чтобы реализовать данный паттерн, нужно запретить возможность другому коду вызывать конструктор. Для этого требуется приватный конструктор. Дальнейшая реализация различается у разных способов. Приведем пример двух способов:

1) через метод `getInstance()` с ленивой инициализацией:

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

2) через `enum`:

```
public enum Singleton {  
    INSTANCE;  
    public Singleton getInstance() {  
        return INSTANCE;  
    }  
}
```

## 5.2. Задание

Реализовать паттерн Singleton как минимум 3-мя способами.

## 6. ПРАКТИЧЕСКАЯ РАБОТА № 6

### Цель работы

Знакомство с реализацией порождающих паттернов проектирования.

### 6.1. Теоретическая часть

*Порождающие паттерны* проектирования отвечают за удобное безопасное создание объектов или групп объектов.

*Паттерн «Фабричный метод»* – определяет интерфейс создания объектов, позволяя подклассам менять тип создаваемых объектов (рисунок 3).

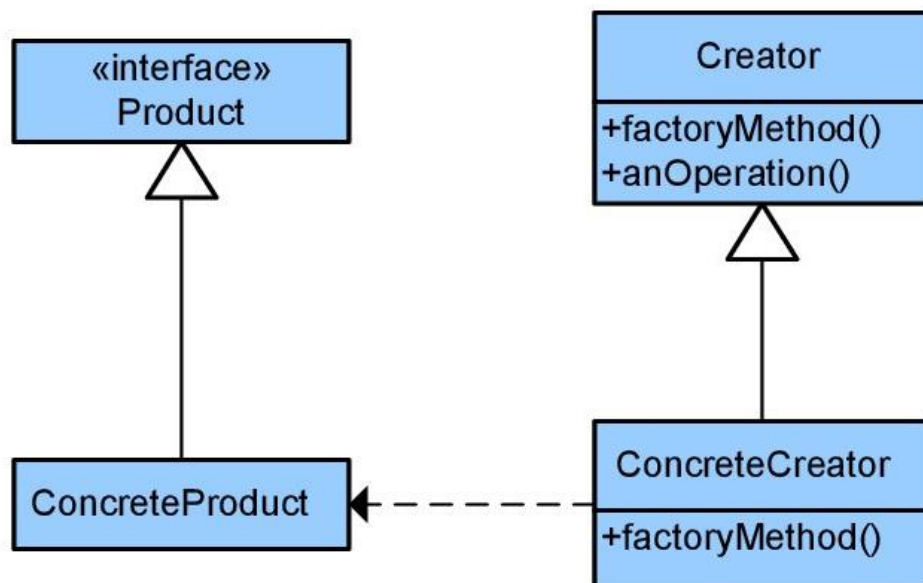


Рисунок 3 – Паттерн «Фабричный метод»

*Паттерн «Абстрактная фабрика»* – позволяет создавать семейства определенных объектов (рисунок 4). Фактически является расширением паттерна «Фабричный метод».

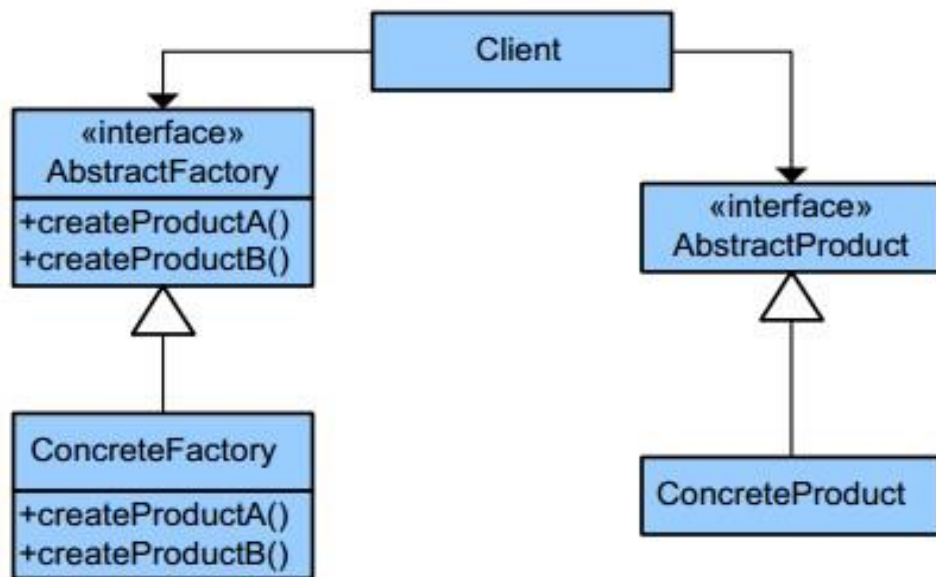


Рисунок 4 – Паттерн «Абстрактная фабрика»

*Паттерн «Строитель»* – разделяет создание объекта на отдельные шаги, а также позволяет использовать один и тот же код создания для получения различных представлений (рисунок 5).

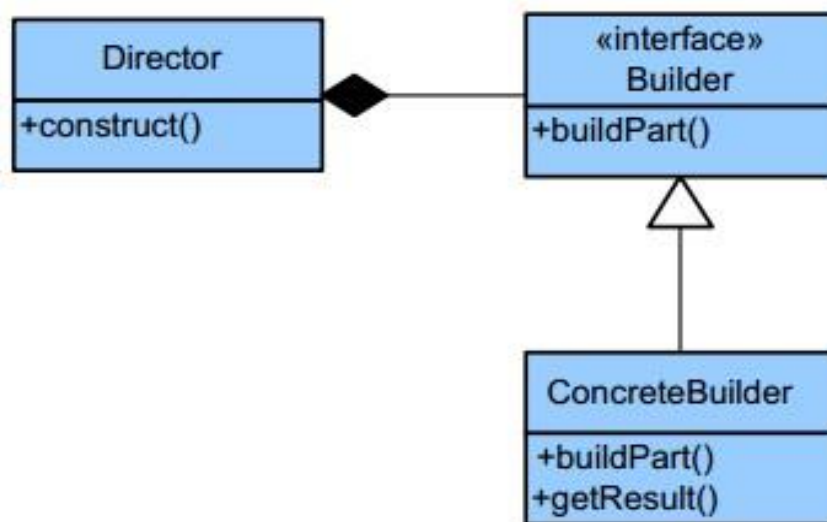


Рисунок 5 – Паттерн «Строитель»

*Паттерн «Прототип»* – позволяет копировать объекты без обращения к приватному состоянию извне (рисунок 6).

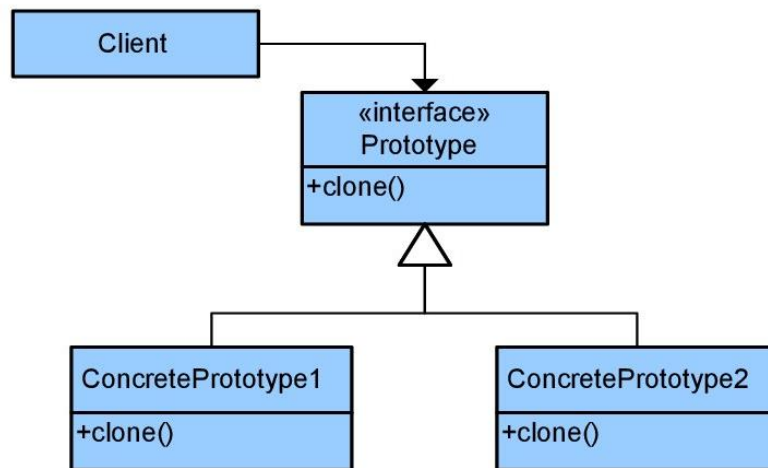


Рисунок 6 – Паттерн «Прототип»

## 6.2. Задание

Написать реализацию паттернов «Фабричный метод», «Абстрактная фабрика», «Строитель», «Прототип».

## 7. ПРАКТИЧЕСКАЯ РАБОТА № 7

### Цель работы

Реализация структурных паттернов проектирования.

### 7.1. Теоретическая часть

Структурные паттерны проектирования играют не менее важную роль, нежели остальные паттерны. Они отвечают за построение удобной структуры и иерархии классов, которая делает код более поддерживаемым. Рассмотрим основные структурные паттерны проектирования.

Паттерн «Адаптер» позволяет какой-то объект с одним интерфейсом подстроить под другой интерфейс (рисунок 7).

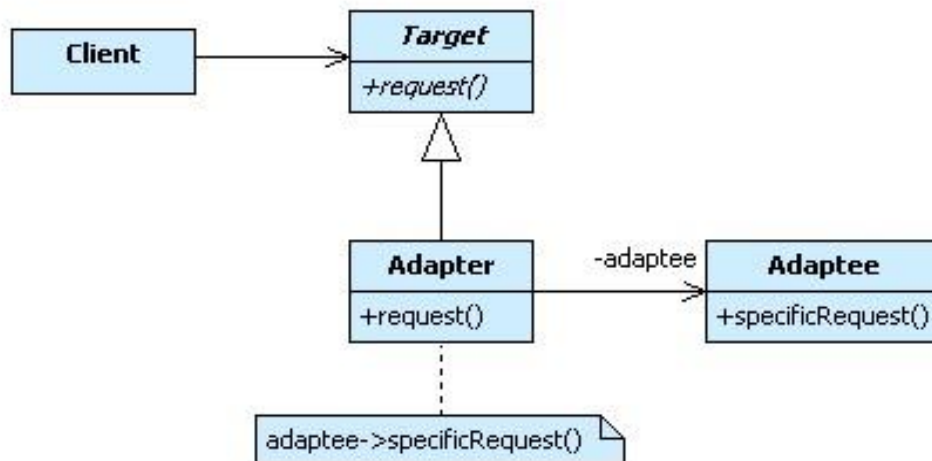


Рисунок 7 – Паттерн «Адаптер»

Паттерн «Мост» разделяет класс на две независимые части – абстракцию и реализацию (рисунок 8).

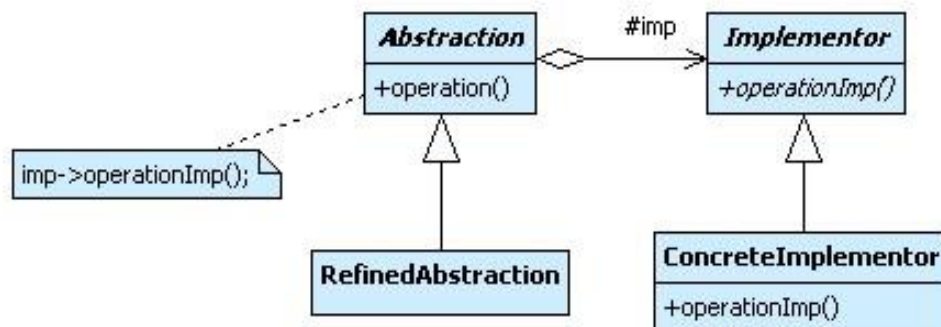


Рисунок 8 – Паттерн «Мост»

Паттерн «Компоновщик» позволяет сгруппировать множество объектов в древовидную структуру (рисунок 9).

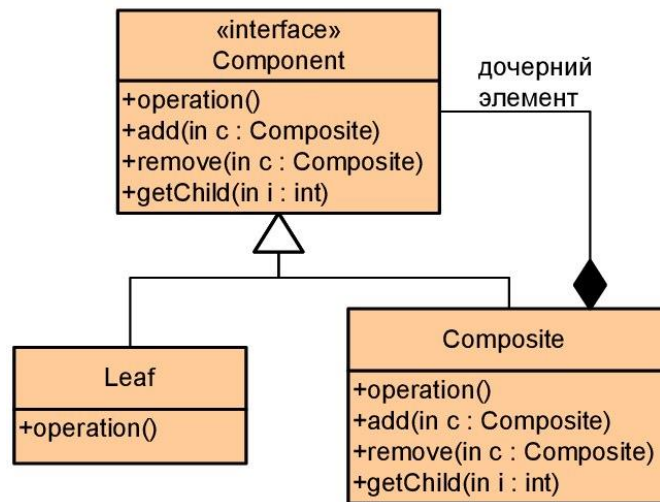


Рисунок 9 – Паттерн «Компоновщик»

Паттерн «Декоратор» позволяет добавлять новую функциональность объекту, является некоторой оберткой над классом (рисунок 10). Не управляет жизненным циклом объекта.

