Master's thesis

Master's Programme in Computer Science

In-depth comparison of BDD testing frameworks for Java

Jone Lång

November 11, 2022

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Jone Lång

Työn nimi — Arbetets titel — Title

In-depth comparison of BDD testing frameworks for Java

Ohjaajat — Handledare — Supervisors

Dr. Antti-Pekka Tuovinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 11, 2022	52 pages, 14 appendix pages

Tiivistelmä — Referat — Abstract

Test automation has a crucial role in modern software development. Automated tests are immensely helpful in quality assurance, catching bugs and giving information on the state of the software. There are many existing frameworks that are designed to assist in creating automated tests. The frameworks can have massively varying purposes and targeted applications and technologies. In this paper, we aim to study a selected group of Behavior Driven Development (BDD) testing frameworks, compare them, identify their strengths and shortcomings, and implement our own testing framework to answer the discovered challenges. Finally, we will evaluate the resulting framework and see if it can meet its requirements. As a result we'll have a better understanding in what kind of tools there are for automating behavior driven tests, what type of different approaches have been and can be taken to implement such frameworks, and what are the benefits and suitable uses of each tool.

ACM Computing Classification System (CCS)
Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → **Software testing and debugging**

Software and its engineering → Software notations and tools

Avainsanat — Nyckelord — Keywords

Software testing, Test automation, BDD, Java

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction

BDD is a software development strategy based on writing automated acceptance tests before implementing the actual feature. The tests are specifically made to be understandable for people in different roles and written in a way that emphasizes the behavior of the system. There are various test automation tools and frameworks that support BDD. The tools have varying purposes, and each has its limitations. The goal of this study was to inspect and compare some of these tools, learn about different approaches to implementing a BDD framework and find out the strengths and weaknesses of each solution.

This study was conducted in the form of design science. Design science research (DSR) is a problem-solving paradigm that seeks to enhance human knowledge by creating new innovative artifacts and generating design knowledge. In DSR, environment refers to the problem space in which the phenomena under study reside (Brocke et al., 2020). In the case of this study, the environment consists of software developers and companies that are leveraging Behavior Driven Development (BDD) in their development process, as well as the existing BDD automation frameworks. Based on the identified weaknesses we saw a need for new BDD testing tool. From tools that support the Java language, probably the most popular ones are Cucumber (*Cucumber*) and JBehave (*JBehave*). Other frequently used and mentioned tools that are similar in design include Concordion (*Concordion*) and a newer tool called Gauge (*Gauge*). In addition, the tools Spock (*Spock*) and easyb (*easyb*) were considered, but ultimately left out as they require test specification to be written in Groovy.

We chose to set the scope on pure Java implementations and focused on the four first-mentioned tools going forward. The four frameworks were first studied individually and then compared. From the comparison we identified some differences between the tools. Each tool was found to have its shortcomings. It was also noted that all of the four tools shared some common weaknesses. Based on these findings, we built a new framework to answer the challenges. Finally, the new framework was evaluated. The new framework was found to meet most of the requirements that were set for it and can be used as an alternative way of implementing BDD. The new tool is comparatively simple, and also lacking in terms of supported features and quality, as it was implemented in a rather short period of time. It might be more sensible to view the tool more as a proof-of-concept than

an actual mature framework.

Chapter 2 introduces software testing and BDD to give a better understanding on BDD and BDD frameworks. In chapter 3 we inspect each tool individually, examining their implementation and features. In chapter 4 we do a comparison between the tools to identify potential shortcomings and limitations. In chapter 5 we use these findings as requirements to construct a new BDD testing framework that attempts to answer the potential challenges of the existing frameworks under study. Chapter 6 is used to verify that the requirements for the new framework are met, and to compare the new tool with the old ones. Finally, chapter 7 summarizes the findings and results of this study.

2 Software testing and Behavior Driven Development

This chapter starts by discussing software testing and test automation in general. It later introduces Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD), which are both predecessors of BDD and a crucial part of its emergence. The later half of the chapter is focused on introducing BDD, its central concepts, and common requirements it sets for BDD frameworks.

2.1 Introduction to software testing

Software testing has an important role in developing software. Testing is one of the most expensive and laborious tasks in software projects. However, in the long run it can save in costs by preventing bugs and errors from entering production (Kasurinen et al., 2010). The role of testing can be said to only increase as the requirements and standards for software grow. There are lots of different ways to categorize testing, perhaps the most basic of which is to split it into manual and automated testing. As the size and complexity of software becomes ever larger, manual testing alone is usually not enough to achieve sufficient quality control. Manual testing refers to a more traditional form of testing, where the person executing the tests has to manually use the program and perform the needed steps to ensure that the program works as intended. Manual testing is slow and therefore expensive. It also doesn't allow inspecting the internal structures of the program during its execution. It is however an excellent way to find bugs and errors, or to verify that some specific feature works (Kasurinen et al., 2010). Humans' creativity and grasp of context helps manual testers perform good, sensible tests and identify issues. Thus, manual testing is still an important part of the testing process, which does not seem to be changing any time soon.

The second approach to software testing is test automation. With automation, tests can be performed quickly and efficiently, always exactly the same way, and relatively reliably. Once the test has been created, it can be executed regularly, even daily, and it will provide valuable information about the state of the software. Once the test has been implemented,

it doesn't usually need to be touched too much afterwards, but occasionally it might need to be updated as the requirements or behavior of the software changes. The development and decline in price of cloud services and computing resources make it possible to run large test suites efficiently and regularly, making test automation an even more tempting option. Furthermore, test automation enables so called white-box testing, where the functionality of the inner constructs of a program is tested by leveraging knowledge of its code and implementation details (Nidhra and Dondeti, 2012). Test automation is rarely suitable for tasks that are seldom performed, as the creation of automated tests is slow and troublesome so the up-front cost is high (Kasurinen et al., 2010). The value of automation lies in repetition, generating value in the long run.

As mentioned earlier, there are more ways to categorize testing. For example, testing can be further divided into unit testing, integration testing, system testing and acceptance testing (Leung and Wong, 1997). Unit tests test a single class or component (Leung and Wong, 1997), while integration tests test the interactions between modules and their interfaces (Leung and White, 1990). These are both white-box testing methods, although Nidhra et al. state that black-box testing techniques can be applied to integration tests too (Nidhra and Dondeti, 2012). System tests can be seen as larger integration tests that also include the user interface, and use it to test the software similarly to how an end user would. They test end to end functionality of the software as a whole, based on its specifications (Briand and Labiche, 2002). Acceptance tests are used to verify that the application meets the business requirements of a specification (Miller and Collins, 2001). They also attempt to ensure that the system works correctly from the customer's point of view (Viktor and Alex, 2018a). Both system testing and acceptance testing are in turn black-box testing methods, where the test code has no visibility of or knowledge about the implementation details of the system being tested (Nidhra and Dondeti, 2012).

2.2 Using tests to drive software development

There are also many ways to utilize test automation in software projects. Test Driven Development (TDD) is a software development strategy that is based on writing automated unit tests prior to implementing the actual functional code. It relies on very short iterations of writing tests before writing code, refactoring and continuous integration. New code should only be written if an automated test fails. TDD aims to improve code quality and reduce code duplication (Janzen and Saiedian, 2005).

Acceptance Test Driven Development (ATDD) is a type of TDD where acceptance tests drive the development process. These tests can be automated and are used to represent stakeholders' requirements. ATDD can help developers transform requirements into test cases and verify the system's functionality. A requirement is satisfied once all acceptance criteria or its associated tests are satisfied (Solis and Wang, 2011). TDD and ATDD are widely used as they have been shown to improve software quality and productivity (Janzen and Saiedian, 2008; Gupta and Jalote, 2007). However, both TDD and ATDD still have their share of issues. Many developers get confused about where to start, what to test, what not to test, how much to test, how to understand why a test fails and what to call their tests (Terhorst-North, 2006). What is more, both TDD and ATDD are focused on verifying the state of the system rather than its desired behavior. Another issue is that the test code is tightly coupled with the systems' implementation (Solis and Wang, 2011). In addition, these approaches use unbounded and unstructured natural language to describe test cases, making them difficult to understand (Terhorst-North, 2006).

2.3 What is BDD?

Behavior Driven Development (BDD), which emerged from TDD, can be regarded as the evolution of TDD and ATDD (Solis and Wang, 2011). BDD can also be described as a flavor of TDD (Viktor and Alex, 2018b). It's an "outside-in" methodology that starts by identifying business outcomes, and then drills down into the features that will achieve those outcomes. BDD is built on the idea that an idea for a requirement can be turned into implemented, tested and production-ready code effectively, as long as the requirement is specific enough that everyone knows what to do and from which they can all agree a common definition of done (Terhorst-North, 2007). BDD is based on the same basic principle of writing tests before the implementation code (Viktor and Alex, 2018b). A major difference between TDD and BDD is the life cycle duration. In TDD, the failing unit tests are fixed in a very quick manner, while in BDD it often takes hours or days to get from a failing test to a passing one (Viktor and Alex, 2018b). Another thing that separates TDD and BDD is the target audience. While TDD and its unit tests are aimed at developers, BDD intends to involve a wider group of people in different roles (Viktor and Alex, 2018b). BDD emphasizes behavior over technical implementation, focusing on defining fine-grained specifications of the behavior of the System Under Test (SUT) (Solis and Wang, 2011). Its main goal is to derive executable specifications of a system

(Terhorst-North, 2006; Solis and Wang, 2011). Although BDD relies on ATDD, it has some advantages over the latter. In BDD, tests are clearly written and easy to understand, since BDD provides a specific ubiquitous language for specifying tests (Solis and Wang, 2011). BDD can help close the gap between business people and technical people by encouraging collaboration across roles, increasing feedback and producing documentation that is automatically checked against the behavior of the system (*Behaviour-Driven Development*). It drives the development and makes it easier to understand what should be done (Viktor and Alex, 2018b).

BDD can be applied to different levels of testing. Its principles can be used even on unit testing level, but its benefits are better on higher levels of testing where the tests can be written and understood by everyone. While TDD can be described as an inside-out approach that starts building up from units towards functionalities, BDD can be seen as outside-in, as it starts with features and builds towards units (Viktor and Alex, 2018b).

BDD uses a story as the basic unit of functionality and delivery. Stories are the result of conversations between project stakeholders, business analysts, developers and testers. A story encompasses a feature and defines its scope and acceptance criteria (Terhorst-North, 2007). It's a description of a requirement and its business benefit, as well as a set of criteria by which everyone can agree that it's done. BDD provides a structure for formatting a story. The template can be seen in figure 2.1. A story consists of a title, a narrative and acceptance criteria, which are realised as scenarios. The title should describe an activity, and the narrative should include a role, a feature and a benefit. The scenarios consists of three types of steps: context steps, event steps, and outcome steps, represented by the keywords *given, when* and *then*. The scenario titles should say how the scenarios differ from one another (Terhorst-North, 2007). Figure 2.2 shows a concrete example of a story with two scenarios. The story is about an account holder withdrawing cash from an ATM. From the two scenario titles we can see that the differences between them is whether the account has sufficient funds, leading to different outcomes. In reality, there would likely be more scenarios to consider, but the example is kept short for the sake of simplicity.

```
Title (one line describing the story)

Narrative:
As a [role]
I want [feature]
So that [benefit]

Acceptance Criteria: (presented as Scenarios)

Scenario 1: Title
Given [context]
  And [some more context]...
When  [event]
Then  [outcome]
  And [another outcome]...

Scenario 2: ...
```

Figure 2.1: An example story template by Daniel Terhorst-North (Terhorst-North, 2007)

```
Story: Account Holder withdraws cash

As an Account Holder
I want to withdraw cash from an ATM
So that I can get money when the bank is closed

Scenario 1: Account has sufficient funds
Given the account balance is $100
 And the card is valid
 And the machine contains enough money
When the Account Holder requests $20
Then the ATM should dispense $20
 And the account balance should be $80
 And the card should be returned

Scenario 2: Account has insufficient funds
Given the account balance is $10
 And the card is valid
 And the machine contains enough money
When the Account Holder requests $20
Then the ATM should not dispense any money
 And the ATM should say there are insufficient funds
 And the account balance should be $10
 And the card should be returned
```

Figure 2.2: An example of a concrete story by Daniel Terhorst-North (Terhorst-North, 2007)

To support the use of BDD in projects, multiple BDD testing frameworks have been created. These frameworks support the BDD syntax and produce a human-readable output. Most such tools also use ubiquitous language as their inputs, meaning that the tests themselves are defined in plain text, often with a restricted syntax. The tools can have varying purposes; some tools are designed for unit testing while others are mainly intended for system testing. BDD frameworks aim to make it easier to adopt BDD in the development process and create tests as defined by BDD.

