Faculdade de Engenharia da Universidade do Porto

Live Acceptance Testing Using Behavior Driven Development

José Pedro da Silva e Sousa Borges

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Hugo Sereno Ferreira, PhD

Second Supervisor: André Restivo, PhD

July 30, 2020

Live Acceptance Testing Using Behavior Driven Development

José Pedro da Silva e Sousa Borges

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: João Pascoal Faria
External Examiner: João Miguel Fernandes
Supervisor: Hugo Sereno Ferreira

July 30, 2020

Abstract

Software development is composed of many different stages, from the initial conception of the idea all the way into project deployment. One of these stages is testing the software to make sure it is behaving as expected.

One common testing methodology is Test-Driven Development (TDD), which focuses on developing features based on previously written high-level code-based tests, that may lead to confusion from project stakeholders. Thus, some unwanted results may appear when delivering a project to a client, as he may not be comfortable with the way some features were done. One way to mitigate this is by using Acceptance Testing, which is a level of testing focused on verifying if the system can be made available for the end-users, by testing the features of the project with respects to the needs of the user and the system requirements.

Behavior Driven Development(BDD) is a software development process that extends from TDD and aims to foster collaboration and communication between developers and project stakeholders.

In BDD, features and scenarios are typically written using Gherkin, which is then interpreted by the Cucumber framework. Gherkin is a human-readable, natural language commonly used in acceptance tests that promotes better understanding between all parties in a project.

Currently, there exists a lack of tools that offer support to the user when they are attempting to write acceptance tests in Gherkin. Based on the concepts aforementioned, this dissertation reports the development of an extension using Visual Studio's Language Server Protocol that helps users in writing Gherkin by providing suggestions during the creation of Gherkin steps and automatically transforms these steps into JavaScript or TypeScript tests, by creating a test file associated with the corresponding feature file. It also provides feedback on tests made for the user's project whenever the user saves the corresponding feature document file. This extension has been focused for Visual Studio Code, but due to the usage of the Language Server Protocol, it has the capability of being adapted onto different Integrated Development Environments.

Keywords: Acceptance Testing, Behavior Driven Development, Gherkin, Language Server Protocol

Resumo

O desenvolvimento de software é composto por muitas fases diferentes, desde a conceção inicial da ideia até ao lançamento do projeto. Uma destas fases é a de testar o software para certificar de que este se comporta como o esperado.

Uma metodologia comum de test é a de *Test-Driven Development* (TDD), que se foca no desenvolvimento de funcionalidades baseadas em tests de alto nível escritos em código, que podem gerar confusão nos *stakeholders* do projeto. Assim, alguns resultados inesperados podem surgir ao entregar um projeto a um cliente, pois este pode não estar confortável com a maneira como algumas funcionalidades foram implementadas. Uma forma de mitigar isto é ao utilizar *Acceptance Testing*, que é um nível de testes focado em verificar se o sistema se pode tornar disponível para o *end-user*, ao testar as funcionalidades do projeto com respeito às necessidades do utilizador e pelos requisitos do sistema.

Behavior Driven Development (BDD) é um processo de desenvolvimento de *software* que é uma extensão a TDD e cujo objetivo é ajudar a colaboração e comunicação entre desenvolvedores e os *stakeholders* do projeto.

Em BDD, funcionalidades e cenários são tipicamente escritos usando Gherkin, que é depois interpretado pela *framework* de Cucumber. Gherkin é uma linguagem natural *human-readable* usada em testes de aceitação e que promove melhor entendimento entre todos os partidos de um projeto.

Atualemente, existe uma falta de ferramentas que consigam oferecer apoio ao utilizador quando este tenta escrever testes de aceitação em Gherkin. Baseado nos conceitos mencionados, esta dissertação reporta o desenvolvimento de uma extensão usando o *Language Server Protocol* do Visual Studio para ajudar os utilizadores a escrever Gherkin ao providenciar sugestões durante a criação de passos de Gherkin e automaticamente transformar os passos em testes de JavaScript ou TypeScript, ao criar um ficheiro de teste associado com o ficheiro *feature* correspondente. Providencia também *feedback* nos testes escritos para o projeto do utilizador sempre que este grava o ficheiro de *feature* correspondente. Esta extensão foi focada para o Visual Studio Code, mas devido à utilização do *Language Server Protocol*, este tem a capacidade de ser adaptado para diferentes *Integrated Development Environments*.

Keywords: Acceptance Testing, Behavior Driven Development, Gherkin, Language Server Protocol

Acknowledgements

To my mom, dad, brothers, sisters in law and nephews, thank you all for always believing in me and for pushing me to be the best I could be. I truly believe that I would not have come this far had it not been for you all. I promise I will always stay *humilde*.