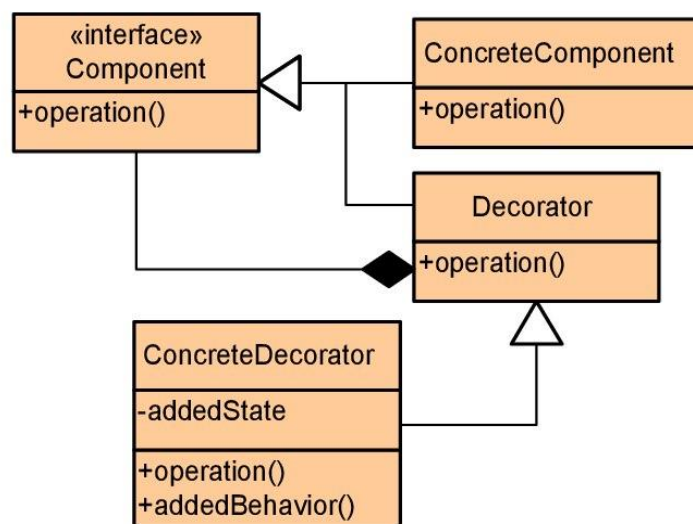


Рисунок 10 – Паттерн «Декоратор»

Паттерн «Фасад» используется для предоставления простой абстракции над некоей сложной системой (рисунок 11).

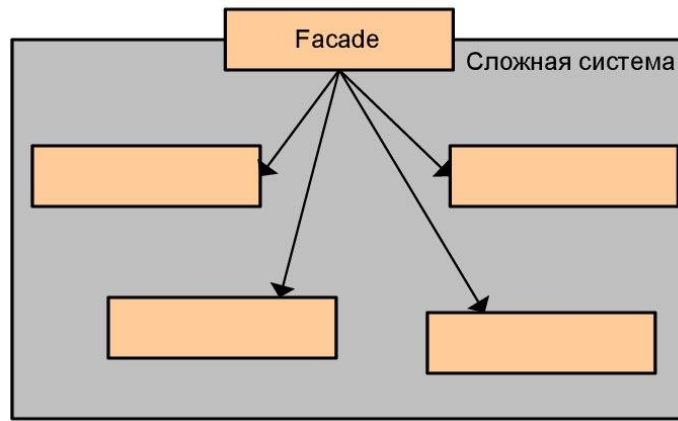


Рисунок 11 – Паттерн «Фасад»

Паттерн «Легковес» используется для экономии памяти, разделяя общее состояние между множеством объектов (рисунок 12). Удобно использовать, когда есть очень много объектов с схожим состоянием.

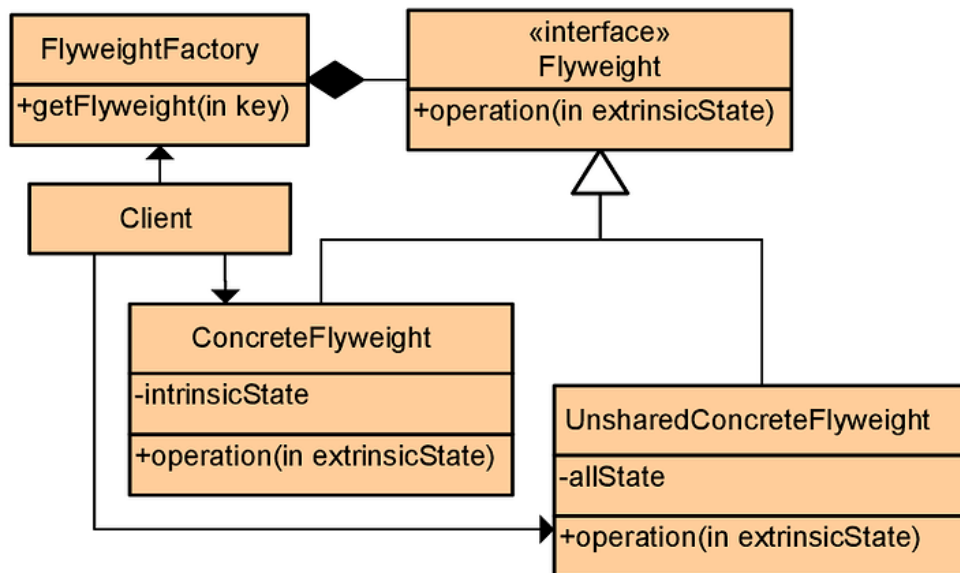


Рисунок 12 – Паттерн «Легковес»

Паттерн «Заместитель» (Прокси) подставляет вместо объектов специальные объекты заместители, добавляя дополнительную логику вокруг вызовов методов (рисунок 13). Может управлять жизненным циклом объекта, который проксирует.

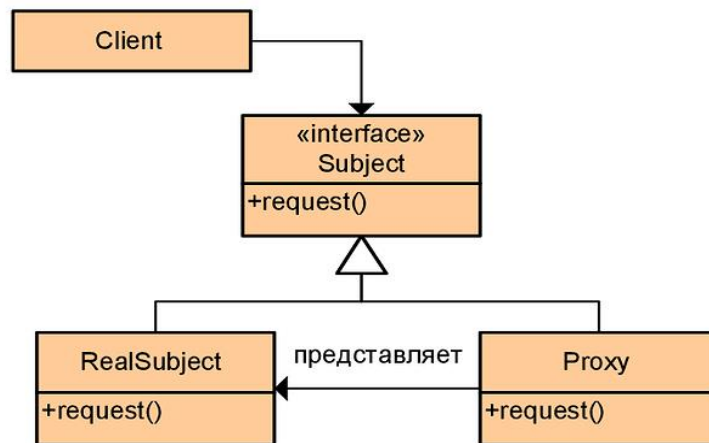


Рисунок 13 – Паттерн «Заместитель»

## 7.2. Задание

Написать реализацию паттерна в соответствии с вариантом индивидуального задания.

## 7.3. Варианты индивидуального задания

- 1) Адаптер, Мост.
- 2) Мост, Компоновщик.
- 3) Компоновщик, Декоратор.
- 4) Декоратор, Фасад.
- 5) Фасад, Легковес.
- 6) Легковес, Заместитель.
- 7) Заместитель, Адаптер.
- 8) Адаптер, Декоратор.
- 9) Декоратор, Легковес.
- 10) Легковес, Компоновщик.
- 11) Компоновщик, Фасад.
- 12) Фасад, Заместитель.
- 13) Заместитель, Компоновщик.
- 14) Компоновщик, Фасад.
- 15) Фасад, Адаптер.



## 8. ПРАКТИЧЕСКАЯ РАБОТА № 8

### Цель работы

Реализация поведенческих паттернов проектирования.

### 8.1. Теоретическая часть

*Поведенческие паттерны* проектирования позволяют расширять поведение системы и взаимодействие различных объектов между собой.

*Паттерн «Цепочка обязанностей»* позволяет передавать запросы по специальной цепочке обработчиков (рисунок 14).

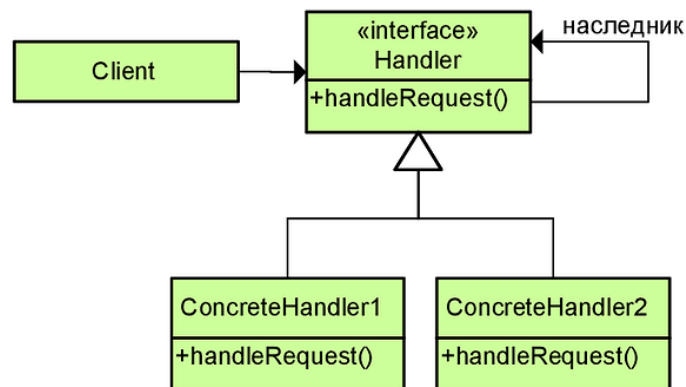


Рисунок 14 – Паттерн «Цепочка обязанностей»

*Паттерн «Команда»* инкапсулирует некий запрос в объект, позволяя передавать их другим объектам для обработки (рисунок 15).

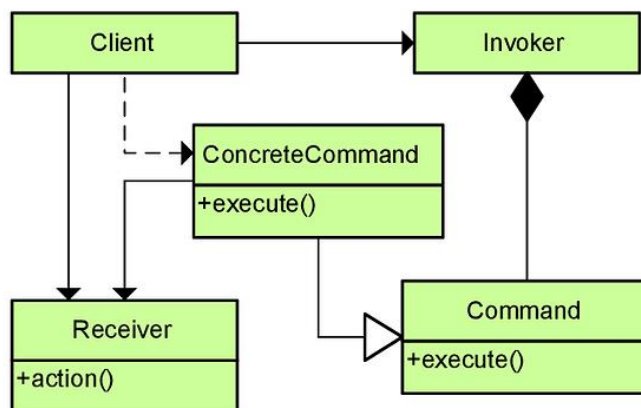


Рисунок 15 – Паттерн «Команда»

Паттерн «Итератор» позволяет обходить множества элементов последовательно (рисунок 16).

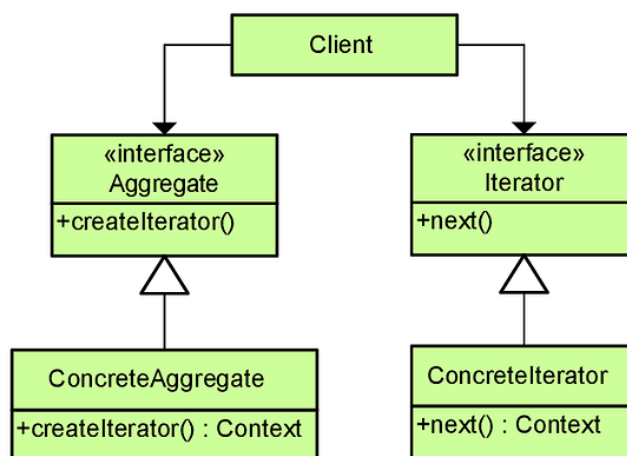


Рисунок 16 – Паттерн «Итератор»

Паттерн «Посредник» перемещает взаимодействие между отдельными объектами в специальный класс-посредник (рисунок 17).

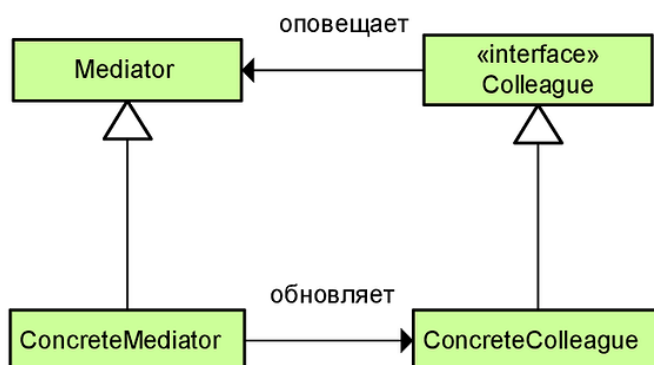


Рисунок 17 – Паттерн «Посредник»

Паттерн «Снимок» позволяет сохранять предыдущие состояние некоторого объекта, не раскрывая его реализации (рисунок 18).

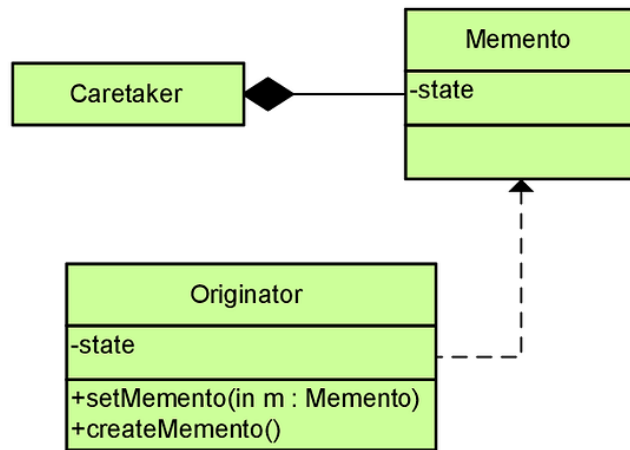


Рисунок 18 – Паттерн «Снимок»

Паттерн «Наблюдатель» используется для создания механизма подписки на события (рисунок 19).

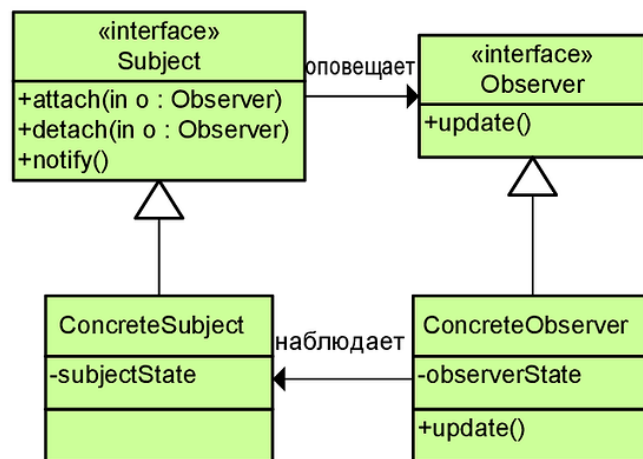


Рисунок 19 – Паттерн «Наблюдатель»

Паттерн «Состояние» позволяет объектам менять свое поведение в зависимости от состояния (рисунок 20).

