Computer Generated Integration Tests using Natural Language Descriptions

Verloop, A.C. (Daniël)

Award date:
2022

Department of Mathematics and Computer Science
Software Engineering and Technology Research Group

Computer Generated Integration Tests using Natural Language Descriptions

Master Thesis

A.C. Verloop

Supervisors:
Prof. Dr. Mark van den Brand
Rinse van Hees MSc.

1.0

Eindhoven, Wednesday 31st August, 2022

Abstract

This thesis defines an architecture for an integration test generation framework that makes use of the Behavior Driven Development approach to software testing. We developed a tool called *FARMER* that is able to generate integration tests. *FARMER* generates integration tests depending on user defined scenarios. User input is provided by writing test scenarios in natural language descriptions using the Gherkin language. Through the use of Natural Language Processing (NLP) and static code analysis, *FARMER* is able to match natural language to source code.

Acknowledgments

First and foremost, I would like to express my thanks to my supervisor, prof. Mark van den Brand, for the excellent guidance through the learning process of this master thesis. He has been of great value, providing useful comments and good feedback for the problems we were trying to solve. I would also like to extend my gratitude to Rinse van Hees for sharing his knowledge and experience on the subjects present in this thesis, as well as to thank him for his valuable feedback.

Second, I would like to thank Marieke Keurntjes for giving helpful insights and supporting me with my planning throughout the project. I am also grateful to all the other graduates in the finance unit at InfoSupport for always being open to review and discuss parts of my thesis.

Lastly, I would like to thank my family for their continuous support and always being there for me.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In this chapter, an introduction to the topic that motivates this thesis is given in section 1.1. Next, a definition of the problem we aim to solve is provided in section 1.2. Finally, the structure of the thesis is defined in section 1.3.

1.1 Motivation

The project is initiated by Info Support[1], an IT-partner and specialist in Artificial Intelligence (AI), cloud services, managed services and IT-training. They are active in sectors of agri-food, retail, energy, finance, mobility, pension & healthcare. Next to this, Info Support is also a big supporter of open-source projects.

One of the tasks of their software developers is to revise or extend already existing software systems. These systems do not always have existing test scenarios to assess the correctness of such a system. Therefore, developers spend a lot of time writing unit tests and integration tests. Integration tests in particular are complex and hard to write. As integration tests often represent the user stories and requirements of the system, business consultants with domain knowledge are better equiped to write these tests compared to developers. For this reason, business consultants should be enabled to write integration tests.

With the Behavior Driven Development (BDD) approach to software testing, test are written in a language called Gherkin. This language contains structured natural language descriptions of test scenarios. These scenarios describe the system under test (SUT) using terms from the targeted business domain language. As such, these tests require domain knowledge and no programming knowledge. In order to execute these test descriptions on the SUT they are coupled with files containing test functions in the source code language of the SUT. The complete set of functions is called the *glue code*. Currently, software testing with BDD contains two steps, first writing the natural language descriptions, and then writing the glue code. The combined result is the test suite and can be used as an integration test suite. The biggest disadvantage of this approach is that maintaining this test suite while the system evolves requires a lot of work. More details on the BDD approach can be found in section 2.3.

Integration test cases are designed to discover bugs or faults that are caused due to incorrect interaction between software modules. If the modules themselves are correct, after system integration, the interaction between modules can lead to incorrect behavior. Integration tests have a larger scope than unit tests, instead of checking small parts of functions, integration tests check large parts of the system under test, replacing mock objects with real implementations. This also means that integration tests have to work through longer method call sequences, meanwhile

[1] https://www.infosupport.com/

checking the state of all modules involved in the test. As a result, integration tests are more complex than unit tests. Higher complexity also means that integration tests are more expensive to develop. Due to the complexity, the creation of an integration test suite by developers is difficult and expensive. As a result, in practice, developers tend to write simple and short test cases that only check basic functionality of a system. As such, automated test generation might be a possible solution for this problem.

1.2 Project Goal

The problem we face is that a generated integration test suite must be able to evolve with a software project. When using test generation algorithms without user input, this is often not possible. The desired user input on the test cases must also come from people that do not necessarily have programming experience. For this reason, we want to use an easy-to-understand Domain Specific Language (DSL), in which the tests are to be written. This will be realized with the help of the BDD approach to software testing, where the test language Gherkin will represent this DSL. The concept behind BDD is to provide a shared platform for collaboration between software developers and business analysts. User stories or integration tests are written in natural language understandable to everyone involved in the development of a system. Developers must transform these natural language descriptions into an executable set of functions written in the source code of the software project, the complete set of functions is often referred to as the 'glue code'. The concepts of BDD are further elaborated in section 2.3. In order to create the integration tests without the need for writing the glue code, a software tool needs to be designed that is able to map the written DSL to executable source code. The most recent developments on this topic were focussed on the Python programming language [27]. However, there has not been significant research on this topic regarding object-oriented languages. Therefore, we want to explore if we can apply the same techniques to the Java programming language.
To achieve this goal, we composed three research questions. The first research question we need to answer is as follows:

> **RQ1.** Can we apply existing integration test generation methods to Java and get equal accuracy compared to existing literature?

The first research question aims at developing a prototype that is able to generate integration tests with equal accuracy compared to previous work [27][16]. There has already been research into algorithms for integration test generation. As such, it is important to have a good overview of the state-of-the-art on the topic. Integration tests can be developed for all mainstream programming languages and frameworks. However, programming languages can be defined as functional or object-oriented languages. Both these types of languages use different concepts on how source code should be written. The focus of this project is on object-oriented languages, specifically Java, but we do not want to disregard promising techniques focussed on functional languages. It is worth noting that previous work in Python [27] did not make use of specific language properties to enhance its accuracy. When applying a method from literature that was not designed to make use of the language properties of Java, there is the possibility that our results might not be the same as described in the literature. Hence, this research question aims at applying findings based on existing literature to the Java language. We then evaluate the results to see if we can get the same results.

The second research question aims at improving the established accuracy of previous work [27][16]. Object-oriented source code contains methods, classes, and parameters that are used to write tests. The main problem regarding test generation is determining if such an artifact should be used in a specific test case. This problem can be seen as finding a specific solution in a search space. In previous work, it has not been attempted to reduce the size of the search space through filtering. For example, methods are defined by their name, parameters, and return type. From this information, a unique signature can be made on which methods can be filtered. To explore this idea

further, we define the second research question as follows:

> **RQ2.** How can we improve upon the results of RQ1 through the use of OO context information?

As seen in section 2.1, integration testing is the second level of the testing pyramid. As such, it is very likely that there is already a comprehensive set of unit tests present in the software project. The third research question is defined as follows:

> **RQ3.** How can we use information on object instantiations present in unit tests to improve the accuracy of our test generation model?

Unit tests often contain code that is also used in integration tests. For object-oriented languages, a unit test always has a "setup" part. In the test setup, the class/object that implements the to be tested function is instantiated. When testing multiple classes in an integration test, both classes also need to be instantiated. We want to explore the possibilities of reusing the class instantiations of unit tests in the integration test generation algorithm. In order to find unit tests that actually contain relevant class information, we intend to use string similarity based on the test function name and the Gherkin descriptions. Based on review of open source unit test suites, we found that unit test names often are similar to Gherkin scenarios. This is further explained in section 5.2.2.

1.3 Thesis Outline

The remainder of this thesis is structured as follows.

- **chapter 2 Preliminaries:** In this chapter, we introduce the basic terminology and background information that is related to this project. Moreover, a motivating example for this project is provided.
- **chapter 3 Methodology:** This chapter describes the research design of this project and steps undertaken throughout its execution.
- **chapter 4 Review of Related Work:** In this chapter lists the existing relevant approaches to generating integration tests.
- **chapter 5 Prototype Overview:** This chapter presents a prototype design for generating integration tests.
- **chapter 6 Prototype Implementation:** This chapter describes how all prototype components are implemented.
- **chapter 7 Evaluation:** This chapter summarizes the evaluation of the prototype.
- **chapter 8 Conclusions and Future Work:** We draw final conclusions in this chapter and list some notes about the future work.

Chapter 2

Preliminaries

In this chapter, background information relevant to this thesis is introduced. First, an explanation is given to the relevance of integration testing, as well as their place in the testing pyramid of a software project. Then, a motivating example is described of transforming integration tests written in natural language to executable instances in Java. Finally, a number of techniques relevant to the example are discussed.

2.1 Integration Testing

Software testing, whether it is manual or automated, enables quality control on software systems. Where unit testing focuses on testing individual functions, integration testing aims at testing multiple modules of the SUT as a group. This type of testing determines whether several classes or functions interact correctly with one another. Integration tests serve multiple purposes. Testing the communication between different modules of a system. Integration testing can also be used to test the data integrity of a system, verifying no inadvertent changes or corruptions have taken place. As the system is created with the user experience in mind, integration tests can also be used to test user-based scenarios, making sure that user stories can be executed correctly on the application. Another useful application of integration testing, is testing the interaction between third party APIs and the system. In Figure 2.1, an example of the testing pyramid is displayed. In general, software systems that are well tested use at least a form of testing that can be represented by these three categories. As can be seen by the figure, the testing scheme is pyramid shaped, hence its name. The base of the pyramid consists of unit tests. Unit tests provide the highest level of granularity and the most detailed level of feedback to developers in case of failures. Unit tests are also more automated, making tests faster and cheaper to create and maintain. At the top of the pyramid, we have End-to-End tests, this type of testing is considered expensive as they are usually not automated. Integration testing is placed in the middle as they can be somewhat automated, but are still more expensive to create and maintain compared to Unit tests. That said, integration tests are necessary, as they test different functionality compared to Unit tests. According to industry-wide experts, to keep a representative test suite that covers the three most important types of testing, it is recommended that the respective volume of the different types of tests follows a pyramid shape.

When creating an integration test suite, there are two types of integration testing: the Big Bang Approach and the Incremental Approach [8]. As the name is suggesting, with the Big Bang Approach, all modules are grouped together and tested as a whole. This approach works best with small systems and yields quick results. However, it comes with the chance of overlooking a component and greater difficulty in locating bugs. Also, all modules need to be complete, so this approach is only suited for a later stage of the development. The Incremental Approach tests the interaction of a few units at a time. There exist two basic methods for this approach: Top-Down

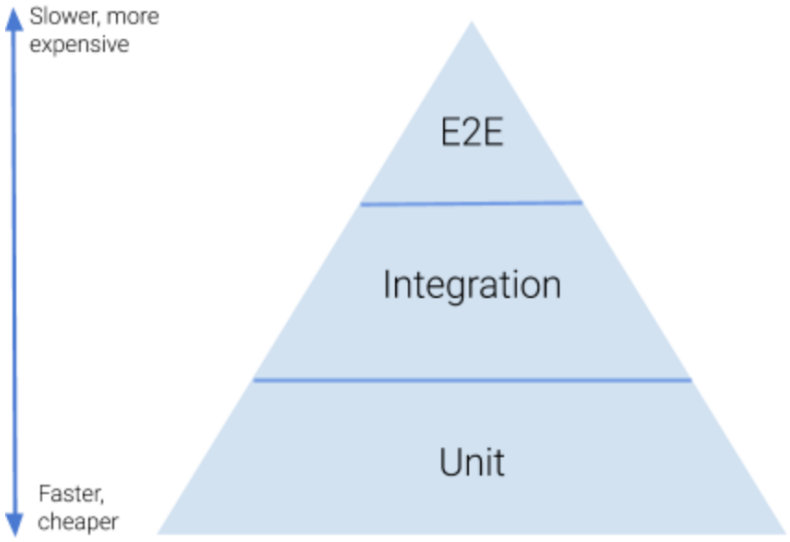

Figure 2.1: An example of the testing pyramid

and Bottom-Up. For Top-Down, high-level modules are tested first and then low-level modules. The main pitfall of this approach is that low-level modules are insufficiently tested. For Bottom-Up, low level modules are tested first and form the foundation for higher-level testing. The main disadvantage of this approach is that it is not possible to create an early working prototype. The advantage of the Incremental Approach is that locating and fixing bugs is easier. Also, the testing does not require a finished product.

2.2 Motivating Example

The focus of this project is to generate integration tests for an existing software system. From the theoretical perspective, we will present several approaches in chapter 4. To help understand the approaches described in this thesis, we will provide a motivating example. The concepts and techniques used with this example will be explained in the sections below.
In this example, we will define a class that represents a bank account. The functionality of this class is simple. The user is able to deposit, withdraw and view their account balance. The implementation code can be found in Listing 2.1.

To test this class, a scenario is written using Gherkin, a test specification framework used to write software tests. In Listing 2.2, a scenario can be found that tests if the written code deposits and withdraws correctly from the bank account. This scenario is described using natural language. The Gherkin structure is maintained using the framework specific keywords, these are highlighted in the example. The text after the *Feature* keyword describes the user story that is linked to this test. As we do two separate actions with the bank account, depositing and withdrawing money, the *And* keyword is used to explicitly separate these actions in the scenario. This keeps the scenario straightforward and organized. Do note that this example is written in the Java programming language, but the concepts used remain the same as with any other programming language.

To be able to execute the created scenarios on the software application, the natural language needs to be transformed to a test that can be executed using the functions of the target system. This part is called creating the 'glue code', as the code that is created in this step essentially 'glues'

```java
public class BankAccount {
    public int balance;

    public BankAccount(int balance) {
        this.balance = balance;
    }

    public int getBalance() {
        return balance;
    }

    public void setBalance(int balance) {
        this.balance = balance;
    }

    public int deposit(int amount) {
        this.balance += amount;
        return this.balance;
    }

    public int withdraw(int amount) {
        this.balance -= amount;
        return this.balance;
    }
}
```

Listing 2.1: The BankAccount class

```
Feature: balance management
    As a customer I want to perform transactions on my bank account
    So that I can manage my finances

    Scenario:
        Given a bank account with initial balance of 0
        When I deposit an amount of 100 into the bank account
        And I withdraw an amount of 50 from the bank account
        Then the balance of the bank account should be 50
```

Listing 2.2: Example Gherkin scenario

the test to the necessary functions to make it executable on the system. We create a separate glue code function for every scenario step. A scenario step contains all lines that belong to a certain keyword. For example, the line *a bank account with initial balance of 0* is the step that belongs to the *Given* keyword. If two scenarios share a step, we use the same function twice to prevent code duplication. The glue code matching the example scenario can be found in Listing 2.3. As each function is annotated with a keyword and context, the Gherkin interpreter is able to automatically execute the correct functions in the correct order for each scenario. This is accomplished with the help of regular expressions.