To my godmother, thank you for all the help in dealing with anxiety and stress throughout the last couple of months. The nightly phone calls were essential in keeping me from staying too overwhelmed.

To my best friends from Gondomar, thank you for all the good times we've spent together. The jokes, basketball games, all-nighters, vacations, trips and dinners at Burger King are all memories that I cherish very much. I can't wait to form some new ones.

To all my basketball teammates throughout the years, thank you for teaching me the importance of teamwork and for providing a way for me to forget about my troubles. The games, practices, locker-room chit chat, final fours and street tournaments have all been really fun to be a part of.

To all the friends made throughout school, thank you for all the childhood memories. Every interaction, big or small, have helped shape the person I am today.

A special thank you to the friends I made at FEUP in the last 5 years. I have no doubt in my mind that I would not be here today had it not been for our friendship. The late night projects, study sessions, sushi and francesinha meals, relaxation sessions in the garden, Rocket League and Left for Dead gaming sessions, trips and every organised event are all memories and moments that I hold close to my heart. During this pandemic period, your help in completing this dissertation was essential and your company online helped keep me sane. I am truly grateful for all your support and friendship.

Last, but not least, I would like to thank my supervisors Hugo Sereno Ferreira and André Restivo for all of their help and patience throughout this year. The insights and advice given were fundamental to finish this dissertation.

José Borges

“Everything I’m not
Made me everything I am”

Kanye West

Contents

List of Figures

List of Tables

Abbreviations

AT	Acceptance Tests
BDD	Behavior Driven Development
DSL	Domain Specific Language
GPL	General Purpose Language
IDE	Integrated Development Environment
LiveMTBD	Live Model Transformation By Demonstration
LSP	Language Server Protocol
npm	Node Package Manager
Regex	Regular Expressions
TDD	Test Driven Development
VSCode	Visual Studio Code
WYSIWYG	What You See Is What You Get

Chapter 1

Introduction

1.1 Context

Software is composed by a set of instructions interpreted by a computer, designed to make human lives better. For the past decades, it has become a tool used by society in their everyday lives, as it has important applications in areas such as communications and transportation, whether it be on the ground or aerial, as well as mundane appliances such as phones, controllers and watches. [AO16] Software development has many steps between initial conception and deployment. A typical development cycle goes from initial discussion and planning, to designing, building, testing and finally deployment and maintenance. Throughout this cycle, teamwork is key as developers and stakeholders work together on the product.

One of the main steps in software development is **testing**. Even though many factors influence the production of reliable software, testing is the primary way for which an industry can evaluate software during development, as it is how most errors, faults and failures are able to be caught and eventually resolved. An easy way to distinguish between them is that user tests certify that the code is done right, whereas Acceptance Tests (AT) certify that the code does the right things [MC02]. For this dissertation, AT are the focus.

AT take into consideration the needs of the user and is used to ensure that a product behaves in the way that the users expect. AT grows with the product and can be used for:

- Capturing user requirements in a directly verifiable way and measure how well the system meets those requirements;
- Exposing problems that unit tests miss;

- Providing a ready-made definition of how finished the product is.

If a system is technically elegant but does not do what users want, it is practically worthless. As such, in theory, a project is completed and ready to be deployed when all AT that are marked as a *must-have* by the client are passed [MC02].

Test Driven Development is a software development process with the goal of achieving clean code that works. The main flow for this technique is demonstrated by Figure 1.1.

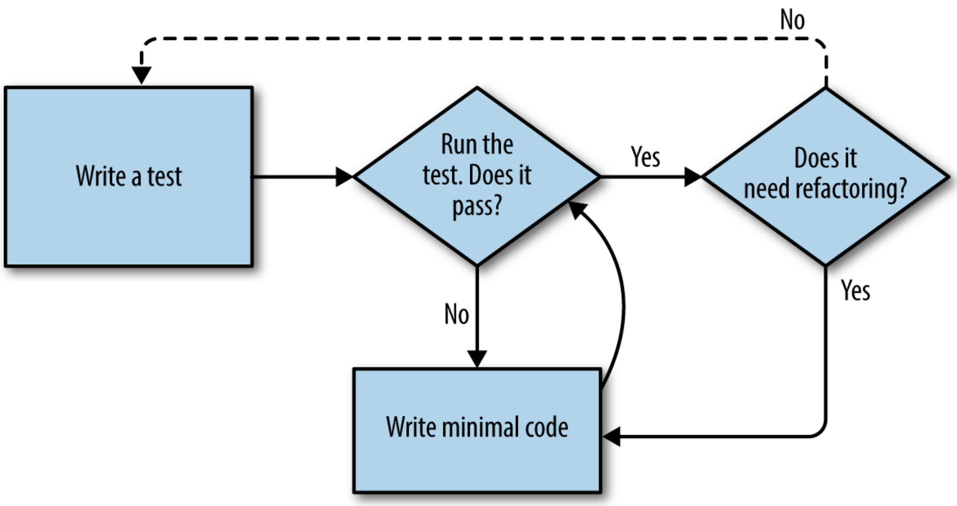


Figure 1.1: Test Driven Development Flow Chart [tdd].