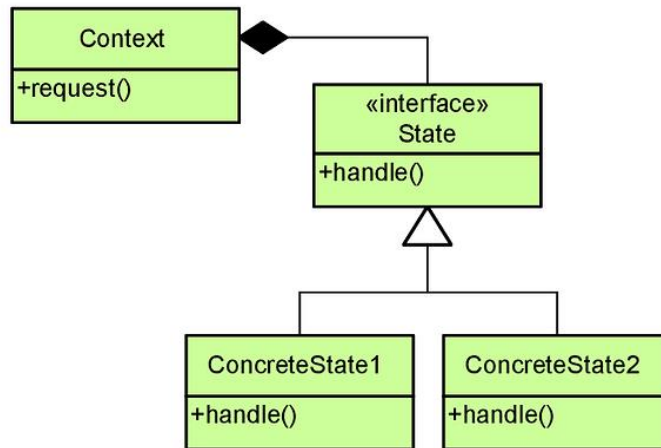


Рисунок 20 – Паттерн «Состояние»

*Паттерн «Стратегия»* позволяет определить семейство различных алгоритмов, которые можно заменять (рисунок 21).

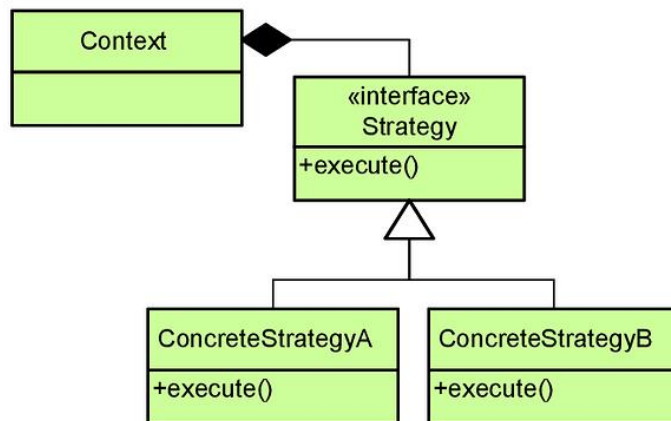


Рисунок 21– Паттерн «Стратегия»

Определяет некоторый алгоритм и позволяет его отдельные шаги делегировать подклассам (рисунок 22).

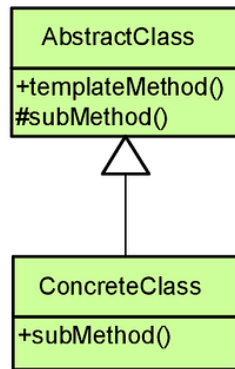


Рисунок 22 – Паттерн «Стратегия»

*Паттерн «Посетитель»* позволяет выполнять одну операцию над группой различных объектов, при этом позволяя создавать новую операцию без изменения классов, над которыми она выполняется (рисунок 23).

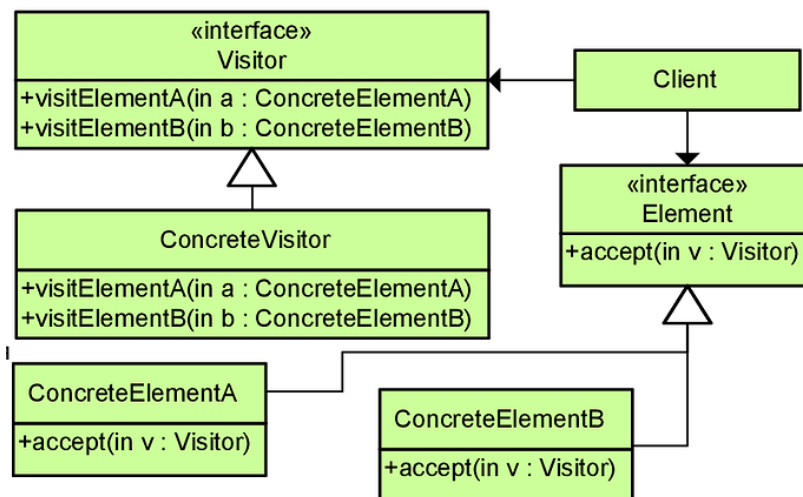


Рисунок 23 – Паттерн «Посетитель»

## 8.2. Задание

Написать реализацию паттерна в соответствии с вариантом индивидуального задания.

## 8.3. Варианты индивидуального задания

- 1) Цепочка обязанностей, Команда.
- 2) Команда, Итератор.

- 3) Итератор, Посредник.
- 4) Посредник, Снимок.
- 5) Снимок, Наблюдатель.
- 6) Наблюдатель, Состояние.
- 7) Состояние, Стратегия.
- 8) Стратегия, Шаблонный метод.
- 9) Шаблонный метод, Посетитель.
- 10) Посетитель, Команда.
- 11) Команда, Стратегия.
- 12) Посредник, Итератор.
- 13) Состояние, Цепочка обязанностей.
- 14) Шаблонный метод, Посетитель.
- 15) Посетитель, Посредник.

## **9. ПРАКТИЧЕСКАЯ РАБОТА № 9**

### **Цель работы**

Знакомство с системой сборки приложения. Gradle.

### **9.1. Теоретическая часть**

Разработка – сложный процесс, и все монотонные процессы так или иначе автоматизируются для того, чтобы программист думал о бизнес-логике, а не об инфраструктуре приложения. Сборка приложения, работа с зависимостями также автоматизирована, и для этого используются системы сборки приложений. Основные функции данных систем:

- 1) организация механизма сборки приложения из исходного кода;
- 2) управление зависимостями;
- 3) предоставление механизмов для настройки алгоритма сборки приложения.

Для Java наиболее известными системами сборки приложений являются Maven и Gradle.

#### **Maven**

Maven основан на использовании xml файла, в котором описывается вся нужная для сборщика приложения информация: зависимости, плагины для изменения сборки, версия самого приложения и версия Java, дополнительная информация.

#### **Gradle**

Gradle основан на groovy файле build.gradle, в котором также описывается основная информация. На рисунке 24 приведен пример build.gradle.

```

plugins {
    id 'org.springframework.boot' version '2.4.0'
    id 'io.spring.dependency-management' version '1.0.10.RELEASE'
    id 'java'
}

group = 'ru.digitalleague'
version = '0.0.1'
sourceCompatibility = '11'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    implementation 'io.jsonwebtoken:jjwt-api:0.11.2'
    implementation 'io.jsonwebtoken:jjwt-impl:0.11.2'
    implementation 'io.jsonwebtoken:jjwt-jackson:0.11.2'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'io.projectreactor:reactor-test'
    testImplementation 'org.springframework.security:spring-security-test'
}

test {
    useJUnitPlatform()
}

```

Рисунок 24 – Пример build.gradle

Рассмотрим основные части данного файла (рисунки 25-32).

```

plugins {
    id 'org.springframework.boot' version '2.4.0'
    id 'io.spring.dependency-management' version '1.0.10.RELEASE'
    id 'java'
}

```

Рисунок 25 – Пример build.gradle, часть 1



В plugins описаны все плагины – специальные дополнения, которые отвечают как за управление зависимостями, так и изменяют алгоритм сборки приложения, а также добавляют различные Gradle Tasks. Что такое Gradle Tasks будет пояснено далее.

```
group = 'ru.digitalleague'
version = '0.0.1'
sourceCompatibility = '11'

configurations {
    compileOnly {
        extendsFrom annotationProcessor
    }
}
```

Рисунок 26 – Пример build.gradle, часть 2

Далее описываются различные конфигурации: название группы пакетов, версия приложения, версия Java, дополнительные конфигурации.

```
repositories {
    mavenCentral()
}
```

Рисунок 27 – Пример build.gradle, часть 3

В repositories описываются репозитории – хранилища, откуда будут загружаться зависимости.

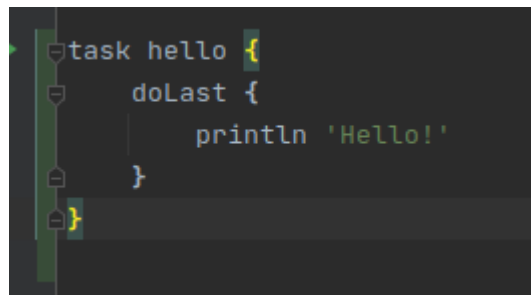
```
dependencies {
    implementation 'org.springframework.boot:spring-boot-starter-security'
    implementation 'org.springframework.boot:spring-boot-starter-webflux'
    implementation 'io.jsonwebtoken:jjwt-api:0.11.2'
    implementation 'io.jsonwebtoken:jjwt-impl:0.11.2'
    implementation 'io.jsonwebtoken:jjwt-jackson:0.11.2'
    compileOnly 'org.projectlombok:lombok'
    annotationProcessor 'org.projectlombok:lombok'
    testImplementation 'org.springframework.boot:spring-boot-starter-test'
    testImplementation 'io.projectreactor:reactor-test'
    testImplementation 'org.springframework.security:spring-security-test'
}
```

Рисунок 28 – Пример build.gradle, часть 4

Далее описываются непосредственно сами зависимости. Есть много типов видимости для зависимостей, вот лишь самые основные:

- `implementation` – зависимость на этапе компиляции, также предоставляется как библиотека для использования на этапе `runtime`;
- `compileOnly` – зависимость только на этапе компиляции;
- `runtimeOnly` – зависимость только на этапе `runtime`;
- `testImplementation` – `implementation` для запуска тестов;
- `annotationProcessor` – использование процессора аннотаций.

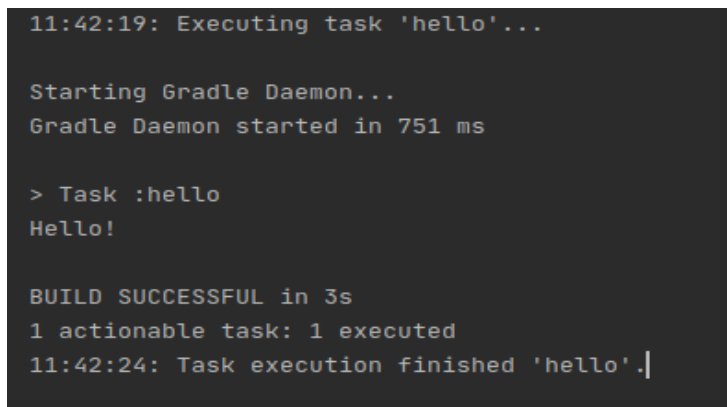
Часто требуется как-то кастомизировать процесс сборки приложения. Для этого в Gradle используются задачи Gradle Tasks. По сути, Gradle task – просто groovy script, который запускается, когда ты его вызываешь. Но удобство в том, что можно строить любые цепочки задач для сборки приложений, а также использовать задачи из плагинов или стандартные задачи Gradle.



```
task hello {  
    doLast {  
        println 'Hello!'  
    }  
}
```

Рисунок 29 – Пример build.gradle, часть 5

Ниже приведен вывод при запуске данной задачи.



```
11:42:19: Executing task 'hello'...  
  
Starting Gradle Daemon...  
Gradle Daemon started in 751 ms  
  
> Task :hello  
Hello!  
  
BUILD SUCCESSFUL in 3s  
1 actionable task: 1 executed  
11:42:24: Task execution finished 'hello'.|
```

Рисунок 30 – Пример build.gradle, часть 6

Далее приведен пример простой цепочки из задач.

```

task hello {
    doLast {
        println 'Hello'
    }
}

task world {
    doLast {
        println ' world!'
    }
}

world.dependsOn hello

```

Рисунок 31 – Пример build.gradle, часть 7

Результат выполнения задачи:

```

11:44:44: Executing task 'world'...

> Task :hello
Hello

> Task :world
 world!

BUILD SUCCESSFUL in 226ms
2 actionable tasks: 2 executed
11:44:45: Task execution finished 'world'.

```

Рисунок 32 – Пример build.gradle, часть 8

Gradle – простой и невероятно удобный инструмент для сборки приложений, и обязательно стоит научиться им пользоваться, познать его силу и удобство.

## 9.2. Задание

Создать приложение, которое выводит какое-то сообщение в консоль. Создать Gradle Task, который создает jar-файл приложения, переносит его в отдельную папку, в которой хранится Dockerfile для jar, а затем создает Docker контейнер из данного jar-файла и запускает его.