If the creation of glue code is not automated, a developer has to write the glue code himself. As the complexity of the system and the test scenarios increases, writing glue code becomes a complicated process and requires a lot of extra time. Therefore, automating this process is desirable. To automate this process, several concepts and techniques can be used. The techniques used within the scope of this project are explained in the sections below.

```
public class MyStepdefs {
    BankAccount bankAccount;

    @Given("a bank account with initial balance of {int}")
    public void aBankAccountWithInitialBalanceOf(int arg0) {
        bankAccount = new BankAccount(arg0);
    }

    @When("we deposit {int} pounds into the account")
    public void weDepositPoundsIntoTheAccount(int arg0) {
        bankAccount.deposit(arg0);
    }

    @Then("the balance should be {int}")
    public void theBalanceShouldBe(int arg0) {
        Assert.assertEquals(bankAccount.getBalance(), arg0);
    }

    @When("we withdraw {int} pounds from the account")
    public void weWithdrawPoundsFromTheAccount(int arg0) {
        bankAccount.withdraw(arg0);
    }

    @Given("a bank account with balance of {int}")
    public void aBankAccountWithBalanceOf(int arg0) {
        bankAccount = new BankAccount(arg0);
    }
}
```

Listing 2.3: Glue code implementation

2.3 Behavior Driven Development

The Gherkin framework is can be categorized as a Behavior Driven Development (BDD) approach to software development. BDD was first introduced by North [19]. BDD combines Test Driven Development (TDD) and Domain Driven Design (DDD) principles to encompass the wider picture of agile analysis and automated acceptance testing for software.

2.3.1 Test Driven Development

Test Driven Development was first introduced by Kent Beck [3]. It is classified as an agile approach to software development. With the TDD development cycle, test cases are defined to specify and prove what the code can do. Beck describes in his book that tests should be written first, before the specified code is written. The set of new tests might initially fail, because required functionality is missing. The developer should then write the simplest code to pass these tests. The complete set of tests can also be used to prove that the functionality of software is preserved after refactoring efforts. At any point in time, the set of tests function as an executable specification [22].

2.3.2 Domain Driven Design

Domain Driven Design was first introduced by Eric Evans [7]. This approach to software design that focuses on modelling software to match a domain according to the input from the domain experts [29]. Fowler defines DDD as an approach that centers the development on programming a domain model that has a rich understanding of the processes and rules of the domain [10]. DDD is a software development approach where the structure of the software source code such as class, method and variable names should match the business domain. By abstracting away from the software design and implementation, DDD provides a powerful solution that leverages the complex needs of the business domain. Using the DDD approach, the development process should not primarily focus on the used technologies. Instead, the primary focus should be on the

business domain and its domain logic. As a result, complex design decisions should be based on a model of the domain. Another design predicate of DDD is that technical and domain experts should collaborate to iteratively refine the conceptual model that addresses domain problems.

2.3.3 Implementation of the BDD approach

BDD argues that the expected behavior of software should be described in testable scenarios that are related to functional requirements. Tests in BDD are structured in a Given-When-Then manner. This is an easy-to-understand method for describing system behavior in natural language:

- *Given* a state of a system. In most cases, this step describes the system context before the user interacts with it. Preconditions are defined in this step. In a single scenario, more than one Given step may be present.
- *When* an event changes the state of the system. This step describes actions on the SUT by another system or the user.
- *Then* the system should produce the expected response. This step is where you describe what the system has to do so that it can be compared to the practical performance of the software. This should represent something measurable, such as the state of the system, a message, or report.

Each scenario describes a test from the users' perspective using natural language. The most common workflow in practice can be found in Figure 2.2. Code steps represent the process of transforming the test scenarios into code that is executable on the SUT. This step can also be referred to as creating 'glue code' or 'step functions'. Using the approach depicted in Figure 2.2, tests have to be created twice. First, the test needs to be written in natural language, then to be able to execute the test on the SUT, the test has to be written again using glue code. This glue code is then transformed into integration tests with the help of regular expressions. This approach is not very efficient, as tests are written twice, the workload and maintenance is doubled. Therefore, automizing this process by using a DSL to automatically generate the integration test desirable. This removes the step of having to create glue code.

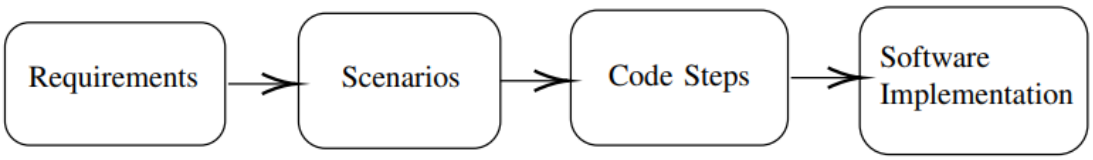

Figure 2.2: The workflow of BDD in practice.(Source: [27])

2.3.4 Gherkin

Gherkin is a DSL that was introduced in parallel with the BDD approach. Gherkin is easy to understand for both developers and business consultants. It is regarded as 'non-technical', which makes collaboration between teams easier. Currently, Gherkin is the most commonly used BDD language in industry. The test scenarios written according to the Gherkin syntax represent the project and business requirements. This ensures that the system always satisfies the requirements defined by the client. In addition to the Given-When-Then structure of BDD, Gherkin also provides the keywords *And* & *But*. This helps with adding more constraints on a scenario and keeps the documentation organized.

2.3.5 Glue code

Gherkin scenarios provide a natural language representation of software tests, but these cannot be executed directly on the SUT. In order to execute these tests, the steps in the pipeline in Figure 2.2 need to be completed. The step called 'Code Steps' is the step in which a source code representation of the Gherkin scenarios is created. The source code representation is often referred to as the *glue code* or *step functions*. An example of glue code can be found in Listing 2.2.

2.3.6 Cucumber

These Gherkin scenarios cannot be executed by themselves. As such, to link the Gherkin scenarios to their corresponding glue code functions and execute them on the SUT, a testing framework is needed. With respect to this project, the Cucumber [25] framework is used for test automation. Cucumber keeps track of which glue code functions belongs to which Gherkin scenario. As a result, it is possible to execute any subset of Gherkin scenarios on the SUT. Also, Cucumber keeps track of all succeeding and failing tests. The framework provides detailed feedback on failed test cases.

2.4 Natural Language Processing

As the tests are written using the Gherkin language, the tests essentially consist of natural language sentences. Humans are capable of capturing the meaning of a word depending on its context without much difficulty. Computers on the other hand have a hard time extracting information from natural language.
The field of analyzing natural language with computer algorithms, is called Natural Language Processing (NLP). NLP is a combination of the fields of linguistics, computer science, and AI concerned with the interactions between computers and human language. In particular, how to program computers to process and analyze natural language data [9]. In the context of this project, NLP is used to analyze and extract useful insights from the Gherkin scenarios. This contains extractions of nouns, verbs, and numbers. But also, words in the sentence are labeled to extract the semantic meaning of each word based on its context. The algorithm used to extract

She	never	like	playing ,	reading	was	her	hobby
PRON	ADV	VERB	VERB	NOUN	AUX	DET	NOUN

Figure 2.3: An example of a POS-tagged sentence

verbs, nouns, and numbers from a sentence with regard to this project is Part-of-Speech (POS) tagging. This algorithm labels all words in a sentence with their grammatical function. These labels contain nouns, adjectives, verbs, etc. The main challenge with this technique is ambiguity, as many words have multiple meanings and therefore can have multiple different tags. A POS tagger's job is to resolve this ambiguity, by giving words correct tags based on their context. An example of a POS-tagged sentence can be found in Figure 2.3. The first POS taggers were based on a linguistic model. However, the latest models are based on statistical analysis and achieve high accuracy [30]. The algorithm used within this project is the default implementation of the NLTK library [4]. This implementation is based on the implementation of Matthew Honnibal [15].

Semantic Role Labeling (SRL) is one of the techniques used to understand the roles of words within a sentence [21]. SRL can be seen as a technique to answer the *"who did what to whom"* question for a sentence. An example can be found in Figure 2.4. This approach creates a machine-understandable representation, which captures the meaning of a sentence. Within the context of this project, SRL is implemented through the use of the AllenNLP library [11]. The specific machine learning model used is the BERT-based model proposed by Shi et al. [24]. The most

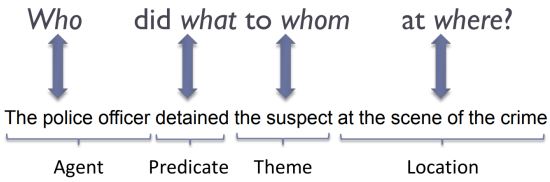

Figure 2.4: Semantic Role Labeling

recent development of NLP models have all been based on neural models that heavily depend on pretraining based on language modeling. Of these models, BERT-models are the latest development. The model described in the paper uses POS-tags as input and is able to perform SRL with state-of-the-art performance. Shi et al. are the first to successfully make use of a BERT-model for SRL.

2.5 Source Code Analysis

In order to generate the correct integration test that are able to be executed on the target software system, source code analysis is needed. Using code analysis software, it is possible to extract the method, functions, and classes of a software project. Specific for this project, we need to perform static code analysis on Java projects. To do so, we make use of the JavaParser library [26].

2.5.1 Static Code Analysis

Static source code analysis refers to the process of analyzing source code without executing that source code. Within the scope of this project, we use the JavaParser library to perform a technique called *Lexical Analysis*. Lexical Analysis converts source code syntax into "tokens". This is done to abstract away from the source code and make it easier to manipulate [6]. Regarding this master thesis, the JavaParser Library performs the Lexical analysis and outputs this in the form of an Abstract Syntax Tree (AST).

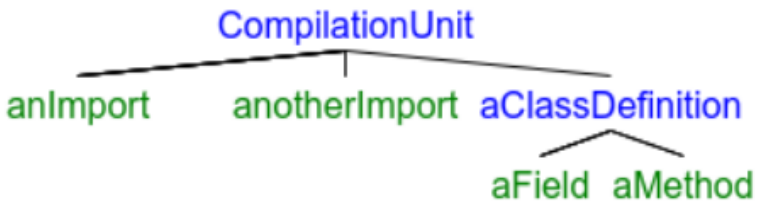

Figure 2.5: Example of an AST.

2.5.2 JavaParser and Abstract Syntax Trees

The JavaParser library analyzes source code by building an AST representation of the target software project. Java source code files can be viewed as a tree. When building such a tree, we start at the file level. We can create tree branches for each code statement present in the file. For a real tree, big branches develop into multiple smaller branches. In the case of source code, we can create smaller branches by breaking down complex code statements into simpler ones. A typical Java source code file contains various imports and at least 1 class definitions. These can all be viewed as direct descendants of the file. In turn, a class statement contains fields and methods. An example of a tree can be found in Figure 2.5.

Important to note is that an AST abstracts away from specific code details such as whitespace, brackets, curly brackets, parenthesis, and semicolons. The AST does not need to store implementation details, as these can be inferred by the tree structure. The structure of code is stored by using the nodes and relationships present in the tree.

Chapter 3

Methodology

This chapter describes the methodology used to develop and evaluate our work. First, an overview of the research design is provided in section 3.1. Then, in section 3.2 we describe how the research methodology applies to this master thesis.

3.1 Research Design

This master thesis consists out of a literature review and a prototype design. The literature review aims to analyze the state of the art regarding automation of the integration test creation process. The main findings of the literature review will be used to support the design of an architecture for an integration test generation workflow. Throughout the course of this project, we have followed an iterative design process. This process is visualized by Figure 3.1. The idea behind this approach

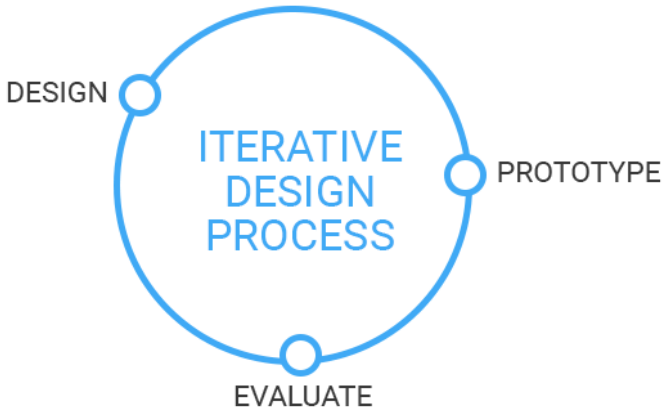

Figure 3.1: The iterative design process.

is that we first design an artifact based on research. Next, the created design is realized during the implementation phase. Finally, to complete a development iteration, the implementation is evaluated based on some sort of testing. These tests can range from user experience tests, but also formal tests such as unit tests or integration tests.
Based on the results of the evaluation phase, the cycle has to be iterated again until the desired result is achieved.

3.2 Research Methodology

To formulate answers to the research questions defined in section 1.2, certain actions need to be undertaken. First, we investigate into the state-of-the-art techniques for integration test generation. The finding following from this literature review will help us to answer the first research question.
Based on the findings of the literature review, a prototype is designed. The main purpose of this design is to illustrate the power of computer generated integration tests. As such, an implementation of the designed prototype will be realized within set goals and requirements. To be able to answer the second research question, the performance of the prototype will be evaluated using a set of open source software projects.
As the prototype resulting from the second research question forms a baseline implementation, it leaves room to extend the design and improve its performance and accuracy. Therefore, with the third research question, we want to explore the possibility of improving the prototype by using source code data presented by existing unit test suites. Therefore, the third research question intends to explore the capabilities of using information regarding class usage in unit tests to create a more accurate class matching algorithm.

Chapter 4

Review of Related Work

This thesis aims to design and implement a system that is able to generate glue code based on Gherkin scenarios. Hence, this chapter explores related work on the topic of computer generated integration tests.
First, an analysis of current approaches that use the Gherkin framework as a base for test generation is conducted.
Second, the third research question of this thesis is about using unit tests to enhance the accuracy of our system. Hence, a review of projects that use unit tests to help with generation of integration tests will be conducted.
Lastly, an analysis of the most recent developments in the field of computer generated tests is conducted.

4.1 BDD Frameworks

Kamalakar et al. describe an approach to automatically generate tests from natural language description [16]. In their paper, they propose an approach of removing the glue code step entirely by using natural language processing (NLP) techniques to analyze the Gherkin scenarios and then mapping them to existing code implementations. The integration tests are then generated by using a probabilistic matcher that uses properties from the code in the software project that the scenario refers to, as well as the results of the NLP engine.
There are two main limitations with the approach of Kamalakar. First, the approach is limited to software projects that have to contain written code or code stubs. This does not allow for using the BDD approach to its fullest extent. Second, the approach was only evaluated against a single test case study. In this case study, software modifications were made to a project to help with the code generation mechanism. Hence, the reliability of this approach has not been tested against multiple or unseen projects.

Storer et al. describe an approach that is able to generate the glue code based solely on the Gherkin scenarios. Where the approach of Storer differs from the approach of Kamalakar is that the approach of Storer does not use any available source code. With Storer's approach, test scenarios written with the Gherkin framework are parsed using a combination of part-of-speech (POS) tagging and semantic role labelling (SRL). Another key component of the approach described by Storer, is that they make use of a change detector. This allows the code steps to be regenerated when changes are made to the targeted software project. The code generator generates step functions using the information provided by the parser. With this approach, Storer et al. are able to achieve an accuracy of about 82%.

There are two limitations with the approach of Storer et al. First, the test data set is drawn from GitHub. While the size of the data set is substantial, many of the software projects seem to be 'toy' or tutorial projects. As a result, most of these projects only contain up to 10 Gherkin scenarios. This may have introduced a bias into the results of the evaluation. As with the previous

paper, this approach has not been tested on scalability. Second, the authors introduced an element of subjectivity into the evaluation by having to manually adjust test scenarios when a mismatch between naming conventions of the Gherkin scenarios and implementation details was present. They argue that only minor adjustments were made. However, in their paper, no criteria are described as to when the Gherkin scenarios are 'good enough'.