As detailed by the image above, in TDD the first step is to write the tests based on the requirements specified in the planning section, then execute the tests and make changes to the code if needed, before restarting the cycle. The tests written in this technique are hard to interpret by non-programmers as they are highly technical and written in code. When developers and project stakeholders discuss the functionalities that the product is expected to have, many ideas can be lost in the process, whether by misunderstanding or by ambiguity [Bec03].

Behavior Driven Development (BDD) is based on Test Driven Development and aims to support collaboration between developers and project stakeholders by creating tests that are easily understood by both parties. This is due to its usage of a non-technical language, Gherkin, to define the AT that will be executed by the developers [PSS+18, SWD12]. Gherkin uses feature files with scenarios written in natural language that are aimed at testing software. One example of a Gherkin feature step, taken from the official documentation [Cucb], is:

```
Feature: Guess the word

  Scenario: Breaker joins a game
    Given the Maker has started a game with the word "silky"
    When the Breaker joins the Maker's game
    Then the Breaker must guess a word with 5 characters
```

Source 1.1: Example of a Gherkin feature step with one scenario written [Cucb]

The BDD methodology is focused in this dissertation and will be explained in detail in section 2.3.

1.2 Problem Definition

As software is becoming increasingly complex, and with a growing user base, the demand for quality has been equally rising, as such, maintaining such software has become, often, the most time-consuming and cumbersome task, when compared to development time [AO16].

Test Driven Development is a popular software development process, but the tests written in this process are typically more difficult to interpret by stakeholders, as they are highly technical and stakeholders may not have programming experience. However, acceptance tests are written in a way which facilitates the understanding of the stakeholders.

Currently, there exists a lack of tools that offer support to the user when they are attempting to write acceptance tests. When writing Gherkin steps, although users may have access to snippets with the help of some extensions [Cucd], these snippets don't really help users in writing full steps, leaving users to manually write every step even if they are similar to previous steps or steps that have already been written in other test files. There is also a lack of feedback in these tools, as users do not have access to the test results after execution, while editing the document where the acceptance tests are written.

The problems characterized in this section will be detailed further in Chapter 4.

1.3 Goals and Motivation

When planning the functionalities that a product should have, miscommunication between developers and stakeholders is bound to happen. The use of technical language may prove difficult for the customers that don't fully understand programming and developers may end up writing tests that don't reflect the wishes of the customers.

This is where Behavior Driven Development comes in and why it was chosen as the focus of the dissertation. With the use of the domain-specific language of Gherkin, communication is improved when compared to other ways of discussing requirements specifications with customers.

When first introduced to BDD, writing high quality scenarios in Gherkin can be challenging due to lack of experience. In order to gain this experience, time and money will need to be invested in training, which may lead to difficulty in convincing customers to use BDD as the main development technique.

While testing, the lack of instant feedback may be time consuming for the development team. With the introduction of live testing, this problem can be mitigated [PSS+18].

Our goal is to implement an extension using Visual Studio's Language Server Protocol that facilitates the way in which users are able to write Gherkin steps and transform them into tests.

1.4 Document Structure

The structure of the document is as follows:

- Chapter 2 details the background work and research done for the project.
- Chapter 3 describes the current state of the art in the areas that the dissertation is based on
- Chapter 4 goes into further detail of the problem and its solution
- Chapter 5 details the tool that has been developed and explains how the Language Server Protocol works and why it was used
- Chapter 6 presents the architecture of the tool and how the communication between client and server is done
- Chapter 7 refers the difficulties encountered throughout development, and mentions the conclusions and the future work that can be done.

Chapter 2

Background

2.1 Software Engineering

Ian Sommerville [Som10] provides the following definition regarding software engineering.

Software engineering is an engineering discipline that is concerned with all aspects of software production.

According to the SWEBOK Guide [BF+14], there are several steps defined during the development cycle which are used when defining and building sound software systems. These stages are [BF+14]:

- **Requirements Analysis** - During this stage, the business requirements for the project are discussed, along with their validation and management.
- **Design** - The design stage is intended to prepare the system and software design, in order to help specify the hardware and system requirements. An internal representation of the project's architecture and the organization of its components is performed.
- **Development** - Developing software involves creating the functionalities of a system through code. This same system will then be verified and validated by debugging actions and tests.
- **Testing** - The code that is written during the Development stage needs to be tested in order to verify if it meets the requirements previously defined. Testers will need to verify the

behaviour of the software and its components and features in various ways. These tests can be performed before, while or after the code is written. This stage will be the main focus of the dissertation.