BDD tools have some key pieces they require to function. If the tool uses plain text

for creating tests, as is the case for many BDD tools, there needs to be some kind of mapping rules to map the textual steps to executable methods. This is usually done by annotating the step implementation methods with a step-annotation that has a string value. The annotation value can then be used to find the correct method to execute for each step. In order to use parameters in steps, the textual values need to be converted to the appropriate type of objects. This is done via parameter converters, that create the objects that will be used when calling the actual step methods. To be able to run the tests, these tools need a test runner. A test runner offers the functionality needed to execute the tests and control the execution. Normally test runners will also output some information about what is being executed and the test results. Another common requirement is a test reporter, that will generate a summary of what was executed and what are the results of each test execution.

In addition, BDD frameworks are often configurable and in some cases a lot of the configuration is required from the user for the tool to function. Examples could be configuring the path to the step implementations so that the tool knows where to look for them, specifying reporting tools, or adding customised parameter converters to support custom types. For BDD frameworks to be maintainable and easier to use, they often rely on IDE plugins to help navigating between stories and step implementations within them.

Automated tests are normally executed regularly in an automated fashion as part of the CI/CD pipeline. What is more, the tests are usually run in large groups referred to as test suites, where a large amount of tests is executed, partly in parallel. To support this, BDD frameworks utilise build tools or separate test runners so that the tests can be run via simple command-line commands. The frameworks support the creation of test suites and are normally implemented so that parallel execution is possible.

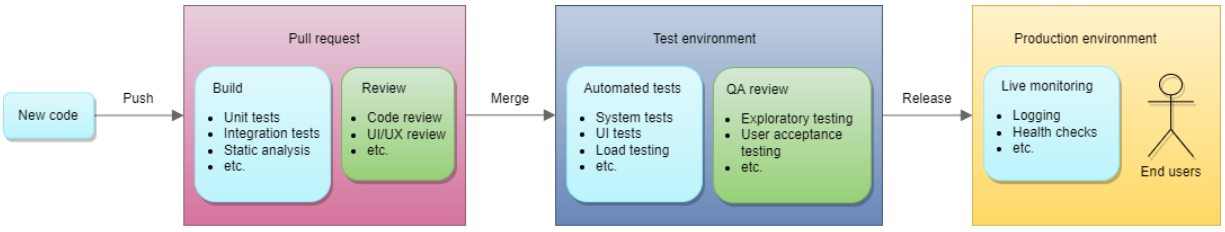

Figure 2.3: Example of a CI/CD pipeline.

Figure 2.3 illustrates a somewhat typical CI/CD pipeline. In the context of this example, the difference between continuous integration and continuous delivery is the transition from the test environment to the production environment. In continuous integration this is done manually, while in continuous delivery the process is automated and often more

frequent. As seen in the figure, unit tests, and sometimes integration tests as well, are run when building the project with the new code changes. It is possible that these lower level tests are written as BDD tests. The new changes are then reviewed. Later on in the Automated tests phase the slower, higher level tests are run. These typically contain the tests where BDD is most utilised, as its abstraction and universal language is perhaps most convenient to apply on higher levels of testing. It is also reasonable to place integration tests in this phase if the test suite is large. After the automated tests, manual testing processes are executed before the changes are deployed to production. Some type of monitoring is typically performed in production environments to further enhance the feedback loop.

With test suites and simple run commands, including BDD to tests to the pipeline is straightforward. Parallel execution helps speed up the pipeline and hence shortens the feedback loop, so that any issues can be resolved faster.

3 Investigating BDD testing frameworks

In this chapter each of the four studied BDD frameworks is inspected individually. The goal is to learn about their implementation and features to be able to do an in-depth comparison in the next chapter.

3.1 JBehave

JBehave is a popular open source BDD framework for Java (*JBehave*). It is used for creating automated acceptance tests with textual user stories written in a BDD syntax. JBehave was originally written to be a replacement for JUnit that would remove any reference to testing and replace it with a vocabulary built around verifying behavior (Terhorst-North, 2006).

3.1.1 Implementation

JBehave is a pure Java implementation that uses special textual story files ending in *.story* to represent user stories. The stories consist of meta data, a title, a narrative, one or more scenarios and human-readable statements that JBehave refers to as elements (*JBehave*). There are two major components that are required for JBehave to run the stories - runners and steps (Viktor and Alex, 2018b). A runner or run configuration is a class that is used to parse a story, run all the scenarios, and generate a test report. It acts as the entry point to a test and is the actual executable file required to run the test. Steps are Java methods that match the written steps in scenarios (Viktor and Alex, 2018b). JBehave uses annotations to bind the textual steps in Story-files to Java methods in Steps-files (*JBehave*).

Stories can be written in either the Gherkin syntax or a custom JBehave syntax. Gherkin is a business readable domain specific language used by multiple BDD frameworks, that is developed by Cucumber. It enables the writing of specifications in plain English (Härlin, 2016). The JBehave syntax is very similar, and supports a lot of the same features. It also has some additional features related to defining the scope of an element, namely being able to set the scope on story and step levels in addition to scenario level, as well as supporting

After-elements, making it possible to execute code mapped to an element after the given scope in addition to the default way of executing it beforehand (*JBehave*). JBehave stories can also contain arbitrary meta data.

Figure 3.1 shows an example of a JBehave story file. The story starts off with a textual narrative, that is only used to enhance readability. It then defines a single scenario for calculating the sum of two numbers. The following three lines, starting with "Given", "When" and "Then", are steps that are mapped to Java methods annotated with a matching String.

```
Narrative:
As a user
I want to use a calculator for calculations
So that I don't have to do them by hand

Scenario: The calculator can calculate the sum of two numbers correctly
Given the user inputs the numbers 1 and 2
When the user chooses the sum function
Then the result is 3
```

Figure 3.1: An example of calculator_story.story with a single scenario

Figure 3.2 shows a Java file that implements the steps used in the example scenario. The class contains three methods, each marked with an annotation that matches the textual steps in the story files. The annotations can also contain parameters, marked with a dollar-sign ($). The method *inputNumbers* takes two integers as parameters, and calls the input function of a Calculator object with the same parameters. The *chooseFunction* method maps the *functionName* parameter to the appropriate method in the Calculator class, and calls that method on the calculator object. The method *checkResult* does an assertion to verify that the actual results calculated and stored in the *chooseFuntion* method is equal to the expected value received as a parameter.

```java
public class CalculatorSteps {

  private Calculator calculator = new Calculator();
  private int actualResult;

  @Given("a user input the numbers $num1 and $num2")
  public void inputNumbers(int num1, int num2) {
    calculator.input(num1, num2);
  }

  @When("the user chooses the $functionName function")
  public void chooseFunction(String functionName) {
    if (functionName.equals("sum")) {
      actualResult = calculator.sum();
    } else {
      // TODO: Implement other functions
    }
  }

  @Then("the result is: $expectedResult")
  public void checkResult(int expectedResult) {
    Assert.assertEquals(expectedResult, actualResult);
  }
}
```

Figure 3.2: Step implementations for the steps used in the story file

Figure 3.3 shows the run configuration file that functions as the entry point for the test. The class extends another class named *JUnitStory* that inherits a JUnit-runnable method named *run*, which is called to execute the test. *CalculatorStory* overrides the *configuration* and *stepsFactory* methods from a class named *ConfigurableEmbedder*. The *Configuration* class provides a variety of ways to customise how stories and steps are located, parsed and run. The *stepsFactory* method is used for creating instances of steps classes that provide the implementations for the steps used in the story. JBehave will find all methods and their annotations within these steps classes and use them to create "step candidates". When running the tests, JBehave will look for matching step definitions only within these candidates.

```
public class CalculatorStory extends JUnitStory {

  @Override
  public Configuration configuration() {
    return new MostUsefulConfiguration();
  }

  @Override
  public InjectableStepsFactory stepsFactory() {
    return new InstanceStepsFactory(
      configuration(),
      new CalculatorSteps()
    );
  }
}
```

Figure 3.3: Run configuration file for the calculator story

3.1.2 Features

JBehave stories can be either classpath resources or external URL-based resources. Stories may be written in any language, as long as you provide custom localized keywords to replace the default English expressions in your configuration. In case a story contains a missing step, JBehave can auto-generate pending steps so that the build is not broken, but this functionality is optional. In JBehave, the matching of a step candidate to the textual steps is a crucial part of its design. Normally the first matching candidate is used to create an executable step, but sometimes a textual step may be mapped to more than one candidates, which can cause problems. To handle such situations, JBehave allows for a customisable prioritising strategy to help decide which step should be executed. JBehave supports auto-conversion of string arguments to any parameter type via custom parameter converters. By default it provides parameter converters for booleans, numbers, strings, lists of strings, dates, enums, JBehave-specific objects called "ExamplesTable", JSON-data and a list of Java 8 date and time types (*JBehave*).

JBehave stories can be run as JUnit (*JUnit*) tests or other annotation-based unit testing frameworks, which enables easy IDE integration (*JBehave*). JBehave provides integration with Ant (*Apache Ant*) and Maven (*Apache Maven*) build tools. It also supports con-

currency, allowing stories to be executed on multiple threads simultaneously (*JBehave*). JBehave provides extensible test reports in HTML, TXT and XML formats, as well as a text-based console output (Okolnychyi and Fögen, 2016; *JBehave*). In addition it supports story report cross-referencing in JSON and XML formats. JBehave also supports annotation-based Steps class specifications and configuration (*JBehave*). JBehave also allows dependency injection to compose configuration and Steps instances via containers like Guice (*Guice*), Needle (*Needle*) or Weld (*Weld*). There is also support for writing configuration and Steps instances with Groovy (*JBehave*).

3.1.3 Analysis

From the extensive list of features we can observe that a big advantage of JBehave is its customisability and configurability. JBehave also has comprehensive documentation to explain the usage of these features. All of the possible configurations are also a weakness in the sense that because of them JBehave is not the easiest framework to use or understand. The fact that each test case requires both a textual story file and a configuration file can be seen as another weakness, since creating step instances and configurations separately for each test splits the test across multiple files, causing unnecessary complexity.

3.2 Cucumber

Cucumber is another popular open source BDD framework originally written in Ruby that also has implementations for Java and JavaScript (*Cucumber*). It can also be used with other languages, such as Python and C# (Lenka et al., 2018). In the context of this paper, we will be focusing on the Java implementation known as Cucumber JVM. Cucumber is designed for BDD and is used to create automated acceptance tests which are represented as textual files written in BDD syntax.

