Linnæus University
Sweden

Bachelors Degree Thesis
Faculty of Technology

Comparing Spring REST api test frameworks

- A comparison study

Author: Leopold Huber
Author: Sebastian Åkerblom
Supervisor: PhD Tobias Ohlsson
Examiner: Diego Perez
Subject: Computer Science
Term: VT2023

Abstract

This bachelor thesis presents a comparison of three Java testing frameworks - JUnit 5, TestNG and Spock - with the purpose of evaluating their suitability in testing REST APIs built with Spring Boot. As the demand for reliable and high-quality software systems continues to grow, automated testing techniques are crucial in ensuring the correct functionality of applications. Our study aims to fill the knowledge gap in the current literature by focusing on unit tests for Java REST APIs running on the Spring framework.

We developed a single Spring Boot application and applied tests written using the three selected testing frameworks. We then compared the performance of the frameworks based on execution time, memory usage and code conciseness. Additionally, we conducted a questionnaire to gather developer preferences for the frameworks.

Our findings reveal that TestNG outperforms JUnit 5 in terms of performance, while Spock requires fewer characters, making it more concise. However, JUnit 5 remains the most well-known and widely used testing framework among developers. The results of our study provide valuable insights into the performance and developer preferences of the selected testing frameworks.

Key words

java, testing, frameworks, Spring, unit tests, JUnit 5, TestNG, Spock

Acknowledgements

We would like to thank our supervisor Tobias Ohlsson and examiner Diego Perez for valuable feedback and cheerful comments related to our research. A special thanks to our teacher Daniel Toll for the effort he put in guiding us to a suitable topic for this paper. We would also like to thank all respondents of our questionnaire, providing us with valuable results.

Contents

1 Introduction

Creating and executing unit tests is a critical part when it comes to developing functioning and maintainable software. In this 15 HEC Bachelor Thesis in Computer science we aim at comparing 3 Java testing frameworks to find out which one to recommend when developing a REST API application built in Spring.

1.1 Background

In today's fast-paced world our lives are more and more connected and dependent on software systems such as web applications, mobile apps and desktop software. Having these systems function in an expected way is therefore crucial for our society to work as a whole.

Defects in software can cause serious effects, such as the loss of data or incorrect responses to system queries. High-profile cases like the 2012 Knight Capital trading glitch [1] shows the importance of robust software testing.

One way to improve the overall quality of a product while saving time otherwise spent on manual testing is by integrating test automation into the software development lifecycle. In order to make informed decision on what test frameworks are suitable for the software project a developer however needs to put some energy and thought on comparing different test frameworks. With help of a suitable test framework a developer will be able to work more efficient with the testing part of a software project, both increasing the robustness of the system while minimizing time and resources spent.

Testing involves a wide range of activities, such as unit-testing, integration testing and system testing. Automated tests can be written for both back-end and front-end code, but to narrow down our research area and fill the knowledge gap that we have spotted we have chosen to focus on unit tests for a Java back-end application running on the Spring Boot framework.

As of February 2023 the Tiobe index ranks Java as the third most popular programming language worldwide according to their ratings system that consists of search queries made for different languages in different browsers. The Tiobe index is updated on a monthly basis, it is maintained by Tiobe Software, a company specializing in software quality assessment [2], [3]. While researching Java test frameworks we have found mainly studies comparing GUI frameworks [4]. We have yet to find a study comparing the three frameworks that we research closer in this study.

Representational State Transfer (REST) APIs have become increasingly prevalent in software development due to their numerous benefits. The use of REST APIs in software development is motivated by their scalability, interoperability, statelessness, cacheability, simplicity and maintainability, making them well-suited for the dynamic demands of modern applications [5].

In the literature review by M. A. Jamil *et al.* [6], the authors dive into the techniques of software testing and mentions that testing is the most critical part of the software development life-cycle. Furthermore, the authors concludes that testing is time-consuming and requires automated testing techniques, further motivating the need for this study that compares automated testing frameworks.

1.2 Related Work

In the paper *An Evaluation of Testing Frameworks for Beginners in JavaScript Programming: An evaluation of testing frameworks with beginners in mind* [7] the author compares three Javascript testing frameworks. Here Jest, AVA and Node TAP are compared based on the main criteria of simplicity, documentation and features. While this study focused on frameworks for another language we still derived inspiration for the methodology used, especially Bunge's scientific method that the author follows. One surprising finding of this study was that while Jest was the overall performer, AVA was not very far behind. While we read this study in order to get inspiration on the research process of a similar project we found it interesting that Jest's and AVA's similar results show that smaller projects that relies on code-contributions can almost keep up with larger projects (Meta is the company that originally developed Jest).

Another paper that is related to our work is *Ramverk för enhetstestning* [8], written by Robin Dahlgren. In this paper TestNG and JUnit 5 is compared. Furthermore, the aim of the study mentioned is to compare Java Testing Frameworks in general, in order to find out what testing framework to use for an imaginary testing course held at a university. We on the other hand aim to provide an answer to which testing framework may be preferred for a Java Spring application. *Ramverk för enhetstestning* concludes that JUnit 5 is the preferred framework when compared to TestNG, in line with the hypothesis of our study.

Meiliana *et al.* [9] performs a similar comparison study as intended for this thesis, however that paper focused on testing frameworks for Graphical User Interface in Android development, namely Espresso, UI Automator, Appium and Calabash, the conclusion of recommended framework there is therefore not applicable to this study.

We are aware that there are numerous gray literature blog posts online comparing the frameworks we mention in this study [10]–[12]. While they provide useful insights on what frameworks a developer may want to consider when writing tests for a Java Spring Application they may lack the comprehensiveness, rigor and objectivity required for a scientific study. Consequently we have chosen to conduct a systematic and in-depth analysis of the frameworks that are the focus of this study, in order to provide a robust and reliable comparison for developers, educators and researchers. Three of the gray literature blog posts that we found relevant are discussed below.

A surprising finding in is the applicability of the TestNG framework as mentioned in the Browserstack article [10]. Compared to JUnit 5, TestNG offers some extra functionality, such as arbitrary thread pools. The study here set TestNG apart from other frameworks and points out that it is an appealing choice for various testing scenarios.

The Enlear Academy article provides a comprehensive list of Popular Java testing frameworks, but lacks direct comparisons of the frameworks mentioned in regards to performance and memory consumption. Additionally, it fails to provide pros and cons of the frameworks discussed, making it harder to chose one framework over another [11].

While the Lambdatest article does provide an extensive overview of many popular Java frameworks along with pros and cons it would be more informative if it included a detailed analysis of the performance of the frameworks mentioned. This would provide the readers with a more comprehensive understanding on how each and every framework compares to each other and which one is suited for specific needs [12].

While researching similar studies using services such as IEEE Explore and Google Scholar we did not find published research comparing JUnit 5, Spock and TestNG in the context of a Java back-end application running on Spring Boot.

1.3 Problem formulation

Other papers have explored different Java testing frameworks aimed for other aspects of the codebase, such as the graphical user interface (GUI) [9]. As mentioned above, there are also gray research articles comparing Java frameworks in general, but our literature research has not found any articles exploring testing frameworks for Spring Boot applications where the back-end runs on Java, and that is the knowledge gap that we aim to fill with this study [13]. The motivation to why we selected what frameworks to research are discussed in the Chapter Scope/Limitation.

With our research we aim on answering these two research questions:

- RQ1.1 - How do Junit 5, TestNG and Spock compare when it comes to unit testing?
- RQ1.2 - Can a single value represent the quality of a testing framework appropriately?
- RQ2 - Which testing framework is preferred by participants who have experience in developing Java Spring applications and why?

1.3.1 Expected Results

Since we have not previously worked with either TestNG or Spock we cannot hypothesize on the performance of the frameworks compared to JUnit 5.

From our personal experience when writing Java projects we believe that JUnit 5 will be the preferred framework by most developers since it is the one we have come across the most times in tutorials, and due to the fact that it comes pre-installed in Spring boot. We therefore expect to answer JUnit 5 to RQ2.

1.4 Motivation

All quality applications need testing to make sure the application fulfills the demands promised and there is a very large amount of different testing frameworks to choose from when testing Java code.

A GitHub search for "Java Testing Frameworks" where we only included projects with at least 100 GitHub stars already yielded over 40 results. A quick Google search on popular frameworks also yields very many results, such as "13 Best Java Testing Frameworks For 2023” [13].

We also feel like a comparison of only 4 frameworks, like in the study by Meiliana *et al.*, motivates writing a research paper on the subject, since a lot of work is involved in deep diving into the implementations of different testing frameworks [9].

As Java is a popular programming language and widely used for developing rigid back end systems, our research area focuses on Java testing frameworks that work well together with the Spring boot framework, which is one of the most popular web frameworks for Java.

A poor choice in testing framework can impact the efficiency and effectiveness of the testing process. By comparing different testing frameworks, developers can make informed decisions on what testing framework to choose for their projects. This will improve quality and reduce time and resources spent.

1.5 Results

In this thesis, we present a comprehensive comparison of three popular test frameworks for Java Spring applications: Spock, TestNG and JUnit 5. Our research yields several types of results, providing valuable insights into the performance and developer preferences.

Through the evaluation of execution time, memory usage and code conciseness, we propose a method for comparing test frameworks, which can be applied to other similar comparisons in the field of software engineering.

Based on the data obtained from running the test frameworks on a Java Spring application and the questionnaire, our study presents results that captures the relationship between the performance metrics and junior developer preferences for the frameworks.

By selecting well-defined result-types relevant to the subject of our study we can validate the results of our research [14]. Furthermore, we improved the reliability of our research methods by having our paper being peer reviewed by other students participating in the course, as well as incorporating feedback from our supervisor.

1.6 Scope/Limitation

We did a research by searching Github for general purpose Java Frameworks and came up with this list:

- JBehave
- JUnit 5
- Serenity
- TestNG
- Selenide
- Gauge
- Geb
- Spock
- Selenium
- Mockito

After we had the initial list we decided on a few criteria that we wanted a framework to fulfill in order for us to evaluate it, these criteria were as follows:

- **IDE Compatible:** An Integrated Development Environment (IDE) can enhance the productivity of a developer by offering features such as autocompletion, debugging tools, and more. This is the reason to why we wanted the tests that we compare to be compatible with IDE choice of this research paper, that is Microsoft Visual Studio Code.

- **Open-source:** With an open source project, the developer has the liberty of understanding how it works by inspecting the source code. He/she can also contribute to the project by making pull requests in order to patch bugs or introduce new features. The open source community of a project can also provide assistance or guidance related to the project.
- **Able to generate test-reports:** Clear and detailed test reports allow developers and quality assurance teams to easily identify failing tests and understand causes of errors.
- **Ability to configure test execution:** This feature is important as it provides the flexibility of running individual tests, a specific suite of tests, or all tests. Also, the ability to configure the order of test execution is a key factor in setting up complex test scenarios.
- **Should be fairly well maintained (updated within the last 3 years):** A well maintained open source project indicates that bugs or issues are resolved promptly. This is essential since we want to ensure that our testing-framework works as intended.
- **Support for unit-tests:** By supporting unit-tests developers are able to catch bugs early. Running unit tests in an iterative manner facilitates the refactoring process and helps with providing a robust software.
- **Consistency:** By ensuring that the application where the tests are written is the same for all three frameworks we can compare the frameworks fairly, without variations in the complexity or functionality of the application.
- **Efficiency:** Developing multiple applications where the tests will be written is time-consuming, and by only developing a single application we have more resources that can be spent on doing meaningful comparisons between the three frameworks that serves as the focus of this study.
- **Reduced complexity:** Comparing frameworks itself can be a daunting tasks since there are very many criteria to evaluate. By keeping the application a constant in our comparisons we avoid a lot of complexities that would arise if the application would vary while the frameworks used for the tests would too.
- **Relevance:** Since the focus of this study is finding out what test-frameworks would be preferable when building a Spring boot application we find little need of developing several spring-boot applications where we apply our tests.

After narrowing down the criteria we have found that only JUnit 5, TestNG, Mockito and Spock remained. Since Mockito is mainly used as a complement to other frameworks in order to provide mocking support we have chosen to focus on the former three frameworks.

Apart from the criteria above we felt that these three frameworks would be interesting to compare since JUnit 5 appears to be the standard unit testing frameworks used for Java development, while TestNG brands itself as an improvement of JUnit. Spock has a rather different syntax compared to the former two, and that is why we also felt that adding that framework would make the comparisons more interesting.

1.7 Testing frameworks

Below are short introductions of all the frameworks that we choose for this study. A small calculator test case is also performed in every framework to better explain how they may be used in practice. As previously mentioned, at an early stage we did not research the Mockito framework further, since that framework is mainly used as a complement to other frameworks, in order to provide mocking/stubbing support. In Appendix B "Testing frameworks not included" the same calculator test is also shown for test frameworks that were not included in this study.

1.7.1 JUnit 5

JUnit 5 is a Java unit-testing framework that integrates seamlessly with build tools such as Maven and Gradle. It provides a wide range of features including annotations, assertions, extensions, parameterized tests, nested tests and dynamic tests.

Unlike previous JUnit iterations, JUnit 5 is composed of several different modules: JUnit Platform, JUnit Jupiter and JUnit Vintage.

JUnit platform is the foundation for launching testing frameworks in Java, it also defines an API that is called TestEngine which is responsible for discovering and executing tests. JUnit platform also has a console launcher that makes it possible to execute tests from the command line.

JUnit Jupiter is an extension model for JUnit 5 and provides a more modern and flexible approach for writing tests. Jupiter supports life-cycle methods, annotations for defining tests among other features. Jupiter is designed to be more modular and customizable compared to JUnit Vintage. The Jupiter sub-project provides a TestEngine for running Jupiter based tests on the platform.

JUnit Vintage provides a TestEngine for running JUnit 3 and JUnit 4 based tests on the platform [15].

```java
@Test
void testAddition() {
    Calculator calculator = new Calculator();
    int result = calculator.add(2, 3);
    assertEquals(5, result);
}
```

1.7.2 TestNG

TestNG is a Java testing frameworks that has taken a lot of inspiration from JUnit, especially from JUnit 4. The framework is described as follows on the official website: "TestNG is a testing framework inspired from JUnit and NUnit but introducing some new functionalities that make it more powerful and easier to use".

Many features of TestNG also exist in JUnit 5, but a few of them are only available in TestNG, such as parallel tests. There are also slight variations in the syntax between the two frameworks, but they fill the same purpose [16]–[18].

```java
@Test
void testAddition() {
    Calculator calculator = new Calculator();
    int result = calculator.add(2, 3);
    Assert.assertEquals(result, 5);
}
```

1.7.3 Spock

Spock is a testing framework for Java and Groovy that aims at combining the best features from JUnit, TestNG and BDD frameworks like Jbehave. It provides ways of writing both specification-style tests and traditional unit tests. It also provides features such as built-in mocking support [19]. Specification-style Spock-test:

```
class CalculatorSpec extends Specification {
    def "addition"() {
        given:
        Calculator calculator = new Calculator()

        when:
        int result = calculator.add(2, 3)

        then:
        result == 5
    }
}
```

Unit-style Spock-test:

```
class CalculatorTest extends Specification {
    void "test addition"() {
        Calculator calculator = new Calculator()
        int result = calculator.add(2, 3)
        assert result == 5
    }
}
```

The main difference between the two style of writing the tests as can see above is the given, when and then statements.