- **Maintenance** - After being deployed and used by customers, there may be a need to update and evolve the software, as it may encounter problems or it may become outdated.

2.2 Testing

The software that is developed today has the potential to reach millions of active users. Therefore, in order to obtain the best user experience possible and to avoid negative feedback it is necessary to thoroughly test the behavior of the software, in order to prevent mistakes, i.e., bugs, from being sent into the business world [AO16]. It is not economically feasible to test every single feature in a project as there are too many combinations of inputs and outputs possible, so developers often need to compromise and find techniques that allow them to test the software as best as possible.

Testing is the process of executing a program with the intent of finding errors. This definition is important because it may steer the developers in a direction that increases the value of the final product. Rather than performing tests that show that the program doesn't fail, tests should be done to showcase where the program fails so that it can be fixed. This approach is of higher value to the product [MSB11].

According to Glenford J Myers et al. [MSB11], a successful test should be one that showcases errors that can be fixed, rather than showing no errors at all. As the concept of a program with no errors is unrealistic, programmers should not be relieved when no errors are found in their software. Instead, they should question rather the test that was performed explored the program well enough. This is the definition of an unsuccessful test, i.e., one that does not properly examine the software. The same test that first found an error can later be defined as successful when the error is no longer present. Testing has been widely used in the industry in different domains and with different approaches [DCPF18, CPFA10, SPRG18, RAM16, RAM17].

Testing can be classified in levels, methods and types. There are four different levels of software testing, which correspond to the different stages where testing is conducted in the software development lifecycle. These levels are [swf, isq]:

- **Unit Testing**. This testing level focuses on testing individual software units, which are the smallest component in a software project, and its goal is to isolate different parts of the software in order to verify if the designed units are behaving as expected. Without this approach, whenever an error is found in other testing levels, testers will still need to spend extra time in searching for the unit that is causing the error.

- **System Testing**. This testing level focuses in verifying if the specified requirements are met when the individual units of the software are grouped together and in exposing any possible faults in the interaction between units.

- **Integration Testing**. A level of testing with the focus of verifying if the specified requirements are met when testing the complete and integrated software.
- **Acceptance Testing**. A level of testing focused on verifying if the system can be made available for the end-users, i.e, verifying if the system is ready to be accepted. This level of testing is done with respects to the needs of the user and the system requirements and business processes.

A more detailed description of Acceptance Testing is found in Subsection 2.2.1.

2.2.1 Acceptance Testing

In this dissertation, we will focus on Acceptance Testing, which are the tests that capture the needs of the customers. These types of tests are captured during the requirement's analysis phase and aim to verify if the software behaves as its users expect. The customers should express the requirements as input to the software, along with the results that they expect to receive. Acceptance tests integrate at the level between the business logic and the user interface, whereas testing methodologies like unit testing are aimed at testing low level units such as methods. In theory, a product is ready to be deployed when it is fully functional in the eyes of its customers, which happens when all tests marked as a must-have have passed [HH08, MC02].

Some acceptance tests are essential performance tests that should be repeated on either a routine or a periodic basis, and these tests provide reference data that the user can refer to during the whole life of the system [SPB+10].

Acceptance Testing usually includes End-to-End testing, meaning that testing often occurs for the whole application flow from start to finish. For example, if the software being tested is the one of a supermarket, users will usually test from adding an item to the cart, to the checkout and eventual shipping process [HVG13].

Behavior Driven Development, which is focused in this dissertation, is a software development process used in Acceptance Testing.

2.3 Behavior Driven Development

As previously stated, Behavior Driven Development (BDD) is an Agile software development process that extends from Test Driven Development (TDD). Mark Shead[She18] defines BDD as:

> *The practice of driving development based on desired behavior of the system usually from the standpoint of user action and expected result.*

This means that BDD focuses on how well the system works when interacting with the user so that the developers and project stakeholders can better observe and control the project and its final stage. In BDD, the test cases are written using natural language rather than source code [SHD+18].