3.2.1 Implementation

Similarly to JBehave, Cucumber uses textual files to represent test cases. In Cucumber's case these are these are called feature files, possessing a *.feature* file extension (Lenka et al., 2018; *Cucumber*). The feature files are written in the Gherkin syntax (*Cucumber*). Cucumber feature files contain a feature description and one or more scenario descriptions,

which consist of textual steps that are mapped to corresponding annotated methods. The features act as executable test scripts in themselves. Cucumber utilizes command-line arguments to find out the location of steps methods. This means that if the project is setup correctly, no separate run configurations are needed. Cucumber will load all step definitions from each file in the given package before the test execution starts. If a match is found, the method is executed with appropriate arguments and the step is marked green to indicate success. If not, the step is marked yellow, indicating undefined, and all subsequent steps are then skipped. Other possible states for steps are pending (if marked as pending with an annotation), failed (when an error is raised), or ambiguous (when there are two or more step matching step definitions) (*Cucumber*).

Figure 3.4 shows an example of a Cucumber feature file. The file defines the name of the feature and a narrative in BDD standard. It then defines a single scenario for calculating the sum of two numbers. The following three lines, starting with "Given", "When" and "Then", are steps that are mapped to Java methods annotated with a matching String.

```
Feature: Calculator
  As a user
  I want to use a calculator for calculations
  So that I don't have to do them by hand

  Scenario: The calculator can calculate the sum of two numbers correctly
    Given the user inputs the numbers 1 and 2
    When the user chooses the sum function
    Then the result is 3
```

Figure 3.4: A Cucumber-equivalent example of calculator_feature.feature with a single scenario

Figure 3.5 shows a Java file that implements the steps used in the example scenario. The implementation is almost identical to the JBehave example presented in figure 3.2, the only difference being the way parameters are marked in annotations. JBehave uses the dollar-sign ($) for this purpose, while Cucumber uses curly brackets ({}).

```
public class CalculatorSteps {

  private Calculator calculator = new Calculator();
  private int actualResult;

  @Given("a user input the numbers {int} and {int}")
  public void inputNumbers(int num1, int num2) {
    calculator.input(num1, num2);
  }

  @When("the user chooses the {string} function")
  public void chooseFunction(String functionName) {
    if (functionName.equals("sum")) {
      actualResult = calculator.sum();
    } else {
      // TODO: Implement other functions
    }
  }

  @Then("the result is: {int}")
  public void checkResult(int expectedResult) {
    Assert.assertEquals(expectedResult, actualResult);
  }
}
```

Figure 3.5: Step implementations for the steps used in the feature file

3.2.2 Features

The Cucumber framework itself doesn't have support for localisation, but the Gherkin language has been translated to multiple (77 at the time of writing this) languages to allow tests to be written in a number of languages. Cucumber provides a an online reporting service called Cucumber Reports, as well as built-in reporting plugins for generating local reports that support formats such as HTML and JSON. It's also possible to define your own custom reporter plugin or use a third-party tool. Cucumber also supports custom parameter types, allowing the conversion of textual tables and strings to any type of object (*Cucumber*). By default Cucumber supports conversion of strings, lists and maps

of strings, and a Cucumber-specific object called "DataTable" (*Cucumber GitHub*).

Cucumber uses JUnit to run its tests. Like JBehave, it allows dependency injection to compose hooks and steps instances via containers. Hooks are blocks of code that can run at various points in the execution cycle of Cucumber. Cucumber too supports executing tests in parallel (*Cucumber*). Cucumber provides integration with Gradle (*Gradle*) and Maven (*Apache Maven*) build tools. It also offers integration with the popular project management tool Jira (*Jira*), and their own tool called CucumberStudio. There are also separate Cucumber plugins for IntelliJ IDEA and Eclipse IDEs. In addition, most popular editors support syntax highlighting for Gherkin (*Cucumber*).

3.2.3 Analysis

Cucumber is a robust framework with extensive documentation and an active community. A big advantage of Cucumber is its simplicity. Creation of tests has been made easy and efficient. The framework is relatively easy to learn, use and understand. At the same time it's still quite flexible, allowing the use of custom reports and parameter types. The Gherkin syntax is simple and makes the tests consistent, but it's also limiting.

3.3 Concordion

Concordion is another open-source testing framework for Java that supports BDD. Similarly to both JBehave and Cucumber, it turns requirements written in plain English into executable tests (Österholm, 2021). Concordion is not strictly restricted to the traditional form of BDD and allows users to write specifications in an arbitrary syntax. The framework takes a different approach by utilising markup languages for writing its specifications.

3.3.1 Implementation

While JBehave and Cucumber use their own custom files extensions and specification languages for writing the requirements, Concordion specifications are written in HTML or Markdown (*Concordion*). The specifications do not need to be written in any specific format but the user can choose what kind of syntax to use (Österholm, 2021). Concordion documentation points out that Gherkin is a good way to start, but once the user becomes familiar with structuring specifications, they may find another way to describe their test

cases that suits them better. Each specification may contain one or more examples, each of which is run and reported as a separate test. Examples are the Concordion-equivalent of scenarios. If no examples are defined, all the commands in the specification are run as a single test. The examples contain commands that are mapped to Java methods in corresponding fixtures by method names. The commands are slightly different from steps in JBehave and Cucumber, as they are not just textual representations for a method, but have their own meaning that Concordion interprets and uses to execute the appropriate code. It is also possible to have commands outside of examples, to be run before each example. Fixtures are Java classes that implement the methods in specification commands. Methods in Concordion fixtures support strings, booleans and numeric parameter types. The possible return types are void, primitives and objects (*Concordion*). The return values are used by some Concordion commands, such as *set* and *assert-equals*.

Figure 3.6 shows an example of a Concordion specification file written in HTML. The *<html>* element contains a Concordion namespace that allows the use of Concordion's commands. The specification also contains a *<link>* element in the header that is used for locating the CSS stylesheet to be used in test reports. The *<body>* element contains a title and a narrative for the specification. The * * element is wrapped with Concordion's example command, which turns the element and the commands within it into a named example. The example is run and reported as a separate test. It contains another title and three *<p>* elements that contain Concordion commands. The first two commands on the first line are Concorions set commands used to set the variables *num1* and *num2* to 1 and 2, respectively. The command on the second line sets a *function* variable to sum. On the third line Concordion's assertEquals command is used to assert that calling the calculate function with *num1*, *num2* and *function* as its parameters is equal to 3.

```
<html xmlns:concordion="http://www.concordion.org/2007/concordion">
   <head>
      <link href = "../concordion.css" rel = "stylesheet" type="text/css" />
   </head>

   <body>
      <h1>Calculator Specification</h1>
      <p>As a user</p>
      <p>I want to use a calculator for calculations</p>
      <p>So that I don't have to do them by hand</p>
      <div class = "example">
         <h3>Example: The calculator can calculate the sum of two numbers
         correctly</h3>
         <p>Given the user inputs the numbers <span concordion:set =
         "#num1">1</span> and <span concordion:set = "#num2">2</span></p>
         <p>When the user chooses the <span concordion:set =
         "#function">sum</span> function</p>
         <p>Then the result is <span concordion:assertEquals =
         "calculate(#num1, #num2, #function)">3</span></p>
      </div>
   </body>

</html>
```

Figure 3.6: HTML version of a Concordion specification calculator_specification.html with a single scenario

Figure 3.7 shows a Concordion fixture class for the calculator. The class is marked with an annotation that is used by Concordion to find fixtures. It implements the *calculate* method used by the example specification. The method takes two integers and a String as its parameters. The numbers are input to the calculator, and the String is used to select which function to call on the calculator, which in the case of our example is *sum*.

```
@RunWith(ConcordionRunner.class)
public class CalculatorFixture {

   Calculator calculator = new Calculator();

   public int calculate(int num1, int num2, String function){
      calculator.input(num1, num2);
      if (function.equals("sum")) {
         return calculator.sum();
      } else {
         // Implement other functions
      }
   }
}
```

Figure 3.7: A Concordion fixture that implements the methods used in the specification

3.3.2 Features

Concordion tests can be run from command line, or by using Gradle or Maven. Concordion supports parallel execution of tests. It allows users to link specifications to one another, forming aggregated results. To better navigate the results, Concordion also supports breadcrumbs. There are Concordion plugins for IntelliJ IDEA and Eclipse. There is also a HTML publisher plugin for Jenkins, that can publish test results, maintain a per-build history and let the user download the output in zip format. Concordion's Extensions API enables users to create their own extensions to add functionality to Concordion, such as new commands, event listeners or modifying the output. In addition to the default way of writing specifications in HTML or Markdown, there is an Excel extension for Concordion, and it's also possible to write a custom extension to handle other formats. To enhance documentation, Concordion provides the ability to embed storyboards, screenshots, logging information, execution time per example, CSS styling and customised status info into test reports. Concordion also allows modification of exception messages, formatting timestamps and run totals of child tests. It supports before and after hooks on example, specification and suite levels (*Concordion*).

3.3.3 Analysis

Concordion is flexible as it doesn't force the use of any particular syntax. This however comes with the risk of the tests being written in an inconsistent way. The Extensions API makes the framework highly robust. Its test reports are highly customisable by design. A unique advantage of Concordion is the ability to include media files in the specifications. Thanks to these features Concordion excels in creating living documentation. One disadvantage of Concordion is the lack of support for custom parameter types. Concordion is not the easiest framework to learn or use, and requires knowing or learning either HTML or Markdown. Its specifications are somewhat technical, and therefore not as easy to read as the plain text alternatives in other frameworks.

3.4 Gauge

Gauge is a cross-platform test automation tool with an implementation for Java that supports authoring test cases in business language (*Gauge*). It's the most recent of the inspected tools. It is not designed strictly for BDD, but can be used as a BDD tool ("Minding the Gap between BDD and Executable Specifications" 2018). Like Concordion, Gauge allows tests to be written in an arbitrary syntax.

3.4.1 Implementation

In Gauge, the test cases, called specifications, can be written in Markdown or plain text and support a .spec or .md file format ("Minding the Gap between BDD and Executable Specifications" 2018; *Gauge*). Like Concordion, Gauge does not require the specifications to be written in any specific syntax. Each specification can have one or more scenarios, each of which consists of one or more steps (Garousi et al., 2020; *Gauge*). Alternatively, the scenarios may contain concepts, which combine one or more steps into a logical group to represent a business intent (*Gauge*). Each test step has a corresponding step implementation that is run when the steps inside a specification are executed (Garousi et al., 2020; *Gauge*). The implementation for a step is located based on a @Step-annotation, similarly to JBehave and Cucumber. A step can also be a context or context step that is defined in a specification prior to a scenario (*Gauge*). Another type of step is a tear down step, which is defined in a specification after the last scenario (*Gauge*). The context and tear down steps are executed for each scenario.

In Gauge, a step in a specification can have four types of parameters. Simple parameters are values within double quotes that are used in steps as is. Dynamic parameters are used as placeholders for actual values either in concepts or when the step uses a data table. Data tables are used to execute scenarios with multiple different values. Table parameters are tables of values used when a single step is executed for multiple values. These parameters can be both simple or dynamic. Finally, there are special parameters that allow the users to pass large and complex data, like tables and files, into the steps. The two special parameter types are called *File* and *CSV*. The supported parameter types in the step implementations are strings, enums, tables, booleans and numeric types by default (*Gauge*).