1.8 Target group

Since the topic of this research is of technical nature the main target group is professionals in the IT-industry who may want to research what testing frameworks to use for their existing or new Java projects.

Another target group resides in the education sector. Students may want an overview of how the different frameworks compare since writing tests is a common requirement in school projects. Teachers are also included as part of our target group since they might be interested in having an overview over testing tools to include in their course content and to recommend to their students.

Lastly, researchers also form a target group for this article. They might be interested in comparative analysis of testing frameworks and knowing what research has been and not been done on the subject.

1.9 Outline

This report is organised as follows. In Chapter 2 we will discuss the methodology used in our research, such as how comparisons are made and why. In Chapter 3 we will provide a deeper theoretical background of the technologies involved, as well as different concepts related to testing, such as white/black box testing, different types of tests and so on. In Chapter 4 we will discuss the implementation done using our methods of choice, and the result of the implementation will be presented in Chapter 5, before being analysed in Chapter 6. In Chapter 7 we will discuss the results. Lastly in Chapter 8 we present the conclusion that we have made based on our discussion and results. Here we will also present suggestions for future research to be made.

2 Method

In this study, we aimed to compare three test frameworks for Java Spring REST APIs - JUnit, Spock, and TestNG - by analyzing their performance and ease of use. To achieve this, we employed a mixed-methods approach, combining experimental research with surveys of developers' preferences and experiences. This approach ensured a comprehensive understanding of the test frameworks in question, allowing for more accurate reliable comparisons. The mixed-methods approach is explained in more detail in the paper "Portal of Research Methods and Methodologies for Research Projects and Degree Projects" by Anne Håkansson [20]. This chapter outlines the research project, which included the development of a simple REST API and associated unit tests, as well as the questionnaire design and distribution process.

By combining the experimental data with the questionnaire responses, we were able to better understand the factors that influenced developers' choices and preferences in selecting test frameworks for Java Spring REST APIs [20].

2.1 Research Project

The research project was divided into two main components. The "Experimental Research" component answers our first research question that is "How to Junit 5, TestNG and Spock compare when it comes to unit testing?". The "Questionnaires" component answers our second research question that is "Which testing framework is preferred by participants who have experience in developing Java Spring applications and why?".

2.1.1 Experimental Research

We developed a simple Java Spring REST API with basic functionality, which served as a common platform for the comparison of the three test frameworks: JUnit 5, Spock and TestNG. We created unit tests for the service layer of the REST API using each of the selected frameworks. The service layer is an architectural layer that encapsulates business logic, calculations, and database operations, serving as a bridge between the data access layer and the presentation layer. Our tests were designed to assess the character count, speed and memory usage of the respective frameworks, providing quantitative data for comparison.

The speed were measured by the test-reports provided by the frameworks run, and the memory usage were taken by utilizing our PerformanceTester Class in the tests:

```java
public class PerformanceTester {
  public static long getUsedMemory() {
    Runtime runtime = Runtime.getRuntime();
    return runtime.totalMemory() - runtime.freeMemory();
  }
}
```

This experimental approach allowed us to obtain objective, measurable results, which could be used to compare the efficiency of the different test frameworks in a controlled environment [20].

The tests included methods for Creating, Modifying and Deleting resources in a CRUD application. We did not test Reading resources separately, since read operations are already performed in the create, modify and delete operations.

We also created tests for edge cases, more specifically operations that tried to delete or modify resources that were not found. Since the "not found" test cases required very little memory usage we did only test them with regards to speed and character count.

When writing tests in the different frameworks we focused on testing the following methods in the ProductService.java file.

```
public EntityModel<Product> modify(Long id, Product
 product) {
  Product existingProduct = repository.findById(id)
      .orElseThrow(() -> new
 ResourceNotFoundException(id));
  existingProduct.setName(product.getName());
  existingProduct.setDescription(product.getDescription());
  existingProduct.setPrice(product.getPrice());
  repository.save(existingProduct);
  return assembler.toModel(existingProduct);
}

public EntityModel<Product> create(Product product) {
  repository.save(product);
  return assembler.toModel(product);
}

public ResponseEntity<?> delete(Long id) {
  Product foundProduct = repository.findById(id)
      .orElseThrow(() -> new
 ResourceNotFoundException(id));
  repository.delete(foundProduct);
  return ResponseEntity.noContent().build();
}
```

2.1.2 Questionnaires

In addition to the experimental research, we designed and distributed a questionnaire to gather qualitative data on developers' preferences and experiences with the three test frameworks. The questionnaire consisted of open-ended and close-ended questions focusing on factors such as ease of use, features, and overall satisfaction.

The results of the questionnaire provided insights into the subjective preferences of developers and helped identify any potential strengths or weaknesses in the frameworks that might not have been apparent from the quantitative data alone. See Appendix section ***A Questions & Answers***.

Reasoning behind the questions of our survey

Below are the reasoning behind the questions that we included in our survey:

- **"Which of the following testing frameworks have you used in your Java Spring projects? (Multiple choice") and "Which testing framework do you prefer using for Java Spring projects? (Multiple choice)":** The two first questions are designed in order to understand what testing frameworks the respondents have used and which they prefer. With this information we can get a broad view of the current state of the testing frameworks landscape.

- **"How familiar are you with JUnit / Spock / TestNG?"** (Multiple choice) With these 3 questions we can gauge the respondent's familiarity with the aforementioned frameworks. The "not familar" to "very familar" scale allows up to capture varying levels of experience with each framework. With this information we can not only understand the distribution of knowledge among the respondents, but also contextualizes other answers they might give about their preferences and experiences.
- **"I find it easy to write tests using my preferred testing framework", "The documentation for my preferred testing framework is clear and helpful", "My preferred testing framework has good integration with Java Spring projects."** These questions are aimed at understanding what factors that contribute to selecting a specific framework.
- **"I prefer using a testing framework that is widely adopted by the industry."** This question was designed in order to better understand the importance of industry adoption when choosing a specific framework. This can indicate the importance of factors such as future proofing ones skill set, or community support by an open source project.
- **"Which criteria is the most important when selecting a testing framework for Java Spring projects?"** The answers from this question will help us better understand what factors developers prioritize when making their choice of test framework. This will give us an idea of what factors developers prioritize when making their choice and help us in comparing the frameworks from these different perspectives.

With multiple-choice questions we ensure comparability of responses and ease of data analysis. The open ended "Other" answer helped with ensuring that unanticipated answers could be captured.

2.2 Research Methods

2.2.1 The Application

For our study we have built a simple REST API in Java using the Spring framework. The application aims to be a foundation for all areas which we aim to test. It is built as a layered Spring application using controller, service and repository layers which isolates most logic in the service layer. The application uses a PostgreSQL database and both the application and database are run using docker compose.

For each testing framework we have made copies of our initial application to ensure that the application and testing remains the same, we are only comparing the testing frameworks. The copies referred to are different branches in a Github repository.

2.3 Reliability and Validity

The methods of both questionnaire and experimental studies are used within the field of computer science, which enhanced the reliability and validity of our research [20]. We designed our research questions to concentrate specifically on the performance and characteristics of the testing frameworks. This focus justified our decision to use a single REST API as the test application, ensuring that the only variable altered between tests was the testing framework itself.

To further safeguard the validity of our study, we adopted a consistent approach in which all performance metrics were collected from tests run on the same system. This approach mitigated the risk of extraneous variables that could potentially influence the results. In this way, we established a controlled environment that allowed us to directly attribute any observed differences in performance to the testing frameworks being evaluated.

Moreover, by making our study design transparent and replicable, including a clear definition of our research questions, a detailed description of our experimental setup, and an explanation of our method for collecting performance metrics, we have enhanced the reliability of our findings. Any future researchers can thus follow our methodology to reproduce our experiments and verify our results.

In addressing RQ2, we sought to understand the testing framework preferences of participants with experience in developing Java Spring applications. We used a questionnaire to capture both the choice and the reasoning behind it. By using a consistent rating scale for responses and administering the questionnaire uniformly to all participants, we assured the reliability of our findings. The questionnaire is also replicable, facilitating future verification of our study.

By adopting these measures, we have made every effort to ensure that our study provides accurate, reliable and valid insights into the efficacy of different testing frameworks for Java Spring REST APIs.

2.4 Threats to Validity

When conducting our research on comparing Java testing frameworks for the Spring Boot Application that we have built we have identified several threats to the validity of our findings. These threats are listed below:

- **Internal validity:** Our previous experience of the different Java frameworks that we compare may introduce bias in the comparisons we do in this study. Our previous experiences working with these frameworks may affect what tests cases are implemented, and how they are implemented.
- **Generalizability:** Our study focuses on what test framework to chose when developing a Spring Boot Application in Java. This decision will limit the generalizability of our findings to other types of Java applications or programming languages.
- **Statistical power and accuracy:** The statistical power and accuracy of our findings is affected by the limited test-cases that we write, and the fact that we only have a single application that the tests are written on. Furthermore, it will also be limited by how many people respond to our questionnaire.
- **Bias in the questionnaire:** Previous experiences of different testing frameworks among the developers who will take part in our questionnaire may influence the answers they provide.
- **Not knowing enough about the respondents in the questionnaire:** We acknowledge that our methodology may not exclusively capture the opinions of experienced developers. However, it should be noted that this was not the primary intention. Our goal was to gather responses from individuals with Java Spring development experience in a broad sense, which could include both

students and more senior developers. Given the nature of the school's Slack channel, we anticipate that the majority of the responses would come from current students, who form the majority of the channel's active members. Nevertheless, we value the input from the more experienced developers in this space. While we expect these individuals to constitute a smaller proportion of the respondents, their experience and insights will provide a valuable perspective on the testing frameworks in use today. Therefore, we maintain that our questionnaire, despite potential threats to validity, remains a meaningful tool for achieving our research goals.

2.5 Dependent and Independent variables

Independent variables are variables that we can evaluate and manipulate in our study, in the context of this study the independent variables would be the testing frameworks themselves that we are comparing. Each of the above frameworks has its own syntax, features and approaches to testing, which can influence the dependent variables that we observe.

The dependent variables are the factors that we analyze and measure in order to compare the performance and suitability of the three testing frameworks. These variables includes:

- **Test writing efficiency:** The amount of time and effort required to write tests in every framework.
- **Test execution speed:** How well the frameworks performs when executing tests.
- **Memory consumption:** Memory usage of the frameworks while executing tests.

2.6 Ethical Considerations

As our study only focus on factual results from our experiments we do not have any ethical considerations regarding to the results of our study or risk of harm, however, since we are doing a survey we have some considerations in regard to the participants of the study.

In order to maintain confidentiality for the questionnaire that we decided to not collect email addresses, or any information but the answers from the participants, in order to make the questionnaire and its participants anonymous. Since the questionnaire was posted on the slack-channel of our school the results will most likely be biased, since there are mostly students who read the messages there. Furthermore, as part of the questionnaire we inform the participants that the answers there will be part of our Bachelor thesis.

2.7 Bunges Scientific Method

To conduct our research in a structural and scientific way we have chosen to work after Bunge's Scientific Method that divides the research process into stages to transform the hypothesis of the work into a scientific article [21], [22].

Bunge's Scientific Method is an interpretation of the research process outlined in the book "Epistemology and Methodology I: Exploring the World, Vol 5'" by Mario

Bunge [22]. We have taken a great deal of inspiration from this interpretation, but cannot say that we have followed it exactly as intended.

All steps below belong to either research, experimentation/application or evaluation. They are an translation of the steps outlined in the interpretation of the method, outlined in the paper "Vetenskaplighet - Utvärdering av tre implementeringsprojekt inom IT Bygg & Fastighet 2002" by Niclas Andersson and Anders Ekholm [21].

1. Identify a problem in an area of research.
2. Describe the problem in clear words.
3. Map the existing information and methods to gather the existing knowledge of the area.
4. Explain a solution based on the background knowledge described in step 3. If the background knowledge is not enough to do this, then go to step 5, otherwise skip step 5.
5. Suggest new theories, ideas or techniques and generate new empirical data for a solution to the problem.
6. Submit an exact or approximate solution to the problem.
7. Extract the consequences of the presented solution.
8. Test the proposed solution.
9. Correct the solution proposal after analyzing the results form step 8 (if needed).
10. Examine the solution with the existing knowledge from step 3 in mind and identify new issues.

3 Theoretical Background

In this chapter we will discuss the tools used as part of this study, as well as the theoretical background when it comes to different methods of testing software. We will discuss different ways of testing, namely unit tests, integration tests, functional tests, system tests as well as acceptance tests.

We will also briefly discuss the testing pyramid that helps us visualise roughly the amount of tests required of different types of tests. We will also discuss the frameworks we have chosen, as well as similar frameworks that were not included in the study.

To finish of this chapter we will also discuss some general concepts that apply for software testing such as Test Driven Development (TDD), Behaviour Driven Development (BDD), as well as White and Black-box testing.

3.1 Functional tests and non-functional tests

Functional tests involves testing the behaviour of the system and ensuring that it meets the requirements specified for the software. For example, functional tests include integration tests to ensure that the software works as expected when multiple component are integrated as a single unit. Another type of functional test is regression testing, where tests are performed after doing a update to the system, to ensure that it still works according to the business or user requirements.

Non-functional tests involves measuring quality metrics of a system such as performance, security and reliability. These aspects can be measured by for example setting the system under heavy traffic-loads and measure the response-time and check whether the system crashes or not.

Usability testing is also a non-functional test where the interface is tested for its ease of use. Security aspects, such as testing authorization and authentication is also classified as non-functional tests.

In summary, functional testing are performed to ensure that the system meets the business and user requirements, while non-functional testing is performed to measure other aspects of the system that is not part of the functional testing, such as performance, security and usability [23], [24].

3.2 Unit tests

Unit tests are tests performed in isolation from the rest of the system. What isolation means here is that the code is tested without any dependencies to other parts of the system, such as databases or web-services. To achieve this, dependencies are usually mocked or stubbed out which means that they are replaced with test doubles that mimic the behavior of the real dependencies but in a controlled way.

By isolating the code that is being tested from other part of the system we can ensure that a test fails due to problems in the actual code and not because of problems in any dependencies of the code [25].

3.3 Integration tests

Software systems today are built up of several modules that needs to work together. While unit testing is the practice of testing these modules in isolation integration testing is the practice of testing the interfaces between the modules and how they work together as a group.

While all unit tests of modules may pass we may still have integration tests failing due to for example mismatching communication protocols or data formats.

Integration tests can be performed on different scales, such as subsystem integration where groups of related components are tested together, or system integration where the entire system is tested as part of a System test [26].

3.4 System tests

System tests is the process of testing an entire system as a whole. It involves functional, as well as non-functional testing to for example measure the performance or security of the complete system.

