МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ

Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Кафедра: "Математическая кибернетика и информационные технологии"

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Информационные технологии и программирование»

Тема: «Инструмент сборки Maven»

Выполнил: студент группы БВТ2401

Язалиев Тимур Исламович

Проверил: Харрасов К. Р.

Москва, 2026

Цель работы

Изучить основы работы с инструментом сборки Maven, научиться управлять зависимостями проекта, настраивать плагины и анализировать код с помощью статических анализаторов.

Ход работы

1. Создание проекта и настройка *exec-maven-plugin*

В соответствии с заданием, был создан новый Maven-проект. Базовая структура проекта сгенерирована автоматически. Код программы был взят из предыдущих лабораторных работ (иерархия классов *Animal, Cat, Parrot, Fish*).

Для удобного запуска приложения непосредственно из Maven в файл *pom.xml* был добавлен плагин `exec-maven-plugin`. Он позволяет выполнить основной класс проекта без необходимости сборки исполняемого JAR-файла.

Фрагмент *pom.xml*:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>3.6.3</version>
      <configuration>
        <mainClass>org.example.Main</mainClass>
        <arguments>
          <argument>-Dfile.encoding=UTF-8</argument>
        </arguments>
      </configuration>
    </plugin>
    <!-- другие плагины -->
  </plugins>
</build>
```

2. Подключение зависимости для логирования (SLF4J + Logback)

Система логирования была интегрирована для замены стандартного вывода *System.out* на более гибкий и информативный вывод. Выбор пал на SLF4J (как фасад) в связке с Logback (как реализация), так как это современный и производительный стек.

Добавленные зависимости в *pom.xml*:

```
<dependencies>
  <!-- SLF4J API -->
  <dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-api</artifactId>
    <version>2.0.17</version>
  </dependency>
```

```
    <!-- Реализация Logback -->
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.5.32</version>
    </dependency>
    <!-- другие зависимости -->
</dependencies>
```

3. Модификация классов для использования логирования

Все классы проекта (*Animal, Cat, Fish, Parrot, Main, JsonUtils*) были модифицированы. Вывод информации на консоль через *System.out.println()* был заменен на вызовы методов логгера (*logger.info(), logger.error()*). В каждый класс добавлен статический логгер.

Пример (*Cat.java*):

```
public class Cat extends Animal {
    private static final Logger logger = LoggerFactory.getLogger(Cat.class);
    // ...

    @Override
    public void makeSound() {
        logger.info("{} говорит: Мяу!", name);
    }
}
```

4. Подключение зависимости для работы с JSON (Jackson)

Для выполнения операций сериализации и десериализации объектов в формат JSON была добавлена библиотека Jackson, а именно её основной модуль *jackson-databind*.

Добавленная зависимость в *pom.xml*:

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.21.0</version>
    <scope>compile</scope>
</dependency>
```

Анализ транзитивных зависимостей:

Библиотека *jackson-databind* имеет свои собственные зависимости, которые Maven подключает автоматически. Чтобы увидеть их полный список, была выполнена команда:

```
mvn dependency:tree
```

В выводе команды можно наблюдать следующие транзитивные зависимости:

- *com.fasterxml.jackson.core:jackson-annotations* - содержит аннотации Jackson.

- *com.fasterxml.jackson.core:jackson-core* - содержит низкоуровневые потоковые API.

Эти зависимости необходимы для работы *jackson-databind* и были загружены в проект автоматически.

5. Создание класса *JsonUtils*

Для инкапсуляции логики работы с JSON был создан утилитарный класс *JsonUtils*. Он использует *ObjectMapper* из библиотеки Jackson для преобразования объектов в JSON-строку и обратно.

Код класса *JsonUtils*:

```
public class JsonUtils {
    private static final Logger logger = LoggerFactory.getLogger(JsonUtils.class);
    private static final ObjectMapper mapper = new ObjectMapper();

    static {
        mapper.enable(SerializationFeature.INDENT_OUTPUT); // Красивый вывод JSON
    }

    public static String toJson(Object object) {
        try {
            return mapper.writeValueAsString(object);
        } catch (Exception e) {
            logger.error("Ошибка сериализации в JSON", e);
            throw new RuntimeException("Ошибка сериализации", e);
        }
    }

    public static <T> T fromJson(String json, Class<T> clazz) {
        try {
            return mapper.readValue(json, clazz);
        } catch (Exception e) {
            logger.error("Ошибка десериализации из JSON", e);
            throw new RuntimeException("Ошибка десериализации", e);
        }
    }
}
```

В классе *Main* была продемонстрирована работа этих методов: объект *cat2* был сериализован в JSON, а затем успешно десериализован обратно.

6. Добавление и настройка плагина SpotBugs

В секцию *build* файла *pom.xml* был добавлен плагин SpotBugs для статического анализа кода с целью выявления потенциальных ошибок.

Добавленный плагин в *pom.xml*:

```
<plugin>
```

```
    <groupId>com.github.spotbugs</groupId>
    <artifactId>spotbugs-maven-plugin</artifactId>
    <version>4.9.8.2</version>
    <configuration>
        <effort>Max</effort>       <!-- Максимальная глубина анализа -->
        <threshold>Medium</threshold> <!-- Сообщать о проблемах уровня Medium и выше --
>
        <failOnError>true</failOnError> <!-- Прерывать сборку при ошибках -->
    </configuration>
    <executions>
        <execution>
            <goals>
                <goal>check</goal> <!-- Автоматический запуск при сборке -->
            </goals>
        </execution>
    </executions>
</plugin>
```

Выполнение анализа и исправление ошибок:

Была выполнена команда для запуска анализатора:

```
mvn spotbugs:check
```

Первоначальный запуск выявил несколько предупреждений:

1. Проблема: Неправильное написание имени логгера (*logger* вместо *LOGGER*).

   ◦ Решение: В классах *Cat, Fish, Parrot* и *Main* статические поля-логгеры были переименованы в соответствии с конвенцией Java (UPPER_CASE).

```
// Было:
private static final Logger *logger* = ...;
// Стало:
private static final Logger LOGGER = ...;
```

2. Проблема: Использование *scanner.nextInt()* и *scanner.nextBoolean()* без последующего вызова *scanner.nextLine()* для поглощения символа новой строки, что приводило к некорректному вводу строк далее.

   ◦ Решение: В классе *Main* после каждого вызова *nextInt()* и *nextBoolean()* был добавлен вызов *scanner.nextLine();*.

После внесения всех исправлений повторный запуск `mvn spotbugs:check` завершился успешно, что подтверждает отсутствие критических ошибок по мнению анализатора.

Вывод

В ходе выполнения лабораторной работы был изучен и применен на практике инструмент сборки Maven. Были освоены ключевые аспекты: управление зависимостями (SLF4J, Logback, Jackson), настройка плагинов (`exec-maven-plugin`, `spotbugs-maven-plugin`) для автоматизации задач, интеграция логирования вместо консольного вывода, а также использование статического анализатора кода SpotBugs для повышения качества и

надежности разрабатываемого приложения. Все пункты задания выполнены в полном объеме.

Ссылка на гит репозиторий с лабораторной работой: https://github.com/Maggistr/4-semestr.git