Figure 3.8 shows an example of a Gauge specification with a single scenario. The title is written in the highest level header in Markdown, represented by the #-symbol. The plain text narrative is written according to the BDD template. In Gauge, lines without formatting are not processed in any way, which is suitable for the narrative. Scenarios are written in the second highest level headers, marked by ##. The actual steps are list items, marked with the *-sign.

```
# Calculator specification

As a user
I want to use a calculator for calculations
So that I don't have to do them by hand

## The calculator can calculate the sum of two numbers correctly

* Given the user inputs the numbers "1" and "2"
* When the user chooses the "sum" function
* Then the result is "3"
```

Figure 3.8: Gauge specification calculator_specification.spec with a single scenario

Figure 3.9 shows an example implementation of the steps used in the example specification. Similarly to JBehave and Cucumber, the methods are annotated with string values that are used to map the textual steps to the appropriate methods.

```
public class CalculatorSteps {

  private Calculator calculator = new Calculator();
  private int actualResult;

  @Step("Given a user input the numbers <num1> and <num2>")
  public void inputNumbers(int num1, int num2) {
    calculator.input(num1, num2);
  }

  @Step("When the user chooses the <functionName> function")
  public void chooseFunction(String functionName) {
    if (functionName.equals("sum")) {
      actualResult = calculator.sum();
    } else {
      // TODO: Implement other functions
    }
  }

  @Step("Then the result is: <expectedResult>")
  public void checkResult(int expectedResult) {
    Assert.assertEquals(expectedResult, actualResult);
  }
}
```

Figure 3.9: Step implementations for the steps used in the spec file

3.4.2 Features

Gauge provides support for creating your own custom parameter parsers to support arbitrary parameter types. It also supports using tags to associate labels with specifications or scenarios, making it easier to search for or filter them. Gauge provides before and after hooks to run test code at step-, scenario-, specification- or suite levels. These hooks can also be tag-based, so that they are only applied to tests with a specific tag. In addition, Gauge has scenario-, specification-, and suite level data stores hold values in memory for a specific lifecycle. Gauge also supports taking screenshots, and customising how they should be captured. The screenshots can also be added in reports, and they can be taken

at any point in time when a specification is run. Gauge has its own test runner for running specifications. It supports parallel execution of specifications. Running tests in parallel creates threads based on the number of available CPU cores by default, but the number of threads can also be specified with a flag. It also offers a command to rerun scenarios that failed in the previous run (*Gauge*).

Gauge can be integrated with any continuous integration tool as it supports first class command line. It is customisable via different types of plugins, and the user is able to create their own plugins. The supported plugin types are language plugins, reporting plugins, IDE plugins and other plugins (*Gauge*). Language plugins enable users to implement specifications in a language of their choice. Reporting plugins create test reports in different formats. Gauge provides reporting plugins out-of-the-box for HTML, XML and Flash formats. IDE plugins make it easier to use Gauge with the IDE in question. Gauge recommends using VSCode with their VSCode plugin. Other plugins can be any arbitrary extension, Gauge provides an example of a plugin called Spectacle that can generate HTML documents from specifications (*Gauge*).

3.4.3 Analysis

Gauge is a new tool that is designed to be flexible. It is highly customisable with its support for custom parameter types and arbitrary plugins to extend its functionality. It also has an extensive list of useful features that are not obvious. Some examples are the support for before and after hooks on every level and data stores that help dealing with variables within a specific scope. Tags are another useful feature that not all similar frameworks have. The free syntax and ability to add plain text at any point in a specification make Gauge robust and powerful. The framework seems relatively easy to use. However, its documentation could be more comprehensive. Because Gauge uses its own test runner, it can not be integrated with some popular tools like JUnit and TestNG.

4 Comparison of BDD frameworks

In this chapter we compare the frameworks to find out their differences, including their strengths and weaknesses. The frameworks are evaluated strictly from the point of view of BDD, so some of their features or capabilities were ignored. Based on the observations, we derive requirements to use for building a new BDD framework.

4.1 Comparison

All of the studied frameworks share many similarities in their implementation and design. They all use plain text files to define tests, and use them to produce human readable outputs in the form of test reports. Each tool does so by mapping some kind of textual step to a Java method that is executed when the test is run. However, each of the four frameworks also has some aspect to it that sets it apart from the others. JBehave tests are based on run configurations that are easily configurable. JBehave's way of locating step implementations is unique in the sense that it is possible to have multiple steps with the same definition, and the user can define prioritising strategies to handle such cases. Cucumber is perhaps the easiest tool to learn and use. It is designed strictly for BDD, and fills its purpose nicely. It is also the only tool to support conditional before and after hooks based on tags. Both JBehave and Cucumber have extensive documentation and are time-tested industry standard tools. Concordion provides the unique ability of styling and formatting test reports and specifications. It also makes it possible to include media files in specifications. Gauge provides data stores to help handling variables within a specific scope. It has the most extensive list of features of all the compared frameworks. Both Concordion and Gauge allow tests to be written in any syntax, giving more freedom to the person writing the tests.

The terminology used in the frameworks varies, which can be seen from table 4.1. A central piece of any BDD tool is the test case, generally referred to as a user story. JBehave calls this a story, in Cucumber it's a feature, and in both Concordion and Gauge, specification is the corresponding term for it. The test case often contains one or more scenarios, which Concordion refers to as an example, while the other tools use the word scenario. The scenarios consist of one or more steps, which is a term that Concordion doesn't really

recognize, but a command is the closest substitute for it. The other tools use the term step.

BDD-related term	**JBehave**	**Cucumber**	**Concordion**	**Gauge**
User story	Story	Feature	Specification	Specification
Scenario	Scenario	Scenario	Example	Scenario
Step	Step	Step	Command	Step

Table 4.1: Terminology used for BDD concepts in the studied frameworks

Table 4.2 lists different features or aspects found in the studied frameworks. The focus is mainly on features that are not implemented by every framework, but rather ones that can be used to find ways to separate them from one another. From the table it can be observed that JBehave and Cucumber, which are specifically designed for BDD, have a forced "Given-when-then" syntax, while Concordion and Gauge allow tests to be written in an arbitrary way. Each approach has its upsides and downsides. A predefined syntax makes the tests consistent and enforces the use of BDD principles. On the other hand, it's very limiting and sometimes it might make more sense to use a different syntax. Consequently. as pure BDD tools, JBehave and Cucumber provide built-in support for BDD, whereas Concordion and Gauge do not, but leave it up to the user to establish a syntax. The table also shows that Concordion is the only tool that does not provide support for custom parameter types. There are ways to work around this, for example one can create an arbitrary object based on a string parameter in the method itself. In the long run this does add a lot of unnecessary boiler plate code however, so having the ability to easily define custom parameter parsers or converters is a useful feature to have. Test report styling and formatting with the combination Markdown, HTML and CSS is unique to Concordion, and a very powerful asset for creating living documentation. The downside to this approach is that the specification files are harder to read and require some technical skills to write. Grouping steps together to form one larger step is a handy feature provided by JBehave and Gauge. JBehave refers to these groups as composite steps, while Gauge use the term concept. This feature can improve code reusability and help make user stories less technical and more concise by increasing the level of abstraction.

Further inspecting the table 4.2, we can observe that each tool supports writing comments in the test cases, although Concordion does not implement any specific way of doing so. Similarly, all tools support using tags in stories, but Concordion does not have a specific implementation for this. JBehave, Concordion and Gauge also support adding

other meta data to tests, and JBehave is the only tool that specifically implements this feature. It can also be observed that JBehave is the only tool that requires a separate run configuration file in order to execute a test. This provides control and adds some flexibility by being able to use different run configurations easily. However, it makes it more troublesome to create new test cases, splits the test case across two separate files, and creates a lot of repeated boiler plate code over time. JBehave and Gauge provide built-in support for step aliases, where the same Java method can have more than one textual representation, implemented as values within an annotation. Concordion also supports this feature naturally, as its specifications refer to method names directly, meaning that it is possible to write anything at all in the specification and still call the same method. This feature can improve the readability of test cases, as there are often multiple ways to express an action, and sometimes the best fit depends on the test case. JBehave is the only tool to support keyword synonyms, which makes it possible to extend its syntax to some extent. For example, you could create a custom synonym "since" for the keyword "given". Gauge implements data stores for managing data in different scopes. Finally, all of the tools support parallel execution which is crucial for cutting down on execution times.

Each framework also scales well. The same steps can be used in multiple different tests to promote re-usability and maintainability. Tests and steps can be organized into arbitrary sub-folder structures to make it easier to locate specific tests, especially for bigger projects. JBehave, Concordion and Gauge provide further flexibility for large projects via step aliases, which allow steps to be re-used with a different definition, instead of having to create a duplicate one. In comparison, Cucumber can be limiting at times. The advantage of Cucumber's approach is that there is less room for errors where a step is mapped to an incorrect method. The fact that each framework supports parallel execution is another important factor to support larger amounts of tests.

Any one of the four tools can be used sufficiently for BDD. For traditional BDD, Cucumber seems like the easiest and most straight-forward option. If more complex configuration is needed while still sticking to strict BDD principles, JBehave may prove more useful. If the focus is on creating living documentation and the team has sufficient technical capabilities, Concordion is a good option. In cases where a team wants more freedom and flexibility, Gauge is a very robust tool with lots of features and therefore another great choice.

Support of features	JBehave	Cucumber	Concordion	Gauge
Free syntax	No	No	Yes	Yes
Built-in BDD support	Yes	Yes	No	No
Custom parameter types	Yes	Yes	No	Yes
Test report styling and formatting	No	No	Yes	No
Grouping steps together	Yes	No	No	Yes
Comments in stories	Yes	Yes	Yes	Yes
Tags in stories	Yes	Yes	Yes	Yes
Meta data in stories	Yes	No	Yes	Yes
Story file is executable	No	Yes	Yes	Yes
Step aliases	Yes	No	Yes*	Yes
Keyword synonyms	Yes	No	No	No
Data stores	No	No	No	Yes
Parallel execution	Yes	Yes	Yes	Yes

Table 4.2: Feature comparison of the studied frameworks

* Concordion does not implement step aliases per se, but is based on executing Concordion commands and therefore does not care what is written around them. In essence the same methods can be called with arbitrary step definitions, which in practice is a more powerful version of a step alias.

4.2 Discoveries

As seen from table 4.2, some tools allow a free syntax, while others set strict rules on how the tests must be written. Considering the ups and downs of each approach, a good solution might be to implement rules for a basic BDD syntax that can then be easily extended. This should essentially combine the benefits of each approach - the rules will support consistency, and being able to extend them improves flexibility. Custom parameter types, test report styling, the grouping of steps, step aliases, and support for comments, tags and meta data in story files can all be seen as useful features with no apparent downside to them. Executable story files help simplify test creation but can be restricting in comparison to JBehave's approach of using separate run configurations for each test. A potential solution is to make the story files executable, but provide an option to specify a configuration file to use, so that it is easy to use different configurations when needed.

All of the tools use human readable, textual files as both their inputs and outputs. The plain text inputs can be seen to increase co-operation between roles by making the tests

understandable for everyone. It can be argued however, that not everyone needs to understand them, as long as the test report is in a human-readable form. While this approach also technically allows test cases to be written without any coding skills, it is not always so. The test writer may need to create new steps and additional test code to support them. Also, debugging any potential issues still requires knowledge about the code. Another issue of this kind of approach is that the test creator has to navigate between files to try and find the step definitions used or to use in their specifications. Parametrised steps make the matter even worse, as simply copying the text and using it in a search query will not suffice. In addition, a tool like JBehave, which allows multiple methods to be annotated with the same text, could cause the wrong step to be found. The issue of navigating between stories and step definitions can be solved to some extent with the help of IDE plugins. Textual story files are also not the best at representing complex data like tables. One consideration is whether the pros of a human readable input outweigh its problems.