There are also a few more types of system tests, such as:

- Functional testing - Compares actual and computed inputs.
- Non-functional testing - Evaluates quality attributes of the system, such as security, usability, reliability and performance.
- Usability testing - Focus on the ease of use of the system from the perspective of an actual user.
- Load testing - How the system as a whole performs under real-life loads.
- Regression testing - Makes sure that changes to the system has not introduced new bugs, or re-introduced bugs that were existing in previous iterations of the system.
- Recovery testing - Tests that the system can recover in the event of a failure.
- Migration testing - Ensures that the system can be migrated between systems without issues.

There are more than 50 types of system testing, but the above are types that a large software development would typically use [27], [28].

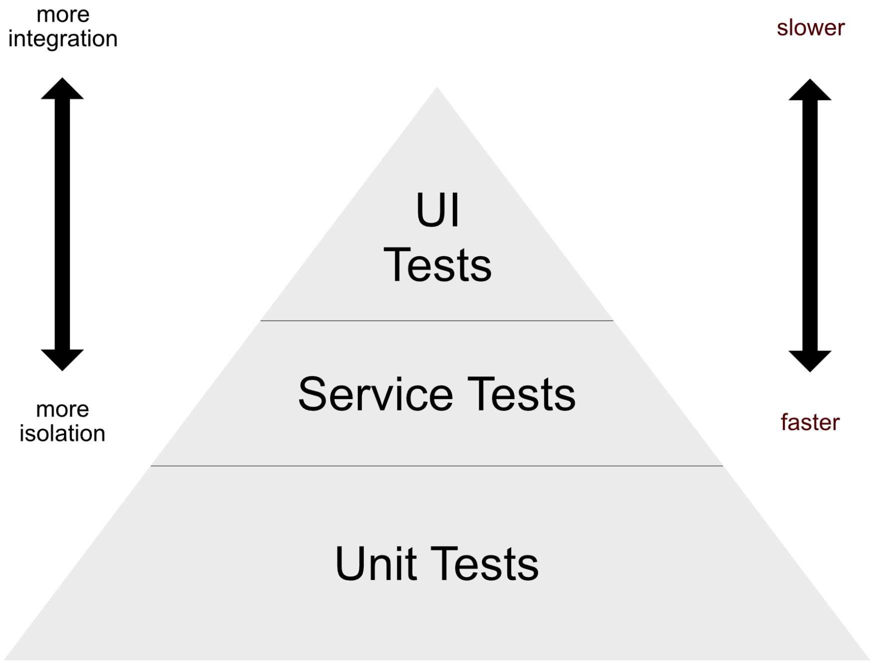
3.5 Acceptance tests

Acceptance testing is the process of analysing whether the system conforms to the requirement specification. It is performed after System tests has been done. At this stage it is decided whether the system will be released to end-users, or if changes still has to be made. Usually black-box testing is performed as part of the acceptance tests. Acceptance tests can be categorized as either internal or external.

Internal acceptance testing, also known as alpha testing is performed by internal members of the organization that developed the system but were not directly involved in the development or testing of the software. These members could include sales or customer support.

External acceptance testing, also known as beta testing is on the other hand performed by users that are not part of the organization that developed the software. External acceptance testing can also be divided in two categories where Customer Acceptance Testing (CAT) are performed by the customer who asked for the software to be developed (the customer can also refer to the company itself, if the software is meant for internal use). The other category is User Acceptance Testing (UAT), also known as Beta Testing. Here the testing is performed by existing or potential end users of the software [29].

3.6 The testing pyramid



The concept of the testing pyramid is a visual metaphor telling you to think about tests in 3 horizontal layers, where the widest section at the bottom is compromised of unit-tests, and where the middle consist of API tests, which depend on a back-end data-store, and finally UI-tests at the top.

The testing pyramid emphasizes the need of a strong foundation of unit tests, that are typically fast and reliable and can catch defects early on in the development process. API tests at the middle are generally also quite fast, but not as fast as unit-tests, and not as reliable, so they should therefore be used in less quantities. Finally, at the top of the pyramid the UI tests are placed. They should be used in lesser quantities compared to the other tests since they are less stable and takes more time to execute due to their multiple dependencies [30], [31].

3.7 Test and Behaviour driven development

Both Test Driven Development (TDD) and Behaviour Driven Development (BDD) are agile ways of developing software [32][33]. Agile development takes an iterative approach to the development process where the system is developed in incremental stages with frequent meetings between the developers and other stakeholders, such as the customer [32].

In contrast, the waterfall approach development is done in a linear manner with much more upfront planning. With this approach each stage serves as a prerequisite for the next stage, and once the next stage has started it may be hard doing changes to the one before [34].

With test driven development the developer writes failing tests, and only after this writes the code to make them pass. The code here is written to satisfy the tests and in that way provides a good code coverage for the tests in the system. Behaviour driven development is an extension of test driven development where the focus in on making sure that the written tests reflect the way the system is expected to behave. To find out the expected behaviour of the code, meetings between stakeholders, such as developers, testers and the customer are needed with frequent intervals. Once the behaviour is settled, the code to implement is written in a test driven manner [33], [35].

Both test driven development and behaviour driven development emphasize the need for frequent iterations. When it comes to test driven development the iterations may be frequent changes in code to make the tests pass, and when it comes to behaviour driven development the iterations may be to collect feedback from the customer and make necessary changes to the expectations on the behaviour of the software [36].

3.8 White and Black Box Testing

Black box testing is the technique of performing tests on a system without knowing about its internals, such as architecture or source code. Black box testing is usually conducted by a tester providing input to a system using its user-interface and examine the output with no knowledge on how the input were operated on.

White box tests on the other hand is done by a tester with knowledge of the systems source code and architecture. The tester typically analyzes the source code and writes tests with it in mind, in order to make the tests cover the as much code as possible. By having knowledge of the source code the tester can write tests with proper boundary conditions [37].

3.9 Validation & Verification

The two important areas of software development, while verification is about building the product right (conformance to specifications), validation is about building the right product (satisfaction of user needs). Together, verification and validation help to build robust, reliable and efficient software. For this thesis it is important to know what validation is since this is the field of testing while verification involves code och requirement reviews [38].

4 Implementation

In this chapter we will discuss the actual implementation of the comparison between the testing frameworks. Areas that we will cover include installation, writing and running tests. We will also briefly cover the tools used in this study.

When writing the tests we will compare aspects of them such as the test structure, exception handling, conditional test execution, mocking and parameterized tests.

At the end of this chapter we will also outline a TES metric that we have implemented in order to answer RQ1.2.

4.1 JUnit 5

4.1.1 Installation of JUnit 5

The package spring-boot-starter-test is the preferred way of many developers to get started with JUnit 5 in Spring Boot Applications. It is a "starter" kit that includes everything needed to get started with running tests. It includes the packages below. The list is cited from the official Spring Boot documentation [39].

- JUnit - The de-facto standard for unit testing Java applications.
- Spring Test and Spring Boot Test - Utilities and integration test support for Spring Boot applications.
- AssertJ - A fluent assertion library.
- Hamcrest - A library of matcher objects (also known as constraints or predicates).
- Mockito - A Java mocking framework.
- JSONassert - An assertion library for JSON.
- JsonPath - XPath for JSON.

We installed the spring-boot-starter-test framework by including it as a dependency in our build.gradle file. Also notice the automatically generated tasks section where JUnitPlatform is specified.

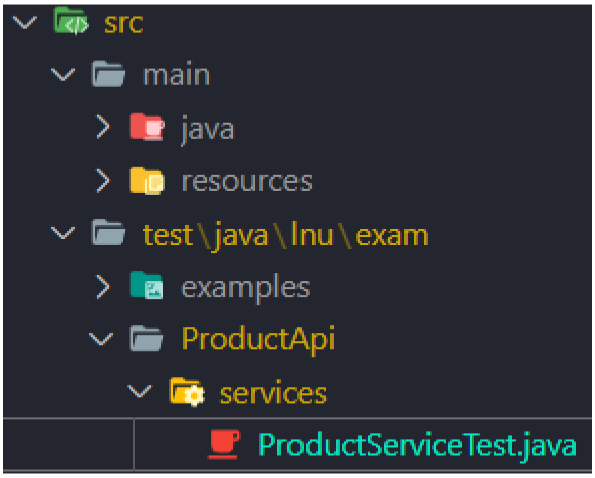
```
dependencies {
    testImplementation
   'org.springframework.boot:spring-boot-starter-test'
}

tasks.named('test') {
   useJUnitPlatform()
}
```

4.1.2 Running tests in JUnit 5

Running the tests in our environment can be done from the command terminal with a simple "gradle test" command.

4.1.3 Test structure in JUnit 5

We will focus on a unit test we will later use to test the frameworks. This test ensures correct functionality of the modify() method in our ProductService class and a complete setup is shown in the Results chapter. The file- and test structure for both JUnit 5 as well as TestNG are the same and should look something like this:



```
src
  main
    java
    resources
  test\java\lnu\exam
    examples
    ProductApi
      services
        ProductServiceTest.java
```

If we were to auto-generate a test it would look like this code below which is a decent start but contains no actual test thus simply providing the basic setup.

```
package lnu.exam.ProductApi.services;

import org.junit.jupiter.api.Test;

public class ProductServiceTest {
    @Test
    public void modify() {

    }
}
```

When writing a unit test, we want to test only a specific "unit" of code. This means that we will likely need to "mock" any dependencies that the unit relies on. To do this, we can use Mockito, which is a popular framework in Spring applications for creating mock objects. It is common to use Mockito together with JUnit 5, which is why they are both included in the spring-boot-starter-test package [39].

For now we will cover a basic test structure and not think about mocking, take note of the code below and notice our comments arrange, act and assert. In the arrange block we create the objects needed for the test, in the act block we call the actual method and save the result and lastly in the assert block we compare our expected result with the actual in an assertion method, in this case assertEquals methods.

```
package lnu.exam.ProductApi.services;
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class ProductServiceTest {

    @Test
    public void create() {
        // arrange
        Product product = new Product("Test Product", 10.0);
        ProductService service = new ProductService();

        // act
        Product result = service.create(product);

        // assert
        assertEquals(product.getName(), result.getName());
        assertEquals(product.getPrice(), result.getPrice(),
   0.001);
    }
}
```

4.1.4 Exception handling in JUnit 5

One of the key features of JUnit 5 when it comes to exception handling is the *assertThrows()* method. With this method we can make tests fail if they are not throwing the expected exception. Here is a sample code snippet:

```
@Test
void testDivideByZero() {
    assertThrows(ArithmeticException.class, () -> {
        int result = 5 / 0;
    });
}
```

In the snippet above we are testing the behaviour when we are dividing a value by zero and expect a ArithmeticException to be thrown. In the case no error of the type is thrown the error will fail.

Other features related to exception handling in JUnit 5 includes *assertDoesNotThrow()* that test that no exceptions are thrown and *expectThrows()*, which works in a similar manner as *assertThrows()*, but returns the thrown exception so that additional tests can be performed on it [40].

4.1.5 Conditional test execution in JUnit 5

With help of annotations JUnit 5 can run, or not run tests based on different conditions. One such condition may be the operating system that the test is running on. We could for example specify that a test should only be executed when running on a Mac operating system. For all other systems the test will be skipped.

```
@EnabledOnOs(OS.MAC)
public class MyConditionalTest {

    @Test
    public void testSomething() {
        // your test code here
    }
}
```

Other similar conditions that decide whether to run or skip tests include @EnabledIfSystemProperty (that enables tests based on the value of a system property), or @EnabledIfEnvironmentVariable that decides whether to run a test based on the value of an environment variable. Examples of how these can be used are found below:

```
@EnabledIfEnvironmentVariable(named =
    "NUMBER_OF_PROCESSORS", matches = "8")
@EnabledIfSystemProperty(named = "run.import.tests", matches
    = "true")
```

4.1.6 Mocking in JUnit 5

As previously stated, when working with mocking in JUnit 5, developers usually install a separate package, Mockito, in order to work with mocks. Here is an example how one may implement mocking in JUnit 5 with help och Mockito:

```
public class MockTest {
  class ClassToMock {
    public String someMethod() {
      return "original result";
    }
  }

  @Test
  public void shouldReturnMockedResult() {
    // Create a mock object of the class to mock
    ClassToMock mockObj = mock(ClassToMock.class);
    when(mockObj.someMethod()).thenReturn("mocked result");

    // Call the method being tested
    String result = mockObj.someMethod();

    // Verify that the method called the mock object as
   expected
    verify(mockObj).someMethod();

    // Assert the result
    assertEquals("mocked result", result);
  }
}
```

In the example above we have ClassToMock that always returns "original result". We then mock it to always return "mocked result" by calling when(mockObj.someMethod()).thenReturn("mocked result"). In the test "mocked result" will now be returned, leading to a successful assert.

4.1.7 Parameterized tests in JUnit 5

Parameterized tests were one of the main features earlier on that JUnit lacked. With JUnit 5 parameterized tests however are supported out of the box. When using parameterized tests in JUnit 5 you need to provide the @ParameterizedTest notation, instead of the usual @Test notation, and you also need to specify a source for the test parameters.

One way to specify this would be to use the @ValueSource parameter like in this example:

```
@ParameterizedTest
@ValueSource(strings = { "foo", "bar", "baz" })
```

```
void testStringLength(String input) {
    assertEquals(3, input.length());
}
```

In the example above the @ValueSource is specifying a list of input values as strings, and the test will be executed one time for every string, so in this case for a total of 3 times.

4.2 Spock

4.2.1 Installation of Spock

Apart from Spock's 'core' and 'spring' packages the installation of Spock also requires "groovy" which is the language used when writing tests in the Spock framework.

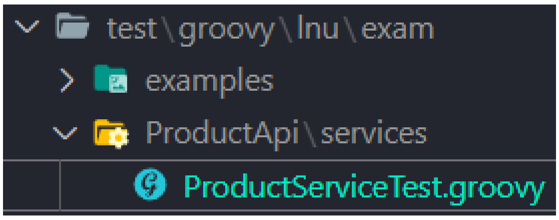
```
plugins {
    // Other plugins...
   id 'groovy'
}

dependencies {
    // Other dependencies...
   testImplementation
   'org.spockframework:spock-core:2.2-groovy-3.0'
   testImplementation
   'org.spockframework:spock-spring:2.2-groovy-3.0'
   testImplementation 'org.codehaus.groovy:groovy-all:3.0.15'
}
```

4.2.2 Running tests in Spock

Similarly to JUnit 5, texts can be executed by a simple gradle test command.

4.2.3 Test structure in Spock

Apart from differences in code Spock also prefers a slightly different file structure, notice the switch from .java to .groovy which is the language used in Spock.



```
test\groovy\lnu\exam
  examples
  ProductApi\services
    ProductServiceTest.groovy
```

```
import spock.lang.Specification

class ProductServiceSpec extends Specification {
    def "create method should return a product with the same
   name and price"() {
        given:
        def product = new Product(name: "Test Product",
   price: 10.0)
        def service = new ProductService()
        when:
        def result = service.create(product)

        then:
        result.name == product.name
        result.price == product.price
    }
}
```

There are a few ways to structure code in Spock, you'll probably notice the keywords given, when, then. These keywords are are the most common and has a logical explanation.