4.2 Using Unit Tests

Pezzé et al. [23] describe an approach to integration test generation that utilizes existing unit tests. Their approach has 3 different phases. First, class dependencies within the SUT are identified in the form of an object relation diagram (ORD). Second, the data flow information within the input test cases is computed. This is then used to segment the test cases into useful blocks. Lastly, new and more complex test cases are created from the code blocks extracted from unit test cases using the ORD and data flow information generated in the first two steps.
The ORD of the SUT is used to identify clusters of dependent classes that will be tested together in an integration test case. The dependencies are generated with a simplified version of the algorithm proposed by Briand et al. [5].

Grechanik et al. designed a tool called ASSIST [14] that automatically obtains models that describe frequently interacting components in software applications. With this approach, they intend to increase the effectiveness of each individual integration test. To extract information from the SUT, they make use of an already existing test suite. After executing the test suite on the SUT, a profiler collects execution traces. The execution traces are then analyzed by the frequent pattern matcher to find frequent patterns of method calls. From these patterns, a model is generated that is used to produce a more effective test suite. For the implementation of the pattern miner, the authors make use of a tool called BIDE [31]. The most relevant part of their paper is the method with which they are able to extract information from already existing unit tests. Information gathered consists out of classes, methods, and variables used within assert statements. Static analysis is used to extract these variables and classes from the source code.

To measure the effectiveness of the generated tests with this approach, the authors use mutation testing. Furthermore, the results of ASSIST are compared to FUSION [23], the approach described by pezzé et al. Although the results of ASSIST do not provide convincing results, it performs on par with FUSION on two of the test applications. Important to note is that ASSIST creates significantly fewer tests compared to FUSION or human generated tests. ASSIST also outperforms FUSION when tested against mutation testing. The main problem with these results is that the applications against which this approach was tested might not be good representatives of the average application used in industry.

4.3 Genetic Algorithms

The most recent developments regarding integration test generation have been in the form of genetic algorithms. There have been several papers published that propose a genetic algorithm approach for generating unit tests. **Ahn et al.** [1] propose an algorithm that is able to generate both unit tests and integration tests. They formulate the problem of test generation as an optimization problem that can be solved using a Search-Based Software Testing (SBST) approach. SBST is the process of automatically generating test data according to a test adequacy criterion using meta-heuristic searching algorithms such as Hill Climbing, Particle Swarm Optimization, Simulated Annealing as well as Genetic Algorithms. Anh et al. state that Genetic Algorithms have become the most widely used technique for the SBST approach. A Genetic Algorithm is based on the natural evolution (crossover & mutation) and selection of the fittest individuals for survival. A genetic algorithm usually consists out of three components: the representation of individuals or chromosomes that encodes a solution to the given problem, the fitness function to evaluate the relevance of each individual, and lastly, the genetic operators including crossover,

mutation, and selection to improve the search population. In our case, an individual is represented by an integration test case. The fitness function's main goal is to find the best results, and is therefore very important. Ahn et al. provide a genetic algorithm for generating test data for OO programs. The exact algorithm details can be found in their paper [1]. The most interesting part of this paper is the performance of the proposed algorithm. The only relevant results are the performance results of the integration test generation. For this, Anh et al. used two case studies to assess how efficient the algorithm is in detecting the errors present in the code. The enhanced algorithm changed to specifically address integration test generation was able to generate a good enough integration test suite to detect all the errors in both systems. The general algorithm used to also generate unit tests was only able to detect 51% and 74% of the errors present in the systems. That said, the enhanced algorithm did take a significantly longer time to create and run the tests. Important to note is that both case studies were Object-Oriented languages, and no details about the size of the projects were published.

4.4 Conclusions

The existing literature described in section 4.1, is used as a base for this project. The goal of both these papers is similar to the goal of this project. The papers in section 4.1 present techniques for test generation with promising results. Therefore, a combination of the techniques used by these papers will form a starting point for this thesis.
In this thesis, we do explore how unit tests can be used to improve test generation accuracy, but not with techniques presented in the related literature described in section 4.2. Although some techniques described in section 4.2 look promising, implementing these techniques would require a lot of time given their complexity. Although we did not implement any of the techniques described in section 4.2, the literature described in that section does provide a nice overview of the existing literature on improving integration tests based on unit tests.
In section 4.3, we describe the most recent developments in the area of test generation. Genetic algorithms are the most recent and best performing technique for generating software tests. However, this technique is very complex to implement within the context of this thesis. Given the goal of this thesis and the existing literature presented in section 4.1, we decided not to use the genetic algorithm approach.

Chapter 5

Prototype Overview

This chapter presents a detailed overview of our approach to integration test generation. The prototype proposed in this chapter is focussed only on the object-oriented language Java. The goal of the prototype is stated, as well as the requirements it is constrained by. Also, an explanation of the architecture of the system and its core elements is described.

5.1 Goals and Requirements

As stated in chapter 1, no good integration test generation tool currently exists in practice. Based on the approaches described in chapter 4, there are several approaches that can be used to achieve this goal. Most of these methods do not allow for user input on what tests are created. However, user input on the generated tests is desired with maintenance in mind.
With the approach described by Storer [27] and Kamalakar [16], the Gherkin language is used to maintain a natural language description of the test suite. The main benefits of this approach are that it allows for user input, as well as easier maintenance and extension of existing tests. The main purpose of the use of this language in both papers is to let users write their tests in structured natural language. As described in the literature research, both papers describe a tool that generates glue code based on these natural language scenarios. However, both papers have their disadvantages. These are discussed in the sections below, as well as section 4.1.

5.1.1 Compared to Previous Work

The approach described by Storer [27] focuses on Python only. With Python, code does not necessarily follow a method and class structure. Also, unlike Java, Python does not use strict variable and function types. As a result, methods used for generating the glue code also use different language specific features. Another important thing to note is that Storer does not use the target projects' source code for generating the integration tests. Instead, their approach relies solely on NLP.
Kamalakar describes an approach [16] that is focussed on the Java language, but uses outdated NLP techniques. Kamalaker does extract source code properties with the use of the Java reflection API. In previous research, Kamalakar developed a wrapper around the Java reflection API which allowed them to easily use it with their paper on test generation. However, this wrapper library is not publicly available. Therefore, regarding this project, we cannot reproduce their approach to code analysis. Another thing to note is that Kamalakar generates JUnit tests and not glue code. However, this only affects the code generation and not the model that maps the Gherkin scenarios to source code.
For our approach, we want to use several aspects presented in both papers. From the paper of Storer, we want to build on their implementation of NLP. As their paper presented promising results, it should establish a good foundation on which our tool can be built. The approach of

Kamalakar provided a promising architecture where both the source code and the natural language descriptions are used to generate the integration tests. Therefore, our prototype will use a very similar architecture. Their architecture can be found in Figure 5.1.

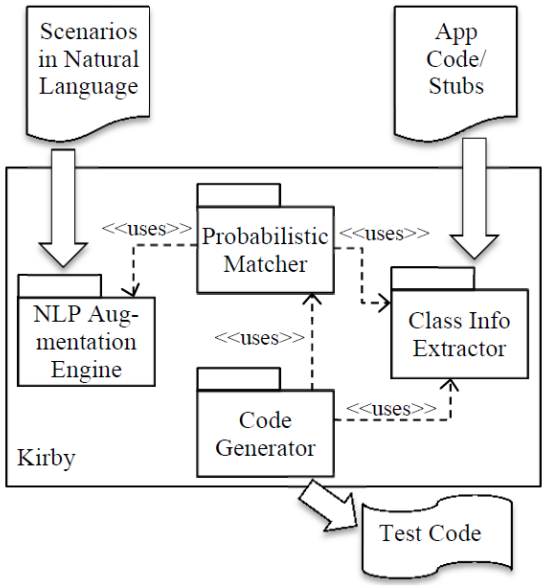

Figure 5.1: The architecture used by Kamalakar [16].

Similar to the approach of Kamalakar we will also make use of the fact that Java follows a strict object-oriented structure. Also, similar to Kamalakar is that our prototype will be developed using Java 8. The choice for Java 8 was to ensure that specific libraries used within this project work only for Java 8. We could use newer versions of Java, but then we cannot guarantee bug-free behavior from the libraries. The prototype will use a code analysis component that keeps track of public methods and also classes and their public fields. Furthermore, we also keep track of method and object parameters. With this approach, we can build a search space based on a target projects' source code. Where our approach differs from previous work is that we attempt to filter the search space based on the use of parameters. From a Gherkin step, we can extract parameters. As methods in Java can be identified by their name and parameters, we can use the extracted parameters to filter out methods that do not support the correct parameter types. This has not been explored in previous work.
Therefore, for this project, the initial goal is to explore search space filtering to improve upon previous work.

As understandability of the generated integration tests is important for this project, the implementation of the prototype will also make use of the Gherkin language. Similar to previous work, the tests are written in the Gherkin language. Based on these natural language test scenarios, the prototype will generate an executable set of functions. The set of generated functions is called the 'glue code', this was explained in section 2.3.5. With the help of Cucumber, as explained in section 2.3, the integration test suite can be executed based on the generated glue code. In Figure 5.2, we can see a simplified schema of the system architecture. The names *Cucumber* and *Gherkin* are cucumber species. To stay within the cucumber analogy, our prototype can be seen as a farmer growing crops. As such, the decision was made to name the prototype *FARMER*. The system follows a multi-component architecture, dividing the different responsibilities between the components. In general, the architecture consists out of the following general layers:

- **NLP Augmentation:** The main responsibility of this layer is to parse and transform the natural language descriptions into labeled information. Based on the output of this

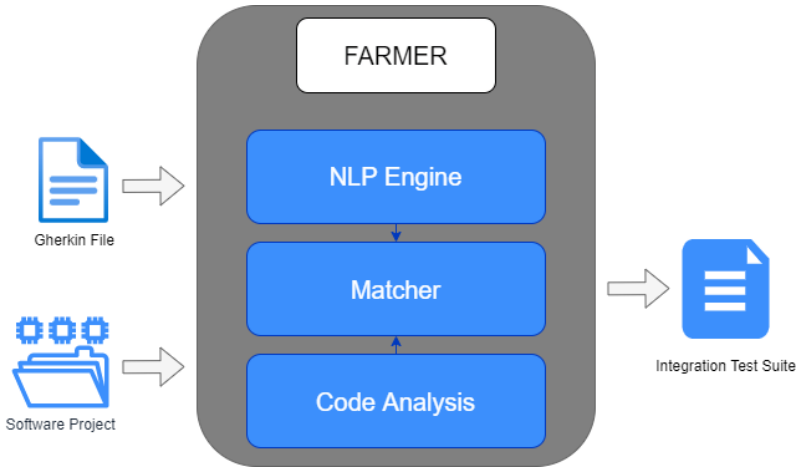

Figure 5.2: Simplified overview of *FARMER* architecture.

component, the system is able to get the needed information from the Gherkin scenarios.

- **Code Analysis:** This layer is responsible for analyzing the SUT and extracting an AST. This component can be queried by the system to retrieve source code properties. These properties are often names of classes and methods, but also parameters.

- **Matcher:** This layer comprises the logic to match the Gherkin scenarios to executable code. It functions as the most influential component for achieving the goal of this project.

5.1.2 Gherkin scenario quality assumptions

The user creates test scenarios using the Gherkin language. This language is based on a certain syntax [1]. As this syntax has matured over the years, there exist multiple ways in which a scenario can be written. To support the complete syntax with *FARMER* would require a lot of work that does not contribute to answering the research questions. In the syntax, there exist multiple keywords that essentially serve the same purpose. The main difference being ease of use. As such, *FARMER* supports a strict subset of the Gherkin syntax:

- Only one step of the keywords Given, When & Then exist per scenario.

- The And keyword is supported.

- *FARMER* only supports the 'Scenario Outline' with 'Examples' keywords to distinguish tables. An example of this can be found in Listing 5.1.

- Other keywords than the keywords described above are not supported by *FARMER*.

Note that with this subset of rules, a user is still able to describe the same behavior compared to the complete Gherkin syntax. When data tables are used, as displayed in Listing 5.1, variable types can be inferred. To keep this process simple, it is assumed that all types in a column are displayed consistently. For example, when numbers are used, do not mix integers and decimal values.
Furthermore, to make the matching process less complex, we assume all test scenarios adhere to the following format:

[1] https://cucumber.io/docs/gherkin/reference/

```
Scenario Outline: customer orders a soda
  Given there is a bar
  And the bar has a soda machine with <liters> of <soda>
  When the customer purchases a <soda>
  Then the machine contains <amount> liters of <soda>

  Examples:
    | liters | soda    | amount |
    | 2.0    | Pepsi   | 1.5    |
    | 1.0    | Fanta   | 0.5    |
```

Listing 5.1: A Gherkin Scenario using a data table.

- If multiple actions/states are to be described, the And step type is used. As a result, each step in a Gherkin scenario should refer to only 1 action or state. The NLP engine struggles to give a correct analysis when prompted with sentences containing multiple actions or states. Hence, when we adhere to this requirement, the NLP engine is the most accurate in its analysis of the natural language sentences.
- Every step type is used for its intended purpose only. This means that the *Given* keyword is only used to describe the initial state of the system, the *When* keyword is only used to describe actions, and the *Then* keyword is only used to describe the expected result.

Another issue which can hamper the accuracy of *FARMER* is the quality of English used to write the scenarios. As such, it is assumed that the language used to write the scenarios meet the following requirements.

- The scenarios are written using proper English. This means no spelling or grammar mistakes that could give the sentence an ambiguous meaning.
- The business domain language must be used when referring to the behavior of the software system.
- When referring to certain implementation details, the language used must be representable of the implemented business process. For instance, when in a business process a vending machine is created first before it receives an inventory, the Gherkin scenarios should describe this process in the same order.

The matching of text to code is influenced by all the factors described above. In a Gherkin scenario, steps build upon previous steps. Therefore, one spelling or grammar mistake in a step could have a snowballing effect on all the remaining steps of a scenario. Also, NLP is only able to process the actual text. As such, hidden implementation details are very hard to discover by the NLP Engine. As such, the development of *FARMER* was based on the requirements described above. Hence, if the Gherkin scenarios adhere to the requirements described above, the prototype has the best chance of generating the correct glue code for all the scenarios.

5.1.3 Java source code requirements

Java 8 source code contains certain language specific feature that are not trivial to deal with. Two of these non-trivial features available in Java 8 are inheritance and lambda functions. For this project, we regard lambda functions out-of-scope. However, we cannot ignore inheritance.

We encounter inheritance in the code analysis layer. Inheritance is relevant only to the method matching algorithm. As class matching is based on constructor and name only, inheritance is not relevant. However, when we want to retrieve all methods that are available to a class, if inheritance is used, not all methods are present in that class. Therefore, to deal with this problem, we need to recursively go through all super-classes in order to find all public methods. The implementation details of this approach can be found in section 6.4.5.