All of the inspected tools also share some weaknesses. They are all somewhat complex in their own way. JBehave, Cucumber and Gauge have to implement parsing, parameter transformations, mapping rules etc. to support their textual story files. Concordion on the other hand needs to implement custom logic for interpreting HTML and Markdown. In addition, simply having to use these markup languages adds a level of complexity to using the framework. A related weakness is that each tool, especially the first three, require a lot of configuration to function. Furhermore, none of the inspected tools tools offer support for unit testing. While BDD is generally not as heavily utilized for unit testing, there are BDD frameworks like JDave (*JDave*) that are specifically designed for this purpose. To support a wider set of uses, it might be sensible for BDD tools to implement some unit testing capabilities.

There are other BDD tools, like aforementioned Spock and easyb, that define the test cases in code, while still producing a human readable output. The advantage of this approach is that there is no need for parsers, parameter converters or mapping rules, which reduces complexity. Consequently, there is also less need for configurations. It also simplifies the process of writing tests in the sense that the programmer can refer to the appropriate methods directly and don't need to lookup the textual step definitions. The downside is that the specifications are more difficult to understand for non-technical people. This can however be answered with the help of Java's annotation processors. Annotation processors can be used to inject code for annotated elements into the Abstract Syntax Tree during

the compilation of a program (Kahlout, 2011). The Java Compiler later combines the original source code with the code injected into the AST and generates the Java bytecode required to execute the program (Kahlout, 2011). By leveraging this feature, it is possible to generate plain text representations of the test when building a project. These can then be referred to and understood by everyone when needed, to help clarify the purpose of a particular test case. The plain text representations should also help in maintaining and documenting the tests.

JBehave, Cucumber and Concordion utilize JUnit for running the tests, while Gauge implements its own test runner. Both approaches have their advantages. A separate runner allows Gauge to have more specific logging information and control when running tests. The main advantage of building on JUnit are the existing features and IDE integrations that come along with it.

Overall, the inspected tools are designed in a way where they try to do a lot of things on the user's behalf to support the writing of tests in plain text. While this approach certainly has its advantages, it also has its drawbacks as discussed above. A different approach that would answer some of these drawbacks would therefore be a more developer-oriented design, where the tests are written in code. The main downside of this approach is that it's less friendly for non-developers with poor coding skills.

This approach raises a question; why not stick to more traditional tests instead of BDD if the ability of writing human readable test cases, and therefore some of the collaboration and visibility potential, is lost. While this is also a valid option, it can be argued that writing tests in BDD is also beneficial for the developers. The stories act as documentation for the system's features and can make it easier to understand the code if the developer is unfamiliar with the system or parts of it. Even the individual step descriptions can help describe what a given method is used for. Another potential benefit of BDD is that it may lead to producing better APIs as it promotes writing testable code (Binamungu et al., 2018). Furthermore, BDD can help developers focus on the end user's needs and implementing only what is necessary to achieve them, reducing potential waste.

Based on the identified weaknesses and alternative approaches, we saw a need for building a new, more developer-oriented BDD tool. While it is clear that this approach has its limitations, the aim is to answer the challenges of the existing tools and provide an alternative for them.

Table 4.3 presents the key points derived from inspecting and comparing the tools that will be used as requirements for building a new BDD tool. The new framework should

Requirement	Description
Specific but extensible BDD syntax	The tool should implement specific rules for writing the tests. This improves consistency and enforces the use of BDD. The rules should still be customisable and extendable to provide flexibility.
Java-based test cases	The test cases are written in Java to simplify the creation of tests and to make the framework easier to learn and use. Handling complex data becomes easier too.
Annotation processor for generating textual representations of test cases	The tool should implement an annotation processor that can be used to generate textual representations of test cases when building the project. This way it is easier for everyone to understand the purpose of a test case.
Support unit testing	The tool should have some support for unit testing as well?
Use JUnit for running the tests	The tool should use JUnit to leverage existing IDE integrations

Table 4.3: Requirements for the new framework

offer a specific but extensible BDD syntax to achieve consistency while providing flexibility at the same time. Tests will be written in Java, which is assumed to make the creation of tests and use of the framework easier for developers and test automation engineers. The framework should leverage annotation processing to offer a plain text representation of each test, in addition to the Java code. This way is to support the collaboration aspect of BDD. The new framework should also have at least some unit testing capabilities, as it can be a useful feature. Finally, it was decided to build the framework on top of JUnit in order to both save time in implementation and benefit from the existing features and IDE integrations.

5 Building a new BDD framework

This chapter presents the new bumbleB framework that was built to answer the challenges identified in the previous chapter. The new framework is quite different in design from the previously inspected tools, but it's still specifically designed for BDD. In this chapter, we introduce bumbleB (*bumbleB*), its intended use and purpose, and go through its implementation and features.

5.1 Introducing the bumbleB framework

In comparison to the other studied frameworks, bumbleB is a more developer-oriented framework. It aims to make it easy for developers and test engineers to create new tests. The framework is designed to be simple to use and learn. It is also made to support different testing levels from unit testing to system level testing. The amount of required configuration was attempted to be kept as low as possible. bumbleB also utilizes some technologies that are not commonly seen in testing frameworks. The purpose of this tool is to offer an alternative approach for implementing BDD.

5.2 Implementation

bumbleB takes some different design choices than most BDD frameworks. Its test cases are implemented as Java code instead of textual story files like the other studied tools. bumbleB is built on top of JUnit. Its @Example-annotation is an extension of JUnit's @Test-annotation, meaning that each example (scenario) is run as its own test. bumbleB depends on JUnit framework to run the tests and capture information about their execution. This information is used for generating HTML test reports.

It makes use of Java's annotation processing in order to generate human readable text files to represent test cases. In addition, bumbleB leverages Aspect Oriented Programming (AOP) via AspectJ (*AspectJ*) to provide access to important information at run time. AOP is a programming paradigm that attempts to increase program modularity by separating cross-cutting concerns. AOP allows inserting new code to specific points during

the execution of a program. New code is defined as advice within aspects, and is combined with the original code using an aspect weaver. Three kinds of weaving is possible; compile-time, post-compile and load-time weaving. bumbleB uses load-time weaving. The use of AOP is motivated by the framework's unconventional approach of using method references within its steps. In bumbleB AOP is used to access information about parameters that are passed to step methods following the method reference.

Figure 5.1 describes the implementation details of the new framework. bumbleB consists of three main modules; the framework, the annotations and the annotation processor. The annotations are used by both the framework and the processor. The processor generates plain text descriptions of tests based on the @Example and @Step annotations when the project is built. The main logic of the framework is contained in the *Framework*-class. It provides the methods required for creating examples and the steps inside them. The *Utils* and *StateHolder* classes offer utilities for the framework. The two aspects are used to update the framework's state with information about the arguments and result of a step, as well as the class name and package name of an example. The *ExampleListener* listens to the test execution and informs the *HtmlReportBuilder* about the result of each test case. When the execution is finished, the *HtmlReportBuilder* will generate a HTML test report for each directory that contains tests.

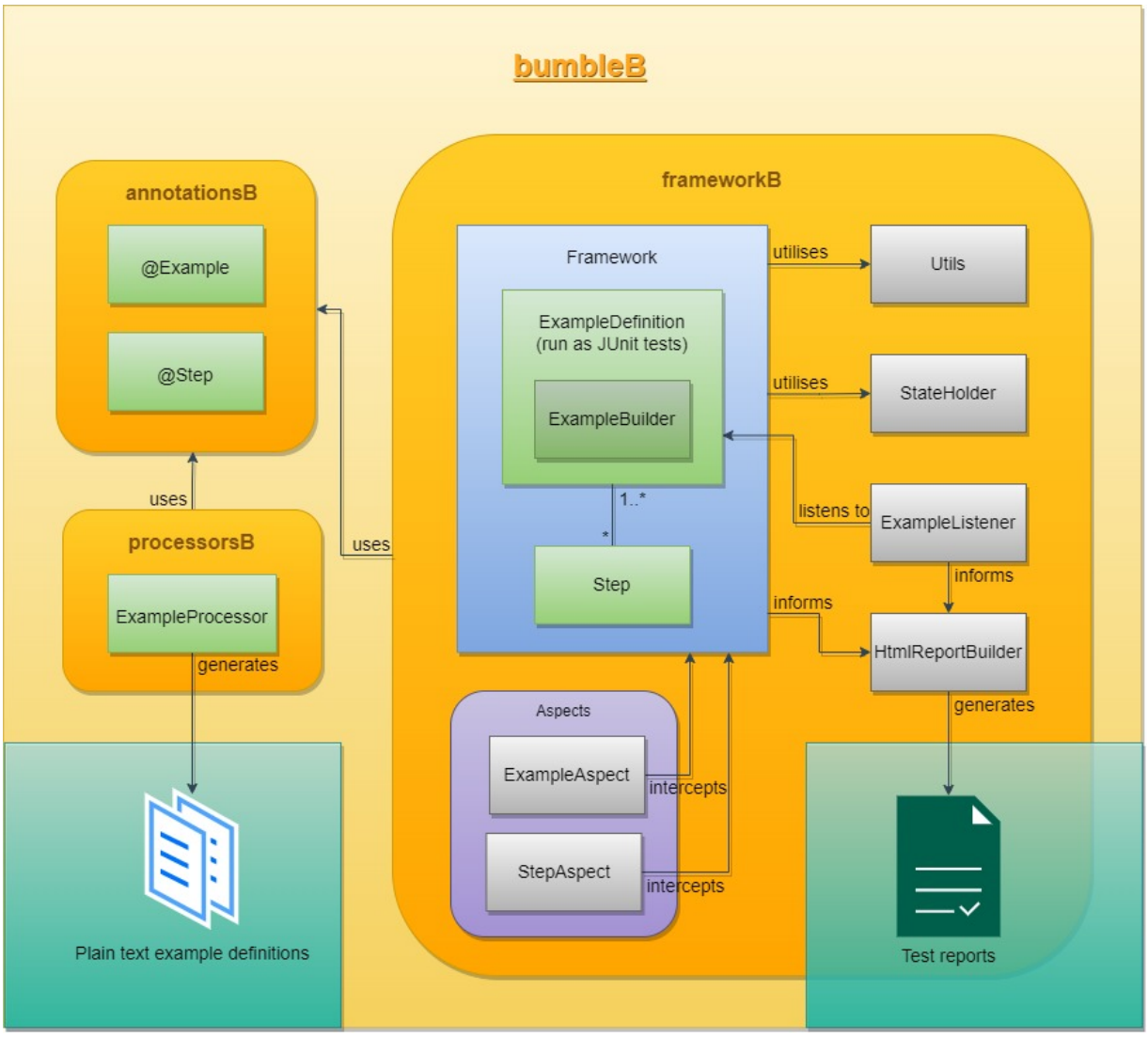

Figure 5.1: A chart that describes the implementation of the bumbleB framework.

Figure 5.2 shows an example of how a BDD story might look like in bumbleB. Each scenario is annotated with the @Example-annotation. An example refers to a single test case or a scenario. The implementation of a story uses an example builder provided by the framework. The builder can accept a name, a narrative and any number of steps to execute. The steps are defined using bumbleB's *given*-, *when*- and *then*-consumers, that take a method reference and zero or more parameters to be passed on to the referenced methods as its arguments. The *build*-method constructs the example that can then be executed via the *run*-method, which executes all the steps in the given order.