- given - variables and objects are created.
- when - logic to be tested are called and saved in result variables.
- then - results are compared to expected outcomes.

4.2.4 Exception handling in Spock

With Spock the @FailsWith annotation can be used to indicate that a declared exception should be thrown. Another way of checking for failing exceptions in Spock is to use the thrown(Class) method to assert a thrown exception. Below is a test implementing both the @FailsWith annotation and thrown() method:

```
class ExpectedFailTest extends Specification {
    @Subject def obj = new MyClass()

    @FailsWith(NullPointerException)
    def "throws exception if method called on null object"()
   {
        given:
        def obj = null

        when:
        obj.someMethod()

        then:
        // Exception should be thrown
    }

    def "throws exception if method called on null object
   and assert message"() {
        given:
        def obj = null

        when:
        obj.someMethod()

        then:
        def t = thrown(NullPointerException)
        t.message == 'Object is null'
    }
}
```

```
class MyClass {
    def someMethod() {
        throw new NullPointerException("Object is null")
    }
}
```

4.2.5 Conditional test execution in Spock

Conditional test support for Spock is provided out if the box with help of the @Requires and @IgnoreIf annotations. Out of the box Spock can check for JVM version, operating system, system properties and environment variables.

The test below will only execute if the system is Java 8 compatible and is not running in a production environment:

```
def jvm = System.getProperties()
def sys = System.getenv()

@IgnoreIf({ !jvm.java8Compatible })
def "addition in Java 8+"() {
given:
ConditionalCalculator calculator = new
   ConditionalCalculator()

when:
int result = calculator.add(2, 3)

then:
result == 5
}

@Requires({ sys["targetEnvironment"] != "prod" })
def "addition in non production environment"() {
given:
ConditionalCalculator calculator = new
   ConditionalCalculator()

when:
int result = calculator.add(5, 5)

then:
result == 10
}
```

4.2.6 Mocking in Spock

One of the key areas that sets Spock apart from JUnit 5 and TestNG is mocking support. While both JUnit and TestNG can be used together with Mockito to enable mocking in tests Spock contains a built-in mocking subsystem.

In Spock you can create a mock object with help of the Mock() method, and then define its behaviour with help of the "»" operator. A simple mock-test can be found below:

```
class MockTest extends Specification {
  def "should return mocked result"() {
    given:
    def mockObj = Mock(ClassToMock)
    mockObj.someMethod() >> "mocked result"

    when:
    def result = mockObj.someMethod()

    then:
    result == "mocked result"
  }
}

class ClassToMock {
  def someMethod() {
    // implementation
  }
}
```

In the example above we are mocking "ClassToMock" using the "Mock(ClassToMock)" method to instantiate the mockObj object. we are defining the behaviour of mockObj with the '»' operator in the "given" block.

In the "when" block we are calling mockObj, and in the "then" block we assert that "mocked result" was returned from the call.

Spock provides a different syntax compared to Mockito. Notice the absence of the given/thenReturn/thenAnswer/ structure in the Spock test above, as apposed to the Mockito one below:

```
// Mockito
MyClass myMockObj = mock(MyClass.class);
given(myMockObj.someOtherMethod()).willReturn("mocked
    result");
assertEquals("mocked result", myMockObj.someOtherMethod());
```

While some may say that the given/thenReturn/thenAnswer constructs helps with readability some other may say that the Spock syntax provides better separation between the blocks with no need of chaining methods together.

4.2.7 Parameterized tests in Spock

Parameterized tests is another area where Spock has some major differences when compared to Junit 5 and TestNG. Spock provides a table-like formatting of input data to make it look natural and easy to read. The @Unroll annotation is used to generate multiple test cases based on the data in the where block.

```
@Unroll
def "should divide two integers (#x / #y =
    #expectedResult)"() {
    when:
    int result = calculator.divide(x, y)

    then:
```

```
    result == expectedResult

    where:
    x | y | expectedResult
    3 | 3 | 1
    4 | 2 | 2
    1 | 1 | 1
}
```

The table like syntax can also be used together with the expect block where methods or constructors are needed to be called like the example below:

```
@Unroll("Discounted price for item #item should be
   #expectedDiscountedPrice")
def "should calculate discounted price for item"() {
    expect:
        item.calculateDiscountedPrice() ==
   expectedDiscountedPrice
    where:
        item                  || expectedDiscountedPrice
        regularItem(100)      || 90
        expensiveItem(1000)   || 900
        discountItem(100, 20) || 80
}
```

In the above example the test will verify that the calculated discounted price of an item matches the expected discounted price.

4.3 TestNG

4.3.1 Installation of TestNG

Similar to JUnit the tasks section in build.gradle is needed but with useTestNG() as stated below. TestNG do not come preconfigured with Spring and because of that we also need the testng package.

```
dependencies {
    // Other imports...
   testImplementation ’org.testng:testng:7.7.0’
}
tasks.named(’test’) {
   useTestNG() {
      useDefaultListeners = true
   }
}
```

4.3.2 Running tests in TestNG

Similarly as with JUnit 5 and Spock, the tests can be run by doing a *gradle test* command.

4.3.3 Test structure in TestNG

Structure and code are identical to JUnit 5 apart from a few minor naming differences and of course the imported methods from the testing framework. The Test annotation is imported from *testng* instead of *junit* in the example in the next section.

4.3.4 Exception handling in TestNG

TestNG supports exception handling with the *expectedExceptions* attribute to handle exceptions. One advantage of this attribute is that it makes it easy to specify multiple exceptions that could be thrown.

One example of this can be seen below:

```
import org.testng.annotations.Test;

@Test(expectedExceptions = {NullPointerException.class,
    ArrayIndexOutOfBoundsException.class})
public void testMultipleExceptions() {
     String str = null;
     int[] arr = new int[5];
     arr[10] = 5;
     str.toLowerCase();
}
```

In the example above we specify that a *NullPointerException* or *ArrayIndexOutOfBoundsException* can be thrown, and as long as one of the exceptions are thrown the test will pass.

```
@Test(expectedExceptions = {NullPointerException.class,
    ArrayIndexOutOfBoundsException.class})
public void testMultipleExceptionsUnexpectedFirst() {
     int result = 5/0; // this line will throw an unexpected
    ArithmeticException, making the test fail.
     String str = null;
     int[] arr = new int[5];
     arr[10] = 5;
     str.toLowerCase();
}
```

In the example above *NullPointerException* and *ArrayIndexOutOfBoundsException* is thrown, as expected, but also *ArithmeticException*, which is not expected, resulting in a failed test.

One interesting thing is that the test will not fail if *NullPointerException* or *ArrayIndexOutOfBoundsException* has already been thrown, as can be seen from the example below, where *ArithmeticException* is placed further down in the test:

```
@Test(expectedExceptions = {NullPointerException.class,
    ArrayIndexOutOfBoundsException.class})
public void testMultipleExceptionsUnexpectedSecond() {
     String str = null;
     int[] arr = new int[5];
     arr[10] = 5;
     int result = 5/0; // this line will throw an unexpected
    ArithmeticException, but the test will pass anyways,
    since ArrayIndexOutOfBoundsException was already thrown.
     str.toLowerCase();
}
```

The test will also pass if we throw both *NullPointerException* and *ArrayIndexOutOfBoundsException* before the unexpected *ArithmeticException* is thrown:

```
@Test(expectedExceptions = {NullPointerException.class,
   ArrayIndexOutOfBoundsException.class})
public void testMultipleExceptionsUnexpectedThird() {
    String str = null;
    int[] arr = new int[5];
    arr[10] = 5;
    str.toLowerCase();
    int result = 5/0; // this line will throw an unexpected
   ArithmeticException, but the test will pass anyways,
   since ArrayIndexOutOfBoundsException and
   NullPointerException was already thrown.
}
```

4.3.5 Conditional test execution in TestNG

With TestNG conditional test execution can be achieved with help of the "enabled" attribute of the @Test annotation.

If enabled is set to true the test will run, otherwise not. We can also use the "dependsOnMethods" attribute to enable or ignore a test depending on the result of a method:

```
@Test(enabled = true, dependsOnMethods = "myTest")
public void dependentTest() {
    // Test code here
}

@Test
public void myTest() {
    if (myFunction() != true) {
        throw new Error("myTest not passed");
    }
}

private boolean myFunction() {
    // Return true or false based on some condition
    return true;
}
```

In the above example above the dependentTest() method will only run if myTest() passes without throwing any error. We can make myTest() throw or not throw errors depending on the results of myFunction(). We could therefore use myFunction() to for example return false or true depending on system properties or similar in order to control what tests are run or not. This logic could also be put directly in myTest() to make the snippet shorter.

4.3.6 Mocking in TestNG

Similarly to JUnit 5, Mockito can be used to provide mocking support to TestNG. Mockito provides utility classes for easy integration with TestNG by providing the "mockito-testng" package on Github.

With the package above installed mocking can be implemented by adding @Listener annotations like below [41]:

```
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.testng.MockitoTestNGListener;
import org.testng.annotations.Listeners;
import org.testng.annotations.Test;

import java.util.Map;

@Listeners(MockitoTestNGListener.class)
public class MyTest {

    @Mock
    Map map;

    @InjectMocks
    SomeType someType;

    @Test
    void test() {
        // ...
    }
}
```

The MockitoTestNGListener will initialize all fields annotated with Mockito annotations. The "mockito-testng" package was originally part of Mockitos core repository but was moved to this specific package in summer 2018.

4.3.7 Parameterized tests in TestNG

TestNG supports Parameterized tests out of the box, allowing developers to run the same tests over and over using different values. One example of a parameterized test implemented in TestNG can be seen below:

```
public class ParameterTest {

    // Define a data provider that returns a set of test data
    @DataProvider(name = "testData")
    public Object[][] testData() {
        return new Object[][] {
            { 2, 3, 5 },
            { -1, 5, 4 },
            { 0, 0, 0 },
            { 100, -50, 50 }
        };
    }

    // Define a test method that accepts parameters from the
   data provider
    @Test(dataProvider = "testData")
    public void testAdd(int a, int b, int expected) {
        // Call the method being tested with the given
   arguments
        int result = MyClass.add(a, b);
        // Assert that the result matches the expected value
        assertEquals(result, expected);
    }
}

// A simple class with a method to be tested
class MyClass {
    public static int add(int a, int b) {
        return a + b;
    }
}
```

In the above example the *testData()* method returns a two dimensional array of 4 rows. Each row contains 3 values where the first and second values are the input values for the *testAdd()* method, and the third is the value that is expected of the *testAdd()* method to return.

With the test code above *testAdd()* will run for a total of 4 times, each time with different input, and with different expectations on the output. With help of the solution above we can run the *testAdd()* method very many times with different input without having to write a separate test method for each test case.

4.4 Tools

The tool used in this study is Visual Studio Code (often abbreviated as VS Code), which is a free and open-source code editor developed and maintained by Microsoft. It has a wide range of extensions and plugins available to enhance its functionality.

To perform the tests in this study we installed the "Extension Pack for Java" from Microsoft that provides support for Java IntelliSense, debugging, testing, Maven/Gradle and more [42].

4.5 Test Efficiency Score (TES)

In order to answer RQ 1.2 we have developed our own formula that we have called Test Efficiency Score (TES). The formula is as follows:

$$\text{TES} = w_1 \cdot C + w_2 \cdot T + w_3 \cdot M$$

Where:

1. w1: Weight assigned to C, between 0 and 1.
2. C: Number of characters in the test suite. Normalized between 0 and 1.
3. w2: Weight assigned T, between 0 and 1.
4. T: Time needed for a framework to execute all tests of the test suite, normalized between 0 and 1.
5. w3: Weight assigned to M, between 0 and 1.
6. M: Memory needed for a framework to execute all tests of the test suite, normalized between 0 and 1.

A developer can write similar test cases across all three frameworks, and then measure the number of characters (C), execution time (T) and memory consumption (M) for every test suite, and normalize these values between 0 and 1. He/She can then apply the weights (w1, w2 and w3) based on the importance that are assigned to each factor to get the final score.

A lower TES indicates a more efficient testing framework, and by applying this formula to each framework it will be possible to measure and compare efficiency of the frameworks in a standardized, objective and consistent manner.

The weightings could for example be as follows: w1 = 0.4, w2 = 0.4, w3 = 0.2. The weightings are not empirically validated as they can be set according to the preferences of the developer.

The normalization of the values are done by dividing each value by the maximum value of each category. An example of this is outlined in Chapter 6, Analysis.

5 Results

5.1 Tests

All tests have the same setup, the Product objects that are needed are created in a separate *setup()/init()* method that runs before all tests. For the memory usage metric we use the Java Runtime object. Memory is recorded before the test runs and once again after the test, in a *tearDown()/cleanup()* method, to measure memory consumption.

This shows the full setup and and cleanup of the Spock test class:

```
class ProductServiceSpec extends Specification {
    @Shared ProductService productService
    @Shared ProductRepository productRepository
    @Shared ProductModelAssembler productAssembler
    @Shared Long productId = 1L
    @Shared Product productOne
    @Shared Product productTwo
    @Shared Long memoryBefore;
    @Shared Long memoryAfter;

    def setup() {
        productRepository = Mock(ProductRepository)
        productAssembler = Mock(ProductModelAssembler)
        productService = new
   ProductService(productRepository, productAssembler)

        productOne = new Product()
        productOne.setId(productId)
        productOne.setName("New product")
        productOne.setDescription("New description")
        productOne.setPrice(100.0)

        productTwo = new Product()
        productTwo.setId(productId)
        productTwo.setName("Updated product")
        productTwo.setDescription("Updated description")
        productTwo.setPrice(200.0)
        memoryBefore = PerformanceTests.getMemoryUsage()
    }

    def cleanup() throws MissingMethodException {
        memoryAfter = PerformanceTests.getMemoryUsage()
        println "Memory used: " + (memoryAfter -
   memoryBefore) + " bytes"
    }
}
```

5.1.1 testCreate()

For detailed results, see *5.1.4 Result overview*.