5.2 The *FARMER* Prototype

The goal of *FARMER* is to generate the integration test suite. In order to achieve this goal, *FARMER* provides a matching process to pair natural language to source code. We use NLP to extract important information from the natural language sentences. Source code analysis is used to create an overview of available methods and classes in the target systems. The results of these two steps are then combined to create a match between natural language and source code. This section presents the architecture to achieve this goal.

5.2.1 Architecture Overview

In this section, the high-level architecture of *FARMER* is presented. A diagram depicting the architecture can be found in Figure 5.3. The architecture is split into 5 main components: *Parser, NLP Analysis, Matcher, Code Analysis,* and *Code Generator*.

- **NLP Engine** layer is represented by the *Parser* and *NLP Analysis* components. These components analyze the test steps written in natural language and extract useful information. The implementation is based upon the paper of Storer et al. [27]. To emphasize the use of their pre-existing code, a red colored dashed line is visible in Figure 5.3.
- **Code Analysis** layer is represented by a similar named component. This component analyzes the target software project, and creates an Abstract Syntax Tree (AST). It then provides functionality to query the AST for specific information. This layer will also encompass a unit test analysis component responsible for extracting specific class information from unit tests.
- **Matcher** layer is represent by the *Text-to-Code Matcher* and *Code Generator* component. The *Matcher* component, contains the main algorithm for matching the Gherkin tests to application code. The code generator parses the output of the *Matcher* component to actual source code and outputs this to a file.

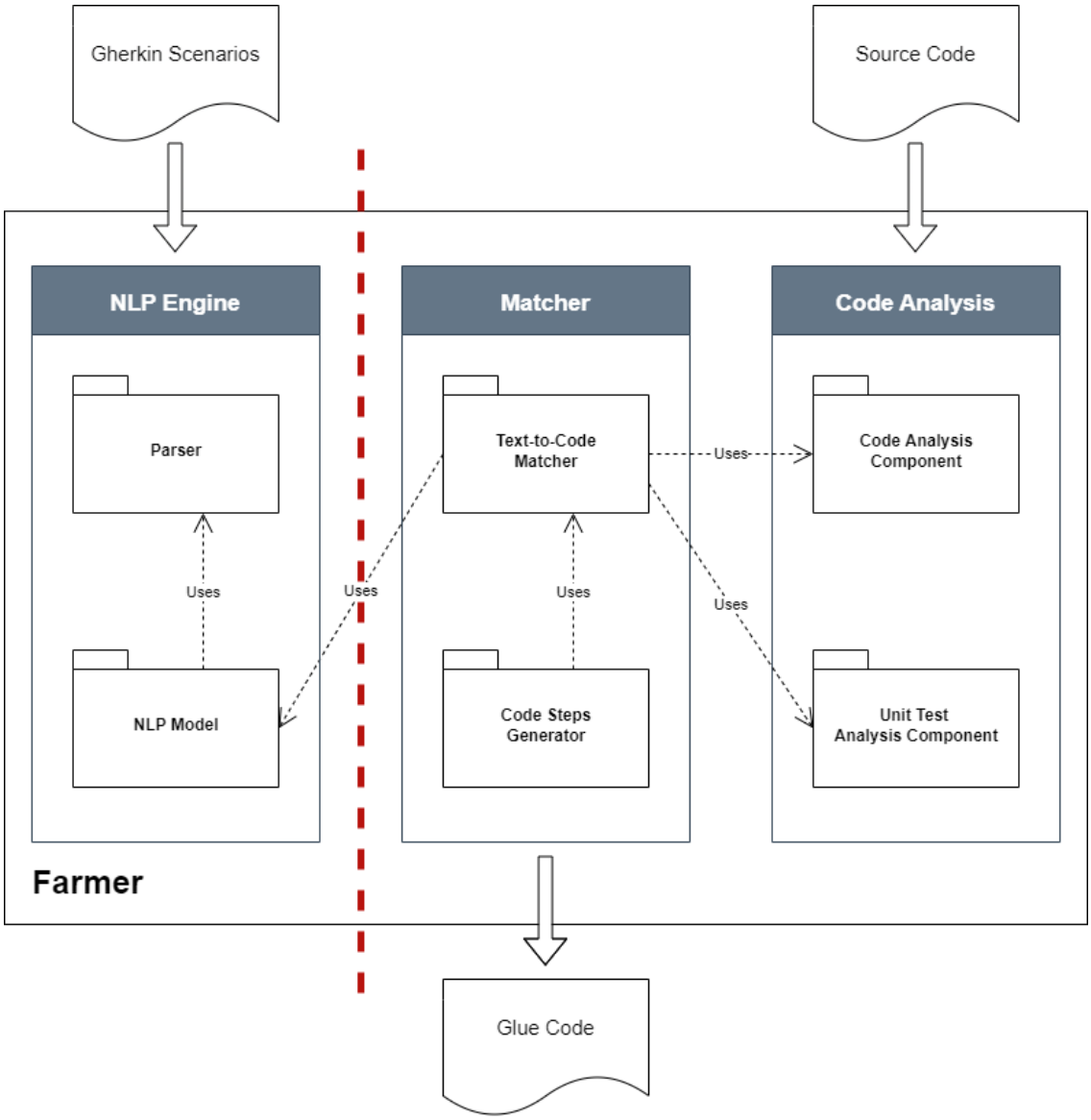

Figure 5.3: High-level architecture of *FARMER*.

5.2.2 Unit Test Analysis

The testing pyramid described in section 2.1 is often regarded as best practice when it comes to software testing. The base of the pyramid consists of unit tests. In Java, unit tests are designed to test the functionality of single methods. However, methods are declared in the context of classes. As such, a typical unit test in Java contains the initialization of a class object, one or more method calls, and a number of assert statements. As described by the third research question in section 1.2, we want to explore the possibility of improving the accuracy of *FARMER* by extracting class information from unit tests.
Based on popular open source projects such as Apache Spark [2] and Google Guava [13], as well as private projects of InfoSupport, we can see that unit test function names are as descriptive as possible. Example function names can be found in Listing 5.2. From these sources, we conclude that if best practices are followed, that the unit test names are representative of the actions taken in the unit test. Also, most of the function names follow a *precondition, action, post-condition* format. As the Gherkin scenarios follow a similar format with their *Given, When,* and *Then* keywords, we would be able to match a unit test to a scenario based on its steps and the test function name. Based on the fact that we are matching sentences with sentences, cosine similarity is a good matching algorithm to start out with. Properties of this algorithm are discussed in section 5.2.3.

```
public async void GivenANonExistingJobTheHandlerShouldReturnNull() {...}

public async void GivenAnExistingJobTheHandlerShouldHandleTheCommand() {...}

public void testCharacterSplitWithDoubleDelimiterOmitEmptyStrings() {...}

public void ValueObjectEqualityShouldDetectEqualValues() {...}

public void testQueryNoMatchWhenExpected()
```

Listing 5.2: Examples of unit test names present in open source projects

To show that this also holds for function names, we analyzed the cosine similarity of the sentence *"Given there is a bar and a machine that contains 5 beans and 3 milk"* and several unit test function names. The results of this experiment can be found in Table 5.1. As we can see in the table, the difference between using identical cardinal digits has only limited impact on the similarity score compared to extending the sentence with non-related words. Also, the difference in similarity between using the parameters of coffee machine and soda machine is quite big. From the table, we can conclude that certain words such as the *"beans and milk"* have big impact on the cosine similarity. The correct use of quantitative words do have an impact on the similarity, but much less impact compared to using different machine descriptions.

Sentence	**Similarity**
given bar get machine with 5 beans and 3 milk	0.601
given bar get coffee machine with 5 beans and 3 milk	0.589
given bar get machine with beans and milk	0.540
given bar get coffee machine with beans and milk	0.531
given bar get machine with 9 beans and 4 milk	0.515
Bar has machine with beans and milk order coffee returns empty machine	0.422
given bar get machine with soda	0.272
given bar get machine with 5 liters of soda	0.246
Bar has machine with coca-cola order soda returns empty machine	0.224

Table 5.1: Cosine similarity of the sentence "Given there is a bar and a machine that contains 5 beans and 3 milk" compared to sentences with different level of detail.

In Java, a unit test's set up consists out of object instantiations. If we match a representative unit

test compared to the Gherkin scenario. We could use these object classes to filter the class search space of the class matching algorithm used within the matcher component. As a result, we expect that the accuracy of *FARMER* will improve.

5.2.3 String Similarity Metrics

Matching source code to natural language is quite challenging, as argued by Kamalakar et al. [16]. In line with their findings, the components in the Matcher layer use 3 different string similarity metrics to match text to pieces of code.

Levenshtein Distance The most basic similarity metric is the edit distance between two strings. To implement this metric, we use normalized Levenshtein distance. Levenshtein distance between two strings is the minimum number of single-character changes needed to change one string into the other. Normalized Levenshtein is the normal Levenshtein distance divided by the length of the longest string. This results into values in the interval [0.0, 1.0]. The use of abbreviations in source code is considerably less penalized using normalized Levenshtein. This algorithm works well in situations where the wording in the source code is very similar to the wording in the Gherkin scenarios, but it will struggle when synonyms are used.

Cosine Similarity For matching a part of a sentence, in case of verb matching, a vector space is more accurate. Therefore, we use cosine similarity to match multi-word strings [12]. Given two multi-word strings represented by the vectors x and y, then cosine similarity is defined by the cosine of the angle between the two vectors. This is defined as the following equation:

$$sim(x, y) = cos(\theta) = \frac{x \cdot y}{||x||||y||} \tag{5.1}$$

Where $||x||$ represents the Euclidean norm of the vector x.

Second Order Similarity To match strings based on semantic similarity, the Matcher layer uses a library called DISCO [17]. This library provides a second order similarity measure between two strings based on usage in big databases like Wikipedia. For example, based on the occurrence of strings in Wikipedia articles, the second order similar words of the word *palms* is the following list of words:

```
palms (0.1345) coconut (0.1059) olive (0.0870) pine (0.0823) citrus (0.0745)
oak (0.0677) mango (0.0652) cocoa (0.0645) banana (0.0627) bananas (0.0623)
trees (0.0570) fingers (0.0560)
```

This list of words can be seen as the second order similarity vector of *palms*. Using these vectors, the second order similarity of two words can be computed.

Depending on the context of the string matching, a combination of these algorithms is used to determine string similarity. We use these algorithms for the following implementations: class matching (see section 6.4.3), method matching (see section 6.4.4), constructor matching (see section 6.4.3), parameter matching (see section 6.4.3 and section 6.4.4), and variable matching (see section 6.4).

Chapter 6

Prototype Implementation

In the previous chapter, the design decisions and background theory regarding *FARMER* was discussed. In this chapter, the technical details regarding the implementation of *FARMER* are discussed. The prototype is implemented using several separated components with isolated tasks. This allows for a modular design where components can be easily added and removed. We discuss the implementation of the components in the following order:

- In section 6.1, the implementations of the NLP Engine is discussed.
- In section 6.2, implementation details of the source code analysis components are given. It consists of information regarding the frameworks that are used, as well as how other components are able to interact with this component.
- In section 6.3, we discuss usage of unit test code analysis to improve the matching process.
- the implementation details of the matching component are described in section 6.4. This part presents details on specific matching algorithms used for matching the Gherkin scenarios to source code. Also, the implementation of search space filtering is discussed.
- the chapter concludes with the implementation details of the code generator in section 6.5.

6.1 NLP Engine

This section describes the implementation of the complete pipeline of extracting the necessary information from natural language using NLP models and a parser.

6.1.1 Parser

The parser is responsible for parsing Gherkin scenarios into separate sentences. This way all the documentation not contributing to the test scenarios will be ignored, leaving only the relevant Given, When & Then steps. When the parser is called to parse a file, it will return an array containing separated step descriptions. Storer et al. [27] have put a link to a GitHub repository in their paper. In this GitHub repository, the source code of their tool, *behave_nicely*, is publicly available. The tool of Storer et al. implements this parser already. Therefore, *FARMER* will use the same implementation.

6.1.2 NLP Analysis

Storer et al. implement a NLP analysis that makes use of part-of-speech (POS) tagging and semantic role labelling (SRL), both of these techniques are described in section 2.4. As explained in section 2.4, these two techniques are implemented using the NLTK library [4] and AllenNLP library [11] for Python.

6.1.2.1 NLP Processing

Building an NLP processing unit from scratch requires a process of model selection, training, and evaluation. As the focus of this thesis is not to build an NLP processing component, *FARMER* will use the same NLP analysis as described by Storer et al. [27]. Based on their results, their NLP component seems to be accurate enough for this project. The NLP analysis component uses the parsed scenarios as input and outputs a detailed analysis of the step description. In this analysis, important words such as verbs and nouns are analyzed in their relation with the rest of the sentence. Using these techniques, the analysis is able to capture the semantic meaning of words and every word is then labeled to what its role is within the sentence. The labels used by NLP models use a combination of the PropBank [20] and TreeBank [18] datasets to label data. In the field of SRL, PropBank is one of the studies widely recognized by the computational linguistics communities. PropBank is the bank of propositions where predicate-argument information of the corpora is annotated, and the semantic roles or arguments that each verb can take are stored. The most important labels regarding this project are Arg0 and Arg1. Arg0 stands for the Agent or Causer in a sentence. Arg1 stands for the Patient or Theme of a sentence. The TreeBank dataset

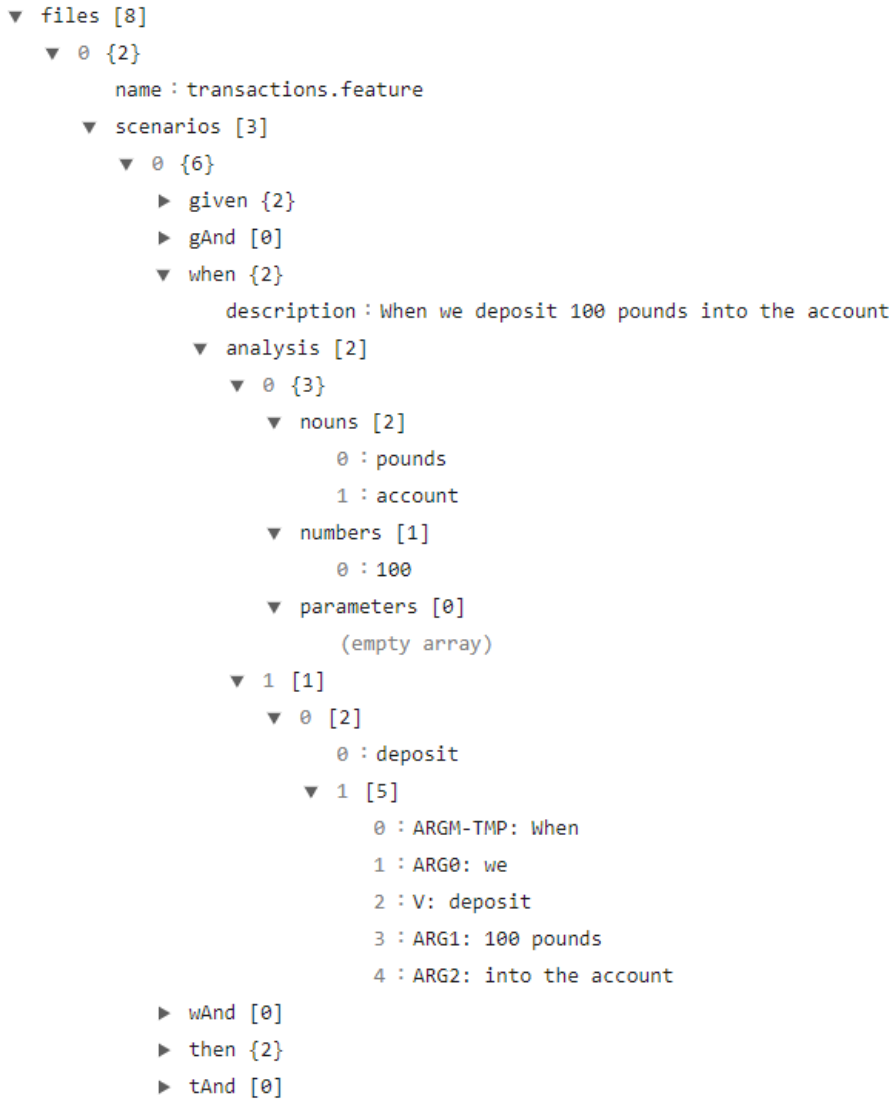

Figure 6.1: Visualization of the JSON format.

is similar to the PropBank dataset. However, the TreeBank dataset differs from the PropBank dataset as the TreeBank dataset offers labels focussed at the word level instead of sentence level. Therefore, the TreeBank dataset is more suited for NLP models that label individual words. Regarding this project, the POS-tagger focuses on individual words and uses the TreeBank dataset. Also, the semantic role labeling model used in this project uses the PropBank dataset.