## 10. ПРАКТИЧЕСКАЯ РАБОТА № 10

### Цель работы

Введение в Spring. Container. Bean. Внедрение зависимостей, основанных на конструкторах и сеттерах. Конфигурация бинов. Автоматическое обнаружение и связывание классов.

### 10.1. Теоретическая часть

В прошлом разрабатываемое приложение практически не обновлялось. Например, игры на телефоны – они выходили один раз, и никто не создавал различные дополнения, патчи для них. В те времена задумываться о поддержке разрабатываемого приложения не имело смысла, но сейчас поддержка кода имеет чрезвычайно огромное значение. Чем больше кода, тем сложнее его поддерживать, с каждой строкой кода растет технический долг приложения.

*Технический долг приложения* – значение стоимости внедрения какого-то дополнительного функционала в существующий проект. Чем больше, сложнее система, тем сложнее добавлять новые фичи, и может доходить до такого, что добавление простейшей логики станет невыполнимой задачей просто из-за объема текущей логики. Поэтому требуется создавать такой код, который будет приносить минимальное количество технического долга. Нужно писать такой код, в котором отдельные элементы будут максимально независимы для упрощения изменения отдельной части логики. В этом и помогает нам инверсия контроля (IoC).

*Инверсия контроля* – принцип, при котором управление отдельными элементами программы передается отдельному контейнеру. Лучшая фраза, что описывает инверсию контроля, это «Don't call us, we call you». Отдельный элемент программы, в нашем случае объект, не выбирает, какой объект он будет использовать в своей логике – за это отвечает некий контейнер, который и предоставляет нужный объект.

Наиболее известной реализацией инверсии контроля является *инъекция зависимостей* (Dependency Injection (DI)) – шаблон, в котором контейнер производит инъекцию требуемых зависимостей (объектов) в свойства другого объекта, которому требуются некие зависимости. Допустим, у нас есть класс человека «Костя», которому нужны штаны, и есть интерфейс штанов и несколько имплементаций. У «Кости» есть метод walk, в котором он выводит в консоль класс штанов, которые надел. Но мы не хотим, чтобы «Костя» сам выбирал, какие штаны ему носить. Нам нужен некий посредник, который определит, какие штаны носить «Косте», создаст нужный объект и произведет инъекцию объекта штанов «Косте»:

```
class Kostya {
    private Trousers trousers;
    public void walk() {
        System.out.println("I'm wearing " +
trousers.getName());
    }
    public void setTrousers(Trousers trousers) {
        this.trousers = trousers;
    }
}
interface Trousers {
    String getName();
}
class Joggers implements Trousers{
    @Override
    public String getName() {
        return "Joggers!";
    }
}
class Pantaloons implements Trousers {
    @Override
    public String getName() {
        return "Pantaloons....";
    }
}
```

```
}
```

И здесь нам пригодится как раз некий контекст, который и произведет инъекцию зависимостей. Здесь на сцену и выходит Spring, точнее, Spring Core с IoC.

### Bean (бин)

В Spring есть понятие Bean (бин). *Bean* – это объект, зависимостями и жизненным циклом которого управляет Spring. У Spring существует некий контекст – `ApplicationContext`, который содержит множество бинов, которые мы хотим использовать. Он управляет ими и производит инъекцию зависимостей там, где требуется.

Но нам нужно как-то объяснить контексту, какие бины мы хотим создать. Для этого нам нужно написать некую конфигурацию с разными бинами. Далее будет приведен пример с Java конфигурацией, так как на данный момент она является стандартом де-факто практически везде, но также конфигурацию можно писать и на Groovy, и при помощи xml файла, и с использованием Kotlin DSL.

Опишем нашу конфигурацию и решим, что «Костя» будет носить джоггеры:

```
@Configuration
public class BeanConfig {

    @Bean
    public Trousers trousers() {
        return new Joggers();
    }

    @Bean
    public Kostya kostya(Trousers trousers) {
        Kostya kostya = new Kostya();
        kostya.setTrousers(trousers);
        return kostya;
    }
}
```

Аннотация `Configuration` показывает, что данный класс является конфигурацией для бинов. Аннотация `Bean` означает, что то, что возвращает данный метод, является бином.

Также можно определить область видимости (scope) бина. Существует много видов областей видимости, сейчас мы ограничимся лишь двумя (singleton и prototype):

1) singleton – создается только один бин и используется для всех остальных. Да, вам не нужно вручную писать логику для того, чтобы класс был синглтоном, об этом позаботится Spring;

2) prototype – может иметь любое количество экземпляров. Создается каждый раз новый бин.

Определить область видимости можно при помощи аннотации `Scope`. Вот пример:

```
@Bean
@Scope("prototype")
public Trousers trousers() {
    return new Joggers();
}
```

Но по умолчанию создается синглтон, что нам и требуется. Поэтому нам не нужно нигде использовать аннотацию `Scope`.

Теперь попробуем получить бин «Кости» и вызвать его метод `walk`:

```
ApplicationContext context = new
AnnotationConfigApplicationContext(BeansConfig.class);
Kostya kostya = context.getBean(Kostya.class);
kostya.walk();
```

И вывод:

```
I'm wearing Joggers!
```

```
Process finished with exit code 0
```

Контекст увидел, что мы создали джоггеры, и сам произвел инъекцию их в бин. Но каждый раз для каждого бина писать где-то в конфигурации метод неудобно, поэтому у Spring существует такое понятие, как

*ComponentScan*. Context сам пробегается по пакету, ищет классы, помеченные специальными аннотациями, и создает их бины. Изменим код нашего приложения для автоматического сканирования. Для начала уберем бины из конфигурации и добавим аннотацию:

```
@Configuration
@ComponentScan
public class BeanConfig {
}
```

Теперь нам нужно как-то пометить нужные классы, чтоб создались их бины. Для этого можно использовать аннотацию *Component*. Поставим аннотацию над классом «Панталон» и «Костя». Но нам требуется также как-то сообщить, что мы хотим добавить зависимость. Контекст же должен как-то понять, нужно ли в данное свойство производить инъекцию. Для этого используется аннотация *Autowired*. Но где ее ставить: над сеттером, над свойством, над конструктором, может вообще над классом?

Существуют различные способы инъекции, рассмотрим каждый из них:

1) через свойство (field injection) – ставим аннотацию над свойством и производится инъекция. Не особо рекомендуется для использования, хотя вполне можно так делать;

2) через сеттер – аннотация над сеттером; данный сеттер вызывается для инъекции бина.

Пример внедрении зависимости через сеттер:

```
@Component
public class Kostya {
    private Trousers trousers;

    public void walk() {
        System.out.println("I'm wearing " +
            trousers.getName());
    }

    @Autowired
    public void setTrousers(Trousers trousers) {
        this.trousers = trousers;
    }
}
```



```
    }  
}
```

3) через конструктор – наиболее удобный способ инъекции, можно даже не писать аннотацию, достаточно лишь создать конструктор с параметром, который нужно получить.

**Пример через конструктор:**

```
@Component  
public class Kostya {  
    private final Trousers trousers;  
    public Kostya(Trousers trousers) {  
        this.trousers = trousers;  
    }  
    public void walk() {  
        System.out.println("I'm wearing " +  
trousers.getName());  
    }  
}
```

**Также добавим аннотацию для панталон и запустим наш код:**

```
@Component  
public class Pantaloons implements Trousers {  
    @Override  
    public String getName() {  
        return "Pantaloons....";  
    }  
}
```

**Вывод:**

```
I'm wearing Pantaloons...
```

```
Process finished with exit code 0
```

Для того, чтобы лучше понять Spring, рекомендуется сайт [baeldung.com](http://baeldung.com), а также официальная документация Spring.

## 10.2. Задание

Создать приложение, в котором создается `ApplicationContext` и из него берётся бин с названием, переданным в качестве аргумента к приложению, и вызывается метод интерфейса, который он имплементирует. Нужно создать по одному бину для каждого класса, определить им название. Проверить, что вызывается при вводе названия каждого из бинов. Классы и интерфейс определяются в соответствии с вариантом индивидуального задания.

## 10.3. Варианты индивидуального задания

1) Интерфейс `Knight` с методом `void fight()`, его имплементации: `StrongKnight`, `WeakKnight`, `KingOfKnights`.

2) Интерфейс `Magican` с методом `doMagic()`, его имплементации: `Voldemort`, `HarryPotter`, `RonWeesly`.

3) Интерфейс `Programmer` с методом `doCoding()`, его имплементации: `Junior`, `Middle`, `Senior`.

4) Интерфейс `Lighter` с методом `doLight()`, его имплементации: `Lamp`, `Flashlight`, `Firefly`.

5) Интерфейс `Musician` с методом `doCoding()`, его имплементации: `Drummer`, `guitarist`, `trombonist`.

6) Интерфейс `Fighter` с методом `doFight()`, его имплементации: `StreetFighter`, `Boxer`, `Judoka`.

7) Интерфейс `Politician` с методом `doPolitic()`, его имплементации: `Trump`, `Biden`, `Merkel`.

8) Интерфейс `SortingAlgorithm` с методом `doSort()`, его имплементации: `MergeSort`, `InsertionSort`, `QuickSort`.

9) Интерфейс `Printer` с методом `doPrint()`, его имплементации: `ConsolePrinter`, `FilePrinter`.

10) Интерфейс `Programmer` с методом `doCoding()`, его имплементации: `Junior`, `Middle`, `Senior`.

11) Интерфейс Programmer с методом doCoding(), его имплементации:  
Junior, Middle, Senior.

12) Интерфейс Programmer с методом doCoding(), его имплементации:  
Junior, Middle, Senior.

13) Интерфейс Programmer с методом doCoding(), его имплементации:  
Junior, Middle, Senior.

14) Интерфейс Programmer с методом doCoding(), его имплементации:  
Junior, Middle, Senior.

15) Интерфейс Programmer с методом doCoding(), его имплементации:  
Junior, Middle, Senior.

## 11. ПРАКТИЧЕСКАЯ РАБОТА № 11

### Цель работы

Разобраться с использованием Spring boot.

### 11.1. Теоретическая часть

У нас есть прекрасные системы сборки приложений, но все равно возникают проблемы с версиями зависимостей. Нужно постоянно следить за версиями, часто возникают проблемы с тем, что одна библиотека зависит от другой с устаревшей версией, а вы используете более новую. Также очень неудобно в Spring постоянно писать один и тот же код, чтобы просто запустить программу с контекстом. Хотелось бы все это автоматизировать. И хотелось бы сделать такие библиотеки, чтобы можно было добавить зависимость на одну интеграцию от Spring, и сразу же можно было ее использовать, так еще и чтобы была конфигурация по умолчанию. И хочется еще, чтобы по одному файлу с настройками поднимались все нужные бины, и можно было сразу писать бизнес-логику. И самое удивительное – такая технология есть. Это *Spring Boot*. Она позволяет просто скачать проект, добавить нужные зависимости, и сразу писать код для бизнес-логики.

### 11.2. Задание

Создать приложение с использованием Spring Boot Starter Initializr (<https://start.spring.io/>) с такими зависимостями:

- Spring Web;
- Lombok;
- Validation;
- Spring boot Actuator.

Запустить приложение и удостовериться, что не появилось никаких ошибок. Добавить все эндпоинты в Actuator, сделать HTTP-запрос на проверку состояния приложения. Собрать jar-файл приложения, запустить и проверить состояние при помощи REST-запроса.

## 12. ПРАКТИЧЕСКАЯ РАБОТА № 12

### Цель работы

Работа с жизненным циклом компонентов. Аннотации `PostConstruct`, `PreDestroy`.

### 12.1. Теоретическая часть

Для использования аннотаций `PostConstruct`, `PreDestroy` требуется Spring boot. Для того, чтобы создать консольное приложение в Spring boot, воспользуйтесь `CommandLineRunner`.

Контекст также контролирует жизненный цикл бина (рисунок 33).