As demonstrated in Figure 2.1 (p. 8) the process is very similar to the TDD cycle. Developers and/or stakeholders should start by selecting an issue that should be tested. After this selection, test

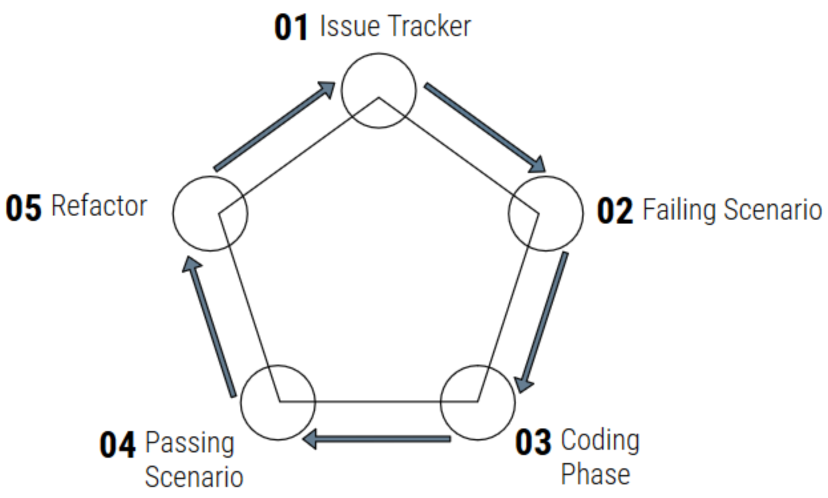


Figure 2.1: Behavior Driven Development Process, adapted from [bdd].

code is written in a way that ensures that the scenario being tested should fail. Developers will then write code until the scenario passes. The last phase is to refactor the code written, and when this phase is finished the cycle will be restarted. The big difference in these cycles is in the types of tests that are done. In TDD, the goal is to test the functionality of the implementation of the source code, whereas in BDD the goal is to verify the behavior of the system from the perspective of the end users.

BDD has the goal of narrowing the communication gaps between team members, as collaboration between developers, quality assurance analysts and non-technical or business analysts is encouraged during the development of a software project. This collaboration will lead to a better understanding from all parties of how the final project should behave [SHD+18].

BDD has great results when utilized, as a study conducted in 2018 [PSS+18] showed that developers had many benefits when using BDD for a period of 5 months, with main benefits being improved communication, collaboration and living documentation.

2.3.1 Gherkin

Gherkin is a domain-specific language that is often used in BDD. Gherkin feature files consist of a list of scenarios that each represent one use case. The number of scenarios is not limited, but all of them should be related to the feature that is to be tested. The feature file should ideally start with a description of what the feature is based on.

Each Gherkin line starts with a keyword. The primary keywords are Feature, Scenario, Given, When, Then, And, But, Scenario Outline and Examples [Cucb]. A description for each keyword is detailed below.

2.3.1.1 Feature

`Feature` should always be the first primary keyword in a Gherkin document. Its purpose is to provide a high-level description of a feature from the software and it is also useful to group several scenarios that are correlated between one another and that relate to the feature. Any text can be added underneath `Feature` to add more description [Cucb]. Source 2.1 presents the typical structure for a feature step in Gherkin, which is composed of a name and an optional description.

```
Feature: Name of the feature

    More information about the feature can be added here. This text will
    be ignored by the Gherkin compiler until another keyword is detected
    in the beginning of a line.
```

Source 2.1: Structure of a feature step written in Gherkin [Cucb].

The feature name and description has no meaning to its compiler as it's only purpose is to help the readers understand its main aspects and importance. As detailed in the example, the description text will be ignored until another keyword is detected in the beginning of a line, meaning that writers have the freedom to write what they feel is necessary. Features also support tags, meaning that a tag can be written above `Feature` in order to group features together [Cucb].

2.3.1.2 Scenario

The keyword `Scenario` is a synonym of the keyword `Example` and these are used to illustrate a business rule. To do this, a list of steps is used and a pattern is followed:

1. Describe an initial context
2. Describe an event
3. Describe an expected outcome

The Gherkin documentation recommends scenarios to have a short number of steps in order to promote better documentation and specification. A scenario is also a test, which means that they are an executable specification of the system [Cucb]. Source 2.2 presents the Keyword scenario, which must always be assigned a name.

```
Scenario: Name of the Scenario, which describes its purpose
```

Source 2.2: Structure of the Scenario step [Cucb].

2.3.1.3 Given

The `Given` keyword is used to introduce the context of the scenario, i.e., it provides information about the initial requirements for the scenario. When executing this step, Cucumber will configure the system according to the description given in the step, meaning that objects may be created and configured and data may be added to a database. More than one `Given` step may be used, but the conditions should be connected using `And` or `But`. Cucumber, the Gherkin compiler, will execute the scenario one step at a time and it doesn't allow the same text to be repeated in more than one step. This allows for less ambiguous scenarios [Cucb]. Source 2.3 presents the Given keyword, which must have a corresponding precondition and be written within a Scenario.

```
Scenario: Name of the Scenario
        Given some precondition
```

Source 2.3: Example of a Gherkin scenario with a Given step [Cucb].