The NLP models are implemented in Python, and *FARMER* is coded in Java, the resulting NLP analysis is written to a JSON file in order to preserve the existing scenario structure. An example of the JSON format can be found in Figure 6.1. The shown JSON representation is that of the Gherkin scenario depicted in Listing 6.1. To give a complete representation of all the labels, the *When* step is shown in detail. As can be seen in Figure 6.1, the Gherkin step description is always preserved, as well as the scenario the step belongs to. On top of this, a step has an analysis property that contains the results of the NLP analysis.

```
Scenario:
    Given a bank account with initial balance of 0
    When I deposit an amount of 100 into the bank account
    And I withdraw an amount of 50 from the bank account
    Then the balance of the bank account should be 50
```

Listing 6.1: A Gherkin scenario of the Banking example

6.1.3 Use of Redundant Model

During testing, there have been cases where the accuracy of the POS-tagger provided by the NLTK library fell short. In this section, we describe the main problem and its solution.

6.1.3.1 Mislabeling of verbs

The problem was encountered when *FARMER* crashed during the development phase. After debugging the program, the issue seemed to be that a sentence did not contain any verbs according to the NLP analysis. The *Matcher* component did not expect this was possible, as a sentence describing a *Then* step must always contain at least one verb. The sentence responsible is listed below, together with its part-of-speech labels.

```
Sentence: "Then the stock reduces in 1 unit"
tags: [('Then', 'RB'), ('the', 'DT'), ('stock', 'NN'), ('reduces', 'NNS'),
           ('in', 'IN'), ('1', 'CD'), ('unit', 'NN')]
```

As we can see, the word 'reduces' is labeled as 'NNS', this label means that this word is classified as a plural noun. Humans immediately recognize that the conjugation 'reduces' of the verb reduce cannot be a noun. As this mislabeling is quite obvious, there are bound to be other verbs that will be mislabeled by this model as well.

6.1.3.2 Solution

To work around this issue, we introduced a second model. This model is the POS-tagger from the Stanford University Library [28]. While testing on the same data, this model seems to make significantly fewer mistakes on verb labeling. Using fuzzy logic to detect whenever the standard POS-tagger has failed, the redundant POS-tagger is activated instead and provides an alternative analysis. The pseudocode for model selection can be found in Algorithm 1. After evaluating the performance of this simple logic switch, there are already considerably fewer cases where the NLP model fails. Important to note is that both models struggle with sentences containing grammar mistakes. That said, the NLTK model seems to be affected less by grammar mistakes compared to the Stanford model. An example where this is the case can be found in the appendix.

Algorithm 1 Part-of-Speech model selection logic

```
tags = NLTK.getPosTags(sentence)
verbs = extractVerbs(tags)
if verbs == empty then
    tags = StanfordNLP.getPosTags(sentence)
end if
```

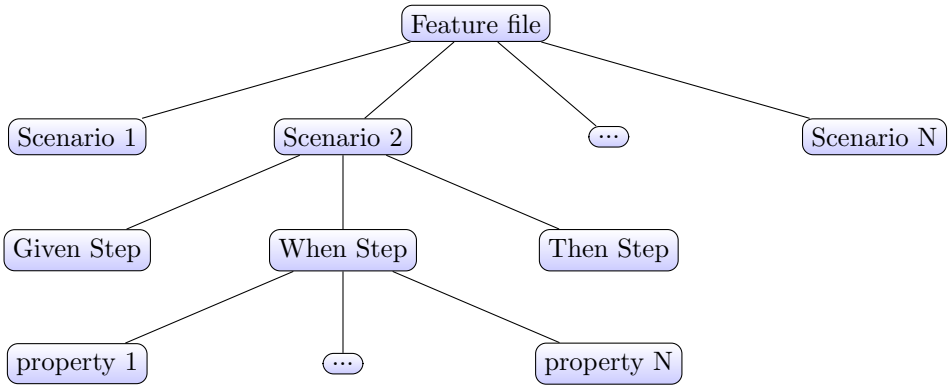

Figure 6.2: The tree representation of a feature file

6.1.4 Data structures

Java does not naturally support JSON format data. In order to store the NLP analysis data, we have designed custom data structures for storing the JSON data. We have created a class *NLPFileReader* that is responsible for converting the JSON file into a tree structure. As JSON is a hierarchical data structure, it naturally lends itself to a tree data structure implementation in Java. Therefore, in this section, we discuss the implementation of the data structure for storing the NLP analysis of Gherkin scenarios.

In order to build the tree structure, we have to decide on a root node. Gherkin scenarios are grouped together in a file, called the feature file. All the glue code belonging to a feature file is also stored in a single file. To preserve this link, we want to build a single tree per feature file. Using a feature file as the root, the Gherkin scenarios will naturally become its direct children. Using the hierarchy of the Gherkin syntax, we can again split on the different Given, When and Then steps. A visual representation of the resulting data structure can be found in Figure 6.2.

6.1.4.1 Step objects

The NLP analysis and the code generation is all done at the Gherkin Step level. To represent these steps in Java, a custom object needs to be designed. For this project, there exist 3 main step types: *Given*, *When*, and *Then* steps. The 3 different step types all require different logic in the matching algorithm. However, they do share a number of properties. Therefore, we decided to create implement this using inheritance. The inheritance structure and the shared properties can be found in Figure 6.3. The properties *nouns, numbers, parameters,* and *srlLabels* cover all the NLP data from JSON file.
Now, there also exists an *And* keyword. The Gherkin *And* keyword is used to split a compound *Given*, *When* or *Then* step into multiple separated steps. This is done to maintain the *each step describes one action* requirement described in section 5.1.2. As such, *And* steps are always related to a 'parent' keyword. To maintain these relations in the scenario tree, these relations are modelled as a parent-child relationship. To deal with the *And* keyword, each step object also has a property *AndSteps* that consists out of a list of *Step* objects representing these *And* steps.

Finally, to easily keep track of the results of the matcher, the property *MatchResult* is used for storing the results of the source code match.

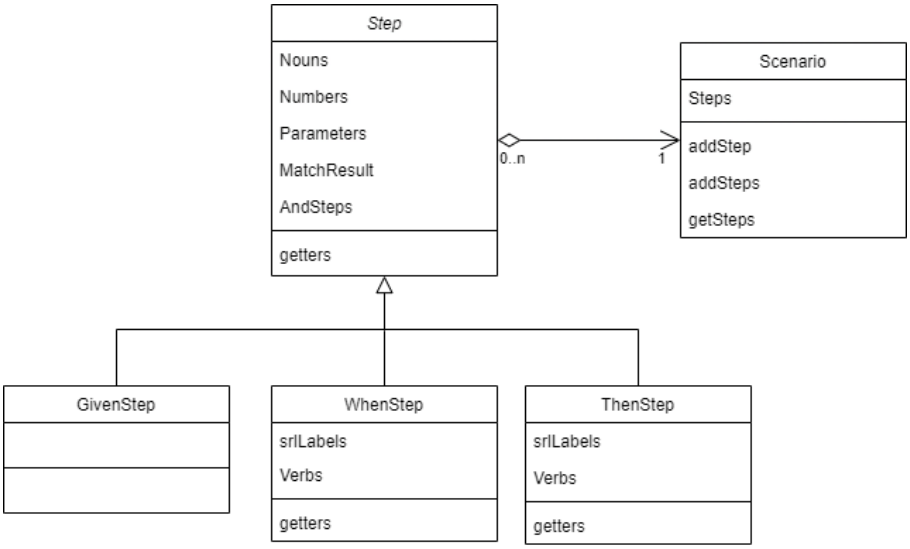

Figure 6.3: The hierarchy of a scenario tree.

6.2 Code Analysis Component

In this section, we will discuss the implementation of the Code Analysis component. This component is part of the Code Analysis layer and responsible for providing functionality to extract class and method information about the SUT. In the section below, the implementation of this component is discussed.

6.2.1 Component Implementation

FARMER uses the JavaParser library [26] to extract an AST of the source code. A simplified visualization of the JavaParser AST can be found in Figure 2.5. In order to extract class names, method names, and parameter types, the Code Analysis component uses the built-in visitor pattern. As we can see, nodes representing a class, constructor, or method have a specific type. This property is used in the algorithms below to extract the needed information from the AST. In Algorithm 2 and Algorithm 3, the algorithms that are implemented by the Code Analysis Component for extracting the names of methods and classes can be found. In order to not repeat the creation of AST's after we have extracted the class names of a file, we save a mapping of class names and their AST. The mapping is used when extracting method names linked to a known class. Otherwise, worst case scenario, to find the name of a certain method, we would have to rebuild an AST for every file before finding the correct method.

Algorithm 2 function to extract all class names out of a source code file

```
function findClassNames(file)
    cu = getCompilationUnit(file)
    for ClassOrInterfaceDeclaration ∈ cu do
        classnames.add(ClassOrInterfaceDeclaration.name)
    end for
end function
```

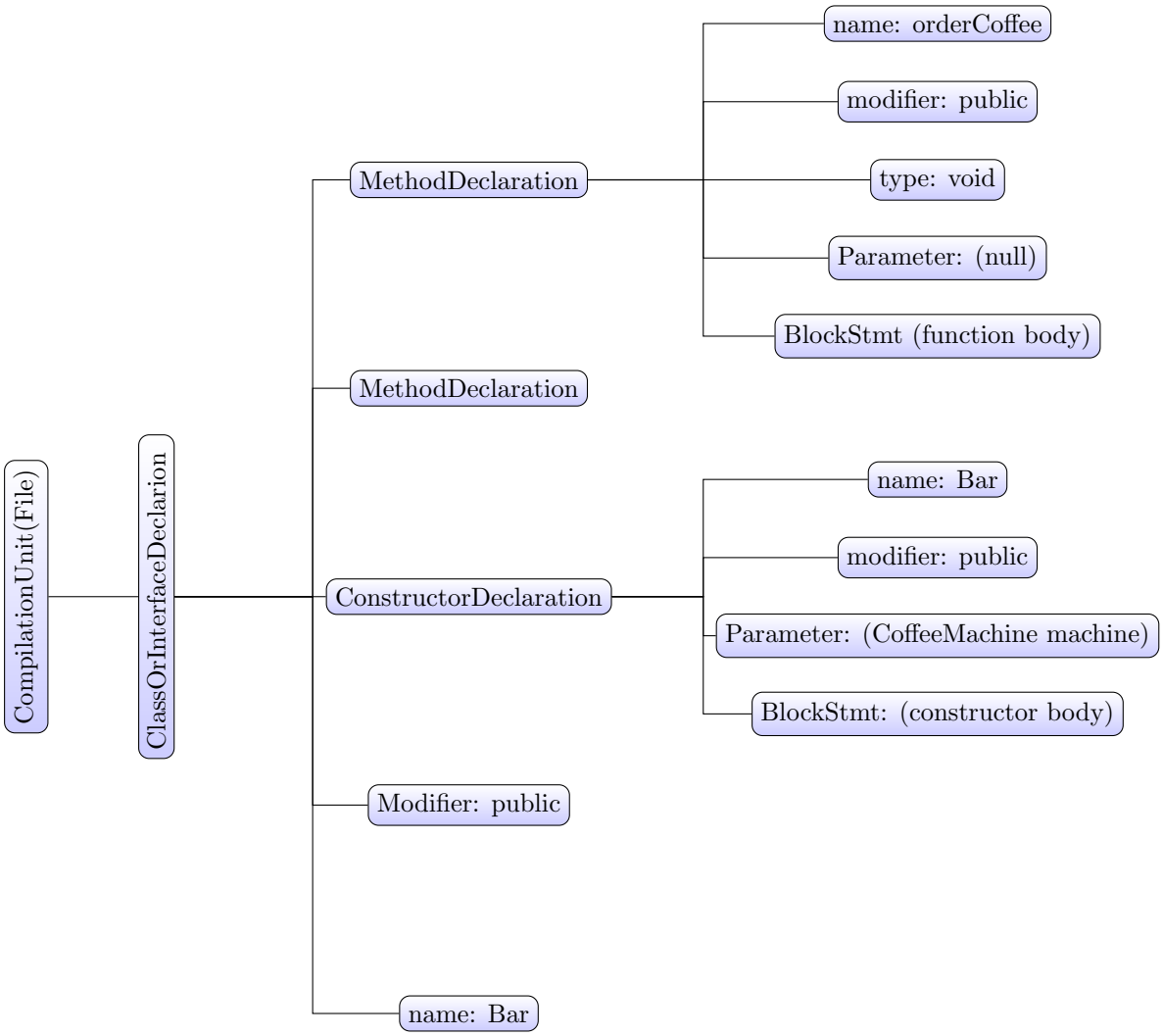

Figure 6.4: Visualization of the JavaParser AST.

The most important feature of the Code Analysis component is that it has functionality to filter the methods and constructors based on their parameters. The JavaParser library provides functionality to query the AST for specific nodes, but not the ability to search for methods or constructors based on a specific parameter set. Therefore, we have extended the functionality of JavaParser by creating functions that are able to provide this functionality. The techniques used for method and constructor filtering are identical, apart from searching for their respective nodes in the AST. As shown by Figure 2.5, both their nodes in the AST contain all the necessary child nodes for parameter filtering. The algorithm for parameter filtering can be found in Algorithm 4. This algorithm is specific to parameters of methods, but as said above, the techniques used are identical to constructor parameter filtering.