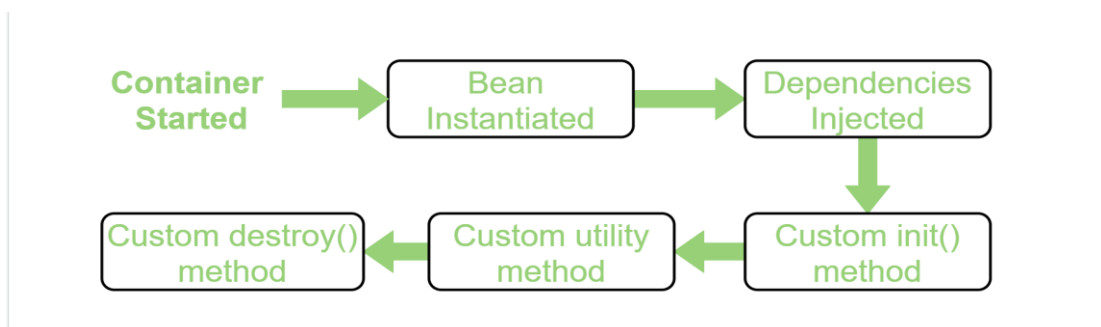


Рисунок 33 – Жизненный цикл бина

Иногда может понадобиться произвести какие-то действия не в конструкторе, так как при вызове конструктора могут не быть инициализированы все бины (в случае если у нас есть `field injection` или `setter injection`), а когда бин уже полностью подготовлен. Для этого и используется аннотация `PostConstruct`.

Для логики при уничтожении бина используется аннотация `PreDestroy`.

Пример кода с аннотациями:

```
@Component
public class PostConstructSample {
    @PostConstruct
    public void init() {
        System.out.println("Bean is ready");
    }
}
```

## **12.2. Задание**

Создать приложение, которое при запуске берет данные из одного файла, хеширует, а при остановке приложения удаляет исходный файл, оставляя только файл с захешированными данными. Названия первого и второго файла передаются в качестве аргументов при запуске. При отсутствии первого файла создает второй файл и записывает в него строку `null`. Реализовать с использованием аннотаций `PostConstruct`, `PreDestroy`.

## 13. ПРАКТИЧЕСКАЯ РАБОТА № 13

### Цель работы

Конфигурирование приложения. Environment.

### 13.1. Теоретическая часть

У приложения может быть достаточно много настроек. Самым банальным примером является строка подключения к базе данных. Не хочется каждый раз лезть в код, когда запускаешь приложение в другой среде – локальной, тестовой, продакшн. И Spring предоставляет невероятно удобный способ конфигурирования приложений. Вся конфигурация находится в одном файле в папке resources – по умолчанию это application.properties, но часто используется application.yml, так как yml формат более читаем. В нем мы можем определять какие-то свойства, которые можем получить в дальнейшем в приложении. Например, добавим в application.yml имя пользователя:

```
program:
  user:
    name: User
```

А теперь попробуем получить эту проперти в классе User

```
@Component
public class User {
    @Value("${program.user.name}")
    private String name;

    @PostConstruct
    public void init() {
        System.out.println(name);
    }
}
```

Запускаем программу, и видим, что наше значение из файла вывелось.

### 13.2. Задание

Создать файл `application.yml` в папке `resources`, добавить в него такие свойства:

- `student.name` – имя студента;
- `student.last_name` – фамилия студента;
- `student.group` – название группы студента.

При запуске приложения выведите данные свойства в консоль при помощи интерфейса `Environment` или аннотации `Value`.



## 14. ПРАКТИЧЕСКАЯ РАБОТА № 14

### Цель работы

Знакомство со Spring MVC. Работа с Rest API в Spring.

### 14.1. Теоретическая часть

Как мы знаем, основная задача сервера – принимать запросы, обрабатывать их и возвращать некий ответ. Чаще всего для этого используется протокол HTTP. Но протокол HTTP не описывает принципов по построению запроса, он является лишь стандартом взаимодействия. Требуется какой-то принцип, удобный, понятный и который легко можно отлаживать. Для этого существует REST API, с которым я советую ознакомиться самостоятельно, ибо информации о нем чрезвычайно много. Я лишь акцентирую внимание контроллеры, созданные для работы с REST API, а также Spring MVC (Model-View-Controller).

Входной точкой для любого запроса является контроллер – класс, аннотированный аннотацией `Controller` или `RestController`. Метод, помеченный аннотацией `RequestMapping`, `GetMapping`, `PostMapping`, `PutMapping` является одной из входных точек для HTTP запроса. Попробуем сделать простой контроллер, который будет возвращать строку.

```
@Controller
public class SimpleController {
    @GetMapping("/hello")
    public @ResponseBody String hello() {
        return "Hello";
    }
}
```

Аннотация `ResponseBody` значит, что то, что вернет метод, будет преобразовано в тело ответа. Главное отличие `RestController` от `Controller` в том, что `RestController` является по сути объединением двух аннотаций – `Controller` и `ResponseBody`. Просто, но эффектно и удобно. Тогда возникает вопрос – зачем нужна просто аннотация `Controller`? Мы же почти всегда хотим

какой-то ответ на запрос. Но в некоторых случаях нам нужно вернуть страницу – статический html или же сгенерированный шаблонизатором. Для этого можно использовать просто контроллер, возвращающий строку. Spring попытается найти какой-то документ по названию, совпадающему со строкой, которую вы вернули. По поводу того, что может возвращать метод контроллера, лучше почитать в документации Spring (<https://docs.spring.io/spring-framework/docs/current/reference/html/web.html>).

Допустим, нам нужно получить тело запроса в формате json, и преобразовать в объект, а затем где-то использовать. Эта проблема решается одной аннотацией и одной настройкой.

```
@Controller
public class SimpleController {
    @GetMapping(value = "/hello", consumes =
MediaType.APPLICATION_JSON_VALUE)
    public @ResponseBody String hello(@RequestBody User
user) {
        return "Hello";
    }
}
```

`consumes` определяет, какой вид в теле запроса потребляет метод. `GetMapping` означает, что он принимает GET запрос.

Контроллеры в Spring – чрезвычайно обширная тема, но очень полезная, они сильно упрощают разработку.

## **14.2. Задание**

Создать отдельный репозиторий Git. Создать простой html-документ, который будет содержать вашу фамилию, имя, номер группы, номер варианта. Создать контроллер, который будет возвращать данный статический документ при переходе на url «/home». Выполнить задание в зависимости с вариантом индивидуального задания.

## **14.3. Варианты индивидуального задания**

1) Создать класс Student с полями `firstName`, `lastName`, `middleName`. Создать класс Group с полем `groupName`. Создать классы-контроллеры для

создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

2) Создать класс Worker с полями firstName, lastName, middleName. Создать класс Manufacture с полями name, address. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

3) Создать класс Book с полями name, creationDate. Создать класс Author с полями firstName, lastName, middleName, birthDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

4) Создать класс Departure с полями type, departureDate. Создать класс PostOffice с полями name, cityName. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

5) Создать класс Game с полями name, creationDate. Создать класс GameAuthor с полями nickname, birthDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

6) Создать класс Post с полями text, creationDate. Создать класс User с полями firstName, lastName, middleName, birthDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

7) Создать класс Item с полями name, creationDate, price. Создать класс Order с полями orderDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

8) Создать класс Level с полями complexity, levelName. Создать класс Game с полями name, creationDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

9) Создать класс Phone с полями name, creationYear. Создать класс Manufacture с полями name, address. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

10) Создать класс Student с полями firstName, lastName, middleName. Создать класс University с полями name, creationDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

11) Создать класс Address с полями addressText, zipCode. Создать класс Building с полями creationDate, type. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

12) Создать класс Card с полями cardNumber, code. Создать класс Bank с полями name, address. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

13) Создать класс Product с полями name, price. Создать класс Market с полями name, address. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

14) Создать класс Patient с полями firstName, lastName. Создать класс Patient с полями firstName, lastName, position. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

15) Создать класс Footballer с полями firstName, lastName. Создать класс Team с полями name, creationDate. Создать классы-контроллеры для создания, удаления объектов и получения всех объектов каждого типа. Сами объекты хранить в памяти.

## 15. ПРАКТИЧЕСКАЯ РАБОТА № 15

### Цель работы

Использование Hibernate в Spring framework.

### 15.1. Теоретическая часть

Зачастую для очень сложной системы требуется какой-то удобный инструмент для взаимодействия с базой данных. Нужно представить таблицы в виде чего-то удобного, с чем легко взаимодействовать. И это, конечно же, объекты. Если таблицу представить в виде класса, а строки в виде некоторых объектов, возникает очень удобная абстракция. И данная абстракция называется Object Relational Mapping (ORM). Существует огромное количество различных ORM, но наиболее известной является Hibernate.

Функции, предоставляемые ORM:

- удобное преобразование данных таблицы в объекты, что существенно упрощает взаимодействие с базой данных (БД);
- управление своими объектами, отслеживание изменений в БД и обновление их. Также при обновлении объектов непосредственно внесение изменений в БД;
- возможность производить lazy loading зависящих объектов;
- возможность переводить JOIN в объект (в некоторых случаях).

Разберем работу с Hibernate сразу на примерах. В качестве базы данных будет использоваться PostgreSQL, но вы можете использовать любую реляционную БД.

### Пример использования Hibernate

Допустим, у нас есть таблица user:

```
create table users (  
  id int,  
  first_name varchar(100),  
  last_name varchar(100)  
);
```

Создадим класс, который будет представлять пользователя:

```
@Entity
@Table(name = "users")
@Getter
@Setter
public class User {
    @Id
    private Long id;
    @Column(name = "first_name")
    private String firstName;
    @Column(name = "last_name")
    private String lastName;
}
```

Теперь нам остается настроить конфигурацию (пока что не будем использовать автоконфигурацию boot, чтобы понять, что вообще требуется Hibernate для старта).

Не забудьте добавить зависимость драйвера вашей базы данных, а также добавьте (если нет) HikariCP. Добавьте данные бины в конфигурации:

```
@Bean
public HikariDataSource dataSource() {
    HikariConfig config = new HikariConfig();
    config.setJdbcUrl("your_url");
    config.setUsername("your_username");
    config.setPassword("your_password");
    return new HikariDataSource(config);
}

@Bean
public LocalSessionFactoryBean factoryBean(DataSource
dataSource) {
    LocalSessionFactoryBean sessionFactoryBean = new
LocalSessionFactoryBean();
    sessionFactoryBean.setDataSource(dataSource);
}
```

```

sessionFactoryBean.setPackagesToScan("your_packages_to_scan");
        Properties properties = new Properties();
        properties.setProperty("hibernate.dialect",
            "your_dialect");
        sessionFactoryBean.setHibernateProperties(properties);
        return sessionFactoryBean;
    }

    @Bean
    public PlatformTransactionManager
platformTransactionManager(LocalSessionFactoryBean sessionFactoryBean){
        HibernateTransactionManager transactionManager = new
        HibernateTransactionManager();
        transactionManager.setSessionFactory(factoryBean.getObject());
        return transactionManager;
    }

```

А теперь попробуем воспользоваться тем, что сделали. Создадим класс, добавим SessionFactory, создадим объект класса Session – он является основным интерфейсом по взаимодействию с БД – нечто, похожее на connection, только адаптер к нему. Для создания запроса воспользуемся языком HQL – специальный язык запросов от Hibernate, очень похожий на SQL.

```

@Component
@RequiredArgsConstructor
public class UserService {
    private final SessionFactory sessionFactory;
    private Session session;

    @PostConstruct
    void init() {
        session = sessionFactory.openSession();
    }

    public List<User> getUsers() {

```

```

        return session.createQuery("select u from User u",
            User.class).getResultList();
    }
}

```

Затем создадим контроллер и попробуем вызвать метод. Он должен вернуть нам пустой массив. Теперь добавим в таблицу пару строк, и опять вызовем данный эндпоинт. Теперь наш массив заполнен значениями.

Для того, чтобы мы могли сохранять программно какие-то объекты или сохранять изменения уже существующих строк, используется метод `saveOrUpdate`.

```

User user = new User();
user.setFirstName("Vasya");
user.setLastName("Dima");
session.saveOrUpdate(user);

```

Только не забудьте добавить генерацию первичного ключа. Воспользуемся способом генерации первичного ключа при помощи `sequence`.

Создаем `sequence`:

```

create sequence users_sequence start 1 increment 1;

```

Добавляем нужные аннотации:

```

@Entity
@Table(name = "users")
@Getter
@Setter
public class User {
    @Id
    @SequenceGenerator(name = "users_seq", sequenceName =
"users_sequence", allocationSize = 1)
    @GeneratedValue(generator = "users_seq", strategy =
GenerationType.SEQUENCE)
    private Long id;
    @Column(name = "first_name")
    private String firstName;
    @Column(name = "last_name")
    private String lastName;
}

```



```
}
```

Готово – теперь при сохранении пользователя при отсутствии у него id он будет автоматически генерироваться. Но, скорее всего, если вы попытаетесь сохранить пользователя, у вас это не получится. Проблема в транзакциях – Hibernate требует для всего текущей транзакции, поэтому добавьте в свой код создание транзакции и коммит:

```
var transaction = session.beginTransaction();  
session.saveOrUpdate(user);  
transaction.commit();
```

О транзакциях подробнее мы поговорим в следующих заданиях.