Figure 5.3 shows an example implementation for the steps used in figure 5.2. It contains methods annotated with the @Step-annotation. The annotation values are used for gen-

erating plain text descriptions with the annotation processor, printing information to the console at run-time, and for generating HTML test reports. Figure 5.4 shows an example of what the annotation processor would output for the example in figure 5.2. The description contains the scenario title, and the steps as defined in the @Step-annotations.

```
public class CalculatorTests {

    private CalculatorSteps calculatorSteps = new CalculatorSteps();

    @Example
    public void calculatorSumTest() {
        builder
                .name("The calculator can calculate the sum of two numbers
                   correctly")
                .narrative("As a user, I want to be able to calculate the sum of
                   two numbers, so that I don't have to do the calculations in
                   my head")
                .steps(
                        given(calculatorSteps::inputNumbers, 1, 2),
                        when(calculatorSteps::chooseFunction, "sum"),
                        then(calculatorSteps::checkResult, 3)
                )
                .build()
                .run();
    }
}
```

Figure 5.2: An example of a bumbleB test case.

```java
public class CalculatorSteps {

  private Calculator calculator = new Calculator();
  private int actualResult;

  @Step("a user input the numbers {num1} and {num2}")
  public void inputNumbers(int num1, int num2) {
    calculator.input(num1, num2);
  }

  @Step("the user chooses the {functionName} function")
  public void chooseFunction(String functionName) {
    if (functionName.equals("sum")) {
      actualResult = calculator.sum();
    } else {
      // TODO: Implement other functions
    }
  }

  @Step("the result is: {expectedResult}")
  public void checkResult(int expectedResult) {
    Assert.assertEquals(expectedResult, actualResult);
  }
}
```

Figure 5.3: Step implementations for the steps used in the bumbleB example.

```
Example: The calculator can calculate the sum of two numbers correctly

Given a user input the numbers {num1} and {num2}
When the user chooses the {functionName} function
Then the result is: {expectedResult}
```

Figure 5.4: bumbleB text file generated by the annotation processor

5.3 Features

Some of the supported features are listed in table 5.1. bumbleB offers a predetermined but easily extendable syntax for writing tests, It has built-in support for BDD. Since tests are written in Java, any type of parameters can be used. Steps can be grouped together by creating a normal step that calls the appropriate steps with appropriate parameters. As the tests are written in Java, all Java comments are supported in stories. It is also possible to create tags as annotations, but there is currently no built-in support for this. Meta data can be added to stories in the form of comments, external files, or String variables. In bumbleB, the story (test case) itself is an executable JUnit test. Step aliases are currently not supported. Keyword synonyms are possible due to the extensible syntax. bumbleB does not support data stores, but it also has no need for such a feature as the data creating, storage and usage can be easily handled in the story with the help of before- and after-hooks provided by JUnit. bumbleB can generate basic test reports that shows general information about the tests executed. Figure 5.5 shows an example of a bumbleB test report. Each row in the table presents a single example. The individual steps can be seen by clicking on the first column. In case of a failure, the failing step is marked as red.

Support of features	JBehave	Cucumber	Concordion	Gauge	bumbleB
Free syntax	No	No	Yes	Yes	Yes**
Built-in BDD support	Yes	Yes	No	No	Yes
Custom parameter types	Yes	Yes	No	Yes	Yes
Test report styling and formatting	No	No	Yes	No	No
Grouping steps together	Yes	No	No	Yes	Yes
Comments in stories	Yes	Yes	Yes	Yes	Yes
Tags in stories	Yes	Yes	Yes	Yes	No
Meta data in stories	Yes	No	Yes	Yes	Yes
Story file is executable	No	Yes	Yes	Yes	Yes
Step aliases	Yes	No	Yes*	Yes	No
Keyword synonyms	Yes	No	No	No	Yes
Data stores	No	No	No	Yes	No
Parallel execution	Yes	Yes	Yes	Yes	Yes
Java-based test cases	No	No	No	No	Yes
Explicit unit test support	No	No	No	No	Yes

Table 5.1: Feature comparison of bumbleB and the studied frameworks.

* Concordion does not implement step aliases per se, but is based on executing Concordion commands and therefore does not care what is written around them. In essence the same methods can be called with arbitrary step definitions, which in practice is a more powerful version of a step alias.
** bumbleB syntax is based on specific rules, but can easily be extended to essentially create a ubiquitous syntax.

Figure 5.5: An example of a bumbleB test report.

In addition, bumbleB comes with a built-in annotation processor to provide textual (.txt)

representations of test cases that are automatically generated when the project is built. bumbleB leverages JUnit for easy IDE integrations for running tests. As bumbleB tests are created with Java via method references, any modern IDE will suggest the possible step-methods to call when typing in the class/object reference and "::", which makes writing tests convenient. The Java-based tests can be written in a natural and easy-to-understand format that resembles the purely textual files used by many other BDD tools.

6 Evaluating the new framework

In this chapter the new bumbleB framework is evaluated to see how viable it is. Each of its requirements is examined to see how well they are met. In addition, there are some general considerations to discuss other aspects of the tool.

6.1 Are the requirements met?

This section focuses on evaluating the requirements that were set for the new tool. Each requirement is inspected separately in their own sub-section below.

6.1.1 Specific but extensible BDD syntax for consistency and flexibility

bumbleB is built on the classic given-when-then syntax. This ensures that tests are created in a proper BDD syntax. It is possible to extend the default implementation and add new keywords or phrases to bumbleB. This allows for some flexibility, as the user is able to create their own language that can sometimes better describe the test cases. bumbleB's support for creating these custom keywords is very limited however, and doing so is relatively laborious. In the future it would be beneficial to implement better support for this feature so that customising the syntax becomes easier.

6.1.2 Java-based test cases for easiness

To evaluate how easy it is to implement tests in bumbleB, a small experiment was conducted. The experiment consisted of a small coding task where the participants implemented a BDD test scenario using two BDD frameworks; bumbleB and Cucumber, as well as a survey where the participants evaluated their experiences. Cucumber was included in the study as a comparison for bumbleB, as Cucumber is one of the most used and well-established BDD tools and uses the traditional design where tests are written in story files. It was therefore decided that it would be a good contestant to compare bumbleB with its different approach to. The hypothesis was that using the framework and handling data

would be easier in bumbleB. The participants were all people who work or have previously worked professionally with test automation using some BDD framework. Unfortunately finding participants and getting their time proved difficult. In the end the task was sent to six potential participants, four of whom did the task and answered the survey.

Figure 6.1 presents the test case that given to the participants in the experiment. The test is deliberately written in an unusual way as to not give a clear advantage to either framework by making the example better suited for one tool. In addition to the test case the participants were provided sample projects that were setup for the task. The projects contained a very simple implementation of a library, another class for books, as well as the required configuration files and folder structure to write the tests. The full instructions can be seen in appendix A. In addition, the participants were given model answers that they could check either after finishing the tasks, or for help if they were unable to complete the task on their own.

```
Story: Books can be searched from the library
Narrative: The customers want to see what books are available

Scenario: The search finds the correct books for a specific author

Given I have added the following books in the library:
-The Devil in the White City, Erik Larson, Crown Publishers, Historical non-fiction, 2003
-Harry Potter and the Philosopher's Stone, J. K. Rowling, Bloomsbury (UK), Fantasy, 1997
-Harry Potter and the Philosopher's Stone, J. K. Rowling, Bloomsbury (UK), Fantasy, 1997
-Harry Potter and the Chamber of Secrets, J. K. Rowling, Bloomsbury (UK), Fantasy, 1998
When I search for books by author "J. K. Rowling"
Then the search results are:
-Book: Harry Potter and the Philosopher's Stone, J. K. Rowling, Bloomsbury (UK), Fantasy, 1997;
 Amount: 2
-Book: Harry Potter and the Chamber of Secrets, J. K. Rowling, Bloomsbury (UK), Fantasy, 1998;
 Amount: 1
```

Figure 6.1: The test case used in the experiment.

Some notable observations were made when inspecting the answers to the survey, the results of which can be seen in appendix B. Firstly, when asked which BDD frameworks they've used before and how much, all the participants reported JBehave as the tool they have used the most. This is notable as JBehave is similar to Cucumber in its design, so the participants were already familiar with the same basic principles. In addition, three out of the four participants had experience in Cucumber too. This could mean that the

experiment favored Cucumber. Another thing to note is that out of the four participants, only one had worked as a software developer. Given how bumbleB is a developer-oriented framework, the participants could have perhaps been chosen differently, to include more people with a developer background to better reflect the target audience.

Looking further at the results in table 6.1, it was noted that the answers to questions about Cucumber were well in line with each other. In the first three questions all four participants evaluated Cucumber exactly the same. The last question about handling data divided the answers so that on a scale of one to five, one being very difficult and five being very easy, handling data with Cucumber was given the scores of 3, 4, 5 and 5.

The answers to questions regarding bumbleB were more divided, as can be seen from the table 6.1. A potentially notable observation was that the person with the least experience with test automation and BDD (1-2 years) was the one who struggled the most with bumbleB. This participant also spent by far the most time on the task. This could mean that their coding skills were sub-optimal for bumbleB, and that the familiarity they had with Cucumber was especially helpful for them in comparison to other participants. It could also mean that bumbleB in general requires more programming skills to use. Overall, bumbleB had comparable scores to Cucumber in most questions, but received a slightly lower total score, as opposed to the hypothesis. As mentioned before, the study likely favored Cucumber and most of the participants were not fully representing the target audience of bumbleB, which potentially played a factor in the outcome. Based on some of the answers, some participants did not fully understand bumbleB and its benefits, which could also be partly due to poor documentation. Another explanation is that the framework was simply not as easy to use as expected and therefore failed in one of its main objectives.

Question	bumbleB answers	Cucumber answers
How easy did it feel to learn and use the framework?	4, 4, 2, 5	4, 4, 4, 4
How efficient did it feel to create tests using the framework?	5, 4, 3, 4	4, 4, 4, 4
How easy did it feel to understand the tests written using the framework?	4, 5, 3, 5	5, 5, 5, 5
How easy did it feel to handle different types of data using the framework?	4, 5, 3, 3	5, 3, 4, 5

Table 6.1: Partial results from the survey, showing the equivalent questions for bumbleB and Cucumber. The answer was given as a score between one and five, one being "very difficult" and five "very easy", or "efficient" in case of the second question.

6.1.3 Annotation processor for collaboration and maintainability

bumbleB comes with a built-in annotation processor that can generate plain text (.txt) files to describe its Java-written tests. The annotation processor is run whenever the project is built, so the descriptions are always available and up-to-date. The plain text files help in collaboration as people can understand them even if they don't have coding skills. This feature also helps with maintaining the tests, as they make it easier to grasp what is being tested and how. The processor works in most basic cases, but is also somewhat limited in its current form. There are cases where parts of the descriptions will be missing due to its limitations. More time and effort would be needed to make it support these edge cases.

6.1.4 Unit testing support for robustness

bumbleB differs from the inspected tools by offering direct support for unit testing. In bumbleB, steps are created as method references passed within bumbleB's built-in step-consumers, which use the method reference to create an instance of bumbleB's Step-class. For higher levels of tests, the method references would normally refer to methods annotated with some textual description. To support unit testing, bumbleB does not require this. Any method references can be passed on instead. To make the tests appear as BDD, the user is able to add a textual description for each step in the test definition. With this feature, any method can be treated as a step and it becomes possible to leverage BDD for unit tests without modifying the original source code.