The steps described by Algorithm 4 are straightforward. First, the correct AST of the target class is retrieved. Then, for each method present in the AST, check if its parameters match the parameter types that we are looking for. The *reduceComplexTypes* subroutine is responsible for breaking down complex Java types, these can be custom Objects, but also collection types. The subroutine reduces each of these complex types to their basic types. With respect to this

Algorithm 3 Extracting method names from JavaParser AST

```
function findMethodNames(className)
    cuClass = cu.getClassNodeAST(className)
    for MethodDeclaration ∈ cuClass do
        methodnames.add(MethodDeclaration.name)
    end for
    return methodnames
end function
```

algorithm, basic types are *double, int, String, long,* etc.

Algorithm 4 Filtering based on parameter types

```
function filterParameters(className, targetParameters)
    cuClass = cu.getClassNodeAST(className)
    for MethodDeclaration ∈ cuClass do
        parameters = MethodDeclaration.getParameters()
        parameters = reduceComplexTypes(parameters)
        if parameters == targetParameters then
            methods.add((MethodDeclaration.name, parameters))
        end if
    end for
    return methods
end function
```

6.3 Unit Test Analysis

In this section, the implementation of the Unit Test analysis component is discussed. The implementation of this component is based on the described design in section 5.2.2. First, we discuss the design of the analysis component. Afterwards, the component implementation is discussed. Finally, an example is given to demonstrate the power of using unit tests in generating the glue code.

6.3.1 Component Implementation

To extract object class names from a unit test, the Unit Test Analysis component implements Algorithm 5. In this algorithm, the first step is to extract the *Given* step, together with its linked *And* steps, from the Gherkin scenario. This will form the base sentence to which we need to find a suitable unit test. Next, the subroutine *getFunctionNames* extracts all unit test function names from the test file using JavaParser. This is done using an implementation of Algorithm 3. Also, the subroutine preprocess the function names to actual English sentences. As we now have the unit tests and the Gherkin sentence, the subroutine *computeBestSimilarity* computes the similarity between the unit test functions and the Gherkin step using cosine similarity. The subroutine will return the function name that is the most similar to the Gherkin step. Finally, the classes present in the matched unit test are extracted with the subroutine *extractClassNames*. This subroutine uses the visitor pattern to find *ObjectCreationExpr* nodes representing the creation of an object type variable. In the last step, the algorithm returns the set of classes used in the unit test. This set of classes can be used by the matcher to filter the search space during the class matching procedure.

Algorithm 5 Extract class information from a matched unit test

```
function extractClassInfo(scenario)
    description = scenario.Given
    unitTests = getFunctionNames()
    bestMatch = computeBestSimilarity(description, unitTests)
    classNames = extractClassNames(bestMatch)
    return classNames
end function
```

6.3.1.1 Example

For this example, we use the Bar project from Table 7.1 in chapter 7. This project represents a bar that has two machines, a coffee machine and a soda machine. These are represented by the classes *CoffeeMachine* and *SodaMachine*. It also contains a unit test file for the class bar, testing its individual functions. Listing 6.3 contains the test functions used for this example. The Gherkin scenario we want to match can be found in Listing 6.2. We use Algorithm 5 to find the set of classes that should be used to match the *Given + And* steps.

```
Scenario: ask for a coffee
    Given there is a bar
    And the machine contains 5 beans and 3 milk
    When the client orders a coffee
    Then the machine at the bar contains 3 milk
    And the machine at the bar contains 4 beans
```

Listing 6.2: Example Gherkin scenario

The first step is to extract the Given sentence from the scenario. In this example, that sentence is "Given there is a bar and the machine contains 5 beans and 3 milk". Next, we compute the cosine similarity between this sentence and all the unit test function names. The cosine similarity of each function is visible in Listing 6.3 behind each function's name.

```
void givenBarGetCoffeeMachineWith5BeansAnd3Milk() {...} //similarity = 0.589

void givenBarWithEmptyCoffeeMachineAddACoffeeMachineWith6Beans() {...} //similarity
    = 0.322

void givenBarThatHasSodaMachineWith5LitersOfCocaColaExpectMachineToContain5Liters()
    {...} //similarity = 0.373

void givenEmptyBarAddASodaMachineWith1LiterOfPepsi() {...} //similarity = 0.276

void givenBarWithCoffeeMachineWith1BeansOrderCoffeeResultsEmptyMachineAfterwards()
    {...} //similarity = 0.355

void givenBarWithEmptyCoffeeMachineOrderCoffeeThrowsIllegalStateException() {...}
    //similarity = 0.201

void givenBarWithEmptySodaMachineFillSodaMachineWith4LitersOfFanta() {...} //
    similarity = 0.254
```

Listing 6.3: The functions present in the test file together with their similarity to the Gherkin scenario

As we can see in Listing 6.3 the function *givenBarGetCoffeeMachineWith5BeansAnd3Milk* is the most similar to the Given step. Now, in the last step of Algorithm 5 all class objects are extracted from the matched unit test function. In this case, the function *givenBarGetCoffeeMachineWith5BeansAnd3Milk* uses the classes *Bar* and *CoffeeMachine*.

This approach filters the *SodaMachine* class. This is especially important for scenarios where more general wording is used to describe a certain class. For the *And* step in Listing 6.2, the coffee machine class can be inferred from the beans and milk, but not from the word machine alone. If no class filtering was applied, the matcher can also match the *SodaMachine* class to this step. The correct class for this step is of course the *CoffeeMachine* class, but this can only be found if we would correctly make use of the context present in the sentences.

6.4 Matcher Component

The text-to-code matcher is responsible for matching the source code to the scenarios. It uses the NLP analysis to extract the key meaning of the sentence, and combines this with the analysis of the AST to find the most representative methods and classes. As a 1-to-1 match is not always possible, this component makes use of matching algorithms to find the best fitting source code implementation.

Algorithm 6 Matcher algorithm for Given steps

```
function matchGiven(step, context)
    classes = matchClass(step)
    constructor = matchConstructor(classes, step)
    context.setClass(constructor.getClass())
    if constructor.getParameters() == empty then
        return constructor
    end if
    parameters = orderParameters(constructor, step)
    constructor.setParameters(parameters)
    return constructor
end function
```

Algorithm 7 Matcher algorithm for When steps

```
function matchWhen(step, context)
    class = context.getClass()
    method = matchMethod(step, class)
    stepParameters = findParameters(step)
    if stepParameters == empty then
        return method
    end if
    parameterTypes = getTypes(stepParameters)
    parameters = orderParameters(stepParameters, parameterTypes)
    method.setParameters(parameters)
    return method
end function
```

6.4.1 General Matching Process

The matching process is separated into three different main methods. One for each of the main Gherkin keywords *Given, When,* and *Then*.
The pseudocode of the *Given* steps can be found in Algorithm 6. Given steps are responsible for instantiating a starting state of a program. In Java, this is done through object instantiations. As such, the function in Algorithm 6 focuses only on class and constructor matching. After a

suitable class and constructor are matched, and the constructor has parameters, the parameters are ordered based on their name and type. This is done to adhere to semantic typing requirements. To complete the match of this type, the returned constructor is formatted into a Rule object, which can be found in Figure 6.5.
When steps are matched by following the steps described by Algorithm 7. As these steps are used to describe actions, the object on which these actions are executed is often left implicit. To still be able to find the correct class, it is inferred from a context variable that keeps track of the class matched to the given step. Next, a method is matched based on the class and step properties. These step properties were defined in section 6.1.4.1. Afterwards, similar to the previous algorithm, parameters are added in a semantically correct order. Finally, the returned method is formatted into a Rule object, which can be found in Figure 6.5.

Algorithm 8 Matcher algorithm for Then steps

```
function matchThen(step, context)
    class = context.getClass()
    field = matchField(step, class)
    method = matchGet(step, class)
    labels = step.getSrlLabels()
    stepParameters == findParameters(step)
    for p ∈ stepParameters do
        if p ∈ labels.ARG1 then
            parameters.add(p)
        end if
        if p ∈ labels.ARG2 then
            checkValues.add(p)
        end if
    end for
    method.setParameters(parameters)
    comparator = getComparator(field, method, checkValues)
    assert = createAssert(field, method, comparator, checkValues)
    return assert
end function
```

To match steps starting with the *Then* keyword, Algorithm 8 is used. These steps are used to validate that the actions performed on the system lead to the correct resulting state. Therefore, the matching algorithm is used to create a suitable assert statement. Similar to the previous step, the object on which the checks have to be done is not always explicitly mentioned. As such, we first infer the class from the context variable. Next, we check if the matched class contains any public fields representative of the to be checked value and find the best match. Public fields do not necessarily exist, therefore a method is also matched for the same purpose. As the matched method needs to be used for an assert statement, the method needs to have the correct return type. We therefore use a more strict filtered set of methods. Otherwise, the logic for matching this method is identical to the logic used in the *matchMethod* function in Algorithm 7 for method matching. After matching a method, we find the correct parameters matching the method parameter types. Finally, we return the assert statement in the form of a *Rule* object.

6.4.1.1 Output

The matcher algorithm outputs a list of Rule objects to a scenario. For every line of code, there exists 1 rule object in the list. A rule object contains attributes with which the Generator component is able to generate glue code. Based on which step is matched, a Rule object is created to represent the resulting match. All possible attributes of a Rule object can be found in Figure 6.5. The properties serve the following purpose:

- The *advice* attribute represents what kind of code is generated based on this object. Advice is an enumeration type, it contains 4 different values: object instantiation, method call, assert statement and pass. The advice variable is used for each step type, and is expected to be not null.
- The *className* attribute represents the name of the class that is operated with. It is used for class Instantiation, method calls, and in assert statements.
- The *parameters* attribute contains a list of Strings. This variable is used to determine the parameters of a method call or object instantiation with a constructor.
- The *fieldName* and *assertExpr* attributes are used only in assert statements. If a class field is to be compared against some value, the *fieldName* attribute is used to indicate the field name. The *assertExpr* attribute is used to indicate what kind of comparison is done, as number comparison is inherently different compared to object or string comparison.
- The *methodName* attribute represents the name of a method that should be called. This can be in the context of a method call representing an action or to retrieve a certain value in the context of an assert statement.

Depending on which step type is matched, certain Rule attributes will contain values. The resulting list of Rules is used by the Code Generator component to base its code generation on.

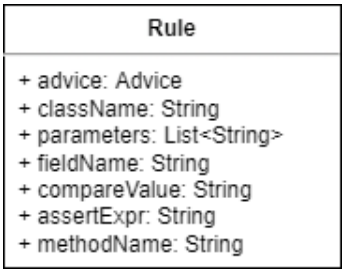

Figure 6.5: The Rule object with its attributes.

6.4.2 Multiline Code

Algorithm 9 Matcher algorithm for multiline code generation

```
function matchMultiline(step, context)
    classes = matchClass(step)
    constructor = matchConstructor(classes, step)
    if constructor.className == context.getClass then
        return matchMethod(step, context)
    end if
    method = matchPairMethod(step, constructor.className, context)
    return method, constructor
end function
```

It can occur that the complexity of a step description is still too high to be captured by 1 line of code. The step descriptions that are most like referring to multiple lines of code are the *And* steps that belong to the *Given* keyword. These steps can denote both actions and variable declarations. Algorithm 9 is able to generate a multiline glue code function. We wanted to show that it was possible to generate multiline glue code given very little logic. In Algorithm 9, we use class matching to determine if the *And* step is describing the same class as the given step or

a different class. In case of matching the same class, the *And* step is likely to specify a certain action to create the correct initial state. As such, we do a method match to generate the correct glue code.
In case of matching a different class to the *And* step, we can instantiate the new class, but we also need to 'connect' the two classes by a method call. The function *matchPairMethod* finds the best matching method for this purpose.

```
Scenario: test if we can set a second cup
  Given there is a bar
  And the coffee machine contains 7 beans and 4 milk
  When the client orders a coffee
  Then the client can still order another coffee
```

Listing 6.4: A Gherkin scenario of a multiline glue code case

An example where the And step is describing a different class can be found in Listing 6.4. The *And* step defines a coffee machine, but the initial *Given* step created a bar. The glue code representing the *And* step is defined in Listing 6.5. As can be seen in the figures, the class *CoffeeMachine* was matched to the *And* step. As the *Bar* class is a different class, we also match a method to this step connecting the two classes. As can be seen in the code listing, the final result is a function with 2 lines of code.

```
Bar bar;
@Given("there is a bar")
public void thereIsABar() {
    bar = new Bar();
}

@And("the coffee machine contains {int} beans and {int} milk")
public void theCoffeeMachineContainsArg0BeansAndArg1Milk(int arg0, int arg1) {
    coffeemachine = new CoffeeMachine(arg0, arg1);
    bar.setCoffeeMachine(coffeemachine);
}
```

Listing 6.5: Multiline glue code function

6.4.3 Class Matching

Here, we describe the implementation of how *FARMER* processes class matching.

6.4.3.1 Initial Algorithm

The first iteration of class matching contained very basic logic that was able to do its job for simple examples. To match a class to a sentence, we compare the nouns in a sentence to the available classes in the target project. To give an indication of confidence in the match, we used a simple edit distance algorithm called Levenshtein distance. This distance metric represents the minimum number of single-character edits required to change one word into the other. It is important to note that with this first iteration matching, constructor parameters were not taken into account. This works very well with simple examples where the nouns of the sentence use the exact wording of the class names. However, below we have an example when there are several nouns in a sentence all pointing to different classes. Due to lack of additional information on the possible matches, it is impossible to determine which class is a better match. Therefore, for simple cases this matching might be enough, as the number of classes is very limited. However, when processing bigger projects with more classes, a different approach is necessary.

```
Gherkin sentence: And it has 10 product in its inventory
Noun-to-Class mapping:    product → Product()
                        inventory → InventoryItem()
```

6.4.3.2 Using a matching context

With the initial approach described in the previous section, class matching is done for each step separately. However, it is likely that the reference to the main class is left implicit within the *When* and *Then* steps. The reason for this is that the state is already explicitly initialized in the *Given* step. This human behavior to writing the natural language descriptions means that the NLP models used with *FARMER* are not able to capture the implicit class references. As a result, it becomes very unlikely that the correct class will be matched to *When* and *Then* steps. To counter this behavior, we introduce a context object. This context object keeps track of all previously matched steps. Using this context, we can now keep track of all the classes matched to the *Given* steps. The advantage of this approach is that it is no longer necessary to first match a class to a *When* or *Then* step. Instead, the most likely class can be retrieved from the context object.

```
Scenario Outline: Have Product and user to receive change
    Given there exists a vending machine
    And it has 10 <product> in its inventory
    When the user inserts the <money> dollars
    And selects the <product>
    And the <product> leaves the machine
    Then the inventory stock must be 9 units
    And the vending machine gives <change> back

    Examples:
      | product      | money | change |
      | "chips"      |  2.50 |   0.50 |
      | "chocolate"  |  3.50 |   1.00 |
      | "cookie"     |  2.00 |   0.25 |
      | "candy"      |  2.00 |   0.50 |
      | "juice"      |  5.00 |   1.75 |
      | "water"      |  3.00 |   1.50 |
      | "coke"       |  3.00 |   0.75 |
      | "pepsi"      |  2.25 |   0.00 |
```

Listing 6.6: A Gherkin Scenario of the VendingMachine project.