Test code JUnit 5:

```
    @Test
```

```
@Order(1)
public void testCreate() {
    // Prepare test data
    Product newProduct = productOne;

    // Prepare saved product (usually this would be
created by the repository)
    Product savedProduct = productTwo;

    // Create expected result
    EntityModel<Product> expected =
EntityModel.of(savedProduct);

    // Set up mock behavior
    when(productRepository.save(any(Product.class)))
        .thenAnswer(invocation -> {
        Product productToSave =
invocation.getArgument(0);
        // Set ID to simulate DB generated ID
        productToSave.setId(savedProduct.getId());
        return productToSave;
    });
    when(productAssembler.toModel(any(Product.class)))
        .thenReturn(expected);

    // Call the service method
    EntityModel<Product> actual =
productService.create(newProduct);

    // Verify the result
    assertEquals(expected, actual);

    // Verify mock interactions
    verify(productRepository).save(any(Product.class));
    verify(productAssembler).toModel(any(Product.class));
}
```

Test code TestNG:

```
@Test(priority = 3)
public void testCreate() {
    // Prepare test data
    Product newProduct = new Product();
    newProduct.setName(productOne.getName());

newProduct.setDescription(productOne.getDescription());
    newProduct.setPrice(productOne.getPrice());

    // Prepare saved product (usually this would be
created by the repository)
    Product savedProduct = productOne;

    // Create expected result
    EntityModel<Product> expected =
EntityModel.of(savedProduct);
```

```
     // Set up mock behavior
     when(productRepository.save(any(Product.class)))
         .thenAnswer(invocation -> {
         Product productToSave =
invocation.getArgument(0);
         // Set ID to simulate DB generated ID
         productToSave.setId(savedProduct.getId());
         return productToSave;
     });
     when(productAssembler.toModel(any(Product.class)))
         .thenReturn(expected);

     // Call the service method
     EntityModel<Product> actual =
productService.create(newProduct);
     // Verify the result
     assertEquals(expected, actual);

     // Verify mock interactions
     verify(productRepository).save(any(Product.class));
     verify(productAssembler).toModel(any(Product.class));
 }
```

Test code Spock:

```
 def "testCreate()"() {
     given:
     EntityModel<Product> expected =
EntityModel.of(productOne)

     when:
     productService.create(productOne)

     then:
     1 * productRepository.save(_) >> productOne
     1 * productAssembler.toModel(_) >> expected
 }
```

5.1.2 testModify()

For detailed results, see *5.1.4 Result overview*.

Test code JUnit 5:

```
 @Test
 @Order(2)
 public void testModify() {
     // Prepare test data
     Product currentProduct = productOne;
     Product updatedProduct = productTwo;

     // Create expected result
     EntityModel<Product> expected =
EntityModel.of(updatedProduct);
```

```
     // Set up mock behavior
     when(productRepository.findById(productId))
             .thenReturn(Optional.of(currentProduct));
     when(productRepository.save(any(Product.class)))
         .thenAnswer(invocation -> {
         Product productToSave =
invocation.getArgument(0);
         productToSave.setId(productId);
         return productToSave;
     });

when(productAssembler.toModel(any(Product.class))).thenReturn(expected);

     // Call the service method
     EntityModel<Product> actual = productService
         .modify(productId, updatedProduct);

     // Verify the result
     assertEquals(actual, expected);

     // Verify mock interactions
     verify(productRepository).findById(productId);
     verify(productRepository).save(any(Product.class));
     verify(productAssembler).toModel(any(Product.class));
 }
```

Test code TestNG:

```
 @Test(priority = 4)
 public void testModify() {
     // Prepare test data
     Long productId = 1L;
     Product currentProduct = productOne;
     Product updatedProduct = productTwo;

     // Create expected result
     EntityModel<Product> expected =
EntityModel.of(updatedProduct);

     // Set up mock behavior
     when(productRepository.findById(productId))
         .thenReturn(Optional.of(currentProduct));
     when(productRepository.save(any(Product.class)))
         .thenAnswer(invocation -> {
         Product productToSave =
invocation.getArgument(0);
         productToSave.setId(productId);
         return productToSave;
     });
     when(productAssembler.toModel(any(Product.class)))
         .thenReturn(expected);

     // Call the service method
     EntityModel<Product> actual =
productService.modify(productId, updatedProduct);
```

```
        // Verify the result
        assertEquals(actual, expected);

        // Verify mock interactions
        verify(productRepository).findById(productId);
        verify(productRepository).save(any(Product.class));
        verify(productAssembler).toModel(any(Product.class));
    }
```

Test code Spock:

```
    def "testModify()"() {
        given:
        EntityModel<Product> expected =
EntityModel.of(productTwo)

        when:
        productService.modify(productId, productTwo)

        then:
        1 * productRepository.findById(productId) >>
Optional.of(productOne)
        1 * productRepository.save(_) >> productTwo
        1 * productAssembler.toModel(_) >> expected
    }
```

5.1.3 testDelete()

For detailed results, see *5.1.4 Result overview*.

Test code JUnit 5:

```
    @Test
    @Order(3)
    public void testDelete() {
        // Prepare test data
        Product existingProduct = productOne;

        // Set up mock behavior
        when(productRepository.findById(productId))
                .thenReturn(Optional.of(existingProduct));

doNothing().when(productRepository).delete(existingProduct);

        // Call the service method
        ResponseEntity<?> responseEntity =
productService.delete(productId);

        // Verify the result - expect no content
        assertEquals(HttpStatus.NO_CONTENT,
responseEntity.getStatusCode());

        // Verify mock interactions
        verify(productRepository).findById(productId);
        verify(productRepository).delete(existingProduct);
    }
```

Test code TestNG:

```
    @Test(priority = 5)
    public void testDelete() {
        // Prepare test data
        Product existingProduct = productOne;

        // Set up mock behavior
        when(productRepository.findById(productId))
            .thenReturn(Optional.of(existingProduct));

   doNothing().when(productRepository).delete(existingProduct);

        // Call the service method
        ResponseEntity<?> responseEntity =
   productService.delete(productId);

        // Verify the result - expect no content
        assertEquals(HttpStatus.NO_CONTENT,
   responseEntity.getStatusCode());

        // Verify mock interactions
        verify(productRepository).findById(productId);
        verify(productRepository).delete(existingProduct);
    }
```

Test code Spock:

```
    def "testDelete()"() {
        when:
        def responseEntity = productService.delete(productId)

        then:
        1 * productRepository.findById(productId) >>
   Optional.of(productOne)
        1 * productRepository.delete(_)
        responseEntity.getStatusCode() ==
   HttpStatus.NO_CONTENT
    }
```

5.1.4 Result overview

We found that memory tests using Runtime resulted in 0 bytes for the exception tests which are *testModify_NotFound()* and *testCreate_NotFound()*, these tests are therefore not presented as part of our memory usage nor time results, they are however part of the total characters used for all frameworks.

testCreate(), testModify() and testDelete() are run 10 times, for each turn both memory usage and time is recorded. The average value for each test is then calculated. The total number of characters for each test suite is recorded, including whitespaces. For all test results, see table in Appendix C.1.

5.2 Questionnaire

The questionnaire was sent out as a message in Linnaeus University Slack general channel with 6058 members combined with current students, teachers and graduated developers. A total of 14 respondents participated in the questionnaire, providing valuable feedback and insights into their experiences with our selected Java Spring test frameworks. The respondents were asked about their familiarity and preferences regarding Spock, TestNG and JUnit.

5.2.1 Previous experience with test frameworks

Out of the 14 respondents, 13 (92.9%) reported having previous experience with JUnit, making it the most familiar framework among the participants. This is not surprising given JUnit is popularity in the Java community. In contrast, only two respondents reported some familiarity with Spock and TestNG, suggesting that these frameworks may be less commonly used or known. See Appendix A.1.

5.2.2 Preferred test framework

When asked about their preferred test framework, 85.7% of respondents indicated a preference for JUnit over the other two frameworks. This preference could be attributed to the respondents' familiarity with JUnit, as well as its wide adoption within the industry. It is important to note that the questionnaire results may not be representative of the overall developer community, as the sample size is small and might not reflect the preferences of a larger, more diverse group of developers. See Appendix A.1.

5.2.3 Familiarity with JUnit

Considering that we can assume that the majority of questionnaire respondents were students or junior developers, it is not surprising that only one respondent reported being "Very familiar" with JUnit. While JUnit is widely recognized and utilized, students may still be in the early stages of their learning and thus may not have gained in-depth knowledge or extensive experience with the framework. As these students progress in their education and professional careers, their familiarity with JUnit and other test frameworks is likely to increase, potentially influencing their preferences and opinions on the most suitable frameworks. See Appendix A.2.

5.2.4 Preference for widely adopted frameworks

All respondents expressed a preference for a testing framework that is widely adopted by the industry. This preference could be driven by the perception that a popular framework has a larger community, more available resources, and better support. Additionally, using a widely adopted framework can potentially facilitate collaboration and improve maintainability, as more developers are likely to be familiar with it. See Appendix A.5.

5.2.5 Important Criteria in Selecting a Test Framework

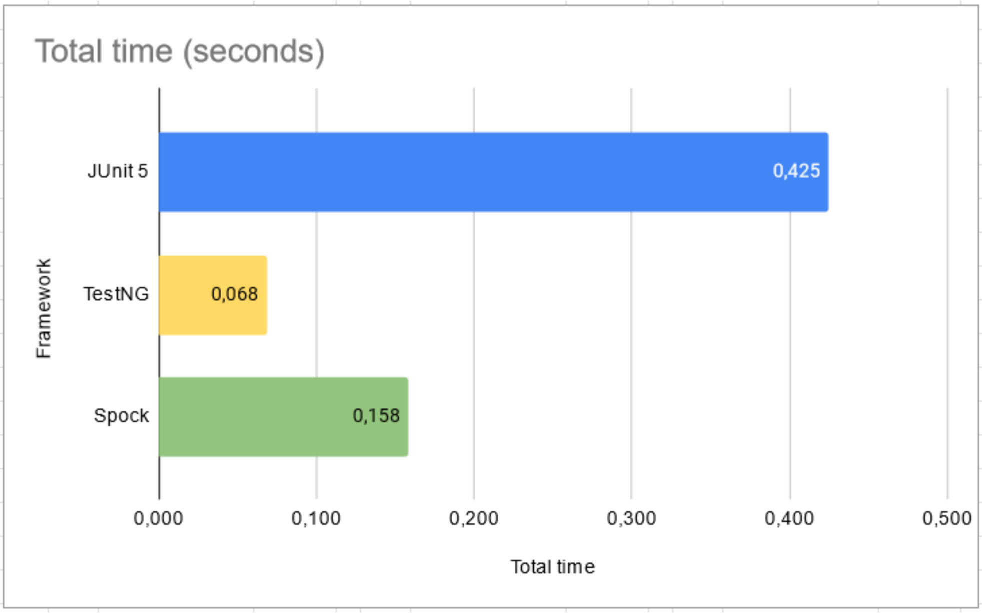
When asked about the most important criteria for selecting a testing framework for Java Spring projects, 42.9% of respondents prioritized ease of use, 28.6% chose features, and 14.3% considered documentation as the most important factor. This suggests that junior developers value a test framework that is user-friendly, feature-rich and well-documented. It is essential to take these criteria into account when selecting a test framework for Java Spring applications, as they can significantly impact the efficiency and effectiveness of the testing process. See Appendix A.5.

6 Analysis

To increase readability of our test results, presented in Chapter *5.1.4 Result overview*, we decided to add the average results together for each framework and then created visual diagrams as seen below in sections 6.1-6.3.

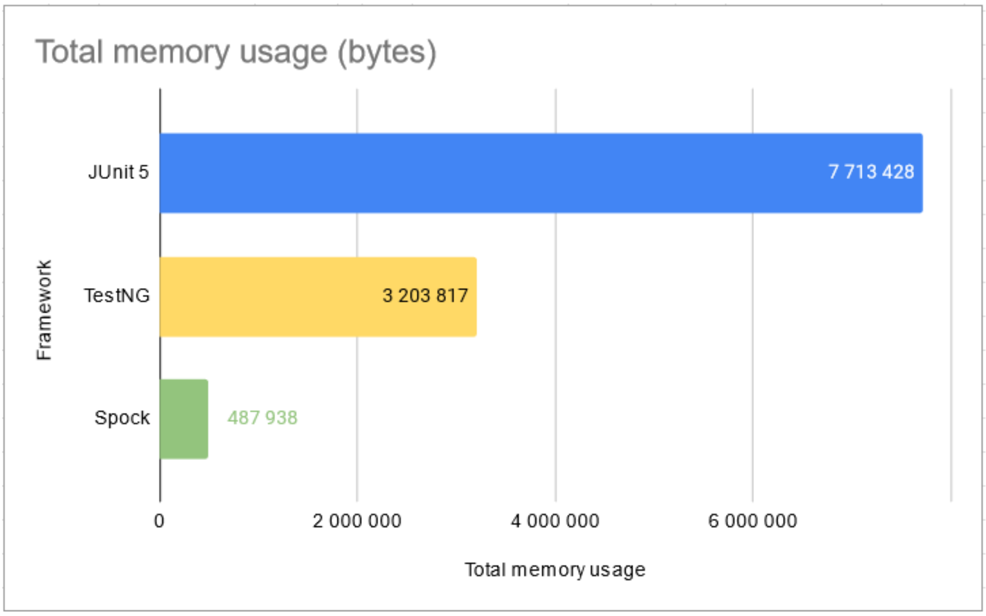
6.1 Execution time

JUnit 5 is delivered with a new Sping project but the results still shows TestNG as being the framework with the quickest execution time with Spock as a close runner up.



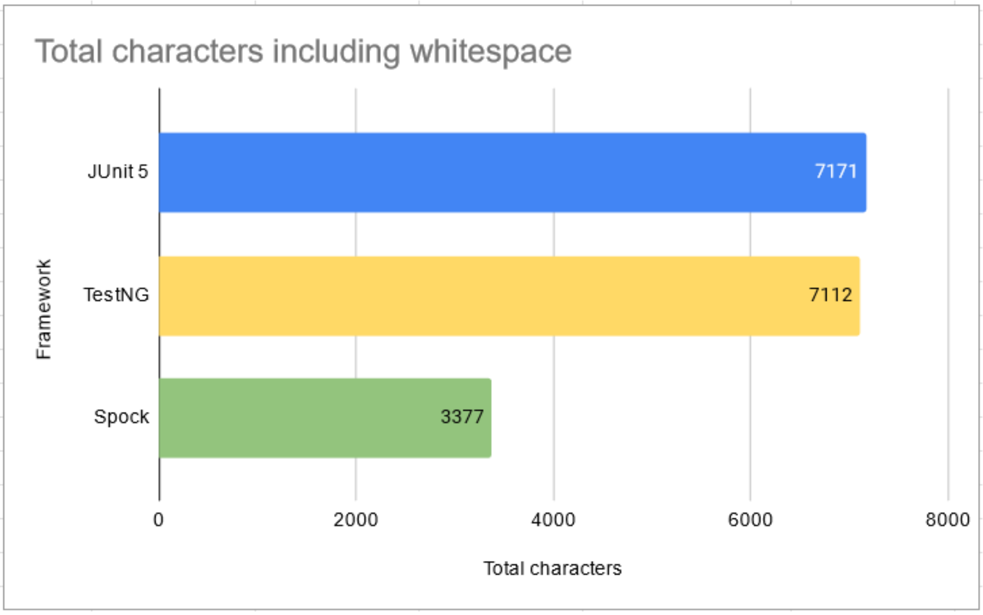
6.2 Memory usage

Surprisingly JUnit 5 continues with the worst results even here, even TestNG shows a considerable amount of memory usage while Spock shows the best results with only 0.49MB of memory usage compared to JUnit 5 and its 7.7MB.



6.3 Code conciseness (characters used)

Our code conciseness metric does not exactly compare the three frameworks but does compare Java and Groovy as programming languages. Groovy is proven to be more descriptive and our test confirms this with our Spock test suite requiring less than half of the characters needed in comparison with JUnit 5 and TestNG.