У объекта, подконтрольного Hibernate, существует свой жизненный цикл (рисунок 34).

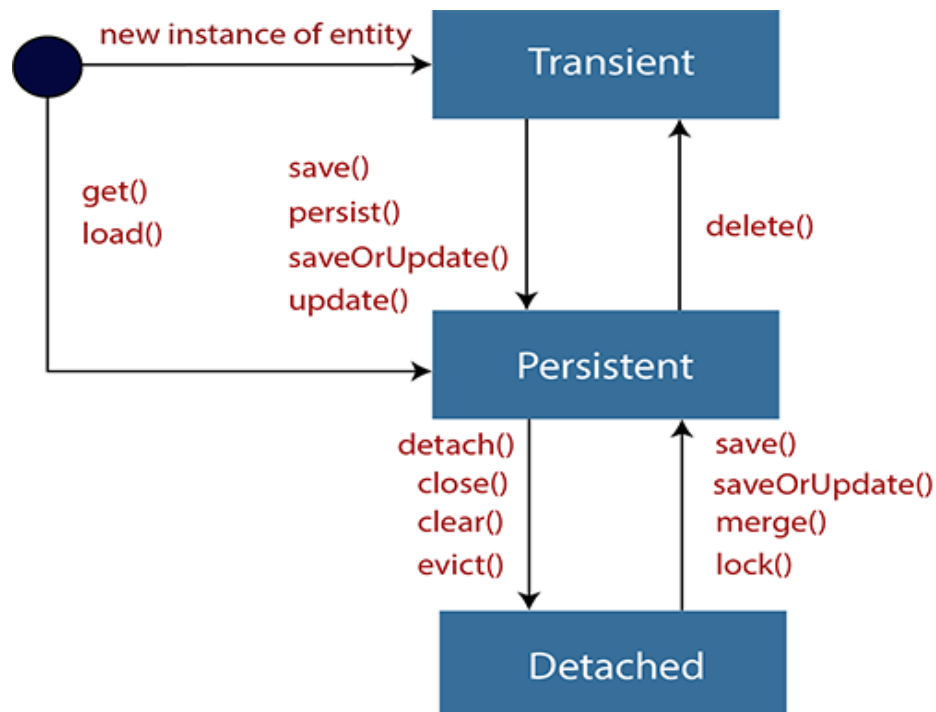


Рисунок 34 – Жизненный цикл объекта, подконтрольного Hibernate

Мы не будем особо углубляться в особенности жизненного цикла, главное, что нужно запомнить, – это в состоянии Persistent все изменения на объекте будут сохраняться в БД, а также происходит dirty checking – проверка, соответствует ли объект строке в БД, не происходили ли изменения на БД. Но даже учитывая, что фактически Hibernate сам видит, что мы что-то поменяли,

и сохраняет изменения, рекомендуется использовать метод `saveOrUpdate` на Persistent объекты для читаемости и понятности кода.

Для лучшего изучения данной темы рекомендуется ознакомиться с официальной документацией Hibernate ([https://docs.jboss.org/hibernate/stable/search/reference/en-US/html\\_single/](https://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/)).

## **15.2. Задание**

Изменить программу с предыдущего задания так, чтобы объекты хранились в базе данных PostgreSQL вместо памяти компьютера.

## 16. ПРАКТИЧЕСКАЯ РАБОТА № 16

### Цель работы

Изучение видов связей между сущностями в Hibernate. Использование транзакций.

### 16.1. Теоретическая часть

Hibernate также предоставляет множество удобного функционала для работы со связями между таблицами. Немного погрузимся в данную тему на практическом примере. Допустим, у нас есть таблица пользователей «users» (возьмем из предыдущего задания) и собаки («dogs»), которые принадлежат пользователям:

```
create table dogs (  
    id int,  
    user_id int,  
    name varchar(100),  
    breed varchar(100)  
);  
  
create sequence dogs_sequence start 1 increment 1;
```

Создадим наш класс собаки, и сразу сделаем так, чтобы мы могли подтянуть пользователя, чья это собака.

```
@Table(name = "dogs")  
@Entity  
@Getter  
@Setter  
public class Dog {  
    @Id  
    private Long id;  
    private String name;  
    private String breed;  
    @ManyToOne  
    public User user;  
}
```

Все, теперь добавим в таблицу парочку собачек (не забудьте установить в значение `user_id` существующего пользователя из таблицы `users`), и попробуем их получить и вызвать метод `getUser()`. Например, вот так:

```
@Service
@RequiredArgsConstructor
public class DogService {
    private final SessionFactory sessionFactory;
    private Session session;
    @PostConstruct
    public void init() {
        session = sessionFactory.openSession();
    }
    public User getUserByDog(Long dogId) {
        return session.createQuery("from Dog where id = :id", Dog.class)
            .setParameter("id", dogId).getSingleResult().getUser();
    }
}
```

И метод в контроллере:

```
@GetMapping(value = "/dog/{dogId}/user")
public @ResponseBody User getDogUser(@PathVariable("dogId")
Long dogId){
    return dogService.getUserByDog(dogId);
}
```

А теперь хотелось бы, чтобы и у пользователя можно было бы сразу получить всех собачек. Подтягиваются они отдельным запросом при вызове непосредственно геттера. Такое поведение «ленивого» подтягивания данных называется *lazy loading*:

```
@Entity
@Table(name = "users")
@Getter
@Setter
public class User {
    @Id
```



```
[
  {
    "id": 5,
    "firstName": "Ivan",
    "lastName": "Shuga",
    "dogs": [
      {
        "id": -1,
        "name": "vasya",
        "breed": "corgi"
      }
    ]
  }
]
```

Вот, теперь все правильно и красиво. Но если все же нам нужно для собаки выводить пользователя? Тогда стоит воспользоваться паттерном Data Transfer Object (DTO) – мы используем отдельный простой класс, в который превращаем наш класс сущности, и возвращаем уже класс DTO.

## 16.2. Задание

Создать связь Один-ко-многим между сущностями из предыдущего задания и проверить работу lazy loading.

## 17. ПРАКТИЧЕСКАЯ РАБОТА № 17

### Цель работы

Знакомство с Criteria API в Hibernate.

### 17.1. Теоретическая часть

HQL – достаточно удобен для написания запросов, и покрывает практически все потребности при разработке. Но есть одно, с чем HQL не справится – динамическое создание запросов. Допустим, нам нужно в зависимости от запроса пользователя по-разному фильтровать данные для ответа. В этом поможет *Criteria API*.

Для полного создания запроса нам потребуется 3 объекта:

- 1) CriteriaBuilder – это, соответственно, сам билдер запроса;
- 2) CriteriaQuery – запрос;
- 3) Root – это основная сущность, для которой делается запрос.

Создадим билдер запроса:

```
CriteriaBuilder builder = session.getCriteriaBuilder();
CriteriaQuery<Dog> dogCriteriaQuery =
builder.createQuery(Dog.class);
Root<Dog> root = dogCriteriaQuery.from(Dog.class);
```

Теперь создадим сам запрос – простую сортировку по породе.

Выглядеть это будет так:

```
dogCriteriaQuery.select(root).orderBy(builder.asc(root.get(
"breed")));
```

А теперь остается получить значения и вернуть как результат метода:

```
Query<Dog> query = session.createQuery(dogCriteriaQuery);
return query.getResultList();
```

Criteria API достаточно громоздкий, но при этом очень подвижный и функциональный способ выполнения запросов. С помощью него можно создавать запросы любой сложности, с JOIN, сортировками, фильтрацией. Поэтому если потребуется динамический построитель запросов, Criteria API – неплохой выбор. Подробнее про Criteria API можно почитать [здесь](#):

<https://docs.jboss.org/hibernate/entitymanager/3.5/reference/en/html/querycriteria.html>.

### **17.2. Задание**

Добавить возможность фильтрации по всем полям всех классов с использованием Criteria API в Hibernate для программы из предыдущего задания. Добавить эндпоинты для каждой фильтрации.



## 18. ПРАКТИЧЕСКАЯ РАБОТА № 18

### Цель работы

Знакомство с репозиториями и сервисами, реализация в проекте. Взаимодействие с Spring Data JPA.

### 18.1. Теоретическая часть

Самой простой, но при этом достаточно эффективной архитектурой является *слоистая архитектура*, при которой приложение делится на отдельные независимые слои. Чаще всего используется 3 слоя:

- 1) слой представлений – в Spring реализуется с использованием контроллеров;
- 2) слой бизнес-логики (сервисов) – реализуется с помощью сервисов;
- 3) слой данных – реализуется с помощью репозиторий.

### Сервисы и репозитории

*Сервис* – специальный слой, в котором хранится вся бизнес-логика. В данном слое должно быть как можно меньше взаимодействия со сторонними библиотеками для большей поддерживаемости. Для данных классов лучше использовать аннотацию *Service*. Она ничем не отличается от *Component*, просто для лучшей выразительности лучше пользоваться ей.

Также наиболее верно использовать интерфейсы на стыке между слоями. Зачем? Это паттерн – инверсия зависимости, для того, чтобы верхние слои не зависели от нижних, и можно было с легкостью подменить одну имплементацию на другую. Поэтому наиболее правильный способ создания репозитория – определение интерфейса, а затем его имплементации с аннотацией *Service*. Пример:

```
public interface UserService {  
    List<User> getUsers();  
    void saveOrUpdate(User user);  
}  
  
@Service  
public class UserServiceImpl implements UserService {
```

```

    public List<User> getUsers() {
        //code
    }

    public void saveOrUpdate(User user){
        //code
    }
}

```

*Репозитории* тоже имеют свою аннотацию – *Repository*. Но также у Spring есть еще одна интереснейшая технология – Spring Data. Рассмотрим данную технологию на Spring Data JPA, так как она используется для реляционных баз данных.

Данная технология основывается как раз-таки на репозиториях, но очень сильно уменьшает количество boilerplate кода. Все, что требуется сделать, это сконфигурировать взаимодействие с базой данных, создать интерфейс репозитория для каждой сущности – и все, можно дальше работать с бизнес-логикой, изредка дополняя методами в интерфейсах для нужного взаимодействия. Для начала возьмем приложение из предыдущего задания, и уберем предыдущую конфигурацию, перенеся полностью конфигурацию в application.yml:

```

spring:
  datasource:
    url: your_url
    username: your_username
    password: your_password

```

И добавим над конфигурацией аннотацию EnableJpaRepositories:

```

@Configuration
@EnableJpaRepositories
public class AppConfig {
}

```

С конфигурацией покончено. Теперь создадим интерфейсы-репозитории, а также добавим возможность поиска по названию породы в Dog:

```

    public interface UserRepository extends JpaRepository<User,
Long> {
        }
    public interface DogRepository extends JpaRepository<Dog,
Long> {
        List<Dog> findAllByBreed(String breed);
    }

```

И все. Spring по названию метода определяет то, каким должен быть запрос. Например, напишем метод по поиску по породе и имени:

```

    public interface DogRepository extends JpaRepository<Dog,
Long> {
        List<Dog> findAllByBreed(String breed);
        List<Dog> findAllByBreedAndName(String breed, String
name);
    }

```

Достаточно просто и удобно. При этом все взаимодействие с БД передается Spring, и не приходится постоянно размышлять о создании сессии, написании HQL даже для простых запросов и т.д. Если все же нужно выполнить более сложный запрос, который не может сделать Spring, или нативный запрос, можно воспользоваться аннотацией Query:

```

    @Query(value = "select dogs.* from dogs join users on
users.id = dogs.user_id where users.first_name = :username",
nativeQuery = true)
    List<Dog> findAllByUserName(String username);

```

Теперь остается лишь воспользоваться Dependency injection (DI) и получить бин данного интерфейса. Spring сам создает имплементацию интерфейса и наполняет нужной логикой. Также JpaRepository имеет стандартные методы, такие как findAll, так что не придется их даже объявлять:

```

@Service
@RequiredArgsConstructor
public class DogService {
    private final DogRepository dogRepository;
    public User getUserByDog(Long dogId) {

```

```

        return dogRepository.findById(dogId).orElseThrow(() ->
            new IllegalStateException("Dog with this id not
            found")).getUser();
    }

    public List<Dog> getAllDogs() {
        return dogRepository.findAll();
    }
}

```

Рекомендуется ознакомиться с документацией Spring для лучшего понимания репозитория (<https://docs.spring.io/spring-data/jpa/docs/2.2.10.RELEASE/reference/html/#reference>).

Spring Data repository – отличная технология для уменьшения boilerplate кода.

## 18.2. Задание

Переписать код предыдущего задания с использованием сервисов и отделения логики контроллера от логики сервиса и репозитория. В программе всё взаимодействие с базой данных должно быть реализовано через репозитории Spring Data Jpa.