The effectiveness of this approach can be shown through the Gherkin scenario in Listing 6.6. In the *Given* steps, the class *VendingMachine* is matched to the *Given* step. The class *InventoryItem* is matched to the corresponding *And* step. If we have to match the class to only the description of the *When* step, the most likely matched class would be one of the classes *Product*, or *Coins*. Now the implicit meaning of this sentence is that we insert money into the vending machine, hence the most logical class to humans would be *VendingMachine*. Now, use the context object instead of matching a class to the step description. The most likely matched class using this context will be *VendingMachine*. Hence, using this context object, we are able to capture the correct implicit relations between step descriptions.

6.4.3.3 Parameter Filtering

With OO languages, classes are instantiated using constructors. These constructors make use of parameters to set an initial state of such a class. Using the amount of parameters of a class and the type of parameters, a signature can be created on which the matched set of classes can be filtered. Gherkin scenarios can contain test data in the form of tables, numbers, or quoted values. By analyzing this test data, it is possible to get an indication on what type and combination of variables is expected to be used within the generated step functions. Building on the assumption that a Gherkin step description describes at most one action or state, we can use this to reduce the number of applicable constructors. As a result, some classes will not contain any applicable constructors. Combining this type of search space filtering with the confidence measure, (resulting from the string similarity calculation), will always result in a decisive match. The lower value the confidence measure has, the better match that class is to the sentence. If we take the same example as shown in the previous section, we get the following result:

```
Gherkin sentence: And it has 10 <product> in its inventory
Class confidence:         Product() → confidence: 1.143
                     InventoryItem() → confidence: 1.231
                    VendingMachine() → confidence: 1.714
                              Coin() → confidence: 1.746

Classes with good constructor signatures: InventoryItem
```

Using the AST analysis, we find that in this specific case, only the constructor of the class *InventoryItem* allows for the use of two parameters. In this case, the confidence measure is superfluous, but if more than 1 class had a possible matching constructor signature, the confidence measure will be the deciding factor.

6.4.3.4 The use of additional similarity metrics

There are two main disadvantages to using Levenshtein distance to compute the similarity of two strings. The first disadvantage is that edit distance algorithms cannot match synonyms. The second disadvantage is that class names consisting of multiple words will be penalized, due to the inherent nature of edit distance. To combat this second problem, a cosine similarity metric is added to the computation of the confidence metric. The two metrics will have equal weight when contributing to the confidence measure. Below, the same example as in the previous sections is shown, but with notably different confidence measures. Do note that again, a lower confidence value is considered better.

```
Gherkin sentence: And it has 10 product in its inventory
Class confidence:         Product() → confidence: 2.343
                     InventoryItem() → confidence: 2.547
                    VendingMachine() → confidence: 3.714
                              Coin() → confidence: 3.746
```

6.4.4 Method matching

The latest approach to method matching used with *FARMER* is the result of extensive experimenting and research of previous work. In the sections below, the algorithms of the two most influential components of the method matching process are discussed.

6.4.4.1 String Matching

Initially, method matching used only verb matching to match methods to Gherkin steps. The string matching was done using only Levenshtein distance, where the best match was the string with the lowest distance score. This approach quickly falls short when the source code contains method names containing multiple words in a camel case fashion. One example of this is the method name “setProduct”.

This name consist out of the verb “set” and the noun “product”. Using only verbs to match this name will not be accurate, as the main information is stored in the noun following the verb. To solve this issue, we can use the labels provided by the SRL labeling model.
For method matching, there are two relevant SRL labels to consider. The first label, “Arg1”, contains the patient or theme of the sentence. The second label, “Arg2”, captures a start/end point, instrument, beneficiary, or attribute of a sentence. Using these two labels and the verb, the most important part of the sentence is extracted. We can split method names on their camel casing to create a list of actual words to compare against. This approach is especially effective when the string matching makes use of cosine similarity, as cosine similarity performs better when matching sequences of words instead of single words.

To be able to match synonyms and closely related words, we use a combination of Levenshtein distance, Cosine similarity, and second order similarity provided by DISCO. These similarity measures are explained in section 5.2.3. The decision to use these three algorithms was based on previous work on the subject by Kamalakar et al. [16].

6.4.5 Dealing with Inheritance

As explained in section 5.1.3, the concept of inheritance is only relevant for the method matching algorithm. Therefore, the algorithm for method matching needs to be able to deal with inheritance. In order to do so, we have designed and implemented Algorithm 10.

In this algorithm, we first deal with the initial class and store its public methods in a list. Then, if the initial class extends a superclass, we recursively go through all superclasses and add every public method belonging to those classes into a list variable. Then we go through all lists and merge them together. When merging the lists, if we encounter an overridden method, we always keep the overridden method of the subclass and discard the method in the superclass.

Algorithm 10 Extracting all methods callable on a class from the JavaParser AST

```
function findMethods(className)
    cuClass = cu.getClassNodeAST(className)
    for MethodDeclaration m ∈ cuClass do
        if m is public then
            methods.add(m)
        end if
    end for
    if className inherits from superClassName then
        methods.addDistinct( findMethods(superClassName) )
    end if
    return methods
end function
```

6.4.5.1 Parameter Filtering

Methods in Java are often parametrized. These parameters and their types can be used to create a specific filter that limits the search space of the matching algorithm. A set of input parameters is extracted from the Gherkin scenario. These, together with the previously matched class, serve as the filter criteria. If the filtering is unsuccessful in creating a reduced search space, the unfiltered search space will be used instead. Using this approach, the chance for a wrong method match is reduced substantially. Below, an example demonstrates the power of parameter filtering. The confidence measure is a value in the range [0.0, 3.0]. For the confidence measures, the same holds as with class matching, a lower confidence value is considered better.

```
Gherkin sentence: And presses the button with the code for product
Parameters: product (String)
Method confidence:     setProduct → confidence: 2.489
                       getProduct → confidence: 2.457
            getInventoryQtyForThe → confidence: 2.544
                  checkForChange → confidence: 2.660
          getAmountMissingMessage → confidence: 2.771
               addItemtoInventory → confidence: 2.796
                  removeInventory → confidence: 2.819
                        setAmount → confidence: 2.831
                        getAmount → confidence: 2.845
```

```
getChange → confidence: 2.888

Parameter Filtered Methods: setProduct, getInventoryQtyForThe, removeInventory
```

If the matcher only takes into account the string similarity, the method matched to this sentence is "getProduct". However, if the matcher uses filtering, the method matched to this sentence is "setProduct". The correct method in this case is "setProduct".
An advantage of method filtering is that the likelihood of matching a function that is able to use the parameters supplied by the Gherkin scenario increases significantly. That said, it is important to keep in mind that the level of strictness in the applied filtering is also a factor.

6.5 Code Generator

The code generator is responsible for generating the final glue code. It takes as input the results of the text-to-code matcher and uses its output to link each Gherkin step to implementation functions, resulting in a functional test suite that correctly represents the scenario descriptions.

To generate syntactically correct Java code, this component makes use of the functionality provided by the JavaParser Library [26]. Using this library, the Generator component builds an AST from the results of the text-to-code Matcher. This AST represents the glue code of the processed scenarios. When all scenarios are processed, the AST is written to file.

6.5.1 Component Implementation

The generator operates in 3 stages. During the first stage, a fresh AST variable is created that holds a skeleton class with a number of predefined imports.
In the second stage, the actual code is added to the AST. To generate the code, the Generator component uses the Rule objects that are outputted by the Matcher component. The implementation code consists out of three different types: object or class instantiations, method calls and assert statements. The properties of the Rule objects are used to distinguish what type of code needs to be generated.
The JavaParser library ensures that the generated code is syntactically correct. However, as JavaParser does not have built-in type solving, semantic correctness of the generated glue code can not be ensured. As a result of this, it is possible to encounter generated code that has compilation errors related to incorrect types.

6.5.1.1 Generating a glue code function in Java

In Listing 6.7, the generated code of the Gherkin step "When we add <amount> of <soda> to the soda machine" is shown. As this step contains the parameters *amount*, and *soda*, the function is also required to use these parameters. The format in which the generated code deals with generation of these parameters is based on the Gherkin and Cucumber rules for Java. There exist the following general rule set:

- Inside the annotation, the words representing the parameters in the description should be replaced by the parameter type surrounded by curly braces.
- The ordering of the function parameters is based on the order of occurrence in the natural language sentence. This requirement is imposed by the Cucumber engine. If this requirement is violated, the Cucumber engine will not be able to correctly execute the test case.
- The Gherkin keyword is represented by the annotation keyword. Subsequently, the keyword is not present in the annotation string description.

Important to note is that the function name is not part of this set of rules. However, to keep the function name as close to the Gherkin sentence as possible, the name is the camel-cased variant of the annotation description. We remove the < and >, and also replace cardinal digits with their parameter name. This is done to stay as close to the auto-generated skeletons of the Java-Cucumber extension.

```
@When("we add {double} of {string} to the soda machine")
public void weAddAmountOfSodaToTheSodaMachine(double amount, String soda) {
    bar.fillSodaMachine(soda, amount);
}
```

Listing 6.7: Example of a generated function

6.6 Conclusion

The previous sections described the implementation based on the design presented in chapter 5. The design and implementation are partly based on existing literature. We added unit test analysis and parameter filtering to improve upon existing work. What remains is to test the effectiveness of the implemented prototype. In the next chapter, we will evaluate *FARMER* against a set of 7 projects to determine the test generation accuracy.

Chapter 7

Evaluation

In this chapter, we evaluate how well *FARMER* is able to generate the Gherkin glue code. First, we describe the evaluation methodology in section 7.1. Then, in section 7.2 the evaluation results are provided. Finally, section 7.3 presents the observations based on the evaluation results.

7.1 Methodology

In order to investigate the correctness of the generated glue code, we decided to use a set of open-source projects and use *FARMER* to generate the glue code given their source code and Gherkin scenarios. To assess the correctness of a generated glue code file, we will perform two checks. First, we will compare the result of executing the existing glue code to the execution of the generated glue code. Both need to have identical sets of succeeding and failing tests. Secondly, a generated test may contain irrelevant code that will result in succeeding tests anyway. For this reason, the syntax and semantics of the glue code will also be compared. We will then compare the statistics of the wrongly and successfully generated tests to provide an accuracy statistic. The accuracy is computed by dividing the amount of correctly generated tests by the total amount of existing tests, see Equation 7.1.

$$accuracy = \frac{\#correctly\ generated\ tests}{\#total\ tests} \tag{7.1}$$

As the process of building *FARMER* was based on theory and experiments, two projects of the test set were used during development. The first test project was the banking example that was shown in chapter 1. The second test project represents a vending machine and is significantly more complex. To validate the final version of *FARMER*, the complete test set was used. One of these projects consisted of 3 separate feature files, we decided to handle each of these files as a separate project, as each of them has a different scope. Therefore, in the sections below, the test set consists of a total of 7 projects.

7.1.1 Test setup

To evaluate the accuracy of *FARMER*'s code generation of a project, we first check if the project's Gherkin files adhere to the requirements listed in section 5.1.2. If this is not the case, these files are adapted with as few changes as possible while maintaining the same coverage of code. When the Gherkin files do adhere to the requirements set by this project, *FARMER* will attempt to generate the step functions. After code generation is complete, we run the original test file as well as the generated test file and compare the results.

7.1.2 Test set

To validate the accuracy of *FARMER*, we use a set of open-source projects available on GitHub. It was very difficult to find suitable test projects that required very few to no changes before being

Project Name	GitHub Link	Commit Hash
Banking	https://github.com/DanielVerloop/Farmer-test-projects	140270a
VendingMachine	https://github.com/alefaissal/CucumberVendingMachine	2cbe716
Calculator	https://github.com/codingstones/cucumber-java	9e7f90b
Cinema	https://github.com/girisharathod/cinema-with-cucumber	50602b3
Bar	https://github.com/DanielVerloop/CoffeeMachine	c138ea9
CoffeeMachine	https://github.com/DanielVerloop/CoffeeMachine	c138ea9
SodaMachine	https://github.com/DanielVerloop/CoffeeMachine	c138ea9

Table 7.1: List of projects comprising the test set with their respective GitHub links.

able to be used for testing our tool. In Table 7.1 the complete list of test projects are shown. Some of these projects needed minor modifications to adhere to the requirements set in chapter 5. The projects with applied changes can be found in the following GitHub repository: link.

7.2 Evaluation Results

Each Gherkin file contains at least 1 scenario. These scenarios consist of a number of steps. Each of these steps is linked to a function in the glue code. To prevent code duplication, steps with identical descriptions are mapped to the same function in the glue code. As such, a glue code function could be responsible for multiple steps. If we were to measure the accuracy in terms of correctly generated scenarios, we would have a metric that is too harsh. For example, if 1 out of 4 steps in a scenario fails, the whole scenario fails. This is too strict, as 75% of the steps are generated correctly. Therefore, we measure the accuracy of *FARMER* in terms of correctly generated steps.
In order to get a good overview of the different techniques used, we gathered results for each of the 3 different parts of this project. First, we show the results by only using techniques from existing literature in section 7.2.1. The results found in this section are provided in the context of research question RQ1. Second, we show the results where *FARMER* uses search space filtering. The results can be found in section 7.2.2. These results must be seen in the context of research question RQ2. Finally, we show how using class data present in unit tests affect the accuracy of *FARMER* in section 7.2.3. This section will provide us with results in the context of research question RQ3.

7.2.1 Results based on previous work

We ran *FARMER* on the test set of Table 7.1. The results from this experiment can be found in Table 7.2. In the table, we can see that *FARMER* is able to generate 124 out of 212 steps correctly. This results in an accuracy of 58%. This is a significant decrease in accuracy compared to previous work [27], which was able to achieve an accuracy of 82%. Do note that previous work was done with the Python programming language. The test set of the work by Storer contained a mix of object-oriented and functional code. Also, our test set is dissimilar to the test set of previous work. As such, there could be multiple explanations for the accuracy difference. As Storer did not rely on source code analysis, the difference in test set in combination with the language difference must be the biggest contributing factors.

7.2.2 Results with filtering enabled

Where *FARMER* differs from previous work, is that it is able to use search space filtering in the constructor and method matching algorithms. As such, we reran the experiment, but this time with filtering enabled. An overview of the accuracy of *FARMER* in this experiment is depicted in Table 7.3. As can be seen in the table, *FARMER* is able to generate 190 out of 212 steps

Project name	Total steps	Successful generation	Unsuccessful generation
VendingMachine	144	72	72
Calculator	6	5	1
Banking	6	6	0
Cinema	6	4	2
Bar	28	19	9
CoffeeMachine	12	10	2
SodaMachine	10	8	2
Total	212	124	88

Table 7.2: List of test projects with the accuracy of the generated glue code without using filtering.

Project name	Total steps	Successful generation	Unsuccessful generation
VendingMachine	144	136	8
Calculator	6	5	1
Banking	6	6	0
Cinema	6	5	1
Bar	28	20	8
CoffeeMachine	12	10	2
SodaMachine	10	8	2
Total	212	190	22

Table 7.3: List of test projects with the accuracy of the generated glue code by making use of filtering.

correctly. This means that the accuracy of *FARMER* on this test set is about 89%. This is a significant improvement compared to the previous experiment. Also, *FARMER* now has a higher accuracy when compared to the accuracy of previous work on the subject[27]. The work of Storer achieved an accuracy of 82%.