6.4 An example use of Test Efficiency Score (TES)

Below is an example for how TES may be used in practice. We will here define the weights ourselves and provide reasoning to why we set certain weights in this example. It is important to note that the weights here are not empirically validated, they are just set in order to demonstrate how TES may be used in practice. The numbers produced in this example are a product of the data we have gathered during our performance and code conciseness tests.

For our TES example we have chosen to weight character count and test execution speed equally. The reasoning behind this is that quick test execution is vital to encourage a developer of writing many tests that are run often, especially if the developer follows a TDD style of developing the software. On the other hand, concise tests also helps with readability, and having short and clean testing code will be of equal importance of motivating the developer with implementing robust test cases throughout the application.

When it comes to memory usage, we see it as less important than the other metrics mentioned above, however, it is still worth measuring and can play a vital role when the tests suite grows large. With this in mind, we have decided on weighting memory usage to half of test execution time and character count of the tests.

The TES for our three frameworks are as follows (lower indicates a more efficient framework):

JUnit 5: 0.9988 TestNG: 0.5789 Spock: 0.6507

6.4.1 Calulation of TES

Below are the calculations used to get the TES results mentioned above.

In order to calculate the TES we follow the given formula:

$$\text{TES} = w_1 \cdot C + w_2 \cdot T + w_3 \cdot M$$

Since Characters and Time should be equally weighted and twice the weight of Memory usage, we have the following weights:

$$w_1\text{(weight for characters)} = w_2\text{(weight for time)} = 2 \cdot w_3\text{(weight for memory)}$$

Since w1 + w2 + w3 = 1, we can calculate the weight for memory:

$$w_3 = \frac{1}{(2+2+1)} = \frac{1}{5} = 0.2$$

Thus, the weights for characters and time are:

$$w_1 = w_2 = 2 \cdot w_3 = 2 \cdot 0.2 = 0.4$$

Now we need to normalize the values of the test, namely Characters (C), Time (T) and Memory usage (M) to a value that is between 0 and 1. we will do this by dividing each value by the maximum value of each category:

Normalized Characters (C):

$$\begin{aligned} &\text{JUnit 5:} && \frac{7171}{7171} = 1 \\ &\text{TestNG:} && \frac{7112}{7171} = 0.9917 \\ &\text{Spock:} && \frac{3377}{7171} = 0.4709 \end{aligned}$$

Normalized Time (T):

$$\begin{aligned} &\text{JUnit 5:} \quad \frac{0.425}{0.425} = 1 \\ &\text{TestNG:} \quad \frac{0.068}{0.425} = 0,1600 \\ &\text{Spock:} \quad \frac{0.158}{0.425} = 0,3717 \end{aligned}$$

Normalized Memory usage (M):

$$\begin{aligned} &\text{JUnit 5:} \quad \frac{7713428}{7713428} = 1 \\ &\text{TestNG:} \quad \frac{3203817}{7713428} = 0.4153 \\ &\text{Spock:} \quad \frac{487938}{7713428} = 0.0632 \end{aligned}$$

With the values above we can now calculate the TES for each framework:

JUnit 5:

$$\text{TES} = 0.4 \cdot 1 + 0.4 \cdot 1 + 0.2 \cdot 1 = 0.4 + 0.4 + 0.2 = 1$$

TestNG:

$$\text{TES} = 0.4 \cdot 0.9917 + 0.4 \cdot 0,1600 + 0.2 \cdot 0.4153 = 0,3966 + 0,0640 + 0,0830 = 0,5436$$

Spock:

$$\text{TES} = 0.4 \cdot 0.4709 + 0.4 \cdot 0,3717 + 0.2 \cdot 0.0632 = 0,1883 + 0,1486 + 0,0126 = 0,3495$$

6.5 Ending notes

Using our data from our performance tests and character conciseness tests we can state that TestNG performs best in terms of execution time, Spock is the most memory-efficient and produces the most concise code, while JUnit 5 is the slowest and uses the most memory. Depending on the priorities and requirements of the project, different aspects might be more important than others, and that will influence the test efficiency score. For instance, if execution time is crucial, TestNG might be the best choice, while if memory efficiency and code conciseness are prioritized, Spock could be the preferred framework. The analysis above is also reflected in the TES, where TestNG and Spock shares similar values, and JUnit has a much higher TES (indicating that it is more inefficient overall than the former two). we have not developed TES in order to provide us with a metric to determine what the most efficient framework is, instead we have developed it to be a tool among other tools that can be used by an developer in order to get a complete picture on what framework to choose for a specific project with specific priorities.

One interesting thing to note about TES is that it promotes short tests. This may increase readability, but if developers put too much weight on lowering the TES they may produce less readable tests by using less descriptive variable names, as well as not writing enough white spaces in the tests.

Another interesting thing is that the TES mentioned above points out JUnit 5 as the clear looser, while it is still the preferred framework according to the questionnaire. Perhaps this may be due to execution speed and memory usage not being very important factors to students who work on smaller projects. For them it is perhaps of larger importance that the test framework of use comes pre-installed with Spring. On larger projects blindly sticking to JUnit could however become a problem. We see it as a risk if students become too used to only using JUnit while not considering the options. Students should be well prepared for real-world scenarios, and there might be cases where JUnit is the inferior choice there.

7 Discussion

Our study has some limitations that we would like to point out. First, we only tested a few methods from the service layer in our Spring Boot application, testing more methods from different layers and components may yield different results. Our sample application is also quite small, a bigger system may show strengths and weaknesses that our study can't show. Second, our TES didn't account for other factors that may influence developers preferences for a framework, such as community support, learning curve or ease of use.

Bias is also an important factor for our questionnaire as none of our respondents showed any significant knowledge in either TestNG nor Spock which makes their choice in JUnit predictable. JUnit also comes preconfigured in Spring which increases its bias when developers and especially junior developers and students start out in Spring projects.

As mentioned in the article by lambdatest Spock is a good choice due to its readability [12]. We believe that this is due to Spock's easy to understand syntax that is more like plain English more than the other frameworks, however, we also believe that the overall shorter syntax is to thank for the increased readability. None of the research we have read points at speed or memory performance as pros or cons of the three frameworks we have compared, suggesting that the main focus of people comparing these frameworks are aspects such as features and syntax.

While it would not mitigate the risks of malicious users answering our questionnaire we could have added a question that verifies that the respondent is a developer or student with experience in Java Spring. Another question that might add value to the results would be a question that indicates the level of the studies the potential student are participating in. We assume students in for example their last term may respond in other way to students in the first or second term.

In conclusion, or study sheds some light on the performance of JUnit 5, TestNG and Spock for tests that are part of a Spring Boot application, as well as developers views on these frameworks. The most suitable frameworks to choose depends on the project in question, and which aspects of a framework that developers value the most. Future research could include more metrics to evaluate these frameworks, including learning curve, ease of use, community support. The generalizability of our findings could also be explored by putting them into different applications and contexts.

8 Conclusion

Below are our research questions for this project again:

- RQ1.1 - How do Junit 5, TestNG and Spock compare when it comes to unit testing?
- RQ1.2 - Can a single value represent the quality of a testing framework appropriately?
- RQ2 - Which testing framework is preferred by participants who have experience in developing Java Spring applications and why?

When measuring the frameworks we found that TestNG has the best performance when it comes to execution time, while Spock is the most memory-efficient. Both frameworks shared a similar TES score, suggesting that TestNG may be preferred when execution speed is a priority, and Spock may be preferred by developers who appreciate concise code and a lower memory consumption. JUnit 5 on the other hand were a lot slower than both Spock and TestNG and also has higher memory usage. JUnit 5 has a lot higher TES score than the other frameworks, indicating that JUnit 5 is less efficient than the other frameworks by a rather larger margin. Despite this our questionnaire indicates that JUnit 5 is the clear leader when it comes to preferred testing-frameworks for Java Spring applications. We believe that this may be due to Junit 5 being pre-installed in Spring Boot, while the other two frameworks are not. It may also be due to junior developers and students being more familar with JUnit 5 compared to the other two frameworks, perhaps because of it being the test framework taught to them as part of their course content.

With the above in mind we would answer RQ1.1 with that TestNG may be preferred when test execution speed is a priority, and that Spock can be preferred when concise tests and low memory consumption is a priority.

In response to RQ1.1, it appears that TestNG may be preferred when speed is a priority, while Spock may be favored for its conciseness and lower memory consumption. For RQ1.2, we propose the TES score as a possible single value representing the quality of a testing framework. However, it is important to acknowledge that this score provides only a partial representation of the quality. The TES score aggregates metrics of execution speed, code conciseness, and memory requirements, but other factors such as ease of use, features, and documentation, as highlighted by our questionnaire respondents, also play crucial roles in the assessment of a framework's quality.

Compared to previous work in this area our study, such as "Ramverk för enhetstestning - För en eventuell kurs på Mittuniversitetet" by Robin Dahlgren [8] where JUnit 5 was the recommended test framework when compared to TestNG, our research provides further detailed differences between JUnit 5 and TestNG while also including Spock. Our questionnaire clearly answers the research question "Which test framework do developers prefer for Java Spring applications and why?". JUnit 5 is the favoured testing framework because of its ease of use, features and documentation.

The analysis of the three test frameworks Spock, TestNG, and JUnit 5 demonstrated that each framework has its advantages. TestNG outperformed the others in terms of execution time, while Spock excelled in memory efficiency and code conciseness.

However, the questionnaire results revealed that the majority of respondents preferred JUnit, possibly due to its widespread industry adoption and their familiarity with the framework.

Additionally, the respondents emphasized the importance of ease of use, features, and documentation when selecting a test framework for Java Spring projects. These criteria should be carefully considered when making a decision on the most suitable testing framework, as they can significantly impact the efficiency and effectiveness of the testing process.

While JUnit is popularity and familiarity among developers may make it a more practical choice for some teams, it is important to weigh its benefits against those of other frameworks based on the specific requirements and goals of a project. The best test framework for a particular Java Spring application will depend on various factors, including the team's experience, the desired features, and the ease of use of the framework.

In conclusion, developers should take into account both the objective measures of testing frameworks and their own preferences and experiences when selecting a testing framework for Java Spring applications. By doing so, they can choose the most appropriate framework that meets their project's needs and maximizes the efficiency and effectiveness of their testing processes.

8.1 Future work

While this study provided valuable insights into the performance and preferences of different testing frameworks for Java Spring applications, there are several areas for future research and exploration to further enhance our understanding and improve the decision-making process.

- **Larger and more diverse sample:** This study relied on a small sample size, mostly compromised of students and junior developers. Future research could include a larger and more diverse sample of developers from various professional backgrounds, levels of experience and industries to provide a more comprehensive understanding of the preferences and experiences within the broader developer community.

- **Additional test frameworks:** This study compared three popular testing frameworks for Java Spring applications. However, there may be other emerging or less well-known testing frameworks that warrant investigation. Future work could include a comparison of additional frameworks, potentially revealing new insights and alternatives for Java testing.

- **In-depth performance analysis:** The current study focused on execution time, memory usage, and code conciseness. Future research could investigate other performance metrics, such as code coverage, scalability, and ease of integration with other tools and libraries, providing a more comprehensive evaluation of testing frameworks.

- **Real-world case studies:** To better understand the practical implications of using different testing frameworks, future work could involve real-world case studies, where various testing frameworks are applied to actual Java Spring projects. This would provide insights into the challenges and benefits encountered during the implementation and maintenance of tests in diverse, real-world contexts.

- **Longitudinal studies:** As the software development landscape continues to evolve, it is important to track the changes and improvements in testing frameworks over time. Longitudinal studies could monitor the progress and development of various testing frameworks, offering valuable information for developers and teams when selecting a testing framework for their projects.

By addressing these areas in future research, we can further deepen our understanding of testing frameworks for Java Spring applications and provide development teams with more comprehensive and accurate information to make informed decisions on the best framework for their specific needs and projects.

References

[1] N. POPPER. "Knight capital's $440 million trading glitch." (2012), [Online]. Available: https://archive.nytimes.com/dealbook.nytimes.com/2012/08/02/knight-capital-says-trading-mishap-cost-it-440-million/ (visited on 05/22/2023).

[2] TIOBE. "Tiobe index." (), [Online]. Available: https://www.tiobe.com/tiobe-index.

[3] TIOBE. "Tiobe rating." (), [Online]. Available: https://www.tiobe.com/tiobe-index/programminglanguages_definition/.

[4] M. Kropp and P. Morales, "Automated gui testing on the android platform," in *On Testing Software and Systems: Short Papers*, 2010, p. 67.

[5] M. N. Huhns and M. P. Singh, "Service-oriented computing: Key concepts and principles," *IEEE Internet computing*, vol. 9, no. 1, pp. 75–81, 2005.

[6] M. A. Jamil, M. Arif, N. S. A. Abubakar, and A. Ahmad, "Software testing techniques: A literature review," in *2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M)*, Jakarta, Indonesia, 2016, pp. 177–182. DOI: 10.1109/ICT4M.2016.045.

[7] G. Aroush, *An evaluation of testing frameworks for beginners injavascript programming : An evaluation of testing frameworks with beginners in mind*, 2022.

[8] R. Dahlgren, *Ramverk för enhetstestning : För en eventuell kurs på mittuniversitetet*, 2022.

[9] Meiliana, I. Septian, R. S. Alianto, and Daniel, "Comparison analysis of android gui testing frameworks by using an experimental study," *Procedia Computer Science*, vol. 35, pp. 736–748, 2018. DOI: https://doi.org/10.1016/j.procs.2018.08.211.

[10] BrowserStack, *Top java testing frameworks every developer must know*, Accessed: 2023-04-03, 2021. [Online]. Available: https://www.browserstack.com/guide/top-java-testing-frameworks.

[11] E. Academy, *Top 10 java testing frameworks every developer should know*, Accessed: 2023-04-03, 2021. [Online]. Available: https://enlear.academy/top-10-java-testing-frameworks-every-developer-should-know-ef818457b08.

[12] LambdaTest, *Top 10 java testing frameworks for 2021*, Accessed: 2023-04-03, 2021. [Online]. Available: https://www.lambdatest.com/blog/top-10-java-testing-frameworks/.

[13] S. Khan, *13 best java testing frameworks for 2023*, https://www.lambdatest.com/blog/best-java-testing-frameworks/, Accessed on February 14, 2023, Jan. 2023.

[14] M. Shaw, "What makes good research in software engineering?" *International Journal on Software Tools for Technology Transfer*, vol. 4, pp. 1–7, 2002.

[15] JUnit. "Junit 5 user guide." (2022), [Online]. Available: https://junit.org/junit5/docs/current/user-guide/#overview-what-is-junit-5 (visited on 02/23/2023).

[16] TestNG Contributors, *TestNG Documentation*, https://testng.org/doc/, Accessed: February 22, 2023.

[17] TutorialsPoint, *TestNG Overview*, https://www.tutorialspoint.com/testng/testng_overview.htm, Accessed: February 22, 2023.

[18] Software Testing Help, *JUnit vs TestNG Comparison: Which is Better?* https://www.softwaretestinghelp.com/junit-vs-testng/#Conclusion, Accessed: February 22, 2023.

[19] Spock Framework Contributors, *Spock Framework Documentation*, https://spockframework.org/, Accessed: February 24, 2023.