2.3.1.4 When

The `When` keyword is used when the writer wants to convey an event or an action triggered either by a user or by a system. The Gherkin documentation recommends that only one `When` keyword should be used per scenario [Cucb]. Source 2.4 presents the When keyword, which must have a corresponding event and be written after the Given preconditions.

```
Scenario: Name of the Scenario
    Given some precondition
    When a specific event or action is triggered
```

Source 2.4: Example of a Gherkin scenario with Given and When steps [Cucb].

2.3.1.5 Then

The `Then` keyword is utilized to describe the outcome of the scenario. This step should be used as an assertion, in order to compare the actual outcome of the system with the outcome that the user expects, i.e., the outcome that the system is supposed to produce. The outcome should be observable to the user and not something buried in the code [Cucb]. Source 2.5 presents the Then keyword, which should have a corresponding result and be written after the Given and When steps.

```
Scenario: Name of the Scenario
```

```
    Given some precondition
    When a specific event or action is triggered
    Then the outcome should be this
```

Source 2.5: Example of a Gherkin scenario with Given and When steps [Cucb].

2.3.1.6 And & But

As stated previously, the keywords `And` and `But` should be used to connect various `Given`, `When`, although this is not recommended, and `Then` descriptions.

With this, a description for all the keywords needed to write a simple gherkin scenario has been provided. Thus, in order to further help understand the gherkin syntax, a concrete example of a gherkin scenario, taken from `behat`[1] is shown in Source 2.6 [Cucb]:

```
Feature: Serve coffee
    In order to earn money
    Customers should be able to
    buy coffee at all times

Scenario: Buy last coffee
    Given there are 1 coffees left in the machine
    And I have deposited 1 dollar
    When I press the coffee button
    Then I should be served a coffee
```

Source 2.6: Example of a Gherkin scenario with Given and When steps [Cucb].

2.3.1.7 Scenario Outline

Another optional keyword is `Scenario Outline` which will be very useful in the development of this dissertation. This keyword allows a scenario to be run several times while changing some parameters. An example taken from the gherkin syntax documentation is detailed in Source 2.7 to clarify the usage of this keyword [Cucb].

```
Scenario Outline: eating
    Given there are <start> cucumbers
    When I eat <eat> cucumbers
    Then I should have <left> cucumbers

    Examples:
```

[1] Behat documentation (Retrieved by: 03 January 2020)

```
        | start | eat | left |
        |    12 |   5 |    7 |
        |    20 |   5 |   15 |
```

Source 2.7: Example of the Scenario Outline keyword in a Gherkin file [Cucb].

Thus, the Gherkin interpreter will run the test for all rows with values that are written into the `Examples` table, while replacing those values in the parameters delimited by the <> symbols. This is particularly useful for scenarios that need to be tested with several values, which is very common while testing software.

Note: As of Gherkin 6, the keyword Rule has been added, but it is not yet fully supported by Cucumber by the time of the writing of this dissertation.

2.3.2 Cucumber

In order to convert Gherkin scenarios into testing code, it is necessary to use a tool named Cucumber [2].

Cucumber uses Regular Expressions (Regex) or Cucumber Expressions, which are very similar to Regex but with a simpler and more natural syntax to read and write, that search for the keywords written in Gherkin and then interpret the free text next to those keywords in order to create code for the tests.

2.3.2.1 Step Definitions

In order to do this, Cucumber uses Step Definitions, which are Java methods linked to Gherkin steps.

For the scenario written in Source 2.8, Cucumber will search for a matching Step Definition to execute. This scenario will match the definition in Source 2.9 [Cucb]:

```
Scenario: Some cukes
  Given I have 48 cukes in my belly
```

Source 2.8: Scenario example [Cucb]

```
package com.example;
import io.cucumber.java.en.Given;

public class StepDefinitions {
    @Given("I have {int} cukes in my belly")
```

[2] Cucumber Documentation (Retrieved by: 03 January 2020)

```
    public void i_have_n_cukes_in_my_belly(int cukes) {
        System.out.format("Cukes: %n\n", cukes);
    }
}
```

Source 2.9: Step definition that matches the scenario written in Source 2.8 [Cucb].

Each step definition should have an expression, either a Regex or a Cucumber expression, associated to it and should with a preposition or adverb associated with Gherkin, such as `Given` or `Then`. These definitions are loaded before Cucumber executes the text in the feature file. Whenever Cucumber analyzes a Gherkin step, it will search for a matching Regex in a step definition, in order to execute this definition with the corresponding arguments [Cucb].

In the above definition, the expression is `I have int cukes in my belly`, where `int` is a capture group that will be passed as an argument to the step definition's method. In this case, the capture group is identical to a registered parameter type for Cucumber, which is the integer. In the above example, this means that the argument for cukes will be passed as an integer.

Regarding the scope of the step definitions, these aren't linked any particular file or scenario, meaning that they may be placed in any file, class or package as the only important aspect for them to be located is the expression.