## 19. ПРАКТИЧЕСКАЯ РАБОТА № 19

### Цель работы

Знакомство с логированием с использованием Logback в Spring.

### 19.1. Теоретическая часть

*Логирование* – очень важная часть разработки. Часто мы не имеем возможности запустить дебаггер, воспроизвести нужную проблему и посмотреть, что происходит в программе в ней. Тем более, когда проблема не воспроизводится, без логирования мы вообще фактически остаемся ни с чем.

Воспользуемся библиотекой Logback для реализации логирования, а также Slf4j вместе с аннотациями Lombok для уменьшения boilerplate. После добавления всех нужных зависимостей (spring boot starter logging), создадим файл logback.xml, и сделаем логирование всей информации в стандартный поток вывода, а также в файл, который при превышении порога или прохождении 30 дней создает новый файл логов, а старый сжимает при помощи gzip. Но перед заполнением logback.xml, рассмотрим уровни логирования:

- Error – выводятся ошибки, то, что может критически повлиять на работу системы;
- Warn – предупреждения, которые не сломают логику, но могут негативно повлиять;
- Info – стандартный вывод информации;
- Debug – информация для дебага;
- Trace – максимально полная информация, очень редко используется, лучше не делайте логирование в файл с уровнем trace, иначе впоследствии логи будут практически нечитаемы.

А теперь заполним logback.xml:

```
<configuration>
  <appender name="STDOUT"
class="ch.qos.logback.core.ConsoleAppender">
```

```

        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level
%logger{36} - %msg%n</pattern>
        </encoder>
    </appender>
    <appender name="FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>application.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <fileNamePattern>application.%d{yyyy-MM-
dd}.gz</fileNamePattern>
            <maxHistory>30</maxHistory>
            <totalSizeCap>3GB</totalSizeCap>
        </rollingPolicy>
        <encoder>
            <pattern>%-4relative [%thread] %-5level %logger{35}
- %msg%n</pattern>
        </encoder>
    </appender>
    <root level="info">
        <appender-ref ref="STDOUT" />
        <appender-ref ref="FILE" />
    </root>
</configuration>

```

Корневой элемент – *configuration*. В Logback есть 3 основных элемента:

- **Logger** – контекст для логирования сообщений, может иметь несколько аппендеров;
- **Appender** – место, куда кладутся логи, например, файлы;
- **Layout** – то, как должны выглядеть логи.

В данном случае мы создаем 2 аппендера: в стандартный поток вывода, а также в файл с созданием файла каждый день, максимальное количество 30, а также максимальный размер 3 ГБайт. В pattern описывается паттерн, как должен выглядеть лог. Это и есть тот самый Layout.

rollingPolicy как раз-таки определяет обновление файла логов, чтобы он не разрастался, и определяет максимальное количество файлов, максимальный размер файлов.

TimeBasedRollingPolicy обозначает, что изменение основывается на времени.

Теперь можно запустить приложение и посмотреть, создан ли файл, записываются ли логи. Если все нормально, остается добавить собственное логирование. Для этого воспользуемся аннотацией Slf4j, и залогироваем какой-либо сервис:

```
@Service
@RequiredArgsConstructor
@Slf4j
public class UserServiceImpl implements UserService {
    private final UserRepository userRepository;
    public List<User> getUsers() {
        log.info("Find all users");
        return userRepository.findAll();
    }
    public void saveOrUpdate(User user){
        log.info("Save user {}", user);
        userRepository.save(user);
    }
}
```

Теперь можно запустить и проверить, все ли залогировалось, а также открыть файл и проверить, записались ли туда логи.

Логирование не является сложным, но имеет огромное значение при отладке программ.

## 19.2. Задание

Создать файл logback.xml, добавить логирование во все методы классов-сервисов.

## 20. ПРАКТИЧЕСКАЯ РАБОТА № 20

### Цель работы

Использование Spring AOP. Pointcut, JoinPoint. Advice.

### 20.1. Теоретическая часть

Нам может потребоваться часто создать такую логику, которая будет присутствовать сразу во множестве разнообразных методов – например, обернуть все методы взаимодействия к БД в транзакцию, или логировать входные параметры всех методов сервисов. Для реализации такой логики идеально подойдет Aspect Oriented Programming (AOP) – парадигма программирования для реализации сквозной логики. Мы можем одну и ту же логику применять сразу к множеству модулей в программе (рисунок 35).

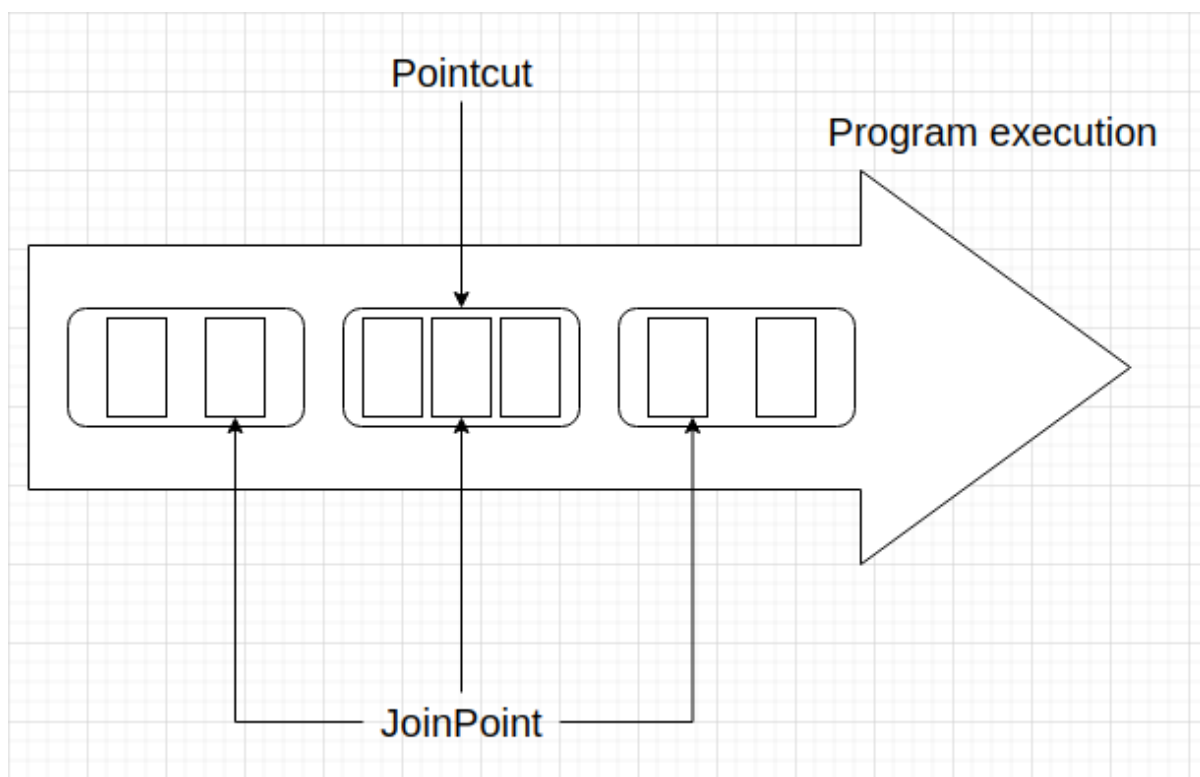


Рисунок 35 – Представление Aspect Oriented Programming (AOP)

В аспектно-ориентированном программировании существует несколько основных понятий:

- Aspect – класс, реализующий сквозную функциональность;



– Advice – блок кода, который должен выполняться в каких-то определенных местах работы программы;

– Pointcut – некий предикат, который описывает ту точку кода, в которой выполняется Advice;

– JoinPoint – конкретная точка выполнения программы, в которой будет исполняться Advice. То есть, Pointcut это множество JoinPoint точек.

А теперь попробуем сделать простой аспект, который будет логировать просто входные параметры всех классов сервисов. Не забудьте добавить аннотацию EnableAspectJAutoProxy над конфигурацией.

```
@Slf4j
@Component
@org.aspectj.lang.annotation.Aspect
public class Aspect {
    @Before("allServiceMethods()")
    public void logParameters(JoinPoint joinPoint) {
        log.info("Parameters: {}", joinPoint.getArgs());
    }
    @Pointcut("within(ru.mirea.springcourse.service.*)")
    public void allServiceMethods() {}
}
```

Аннотация Aspect создает специальный аспект, в котором будут описываться Pointcut, Advice.

Before – аннотация, которая обозначает Advice, который будет выполняться перед выполнением некоего метода.

Есть различные аннотации, например, After, AfterThrowing, Around, AfterReturning.

Pointcut описывает набор точек выполнения программы – методов, в которых выполняется Advice:

- within – объекты заданного типа или классов пакета или подпакетов;
- execution – по имени метода;
- this – прокси реализует заданный тип;
- bean – имеет определённый идентификатор или имя;

– annotation – помечены указанной аннотацией.

Теперь можно запустить приложение, и проверить работоспособность аспектов.

## **20.2. Задание**

Для приложения из предыдущего задания добавить логирование времени выполнения каждого метода сервиса с использованием Spring AOP.

## 21. ПРАКТИЧЕСКАЯ РАБОТА № 21

### Цель работы

Проксирование. Аннотация Transactional. Аннотация Async.

### 21.1. Теоретическая часть

В данной работе мы изучим проксирование, на котором основаны многие функциональные возможности в Spring, например, AOP, планирование заданий, асинхронность.

Логично, что проксирование основано на паттерне Прокси. Создается объект, который подменяет исходный, с дополнительной логикой. При этом проксирование может быть многоуровневое, прокси может проксировать другой прокси, так как он просто внутри использует предыдущий объект.

Есть различные имплементации проксирования: JDK-проксирование, CGLIB. Также существует AspectJ-проксирование на этапе компиляции, но мы его не будем сейчас затрагивать. JDK имплементация основана на том, что создается объект, который имплементирует все интерфейсы класса, который имплементирует целевой объект. При таком типе проксирования можно использовать только public методы.

CGLIB имплементация основана на том, что прокси наследуется от целевого класса, и таким образом может использовать методы. В таком случае можно проксировать все методы, кроме private.

Изначально Spring пытается воспользоваться проксированием на основе интерфейсов, если не получается, то использует CGLIB. Но с версии Spring boot 2 по умолчанию используется CGLIB, что имеет большое значение, так как если мы будем использовать Spring с Kotlin, то нужно не забывать о специальном extension, который все классы делает open, иначе Spring просто не сможет проксировать.

Аннотация Transactional позволяет оборачивать метод в транзакцию, не прописывая дополнительный код. Если на простом уровне, то он просто оборачивает метод в такой код:

```
var transaction = session.beginTransaction();  
    try {  
        invokeMethod();  
        transaction.commit();  
    } catch (Exception e) {  
        transaction.rollback();  
    }  
}
```

Эта аннотация уменьшает код, при этом передавая всю работу с транзакциями на Spring. Также у аннотации есть параметр `readonly`, который стоит ставить в `true`, если не производится никаких изменений в БД, а только чтение. Это убирает различные дополнительные проверки, улучшает производительность метода.

Аннотация `Async` делает метод асинхронным. Он исполняет метод в отдельном треде.

## **21.2. Задание**

Для приложения из предыдущего задания пометить все классы сервисов, в которых происходит взаимодействие с базой данных, как `Transactional`. Добавить отправку информации о сохранении каждого объекта по электронной почте, создав отдельный класс `EmailService` с асинхронными методами отправки сообщений. Для асинхронности методов используйте аннотацию `Async`.

## 22. ПРАКТИЧЕСКАЯ РАБОТА № 22

### Цель работы

Планирование заданий. Scheduler в Spring.

### 22.1. Теоретическая часть

Иногда требуется выполнять какие-то запланированные задания, которые выполняются в определенный момент времени – например, очистку таблицы от устаревших значений, подгрузка новых значений, инвалидация кэша. Для этого используется Scheduler. Не забудьте добавить аннотацию `EnableScheduling` над конфигурацией. Теперь создадим класс `SchedulerService`:

```
@Service
public class SchedulerServiceImpl implements
SchedulerService {
    @Scheduled(cron = "0 * * * * *")
    @Override
    public void doScheduledTask() {
        System.out.println("Scheduled task");
    }
}
```

Для того, чтобы создать запланированное задание, нужна аннотация `Scheduled`. Данный метод затем проксируется и запускается по расписанию. `cron` означает `cron`-значение, описывающее время запуска. В данном случае оно будет выполняться каждую минуту.