Figure 6.2 presents an example of bumbleB's unit testing capabilities. Instead of referring to a steps object, the test creates an instance of the class to be tested. Its methods are then called directly within the *given*-, *when*-, and *then*-methods. The *describeAs*-method is used to add textual descriptions for each step, where the parameter names in brackets are replaced with the actual parameters. Furthermore, the third step leverages bumbleB's *satisfies*-method to make an assertion that compares the return value of the calculator's *getResult*-method with the expected value 3.

Figure 6.3 shows the implementation for the methods used in the example from figure 6.2. This simple calculator takes two integers as its inputs, and a string value to choose which function to perform on the numbers. The *getResult*-method will perform the appropriate calculation and return the result.

```
public class CalculatorTests {

    private Calculator calculator = new Calculator();

    @Example
    public void calculatorSumTest() {
        builder
                .name("The calculator can calculate the sum of two numbers
                   correctly")
                .steps(
                        given(calculator::inputNumbers, 1, 2)
                                .describeAs("the user inputs numbers {num1} and
                                   {num2}"),
                        when(calculator::chooseFunction, "sum")
                                .describeAs("the user chooses function {fun}"),
                        then(calculator::getResult)
                                .satisfies(Assert::assertEquals, 3)
                                .describeAs("the result is equal to [result]"),
                )
                .build()
                .run();
    }
}
```

Figure 6.2: An example of a bumbleB unit test.

```
public class Calculator {

    private int arg1;
    private int arg2;
    private String function;

    public Calculator() {
        this.arg1 = 0;
        this.arg2 = 0;
        this.function = "";
    }

    public void inputNumbers(int arg1, int arg2) {
        this.arg1 = arg1;
        this.arg2 = arg2;
    }

    public void chooseFunction(String function) {
        this.function = function;
    }

    public int getResult() {
        if (this.function.equals("sum") {
            return arg1 + arg2;
        }
        // TODO: implement other functions
        return -1;
    }
}
```

Figure 6.3: The methods used in the unit test example in figure 6.2

6.1.5 JUnit for existing features and integrations

bumbleB uses JUnit framework for running its tests. Leveraging JUnit drastically reduces the amount of code and functionality required for bumbleB. In addition, bumbleB becomes much more user-friendly with all the existing plugins and integrations that exist for JUnit.

6.2 General discussion

Overall the new bumbleB framework is still very much a work in progress, and lacking in terms of customisability, features and even just quality. The tool manages to answer its requirements relatively well, but more work is needed to bring it to a level where it can be considered a reliable automation framework. In the time span of this thesis, it was not possible to reach that level, but it may eventually be reached in the future. bumbleB does however act as a proof-of-concept and manages to show that a different, more developer-centric approach can be taken to implement a BDD framework. Regarding the ease of use, more data is needed to properly assess bumbleB's success in that aspect.

7 Conclusions

In this thesis, we studied BDD is and how it can be supported by test automation tools. Four selected frameworks were then inspected in depth, and they were compared to each other. It was observed that any of the inspected tools can be sufficiently for implementing BDD, and that they are relatively similar in their design and implementation. Nevertheless, each tool was found to have its purpose and unique advantages. Furthermore, being similar in their implementation, the tools were also found to share some limitations, and some alternative approaches were identified. These alternative ideas were used as requirements for building a new BDD automation tool called bumbleB. bumbleB was found to answer most of its requirements successfully, but have its own separate limitations and weaknesses. The results of the small survey B that was conducted point to bumbleB potentially requiring more programming knowledge than tools like Cucumber. It was concluded that bumbleB has the potential be a reasonable alternative in the right scenario where its requirements are met. A suitable context for using this tool might be a workplace where the test engineers have sufficient coding skills. It was noted however, that the new bumbleB tool is still at an early stage and not reliable enough to be taken in to use at this time. It was also noted that more data would be needed to properly evaluate bumbleB as a framework.

To sum up, it was shown that different approaches can be taken to implement a BDD testing tool. As always, each approach and decision has its upsides and downsides. Therefore, when choosing a BDD framework, one should carefully consider the context and the requirements set by their project and organisation. There are many different BDD tools to choose from, so finding a suitable tool should be possible in most cases. In very specific contexts, it may be reasonable to implement a new tool. For such situations, the observations and contributions of this study should prove helpful.

Bibliography

Apache Ant. [Online; accessed 28-February-2022]. URL: https://ant.apache.org/.

Apache Maven. [Online; accessed 28-February-2022]. URL: https://maven.apache.org/.

AspectJ. [Online; accessed 03-May-2022]. URL: https://www.eclipse.org/aspectj/.

Behaviour-Driven Development. [Online; accessed 24-February-2022]. URL: https://cucumber.io/docs/bdd/.

Binamungu, L. P., Embury, S. M., and Konstantinou, N. (2018). "Maintaining behaviour driven development specifications: Challenges and opportunities". In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, pp. 175–184.

Briand, L. and Labiche, Y. (2002). "A UML-based approach to system testing". In: *Software and systems modeling* 1.1, pp. 10–42.

Brocke, J. v., Hevner, A., and Maedche, A. (Sept. 2020). "Introduction to Design Science Research". In: pp. 1–13. ISBN: 978-3-030-46780-7. DOI: 10.1007/978-3-030-46781-4_1.

Concordion. [Online; accessed 7-March-2022]. URL: https://concordion.org/.

Cucumber. [Online; accessed 24-February-2022]. URL: https://cucumber.io.

Cucumber GitHub. [Online; accessed 24-February-2022]. URL: https://github.com/cucumber/cucumber-jvm.

easyb. [Online; accessed 24-February-2022]. URL: https://easyb.io/v1/index.html.

Garousi, V., Keleş, A. B., Balaman, Y., and Güler, Z. Ö. (2020). "Test Automation with the Gauge Framework: Experience and Best Practices". In: *Computational Science and Its Applications – ICCSA 2020*. Ed. by O. Gervasi, B. Murgante, S. Misra, C. Garau, I. Blečić, D. Taniar, B. O. Apduhan, A. M. A. Rocha, E. Tarantino, C. M. Torre, and Y. Karaca. Cham: Springer International Publishing, pp. 458–470. ISBN: 978-3-030-58802-1.

Gauge. [Online; accessed 8-March-2022]. URL: https://gauge.org/.

Gradle. [Online; accessed 03-March-2022]. URL: https://gradle.org/.

Guice. [Online; accessed 28-February-2022]. URL: https://github.com/google/guice.

Gupta, A. and Jalote, P. (2007). "An experimental evaluation of the effectiveness and efficiency of the test driven development". In: *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, pp. 285–294.

Härlin, M. (2016). *Testing and Gherkin in agile projects*.

Janzen, D. and Saiedian, H. (2005). "Test-driven development concepts, taxonomy, and future direction". In: *Computer* 38.9, pp. 43–50.

– (2008). "Does test-driven development really improve software design quality?" In: *Ieee Software* 25.2, pp. 77–84.

JBehave. [Online; accessed 24-February-2022]. URL: https://jbehave.org/.

JDave. [Online; accessed 24-February-2022]. URL: https://github.com/jdave/JDave.

Jira. [Online; accessed 03-March-2022]. URL: https://www.atlassian.com/software/jira.

JUnit. [Online; accessed 28-February-2022]. URL: https://junit.org/junit5/.

Kahlout, G. (2011). "Implementing patterns with annotations". In: *Proceedings of the 2nd Asian Conference on Pattern Languages of Programs*, pp. 1–6.

Kasurinen, J., Taipale, O., and Smolander, K. (2010). "Software test automation in practice: empirical observations". In: *Advances in Software Engineering* 2010.

Lång, J. *bumbleB*. [Online; accessed 2-June-2022]. URL: https://github.com/jiial/bumbleB.

Lenka, R. K., Kumar, S., and Mamgain, S. (2018). "Behavior Driven Development: Tools and Challenges". In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, pp. 1032–1037. DOI: 10.1109/ICACCCN.2018.8748595.

Leung, H. and White, L. (1990). "A study of integration testing and software regression at the integration level". In: *Proceedings. Conference on Software Maintenance 1990*, pp. 290–301. DOI: 10.1109/ICSM.1990.131377.

Leung, H. K. and Wong, P. W. (1997). "A study of user acceptance tests". In: *Software quality journal* 6.2, pp. 137–149.

Miller, R. and Collins, C. T. (2001). "Acceptance testing". In: *Proc. XPUniverse* 238.

"Minding the Gap between BDD and Executable Specifications" (2018). In: [Online; accessed 8-March-2022]. URL: https://gauge.org/2018/11/12/bdd-vs-executable-specifications/.

Needle. [Online; accessed 28-February-2022]. URL: https://github.com/uber/needle.

Nidhra, S. and Dondeti, J. (2012). "Black box and white box testing techniques-a literature review". In: *International Journal of Embedded Systems and Applications (IJESA)* 2.2, pp. 29–50.

Okolnychyi, A. and Fögen, K. (2016). "A study of tools for behavior-driven development". In: *Full-scale Software Engineering/Current Trends in Release Engineering*, p. 7.

Österholm, V. (2021). "Overview of Behaviour-Driven Development tools for web applications". In.

Solis, C. and Wang, X. (2011). "A study of the characteristics of behaviour driven development". In: *2011 37th EUROMICRO conference on software engineering and advanced applications*. IEEE, pp. 383–387.

Spock. [Online; accessed 24-February-2022]. URL: https://spockframework.org/.

Terhorst-North, D. (2006). "Introducing BDD". In: [Online; accessed 24-February-2022]. URL: http://dannorth.net/introducing-bdd/.

– (2007). "What's in a Story". In: [Online; accessed 25-March-2022]. URL: http://dannorth.net/whats-in-a-story/.

Viktor, F. and Alex, G. (2018a). *Test-Driven Java Development, Second Edition : Invoke TDD Principles for End-to-end Application Development, 2nd Edition*. Vol. 2nd ed. Packt Publishing, p. 87. ISBN: 9781788836111.

– (2018b). *Test-Driven Java Development, Second Edition : Invoke TDD Principles for End-to-end Application Development, 2nd Edition*. Vol. 2nd ed. Packt Publishing, pp. 189–209. ISBN: 9781788836111.

Weld. [Online; accessed 28-February-2022]. URL: https://weld.cdi-spec.org/.

Appendix A bumbleB and Cucumber experiment: instructions for the coding assignment

Task:

Implement a simple scenario using both bumbleB and Cucumber frameworks and then fill out a survey

Requirements:

- Java 17
- IDE of your choice (tested with IntelliJ Idea Ultimate)

Download zip files for example projects (both bumbleB and Cucumber), links to the repositories below:

https://github.com/jiial/experiment_bumbleB

https://github.com/jiial/experiment_cucumber

Extract the files and then right click -> open the resulting folder as an IntelliJ project (or open in another IDE if you're not using IntelliJ)

The projects are identical except for the BDD framework used.

Implement the following test case in both projects:

Story: Books can be searched from the library

Narrative: The customers want to see what books are available

Scenario: The search finds the correct books for a specific author

Given I have added the following books in the library

-The Devil in the White City, Erik Larson, Crown Publishers, Historical non-fiction, 2003

-Harry Potter and the Philosopher's Stone, J. K. Rowling, Bloomsbury (UK), Fantasy, 1997

-Harry Potter and the Philosopher's Stone, J. K. Rowling, Bloomsbury (UK), Fantasy, 1997

-Harry Potter and the Chamber of Secrets, J. K. Rowling, Bloomsbury (UK), Fantasy, 1998

When I search for books by author "J. K. Rowling"

Then the search results are

-Book: Harry Potter and the Philosopher's Stone, J. K. Rowling, Bloomsbury (UK), Fantasy, 1997; Amount: 2

-Book: Harry Potter and the Chamber of Secrets, J. K. Rowling, Bloomsbury (UK), Fantasy, 1998; Amount: 1

You can find the documentation for each tool for more information and examples about the tools:

https://github.com/jiial/bumbleB (some examples and general info)

https://github.com/jiial/bumbleB/blob/main/Documentation.md#2-writing-tests (more detailed info about writing tests)

https://cucumber.io/docs/cucumber/

You do not need to create any additional files. Just implement the required steps and the story/test file to run.