If Cucumber finds a Gherkin step without a step definition to match with, it will print out to the user a snippet with an expression that matches the one written in the step. This expression will be complete with capture groups if any saved parameter types are found.

After running a step definition, Cucumber may return one of six different results [Cucb]:

- **Success**. When the step definition does not produce an error.
- **Undefined**. When Cucumber analyzes a Gherkin step and does not find a matching step definition. All subsquent scenario steps are skipped.
- **Pending**. This result indicates that the step is not yet completed, meaning that the developer still has work to do.
- **Failed**. When the step definition produces an error.
- **Skipped**. Steps that come after steps marked as `undefined`, `pending` or `failed` are never executed.
- **Ambiguous**. Steps that are equal to other steps. These steps will raise an Exception due to Cucumber requiring step definitions to be unique.

Cucumber also color codes these results, as `Success` steps are marked as Green, `Undefined` and `Pending` as yellow, `Failed` as red and `Skipped` as cyan.

2.3.2.2 Hooks

Developers may also write `Hooks`, which are blocks of code that run at any time during the Cucumber execution cycle, in any class. These hooks can be divided into `scenario` and `step` hooks. Scenario hooks are the ones that are run for every scenario and they can be run either before or after they execute a scenario. Step hooks are run for every step definition and they can be equally run before or after they execute each step.

For developers that want to have more control over how the test code is executed, Cucumber also offers the possibility of creating `Conditional Hooks`. These hooks can also be divided into before and after hooks and they are run based on the tags of each scenario [Cucb].

2.3.2.3 Tags

In order to better organize and document the code, scenarios and features can be grouped with tags. This is particularly useful when developers when to run only a subset of the scenarios and when they want to restrict the hooks that are run. Scenarios may have multiple tags and they must be placed above the `Feature`, `Scenario`, `Scenario Outline` and `Examples` keywords. These tags will be inherited by the child elements of each keyword [Cucb].

2.3.3 Domain Specific Languages

BDD uses as a natural language named Gherkin, whose syntax and and definition is provided in the subsection below.

Ghosh defines a Domain Specific Language (DSL) as [Gho10]

> *A computer programming language of limited expressiveness focused on a particular domain.*

DSLs are programming or specification languages that target a specific problem domain. Whenever a programmer uses General Purpose Languages (GPL), such as Java or C, many problems are difficult to solve, as these languages do not specialize in any problem in particular. When a programmer is using a GPL and has a problem in implementing a feature, if the domain of said problem is covered by a particular DSL then using that DSL will result in a much faster and more efficient solution than trying to use a GPL. DSLs are also beginner friendly, as they are meant to be learned by nonprogrammers, which means that they must be more intuitive and readable than GPLs [Bet16].

Fowler [Fow10] states that there are four key elements to the definition of DSLs:

- **Computer Programming Languages**. A DSL that is used by humans to give instructions to computers over what to do.
- **Language Nature**. A DSL that should have a sense of fluency and expressivness, so as to be coherent and understandable.

- **Limited Expressiveness**. A DSL should support a bare minimum of features needed to support its domain.
- **Domain Focus**. DSLs should have a clear focus on its domain, as this is what makes a limited language worthwhile.

2.4 Live Programming

Live programming is the process through which a programmer can immediately see the results of his program as it is being coded, since it is re-executed continuously during the editing phase [McD13].

During the development of a product, a lot of time can be wasted in compilation and running steps, especially during small changes to the software. By replacing the edit-compile-test cycle with live programming, this time can be effectively lowered as the time required for building and running the program is significantly reduced. This is very advantageous as it frees time for developers to focus on other activities and it can also lead to a more fluid software development [Gol04].

Even though Live Programming can appear in different stages of the software development cycle [ARC+19], in this dissertation it will be focused on the testing phase.

2.4.1 Liveness Levels

Liveness levels help in determining how visual programming systems can be classified in accordance to the degree of which they present live feedback to the user.

Steven L. Tanimoto introduced this concept in 1990 and introduced four different liveness levels. While this concept is still being used in present times, Tanimoto did make a few changes to it in 2013, by introducing a level five and a level six.

The new model states that there are six levels of liveness, which are shown in Figure 2.2 (p. 16) [Tan13, Tan90]:

- **The First Level** is merely informative, as it consists only in a visual representation of a program, e.g., flowcharts, that help the user in understanding and documenting the program.
- **Level Two** is where programmers can ask for a response after they develop their software system and can then check the results when the response arrives shortly afterwards. In this level, if a flowchart is created, it is executable and more informative than on the previous level as it contains enough details so that the program is completely specified.
- **Level Three** is where the developer doesn't need to ask for the results, as the program would respond sometime after the programmer did something. This means that whenever the user changes something, the system would re-execute the appropriate parts of the program.
- **Level Four** is where the system keeps constantly running the program, modifying its behavior as soon as the programmer makes changes to it. This means that the programmer has access to immediate feedback from the system.