### 22.2. Задание

Для приложения из предыдущего задания создать класс-сервис с методом, который будет вызываться каждые 30 минут и очищать определённую директорию, а затем создавать по файлу для каждой из сущностей и загружать туда все данные из базы данных. Также добавить возможность вызывать данный метод с использованием `Java Management Extensions (JMX)`.

## 23. ПРАКТИЧЕСКАЯ РАБОТА № 23

### Цель работы

Использование Spring Security для аутентификации и авторизации пользователей.

### 23.1. Теоретическая часть

Каждое приложение должно так или иначе поддерживать защиту от несанкционированного доступа. Для этого предоставлена прекрасная технология *Spring Security*.

Самым основным элементом Security является *Filter Chain* – цепочка фильтров. Вся безопасность реализуется через данную цепочку фильтров. По очереди каждый фильтр проверяет запрос, получает нужные данные, в случае чего блокирует дальнейшую фильтрацию. Также вместо отдельного фильтра может быть *FilterChainProxy*, который хранит в себе семейство иных фильтров.

Также в фильтре можно создать имплементацию интерфейса *Authentication*, которую можно положить в *SecurityContextHolder*. Дело в том, что для каждого треда хранится отдельный экземпляр *SecurityContextHolder*, в котором хранится аутентификация для пользователя. Благодаря этому мы, например, можем получить *Principal* в контроллере.

Но это не все – есть огромное количество самых разнообразных абстракций в Spring security, однако на данный момент мы затронем самые базовые конфигурации для Security.

```
@Configuration
public class SecurityConfig extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws
Exception {
    }
}
```

Класс, наследующийся от `WebSecurityConfigurerAdapter`, является фактически входной точкой со всей конфигурацией. Для начала запретим все эндпоинты, кроме `/login`, `/logout` для неавторизованных пользователей, а также отключим Cross Origin Resource Sharing (CORS) и Cross Site Request Forgery (CSRF). Они сейчас не имеют значение, но могут усложнить изучение Spring Security:

```
@Configuration
public class SecurityConfig extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.csrf().disable().cors().disable()
            .authorizeRequests().antMatchers("/login",
"/logout").permitAll()
            .anyRequest().authenticated();
    }
}
```

Теперь хотелось бы добавить форму логина, а также какой-то источник логинов и паролей пользователей:

```
@Configuration
public class SecurityConfig extends
WebSecurityConfigurerAdapter {
    @Override
    protected void configure(HttpSecurity http) throws
Exception {
        http.csrf().disable().cors().disable()
            .authorizeRequests().antMatchers("/login",
"/logout").permitAll()
            .anyRequest().authenticated()
            .and().formLogin()
            .and().userDetailsService(userDetailsService());
    }
}
```

```

public UserDetailsService userDetailsService() {
    InMemoryUserDetailsManager userDetailsService = new
InMemoryUserDetailsManager();

    userDetailsService.createUser(new User("user",
"password", List.of(new SimpleGrantedAuthority("ROLE_USER"))));
    return userDetailsService;
}

@Bean
PasswordEncoder encoder() {
    return NoOpPasswordEncoder.getInstance();
}
}

```

*PasswordEncoder* – класс, который отвечает за кодирование паролей. В данном случае используется *NoOpPasswordEncoder*, но никогда его не стоит использовать в реальном проекте – он никак не шифрует пароли. В данном случае он используется просто для простоты. А теперь запустим проект и попробуем перейти на защищенный эндпоинт. Нас отправит на `/login` (рисунок 36).



The image shows a login form on a light gray background. At the top, the text "Please sign in" is displayed in a large, dark font. Below this, there are two input fields: the first is labeled "Username" and the second is labeled "Password". Both fields have a light gray border and a small blue outline. Below the password field is a blue button with the text "Sign in" in white.

Рисунок 36 – Форма авторизации

Попробуем ввести наш логин «user» и пароль «password». После логина перейдем на защищенный эндпоинт, и увидим информацию.



Также нам хотелось бы еще где-то хранить сессию. Для этого используется Spring Session, который активируется добавлением одной зависимости – `spring session jdbc`.

### **23.2. Задание**

В приложении из предыдущего задания добавить возможность регистрации и авторизации пользователей, хранение cookie сессий в базе данных PostgreSQL, хеширование паролей алгоритмом Bcrypt, защиту всех запросов, кроме запросов на авторизацию и регистрацию, от неавторизованных пользователей.

## 24. ПРАКТИЧЕСКАЯ РАБОТА № 24

### Цель работы

Тестирование в Spring Framework с использованием Junit.

### 24.1. Теоретическая часть

В данном задании мы обратим внимание на модульное тестирование – такой вид тестирования, в котором тестируется отдельный модуль, класс, в изоляции от остальных, на правильное функционирование. Модульное тестирование важно не на стадии создания данных тестов, а при дальнейшем развитии кодовой базы, когда тесты не дадут сломаться тому, что уже работает. Для написания модульных тестов будем использовать библиотеку JUnit, которая тоже прекрасно интегрирована с Spring.

Также нам потребуется создание mock объектов – реализаций некоторых объектов-пустышек, не выполняющих логики, либо симулирующие выполнение какой-то логики. Также мы можем определять, что они будут возвращать, для удобства тестирования. Мы будем тестировать UserService из предыдущего задания.

Для начала создадим класс в папке test UserServiceImplTest, и добавим два метода – getUsers, saveOrUpdate, и пометим аннотацией Test.

```
class UserServiceImplTest {  
    @Test  
    void getUsers() {  
    }  
    @Test  
    void saveOrUpdate() {  
    }  
}
```

Уже сейчас мы можем запустить, и тесты успешно пройдут. Но нам нужно все же проверить какую-то логику. Но для начала нам нужно сделать mock объект интерфейса UserRepository. Для этого добавим аннотацию

**ExtendWith** для активации Mockito, а также добавим свойство **UserRepository** с аннотацией **Mock**:

```
@ExtendWith(MockitoExtension.class)
class UserServiceImplTest {
    @Mock
    private UserRepository userRepository;
    @Test
    void getUsers() {
    }
    @Test
    void saveOrUpdate() {
    }
}
```

**Теперь уже добавим логику в тест getUsers:**

```
@ExtendWith(MockitoExtension.class)
class UserServiceImplTest {
    @Mock
    private UserRepository userRepository;
    @Test
    void getUsers() {
        User user = new User();
        user.setFirstName("Vasya");
        User user2 = new User();
        user2.setFirstName("Dima");
        Mockito.when(userRepository.findAll()).thenReturn(List.of(user,
user2));

        UserService userService = new
UserServiceImpl(userRepository);
        Assertions.assertEquals(2,
userService.getUsers().size());
        Assertions.assertEquals("Vasya",
userService.getUsers().get(0).getFirstName());
    }
    @Test
    void saveOrUpdate() {
    }
}
```

```
    }  
}
```

Assertions отвечает за проверку данных – являются ли они теми, которые мы ожидаем от программы, или нет. Теперь добавим логику для метода `saveOrUpdate`:

```
@ExtendWith(MockitoExtension.class)  
class UserServiceImplTest {  
    @Mock  
    private UserRepository userRepository;  
    @Captor  
    ArgumentCaptor<User> captor;  
    @Test  
    void getUsers() {  
        User user = new User();  
        user.setFirstName("Vasya");  
        User user2 = new User();  
        user2.setFirstName("Dima");  
        Mockito.when(userRepository.findAll()).thenReturn(List.of(user,  
user2));  
        UserService userService = new  
        UserServiceImpl(userRepository);  
        Assertions.assertEquals(2,  
userService.getUsers().size());  
        Assertions.assertEquals("Vasya",  
userService.getUsers().get(0).getFirstName());  
    }  
    @Test  
    void saveOrUpdate() {  
        User user = new User();  
        user.setFirstName("Vitya");  
        UserService userService = new  
        UserServiceImpl(userRepository);  
        userService.saveOrUpdate(user);  
        Mockito.verify(userRepository).save(captor.capture());  
        User captured = captor.getValue();
```

```
        assertEquals("Vitya", captured.getFirstName());  
    }  
}
```

ArgumentCaptor отвечает за перехват аргументов. Нам нужно узнать, поменялся ли как-то объект User в процессе выполнения метода UserService, поэтому нам нужно перехватить аргумент и проверить его.

Тестирование также важно, как и сама разработка, ведь любой код в первую очередь пишется для того, чтобы его кто-то использовал — как приложение, как библиотеку, не важно. Код всегда должен выполнять бизнес-требования, иначе и само программирование становится бесполезным.

## **24.2. Задание**

Написать модульное тестирование для всех классов сервисов приложения из предыдущего задания.

## **25. ОПИСАНИЕ ВЫПОЛНЕНИЯ РАБОТ**

### **25.1. Последовательность выполнения практической работы**

Для выполнения практических работ необходимо:

1) на рабочем месте, где будут выполняться практические работы, установить следующие средства разработки:

а) OpenJDK версии 11 и выше;

б) IntelliJ IDEA Community Edition (желательно Ultimate Edition, можно получить по студенческой лицензии на сайте JetBrains);

в) Docker;

2) создать пустой проект в IntelliJ IDEA;

3) создать пустой класс, содержащий статический main метод;

4) запустить созданную программу и убедиться в работоспособности.

### **25.2. Порядок выполнения индивидуального задания**

Индивидуальное задание должно быть оформлено в отдельном проекте.

Для проверки работоспособности выполненного индивидуального задания следует использовать отдельный класс с методом main.

## **26. ЗАЩИТА ПРАКТИЧЕСКИХ РАБОТ**

### **26.1. Результат практической работы**

Результатом практической работы является:

- 1) рабочий проект, выполненный в соответствии с заданием практической работы;
- 2) отчет, содержащий все этапы выполненной работы, оформленный по примеру.

### **26.2. Этапы защиты практической работы**

Этапы защиты практической работы:

- 1) демонстрация рабочего проекта, выполненного в соответствии с заданием;
- 2) ответы на дополнительные вопросы по рабочему проекту (студент должен владеть теоретической базой, свободно читать и комментировать строки листинга программы, уметь формулировать выводы о проделанной работе);
- 3) отчет по практической работе предоставляется в электронном виде.

## ПЕРЕЧЕНЬ СОКРАЩЕНИЙ

AOP	–	Aspect Oriented Programming
API	–	Application programming Interface
CORS	–	Cross Origin Resource Sharing
CSRF	–	Cross Site Request Forgery
DI	–	Dependency Injection
DSL	–	Domain specific language
DTO	–	Data Transfer Object
HQL	–	Hibernate Query Language
HTTP	–	HyperText Transfer Protocol
IoC	–	Inversion of Control
JDK	–	Java Development Kit
JMX	–	Java Management Extensions
JPA	–	Java Persistence API
JVM	–	Java Virtual Machine
MVC	–	Model-View-Controller
ORM	–	Object Relational Mapping
SQL	–	Structured query language
xml	–	eXtensible Markup Language
БД	–	база данных



## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Стелтинг С., Маасен О. Применение шаблонов Java. Библиотека профессионала.: Пер. с англ. — М.: Издательский дом "Вильямс", 2002. — 576 с.: ил. — Парал. тит. англ.
2. Functional Interfaces in Java: Fundamentals and Examples 1st ed. Edition, Kindle Edition [Электронный ресурс]. URL: <https://www.amazon.com/Functional-Interfaces-Java-Fundamentals-Examples-ebook/dp/B07NRHQSCW> (дата обращения: 29.01.21). Заголовок с экрана.
3. Hibernate Search 6.0.0.Final: Reference Documentation [Электронный ресурс]. URL: [https://docs.jboss.org/hibernate/stable/search/reference/en-US/html\\_single/](https://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/) (дата обращения: 29.01.21). Заголовок с экрана.
4. Паттерны проектирования на Java. Каталог Java-примеров. [Электронный ресурс]. URL: <https://refactoring.guru/ru/design-patterns/java> (дата обращения: 29.01.21). Заголовок с экрана.
5. Руководство по Spring [Электронный ресурс]. URL: <https://proselyte.net/tutorials/spring-tutorial-full-version/> (дата обращения: 29.01.21). Заголовок с экрана.
6. The Reactive Manifesto [Электронный ресурс]. URL: <https://www.reactivemanifesto.org/> (дата обращения: 29.01.21). Заголовок с экрана.
7. Spring Framework Documentation [Электронный ресурс]. URL: <https://docs.spring.io/spring-framework/docs/current/reference/html/web.html> (дата обращения: 29.01.21). Заголовок с экрана.
8. Hibernate Search 6.0.0. Final: Reference Documentation [Электронный ресурс]. URL: [https://docs.jboss.org/hibernate/stable/search/reference/en-US/html\\_single/](https://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/) (дата обращения: 29.01.21). Заголовок с экрана.