You will be given model solutions that you can check after finishing the task or if you need help with finishing it. It's important that you **first try implementing the test yourself to get a feel for each framework**.

Try to keep track of approximately how long you spent on each assignment (bumbleB and Cucumber). You can set a time limit to yourself so that if you are not finished by then, e.g. in 20 minutes (per assignment), you can just check out the model solutions and move on. There will be questions in the survey about how long you spent with each framework.

So, **it's not necessary to finish the task, as long as you've tried so you are able to provide feedback in the survey.**

NOTE: to run your bumbleB test, **you should use a maven run configuration**, e.g. "**clean install**". It is also possible to use JUnit run configurations but that requires changing some settings in IntelliJ and will not generate a test report as that functionality is tied to maven.

For Cucumber, you can use JUnit or Maven run configs.

After you have implemented the tests, **please answer the survey that I invite you to**.

Appendix B bumbleB and Cucumber experiment: full survey questions, choices and answers (n=4)

Question	Choices	Participant 1	Participant 2	Participant 3	Participant 4
How much experience do you have with test automation using BDD tools?*	{Less than 1 year; 1-2 years; 3-4 years; 5 or more years}	5 or more years	5 or more years	1-2 years	5 or more years
Which BDD tools/frameworks have you worked with before? List the tools from the one you are most experienced with to the one you have used the least.*	Open question	Jbehave Cucumber Serenity	- JBehave (4 years) - Robot framework with a custom BDD like steps library (1 year) - Rspec (days to weeks) - A self-built shell script BDD-like test framework (days to weeks) - Mocha/Jasmine/ gazillion other spin-offs or clones (days to weeks) - Spock or Spock-like frameworks (days to weeks)	Jbehave, TestNG, Cucumber	JBehave, Cucumber, Serenity BDD
How easy did it feel to learn and use the framework? - bumbleB*	{1; 2; 3; 4; 5} - 1 being very difficult and 5 being very easy	4	4	2	5
How easy did it feel to learn and use the framework? - Cucumber*	{1; 2; 3; 4; 5} - 1 being very difficult and 5 being very easy	4	4	4	4
How efficient did it feel to create tests using the framework? - bumbleB*	{1; 2; 3; 4; 5} - 1 being very inefficient and 5 being very efficient	5	4	3	4
How efficient did it feel to create tests using the framework? - Cucumber*	{1; 2; 3; 4; 5} - 1 being very inefficient and 5 being very efficient	4	4	4	4
How easy did it feel to understand the tests written using the framework? (what the test is doing, which classes and methods are	{1; 2; 3; 4; 5} - 1 being very difficult and 5 being very easy	4	5	3	5

used/called etc.) - bumbleB*					
How easy did it feel to understand the tests written using the framework? (what the test is doing, which classes and methods are used/called etc.) - Cucumber*	{1; 2; 3; 4; 5} - 1 being very difficult and 5 being very easy	5	5	5	5
How easy did it feel to handle different types of data using the framework? (and how easy would it be to support other arbitrary types of data/objects) - bumbleB*	{1; 2; 3; 4; 5} - 1 being very difficult and 5 being very easy	4	5	3	3
How easy did it feel to handle different types of data using the framework? (and how easy would it be to support other arbitrary types of data/objects) - Cucumber*	{1; 2; 3; 4; 5} - 1 being very difficult and 5 being very easy	5	3	4	5

Did you find any pros to using bumbleB compared to your previous experience?	Open question	Can be used to implement unit tests	This syntax is close to how I've wanted to write BDD tests on Java. Using e.g. JBehave extensively I've always felt that the story file as a test definition is a complication instead of a benefit. If I understood correctly, BumbleB would provide test results in the story file format. This in turn, I feel, is a good use of the story file format. Namely, using the more human readable format for explaining the result and the programmer readable format for the definition of the test. Having two different ways of writing the tests was an interesting feature. This would allow using just one framework for more like Acceptance Testing type and for unit testing. In my experience this doesn't happen in the Java world. I might've been hanging out in the wrong circles. The builder setup replaced some unclarity from the process that annotations often bring. Of course it possible and even easy to learn how the annotations work and things will be just fine. However, as a programmer I appreciate programming most of all with code, as then everything behaves as the code behaves, and everything can be understood.	- flexible cases due to usage of generic types by methods - usage of relatively smaller amount of code when properly defining blocks	No need to create separate story file (JBehave) and the framework can generate a human-readable .txt file for each test

Did you find any cons to using bumbleB compared to your previous experience?	Open question		There wasn't a bundled a bundled assertion library with BumbleB, OR I didn't realise there was. I did have trouble getting my IntelliJ and Java working after a long break so I assume I lost some niceties that were there. To be fair, I'm not sure any other Java testing library provides good assertions either. However, I do feel that in some other ecosystems this would be the case. For both accounts my memory is hazy, though. I might be wrong. The builder setup suffered from the (very usual) tiny bloat of having the ".build()" part. Perhaps it is needed for technical reasons. Maybe there are more extensive features I didn't get to enjoy. I'd probably create a wrapper to skip it. E.g. test.name("hello world").narrative("gonna hello you all").steps(shenanigans::shenanigate).run().	Implementation of BumbleB was a bit more time consuming than Cucumber probably because it was my first approach to use it.	There's no out-of-the-box solution for Data Tables With JBehave: it supports multi-line parameters out-of-the-box and the user only needs to declare the parameter type as ExamplesTable for it to be automatically parsed as a tabular structure. Examples: | title | author | publisher | genre | publicationYear | | The Devil in the White City| Erik Larson | Crown Publishers | Historical non-fiction | 2003| | Harry Potter and the Philosopher's Stone | J. K. Rowling | Bloomsbury (UK) | Fantasy| 1997| | Harry Potter and the Philosopher's Stone | J. K. Rowling | Bloomsbury (UK) | Fantasy | 1997| | Harry Potter and the Chamber of Secrets | J. K. Rowling | Bloomsbury (UK) | Fantasy| 1998| .... | Book 100 | Someone | Publisher | Genre| Year| With bumbleB: private final Book book1 = new Book("The Devil in the White City", "Erik Larson", "Crown Publishers", Genre.HISTORICAL_NON_FICTION, 2003); private final Book book2 = new Book("Harry Potter

					and the Philosopher's Stone", "J. K. Rowling", "Bloomsbury (UK)", Genre.FANTASY, 1997);
					private final Book book3 = new Book("Harry Potter and the Chamber of Secrets", "J. K. Rowling", "Bloomsbury (UK)", Genre.FANTASY, 1998);
					....
					private final Book book100 = new Book("Book 100", "someone", "Publisher", "Genre", Year);
					
					And then you need to create a List like this:
					private final List<Book> books = List.of(book1, book2, book2, book3, ..., book100);

Did you find any pros to using bumbleB when compared to Cucumber?	Open question	Not need to keep and maintain story files	No converter-generator-mapping-magic between Java and TableData or other Story file variables. It is of course nice when someone else has purpose built domain specific converters and such. It still has to be done. I'm absolutely not against adapter patterns in general. However, given my dislike of the story file as a test definition, everything related to that gets minus points as well. With BumbleB I was back to writing actual Java (not that I was actually writing Java - instead I found myself writing chimera code based on whatever was going on in my mind at the time - but thanks to IntelliJ some of it was correct in the end). Then, all my non-programmer colleagues would still get the benefit of the human readable test results that cucumber offers while I can stay a programmer and use all the amazing tools that are available (not you Cucumber! I'm talking IntelliJ idea, jdeps, static typing, ...). In fact, even my programmer colleagues would benefit from the human readable result when they need information at a glance. Everybody wins. Except Cucumber.	You don't need to have this additional abstraction layer responsible for translating natural language to code.	No need to create feature file (Cucumber) and the framework can generate a human-readable .txt file for each test

Did you find any cons to using bumbleB when compared to Cucumber?	Open question		After years of JBehave it seemed so easy to get going with Cucumber, understanding exactly what was going on, copy pasting the task's test definition into the story file and feeling like I'm basically done. (Though, of course, I wasn't.) With bumbleB I still copy pasted the scenarios and such into the java file and started rewriting it into Java. Technically, with Cucumber I would quite easily be able to create some step skeletons and run the test until it fails. Then code some more and keep going. Without the story file though, IntelliJ was all red for the whole 20 minutes I spent on it. Red means danger. Danger means stress. Stress means monkeybrains and getting nowhere fast when coding. The fix would be of course to add the information from somewhere else piece by piece. That somewhere could be the task document, a ticket, a notes app, a throwaway text file or block comments in the code. Though, that sound awful lot like a story file. However, a block comment is an ok fix for a problem that may never go away. We'll always be translating human needs to code, which is possibly why the story files were created as a tool begin with.	Understanding the code structure was easier in Cucumber, probably because I did not had enough time to get used to using BumbleB.	

Are there any things you want to mention that were especially good about bumbleB?	Open question		BumbleB is a nice demo showing that BDD is not the same as a story file. Still providing a story file like test report is a great opportunity. Even more, Behaviour Driven Development (well, Testing is what we're dealing with here) is not even the Given-When-Then syntax, although I suppose these days they are considered one and the same when working as a googling engineer. While that of course is a whole other conversation entirely, the point that bumbleB seems to be getting at is that you can program in a very effective way without sacrificing readability of the code and the understandability of the results. Given-When-Then, for example, provides some of that readability by default.	Usage of blocks instead of standard page objects gives somehow new fresh perspective when using test automation.	
Do you have any suggestions on what could be improved about bumbleB and/or what it's missing?	Open question		If you continue deeper into reporting, I suggest you don't benchmark JBehave's reporting. Instead look into every other interesting ecosystem out there and find some creative ideas to bring back with you into the Java world. Perhaps also a human readable reporting focused suite design would be something to look into. It seems to me that suites are these days either e.g. Java files that define which tests to run, or long it().describe().should().	I think that there could be much more informative materials on the web about this framework.	Documentations are still lacking some information. I managed to write the test because I already understand BDD concept and Given When Then annotations. However, a newbie who is introduced to bumbleB framework might be struggling when writing the test.

			yoda() spells. Both leave me wanting. When you have thousands of tests, do you really understand what is going on? Does anyone?		
Which order did you do the coding assignments in?*	{bumbleB, Cucumber; Cucumber, bumbleB; I worked on both simultaneously}	Cucumber, bumbleB	bumbleB, Cucumber	Cucumber, bumbleB	bumbleB, Cucumber
Do you think that being already familiar with the test case and the sample project helped you when doing the assignment for the second time with another framework?*	{Did not help; Helped a little bit; Helped a lot}	Helped a little bit	Helped a little bit	Helped a lot	Helped a lot
Approximately how long did you spend on the bumbleB part of the coding exercise?*	Open question	30 minutes	25 minutes including setting up IntelliJ, got mostly the steps written without running anything	2x longer than Cucumber	1 hour
Approximately how long did you spend on the Cucumber part of the coding exercise?*	Open question	40 minutes	15 minutes before I felt I'd gone close enough to the point of no return (that would be writing the converters)	1,5 hour	30 minutes

* Mandatory questions