[20] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects," in *The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing WORLDCOMP 2013; Las Vegas, Nevada, USA, 22-25 July*, CSREA Press USA, 2013, pp. 67–73. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:677684/FULLTEXT02.

[21] N. Andersson and A. Ekholm, *Vetenskaplighet - Utvärdering av tre implementeringsprojekt inom IT Bygg & Fastighet 2002*, svenska. Institutionen för Byggande och Arkitektur, Lunds Universitet, 2002.

[22] M. Bunge, *Epistemology & Methodology I:: Exploring the World*. Springer Science & Business Media, 2012, vol. 5.

[23] ISO/IEC/IEEE, "Ieee/iso/iec international standard - software and systems engineering–software testing–part 4: Test techniques," *ISO/IEC/IEEE 29119-4:2021(E)*, pp. 1–148, 2021. DOI: 10.1109/IEEESTD.2021.9591574.

[24] SoftwareTestingHelp. "Functional testing vs non-functional testing: What's the difference?" (2018), [Online]. Available: https://www.softwaretestinghelp.com/functional-testing-vs-non-functional-testing/ (visited on 02/22/2023).

[25] K. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*. John Wiley Sons Inc., 2008, p. 5.

[26] *Integration testing*, https://www.techtarget.com/searchsoftwarequality/definition/integration-testing, (Accessed on 2022-05-21).

[27] Guru99. "System testing." (n.d.), [Online]. Available: https://www.guru99.com/system-testing.html (visited on 02/22/2023).

[28] Guru99. "Types of software testing." (n.d.), [Online]. Available: https://www.guru99.com/types-of-software-testing.html (visited on 02/22/2023).

[29] S. T. Fundamentals. "Acceptance testing." (n.d.), [Online]. Available: https://softwaretestingfundamentals.com/acceptance-testing/ (visited on 02/22/2023).

[30] K. Jackvony. "An introduction to the automation test wheel." (Sep. 2019), [Online]. Available: https://www.ministryoftesting.com/articles/3250eb6c (visited on 02/23/2023).

[31] M. Cohn, *Succeeding with Agile: Software Development Using Scrum* (A Mike Cohen signature book). Addison-Wesley, 2010, ISBN: 9780321579362. [Online]. Available: https://books.google.com.ar/books?id=IdT6AgAAQBAJ.

[32] Agile Alliance. "What is Agile Software Development?" (2022), [Online]. Available: https://www.agilealliance.org/agile101 (visited on 02/25/2023).

[33] M. Rouse, *Behavior-driven development (bdd)*, https://www.techtarget.com/searchsoftwarequality/definition/Behavior-driven-development-BDD, Accessed: March 1, 2023.

[34] K. Petersen, C. Wohlin, and D. Baca, "The waterfall model in large-scale development," in *Proceedings of the 3rd ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, 2009, pp. 386–388. [Online]. Available: https://www.diva-portal.org/smash/record.jsf?pid=diva2:835760 (visited on 02/25/2023).

[35] S. W. Ambler, *Test driven development (tdd)*, http://www.agiledata.org/essays/tdd.html, Accessed on March 10, 2023, 2005.

[36] Agile Alliance. "Test-Driven Development (TDD)." (2022), [Online]. Available: https://www.agilealliance.org/glossary/tdd (visited on 03/01/2023).

[37] A. Tarlinder, *Developer Testing: Building Quality Into Software* (A Mike Cohn signature book). Addison-Wesley, 2016, ISBN: 9780134291062. [Online]. Available: https://books.google.se/books?id=bmDFjgEACAAJ.

[38] I. Sommerville, *Software Engineering*, 10th ed. Pearson, 2016.

[39] S. B. Contributors. "Testing - spring boot reference guide," Spring Boot. (Jun. 2017), [Online]. Available: https://docs.spring.io/spring-boot/docs/1.5.7.RELEASE/reference/html/boot-features-testing.html (visited on 02/28/2023).

[40] JUnit 5 Contributors, *Junit jupiter api assertions*, https://junit.org/junit5/docs/5.0.1/api/org/junit/jupiter/api/Assertions.html, Accessed: February 28, 2023, 2017.

[41] Mockito, *Mockito-testng*, https://github.com/mockito/mockito-testng, accessed 2023.

[42] *Visual studio code java pack*, Available online: https://marketplace.visualstudio.com/items?itemName=vscjava.vscode-java-pack.

[43] JBehave Contributors, *JBehave Documentation*, https://jbehave.org/, Accessed: February 23, 2023.

[44] Serenity BDD Contributors, *Serenity BDD Documentation*, https://serenity-bdd.info/docs/serenity/, Accessed: February 23, 2023.

[45] Selenium Contributors, *Selenium*, https://www.selenium.dev/, Accessed 10 May 2023.

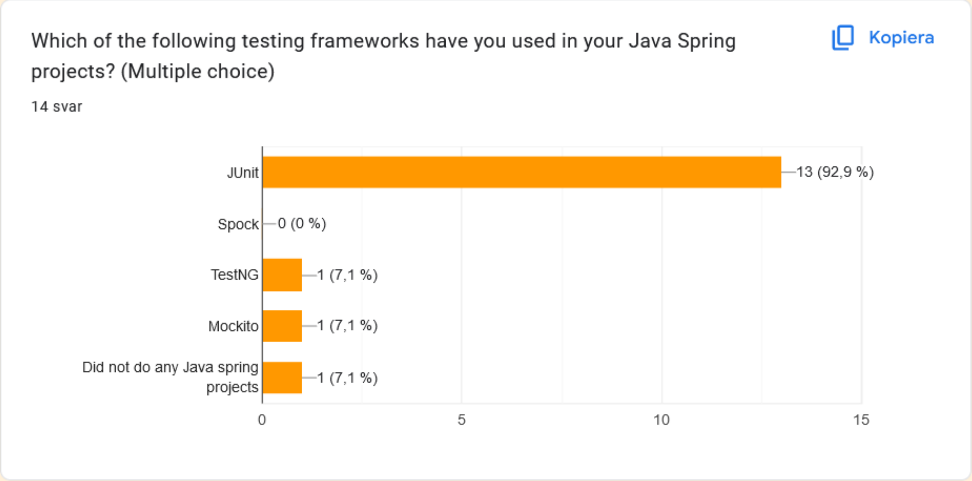
[46] Selenide Contributors, *Selenide Documentation*, https://selenide.org/, Accessed: February 23, 2023.

[47] Microsoft Contributors, *Gauge with End-to-End Testing*, https://microsoft.github.io/code-with-engineering-playbook/automated-testing/e2e-testing/recipes/gauge-framework/, Accessed: February 23, 2023.

[48] Geb Contributors, *Geb Documentation*, https://www.gebish.org/, Accessed: February 23, 2023.

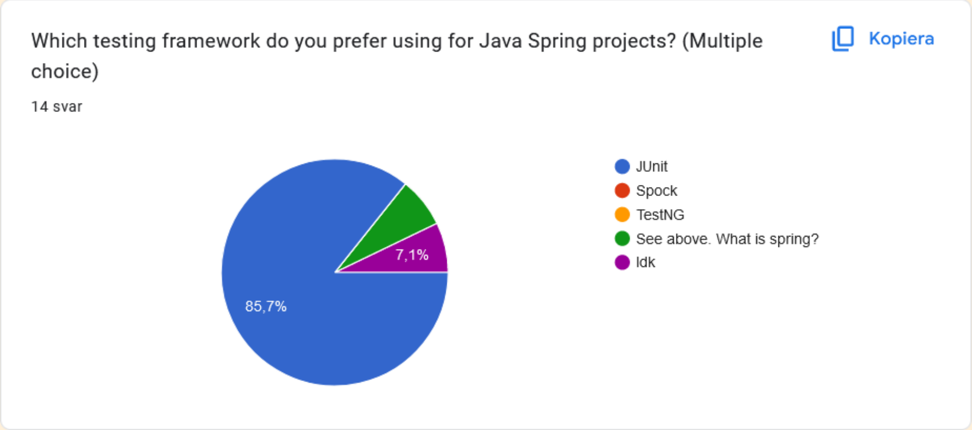
A Questions & Answers

A.1



Which of the following testing frameworks have you used in your Java Spring projects? (Multiple choice)

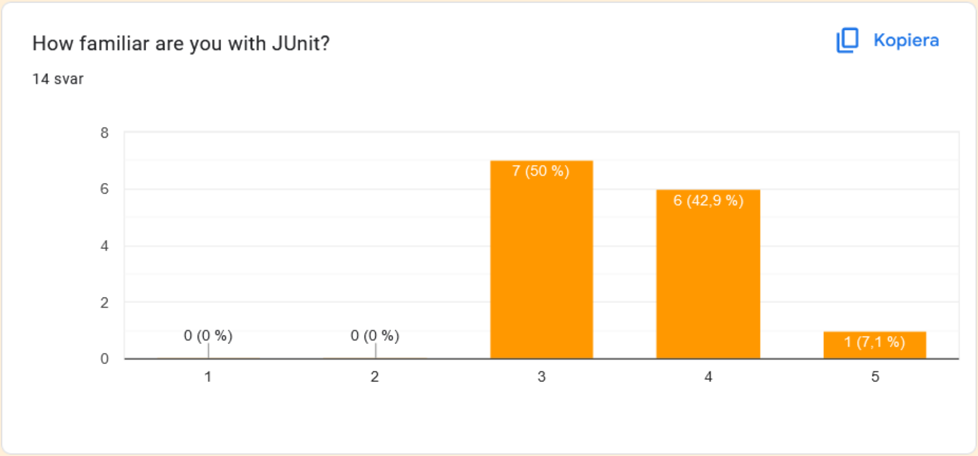
14 svar



Which testing framework do you prefer using for Java Spring projects? (Multiple choice)

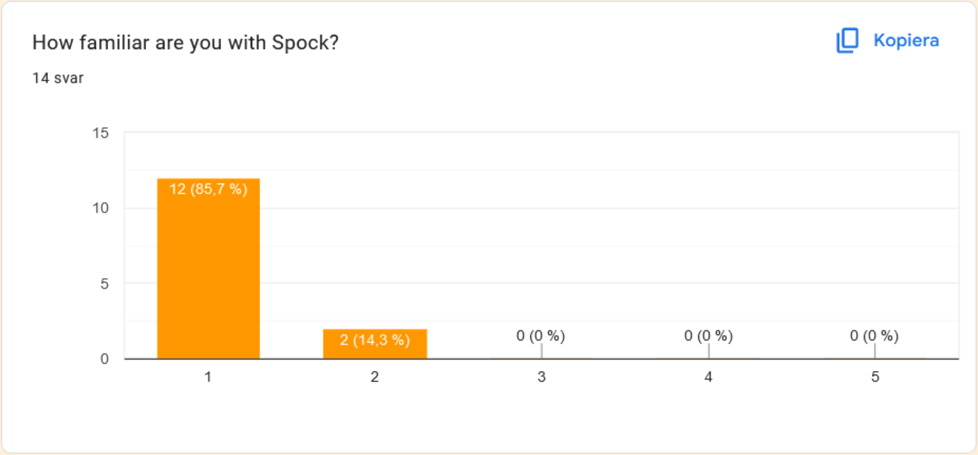
14 svar

A.2



How familiar are you with JUnit?

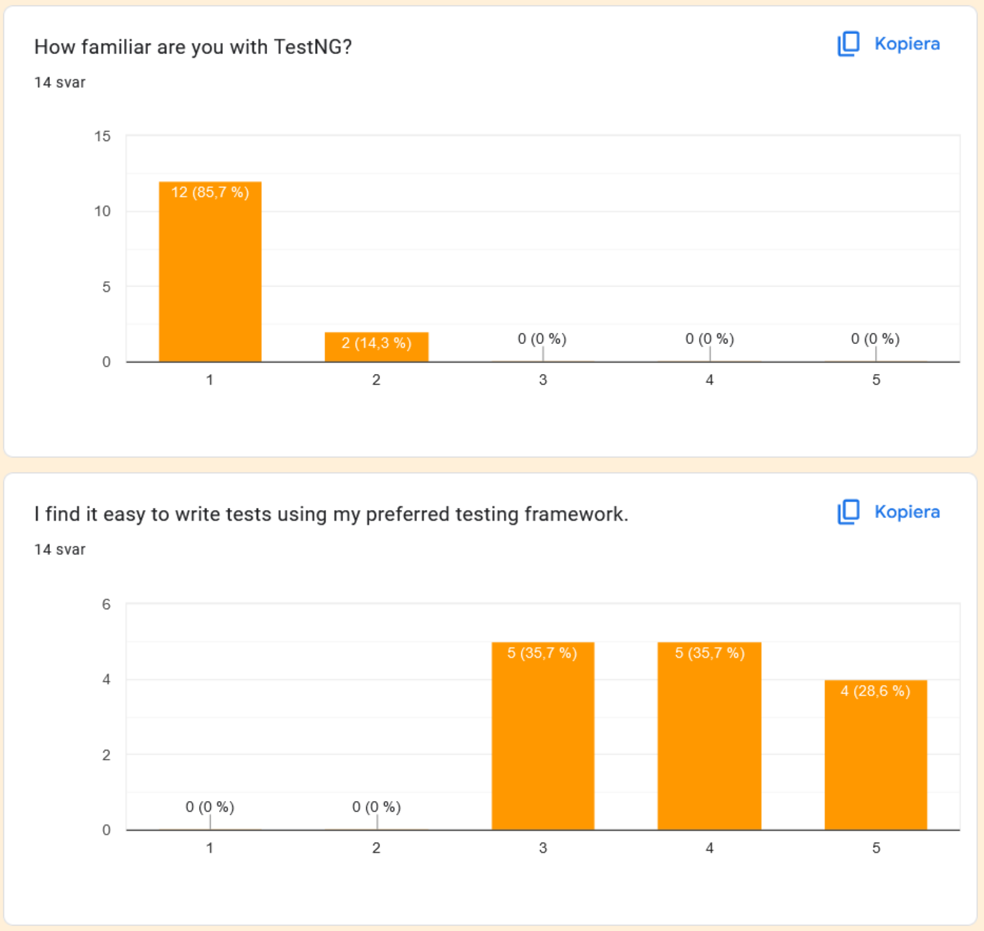
14 svar



How familiar are you with Spock?