We analyzed the 20 wrongly generated steps and were able to categorize the mistakes in Table 7.4. These 20 steps are represented by 11 glue code functions. As explained above, multiple steps with an identical description map to the same glue code function. As such, the total number of unsuccessfully generated steps is higher than the amount of erroneous functions. We have recorded the keyword of the erroneous function, as well as the type of error made. The first category, "Wrong

Error Category	Amount of functions
Total amount of erroneous functions	11
Wrong method/class matched	8
Unsound parameter ordering	3
Erroneous *Given* function	2
Erroneous *Then* function	9

Table 7.4: Categorized erroneous functions present in the generated glue code of *FARMER*.

method/class matched", is straightforward. The generated code does not use a method or class that could lead to the correct result. The second category, "Unsound parameter ordering", means that the syntax of the generated code is correct, but the method calls use the wrong parameters, or the ordering of the parameters is incorrect. The last two categories, "Erroneous *Given* function" and "Erroneous *Given* function", refer to what type of Gherkin steps contained mistakes. The

Given and *Then* keywords were the only keywords that contained mistakes. As we can see, most of the mistakes were made in glue code of functions belonging to the *Then* keyword. These functions contain assert statements. In Listing 7.1, we see one of the few mistakes made in a generated function annotated by the *Given* keyword. The class matching algorithm matched an incorrect class to the step description. As such, the step was interpreted as describing the initialization of a new object instead of adding details to the existing object. This example shows one of the two erroneous function in the *CoffeeMachine* project. As can be seen in Table 7.5 in the section below, with unit test analysis we are able to correct this mistake and generate the correct glue code function.

```
//Erroneous function
@And("we put {int} milk into the machine")
public void wePutArg0MilkIntoTheMachine(int arg0) {
    sodamachine = new SodaMachine();
}

//Correct function
@And("we put {int} milk into the machine")
public void wePutMilkIntoTheMachine(int arg0) {
    machine.setMilk(arg0);
}
```

Listing 7.1: Example of a *Given* glue code function that *FARMER* was unable to correctly generate compared to the correct solution

As shown by Listing 7.2, matching a wrong function also has the result that the assert statement checks the wrong properties of the result. In this example, not only did *FARMER* match an incorrect function, the parameters are also used incorrectly. The function *askForSoda* takes only a string parameter, not an integer. This is just one example that shows some of the issues that *FARMER* still has to overcome. It also occurred that the correct functions were used, but the comparison inside the assert statement was incorrect. Either the expected value was incorrect, or the comparison operator was incorrect.

```
//Erroneous function
@And("the soda machine contains {double} of {string}")
public void theSodaMachineContainsAmountOfSoda(double amount, String soda) {
    Assert.assertTrue(bar.askForSoda(amount).equals(soda));
}

//Correct function
@And("the soda machine contains {double} of {string}")
public void theSodaMachineContainsAmountOfSoda(double amount, String soda) {
    assert bar.getSodaMachine().getInventory(soda) == amount;
}
```

Listing 7.2: Example of a *Then* glue code function that *FARMER* was unable to correctly generate compared to the correct solution

7.2.3 Using Unit Tests

As part of the development of *FARMER*, we investigated if class analysis on available unit tests would improve the accuracy of *FARMER*. Only two projects were accompanied by a unit test file. These were the *Bar* and *CoffeeMachine* projects from Table 7.1. We ran *FARMER* on both these with the unit test analysis to see if the accuracy would increase. The results can be found in Table 7.5 As can be seen in the table, the accuracy of *FARMER* improved by two step functions. One of these two step functions that we know generate correctly can be found in Listing 7.1. On the whole test set of 7 projects, this increases the accuracy of *FARMER* by 1%. The accuracy of *FARMER* is now 90%. That said, the use of class information present in unit tests has only been tested on two test projects, so it is unknown if this also holds for other test projects as well as unseen projects.

Project	Total Steps	Without Unit Test Analysis	With Unit Test Analysis
Bar	28	20	20
CoffeeMachine	12	10	12

Table 7.5: The improvement in accuracy of *FARMER* by making use of class information present in unit tests.

7.3 Discussion

Based on the presented results in the previous section, we can make the following observations about *FARMER*:

- As we only used open source test projects, the test set used to evaluate the accuracy of *FARMER* is limited. Next to this, some projects were adapted as they did not fully adhere to the requirements listed in section 5.1.2.
- Based on the results presented in section 7.2.1. The accuracy presented in the work of Storer [27] does not translate to Java. Also, the work of Kamalakar [16] does not scale well on bigger projects. When *FARMER* only makes use of techniques presented in previous work, given our test set, *FARMER* performs significantly worse compared to the results of previous work.
- The results in section 7.2.2 show that using parameter filtering allows *FARMER* to achieve an accuracy that is comparable to the accuracy of previous work.
- Based on the results in section 7.2.2, most mistakes made by *FARMER* are related to assert statements. When *FARMER* was unable to correctly generate an assert statement, it came down to either unsound parameter ordering, or using the wrong method or variables.
- The results in section 7.2.3 show that using class information present in unit tests does not decrease the accuracy of *FARMER*.

Chapter 8

Conclusions and Future Work

This chapter provides conclusions based on the literature study, implementation, and evaluation results. The chapter concludes with potential future work for further research on the subject of this thesis.

8.1 Conclusions

This section iterates over the research questions defined in section 1.2 and answers those questions based on the literature study, implementation, and evaluation results.

> **RQ1.** Can we apply existing integration test generation methods to Java and get equal accuracy compared to existing literature?

We were able to generate integration tests based on techniques described in previous work by Storer [27] and Kamalakar [16]. With these techniques, we were able to achieve an accuracy of 58% on our test dataset. This is significantly lower than the 82% accuracy described in previous work.

Based on the literature study, multiple approaches for generating integration tests exist. However, most of these approaches do not allow for users to influence the test generation process. InfoSupport uses the BDD approach to software testing. With this approach, users write test scenarios using natural language. In the literature study, we have discussed previous work on generating integration tests based on natural language scenarios. However, the existing literature is very limited on this topic.
The papers of Storer [27] and Kamalakar [16] describe an approach to generating integration tests based on natural language description written in Gherkin. As their results were promising, we decided to create a prototype implementation based on techniques described in their papers. The details of this implementation can be found in chapter 5 and chapter 6.
We developed a prototype called *FARMER* based on a 3 layer architecture. An NLP layer for processing the natural language descriptions provided by the Gherkin language. The Code Analysis layer that provides static analysis on the SUT. And a Matcher layer that used the results of the NLP and code analysis to generate the integration tests.

In chapter 7, we evaluated the performance of *FARMER*. We saw that based on the techniques described in literature, we achieved an accuracy significantly lower when compared to the results of the literature [27][16]. Multiple factors could have contributed to this result. Firstly, the tests set of the literature is quite different compared to our test set. Where the literature mostly used small, simple projects to assess their performance, we used bigger, more complex projects in our evaluation. Another valid point to make is that the approach of Storer was implemented using Python, whereas *FARMER* is implemented using Java. As such, the language dissimilarity could also have contributed into the accuracy difference.

> **RQ2.** How can we improve upon the results of RQ1 through the use of OO context information?

We improved upon RQ1 by implementing using search space filtering. With this technique, we are able to filter out methods and classes that could never lead to a correctly generated test. With this approach, we were able to significantly improve the accuracy of *FARMER* to 89%. This is a significant increase compared to RQ1.

In order to improve upon previous work, we implemented a search space filtering method in section 6.4. Based on the NLP analysis, we already know what parameters are supposed to be used by the method or constructor we are looking for. Initially, the code analysis provided all public methods or constructors belonging to a certain class, and we matched only on the name of the method. This provided us with a search space that contained many methods or classes that could result in an incorrect match. To reduce the likelihood of a wrong match, we implemented a filtering algorithm that filtered the set of methods or constructors returned by the code analysis based on their parameter signatures.
As described in chapter 7, this significantly improved the accuracy of *FARMER*. With this improvement, the accuracy of *FARMER* is now comparable to the accuracy of previous work. In section 7.2.2, the analysis of the remaining tests that *FARMER* is unable to generate correctly show that *FARMER* has the most difficulty generating correct assert statements.

> **RQ3.** How can we use information on object instantiations present in unit tests to improve the accuracy of our test generation model?

We can match unit test names to the Gherkin scenarios using string similarity. We then extract all object instantiations from the most confident match, creating a set of likely class matches. This helps reduce the search space for our class matching algorithm. Based on our evaluation, initial results show a promising increase in accuracy. However, given the small test set, further testing is needed before a definitive conclusion can be drawn.

When best practices are followed, when it comes to software testing, a version of the testing pyramid described in section 2.1 is often used. This states that next to integration tests, unit tests are often also used to test the software project. As such, in the third research question, we were interested to see if *FARMER* could benefit from an already existing unit test suite.
In chapter 5, we describe how we analyze unit tests to extract class information. As described in chapter 7, only two test projects contained unit tests. The results presented show a slight improvement in accuracy. However, the set of 2 test projects provides too small of a test set to draw definitive conclusions from. That said, we certainly did not decrease the accuracy of *FARMER* by adding unit test analysis. With the current implementation, we only extract class information. As a result, we can only improve the accuracy of the class matching algorithm. All we can say is that the initial results look promising, but further testing is needed to reach a definitive answer.

8.2 Future Work

This thesis has been subject to time limitations. As such, the achievements of this thesis have been bound by limited resources. For future work, there are several directions to continue this research:

1. As described in chapter 6, the current models used in the NLP engine are not accurate enough to provide a correct analysis on a consistent basis. The accuracy of these NLP models were part of the reason to create strict language requirements in section 5.1.2. In the future, we recommend implementing a NLP model that is more accurate compared to the currently used models. This way, language requirements can be relaxed while maintaining or improving the accuracy of the generated tests.

2. As described in section 7.2, most of the remaining steps that *FARMER* is unable to generate correctly have to do with assert statements. Therefore, we recommend developing a better matching algorithm that is able to improve upon the existing matching algorithm of *FARMER*.

3. As mentioned in section 6.4, *FARMER* supports basic multi-line code generation for *Given* step functions. This proves that it is possible to generate more complex functions, but the current implementation is very limited. In order to process more Gherkin scenarios, multi-line code generation across all step functions will be required. Furthermore, providing multi-line code generation to the current supported complexity will improve the readability of the generated tests significantly. One such example is first instantiating an object before using it as a parameter. Hence, we recommend implementing multi-line code generation across all step types.

4. In section 5.2.2, we describe how we use unit test analysis to improve the accuracy of *FARMER*. However, regarding this thesis, we only focus on class information present in these unit tests. These unit tests contain much more information in the form of assert statements and methods used. As *FARMER* already benefitted from the class information alone, it is worth finding out if the same holds for assert statements or method calls. Therefore, for further research, we recommend to further explore the benefits of unit test analysis.

Bibliography

[1] Bui Thi Mai Anh. Enhanced genetic algorithm for automatic generation of unit and integration test suite. In *2020 RIVF International Conference on Computing and Communication Technologies (RIVF)*, pages 1–6. IEEE, 2020. 15, 16

[2] Apache. Spark. https://github.com/apache/spark, 2022. 23

[3] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003. 7

[4] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. " O'Reilly Media, Inc.", 2009. 9, 25

[5] Lionel C Briand, Yvan Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, 2003. 15

[6] Ryan Dewhurst. Static code analysis. https://owasp.org/www-community/controls/Static_Code_Analysis. 10

[7] Eric Evans and Eric J Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004. 7

[8] Wikimedia Foundation. Integration testing. https://en.wikipedia.org/wiki/Integration_testing, Oct 2021. 4

[9] Wikimedia Foundation. Natural language processing. https://en.wikipedia.org/wiki/Natural_language_processing, Feb 2022. 9

[10] Martin Fowler. Domain driven design. https://martinfowler.com/bliki/DomainDrivenDesign.html, Apr 2020. 7

[11] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson Liu, Matthew Peters, Michael Schmitz, and Luke Zettlemoyer. Allennlp: A deep semantic natural language processing platform. *arXiv preprint arXiv:1803.07640*, 2018. 9, 25

[12] Wael H Gomaa, Aly A Fahmy, et al. A survey of text similarity approaches. *international journal of Computer Applications*, 68(13):13–18, 2013. 24

[13] Google. Guava. https://github.com/google/guava, 2022. 23

[14] Mark Grechanik and Gurudev Devanla. Generating integration tests automatically using frequent patterns of method execution sequences. In *SEKE*, pages 209–280, 2019. 15

[15] Matthew Honnibal. A good pos tagger in about 200 lines of python, Mar 2015. 9

[16] Sunil Kamalakar, Stephen H Edwards, and Tung M Dao. Automatically generating tests from natural language descriptions of software behavior. In *ENASE*, pages 238–245, 2013. 2, 14, 17, 18, 24, 39, 46, 47

[17] Peter Kolb. !gscs. *Proceedings of KONVENS-2008, Berlin*, 156, 2008. 24

[18] Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Using Large Corpora*, page 273, 1994. 26

[19] Dan North et al. Introducing bdd. *Better Software*, 12, 2006. 7

[20] Martha Palmer, Daniel Gildea, and Paul Kingsbury. The proposition bank: An annotated corpus of semantic roles. *Computational linguistics*, 31(1):71–106, 2005. 26

[21] Martha Palmer, Daniel Gildea, and Nianwen Xue. Semantic role labeling. *Synthesis Lectures on Human Language Technologies*, 3(1):1–103, 2010. 9

[22] Mataz Pancur, Mojca Ciglaric, M Trampus, and Tone Vidmar. Towards empirical evaluation of test-driven development in a university environment. In *The IEEE Region 8 EUROCON 2003. Computer as a Tool.*, volume 2, pages 83–86. IEEE, 2003. 7

[23] Mauro Pezze, Konstantin Rubinov, and Jochen Wuttke. Generating effective integration test cases from unit ones. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 11–20. IEEE, 2013. 15

[24] Peng Shi and Jimmy Lin. Simple bert models for relation extraction and semantic role labeling. *arXiv preprint arXiv:1904.05255*, 2019. 9

[25] SmartBear. Cucumber. `https://cucumber.io/`, 2022. 9

[26] Nicholas Smith, Danny Van Bruggen, and Federico Tomassetti. Javaparser: visited. *Leanpub, oct. de*, 2017. 10, 29, 40

[27] Tim Storer and Ruxandra Bob. Behave nicely! automatic generation of code for behaviour driven development test suites. In *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 228–237. IEEE, 2019. 2, 8, 17, 21, 25, 26, 43, 44, 46, 47

[28] Kristina Toutanova, Dan Klein, Christopher D Manning, and Yoram Singer. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 252–259, 2003. 27

[29] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013. 7

[30] Atro Voutilainen. Part-of-speech tagging. *The Oxford handbook of computational linguistics*, pages 219–232, 2003. 9

[31] Jianyong Wang and Jiawei Han. Bide: Efficient mining of frequent closed sequences. In *Proceedings. 20th international conference on data engineering*, pages 79–90. IEEE, 2004. 15

Appendix A

Project source code

The source code of *FARMER* can be found here. The NLP Engine code can be found at here