14 svar

A.3



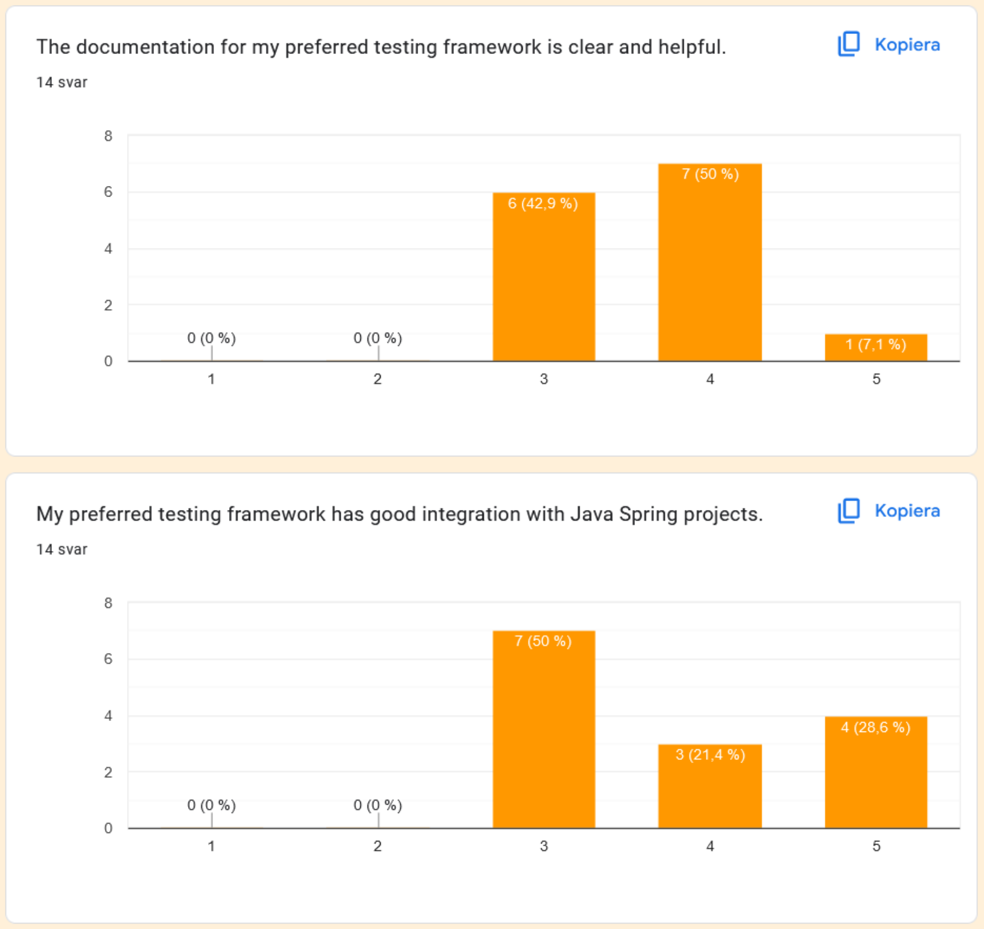
How familiar are you with TestNG?

14 svar

I find it easy to write tests using my preferred testing framework.

14 svar

A.4



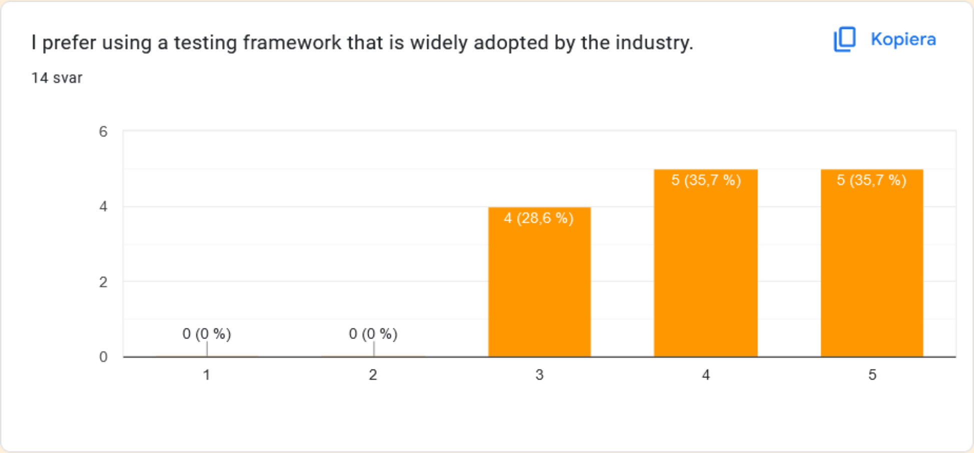
The documentation for my preferred testing framework is clear and helpful.

14 svar

My preferred testing framework has good integration with Java Spring projects.

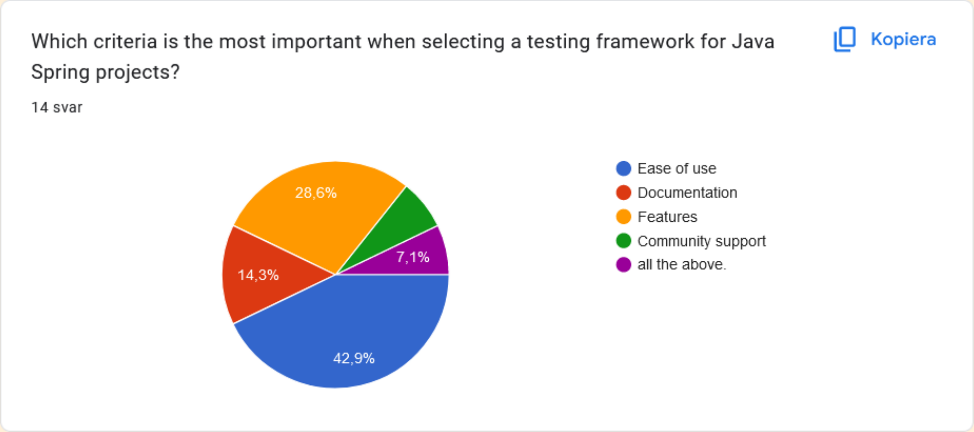
14 svar

A.5



I prefer using a testing framework that is widely adopted by the industry.

14 svar



Which criteria is the most important when selecting a testing framework for Java Spring projects?

14 svar

B Testing frameworks not included

B.1 JBehave

JBehave is a framework used for behaviour-driven development (BDD) that provides a way of writing tests in a way that non-technical stakeholders find it easier understanding. It shifts the vocabulary from test-based to behaviour-based [43].

```
Scenario: Addition
Given a calculator
When I add 2 and 3
Then the result should be 5
```

B.2 Serenity

Serenity is a open source framework built on top of JUnit that helps with automating acceptance tests, this is often done by automating UI actions such as clicks and scrolls on a website. It also helps generating BDD-style living documentation [44].

```
@Step
void testAddition() {
    Calculator calculator = new Calculator();
    int result = calculator.add(2, 3);
    assertThat(result).isEqualTo(5);
}
```

B.3 Selenium

Selenium is a test-automating framework that is focused on bringing a clear and intuitive API for creating reliable tests. It features robust selectors and straightforward setup, as well as features such as implicit waiting and support for managing asynchronous operations [45].

```
void testAddition() {
    WebDriver driver = new ChromeDriver();
    driver.get("https://example.com/calculator");

    CalculatorPage calculatorPage = new
   CalculatorPage(driver);
    calculatorPage.add(2, 3);
    calculatorPage.assertResult("5");

    driver.quit();
}
```

B.4 Selenide

Selenide is a test-automation framework powered by Selenium WebDriver that aims on bringing a concise and expressive API for writing stable tests. It includes powerful selectors and simple configuration and aims to simplify the process of writing selenium tests. It includes features such as automatic waiting and built in support of handling asynchronous operations [46].

```
void testAddition() {
    CalculatorPage calculatorPage =
   open("https://example.com/calculator",
   CalculatorPage.class);
    calculatorPage.add(2, 3);
    calculatorPage.assertResult("5");
}
```

B.5 Gauge

Gauge is a framework for writing and running end to end (E2E) tests. A few of the key features of gauge include simple, flexible and rich syntax that is based on markdown. The framework provides consistent cross-platform/language support and is based on a modular architecture with plugin support [47].

```
## Addition
* Calculator.add(2, 3) -> 5
```

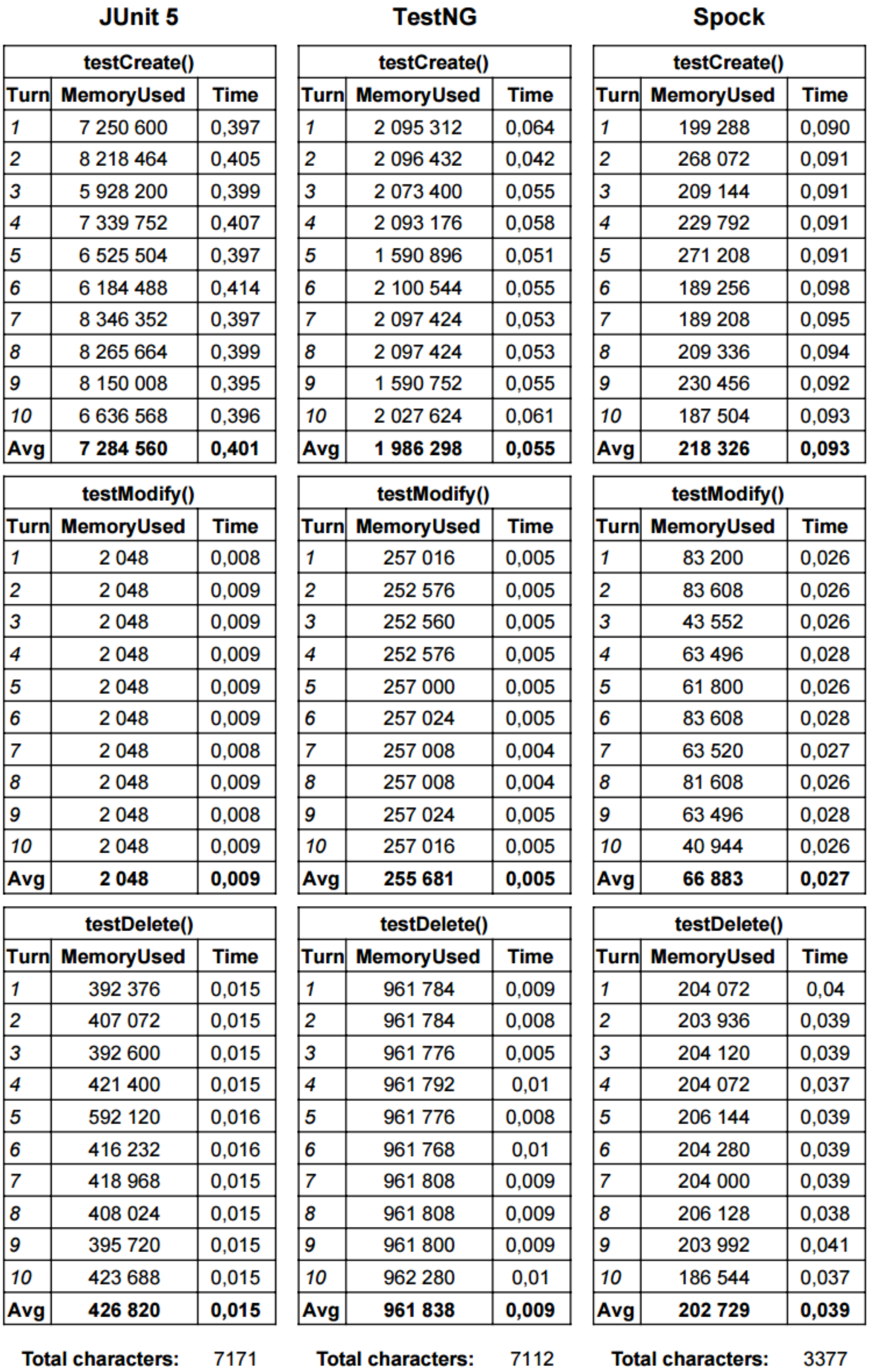
B.6 Geb

Geb is a browser automation library that aims at providing a more expressive and concise API than Selenium. The framework includes features such as automatic waiting, page objects and built in support for handling Ajax requests. The following snippet is taken from the official website of Geb: "It brings together the power of WebDriver, the elegance of jQuery content selection, the robustness of Page Object modelling and the expressiveness of the Groovy language." [48].

```
void testAddition() {
    CalculatorPage calculatorPage = to CalculatorPage
    calculatorPage.add(2, 3)
    assert calculatorPage.result.text() == "5"
}
```

C Results

C.1



JUnit 5			TestNG			Spock		
testCreate()			testCreate()			testCreate()		
Turn	MemoryUsed	Time	Turn	MemoryUsed	Time	Turn	MemoryUsed	Time
1	7 250 600	0,397	1	2 095 312	0,064	1	199 288	0,090
2	8 218 464	0,405	2	2 096 432	0,042	2	268 072	0,091
3	5 928 200	0,399	3	2 073 400	0,055	3	209 144	0,091
4	7 339 752	0,407	4	2 093 176	0,058	4	229 792	0,091
5	6 525 504	0,397	5	1 590 896	0,051	5	271 208	0,091
6	6 184 488	0,414	6	2 100 544	0,055	6	189 256	0,098
7	8 346 352	0,397	7	2 097 424	0,053	7	189 208	0,095
8	8 265 664	0,399	8	2 097 424	0,053	8	209 336	0,094
9	8 150 008	0,395	9	1 590 752	0,055	9	230 456	0,092
10	6 636 568	0,396	10	2 027 624	0,061	10	187 504	0,093
Avg	**7 284 560**	**0,401**	**Avg**	**1 986 298**	**0,055**	**Avg**	**218 326**	**0,093**

testModify()			testModify()			testModify()		
Turn	MemoryUsed	Time	Turn	MemoryUsed	Time	Turn	MemoryUsed	Time
1	2 048	0,008	1	257 016	0,005	1	83 200	0,026
2	2 048	0,009	2	252 576	0,005	2	83 608	0,026
3	2 048	0,009	3	252 560	0,005	3	43 552	0,026
4	2 048	0,009	4	252 576	0,005	4	63 496	0,028
5	2 048	0,009	5	257 000	0,005	5	61 800	0,026
6	2 048	0,009	6	257 024	0,005	6	83 608	0,028
7	2 048	0,008	7	257 008	0,004	7	63 520	0,027
8	2 048	0,009	8	257 008	0,004	8	81 608	0,026
9	2 048	0,008	9	257 024	0,005	9	63 496	0,028
10	2 048	0,009	10	257 016	0,005	10	40 944	0,026
Avg	**2 048**	**0,009**	**Avg**	**255 681**	**0,005**	**Avg**	**66 883**	**0,027**

testDelete()			testDelete()			testDelete()		
Turn	MemoryUsed	Time	Turn	MemoryUsed	Time	Turn	MemoryUsed	Time
1	392 376	0,015	1	961 784	0,009	1	204 072	0,04
2	407 072	0,015	2	961 784	0,008	2	203 936	0,039
3	392 600	0,015	3	961 776	0,005	3	204 120	0,039
4	421 400	0,015	4	961 792	0,01	4	204 072	0,037
5	592 120	0,016	5	961 776	0,008	5	206 144	0,039
6	416 232	0,016	6	961 768	0,01	6	204 280	0,039
7	418 968	0,015	7	961 808	0,009	7	204 000	0,039
8	408 024	0,015	8	961 808	0,009	8	206 128	0,038
9	395 720	0,015	9	961 800	0,009	9	203 992	0,041
10	423 688	0,015	10	962 280	0,01	10	186 544	0,037
Avg	**426 820**	**0,015**	**Avg**	**961 838**	**0,009**	**Avg**	**202 729**	**0,039**

Total characters:	7171	**Total characters:**	7112	**Total characters:**	3377