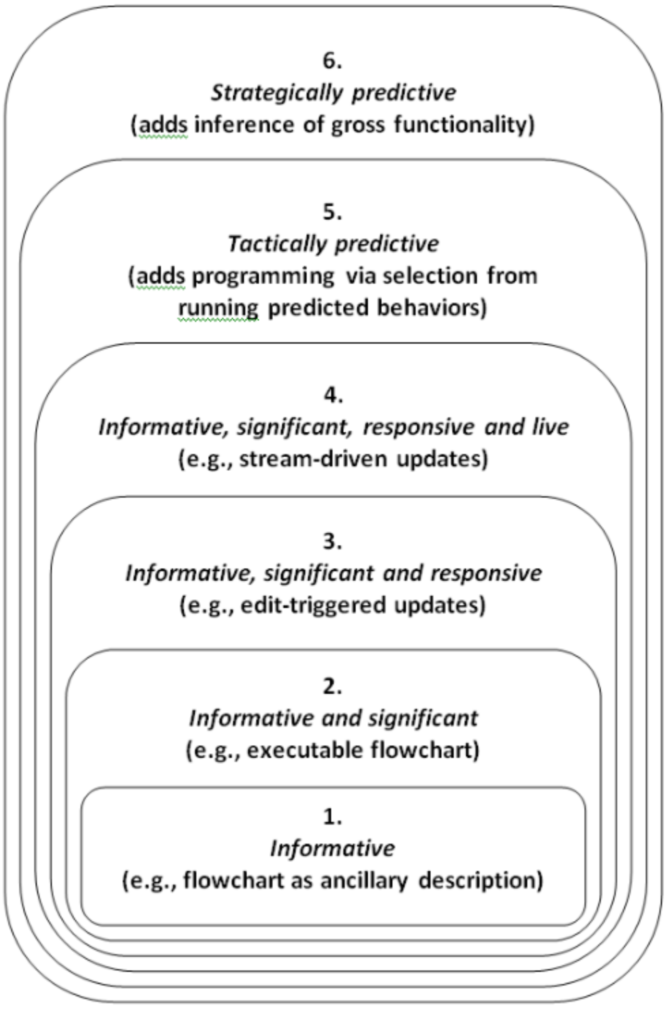

6.
Strategically predictive
(adds inference of gross functionality)
5.
Tactically predictive
(adds programming via selection from running predicted behaviors)
4.
Informative, significant, responsive and live
(e.g., stream-driven updates)
3.
Informative, significant and responsive
(e.g., edit-triggered updates)
2.
Informative and significant
(e.g., executable flowchart)
1.
Informative
(e.g., flowchart as ancillary description)

Figure 2.2: Liveness levels according to Tanimoto [Tan90]

- **Level Five** is where the system is not only able to give instant feedback to the programmer but is also able be a step ahead of them by predicting their next actions. The system executes one or more versions of these predictions and the programmer has full control over any of them.
- **In Level Six** the system is able to give more strategic suggestions to the user by analyzing the current program with a large knowledge base.

2.4.2 Criticisms

The benefits of live programming have been described above, with the main ones being time saving and fluidity while programming. However, live programming also has its share of criticisms. One of which is that it is not necessary in programs that run in very a short amount of time and for those that run for a long time, editing a part of a program where execution has passed and to which it will not return is unnecessary and won't improve the liveness of the system. Another criticism is that liveness is very computationally heavy and uses a large amount of resources due to editing, compiling and running frequently and simultaneously.

Steven L. Tanimoto has addressed these issues. For short programs, liveness can be meaningful when adjusting the run-test configuration and enabling "Auto-repeat" mode, which allows for continuously running the system until the users requests it to stop, leading to instantaneous changes. As for the problem stated for bigger programs, Tanimoto suggests the usage of breakpoints and the definition of sections with a *start* and *end* location, so as to state to the program the parts of code that are to be continuously executed after each edit. The criticism for the large amount of resources that liveness requires is now mitigated, since modern computers can handle editing, compiling and running requirements with great performance results [Tan13].

2.4.3 Summary

This chapter introduced and detailed the most important concepts regarding this dissertation. These concepts include the software engineering process with a special focus on the testing phase, as it is the main area discussed in this dissertation. Acceptance testing is explained, which leads into BDD. The core for this methodology is detailed, together with a description of its main components - Gherkin and Cucumber. The last concepts to be described is live programming and liveness levels, along with its criticisms.

Chapter 3

State of the Art

3.1 Integrated Development Environments

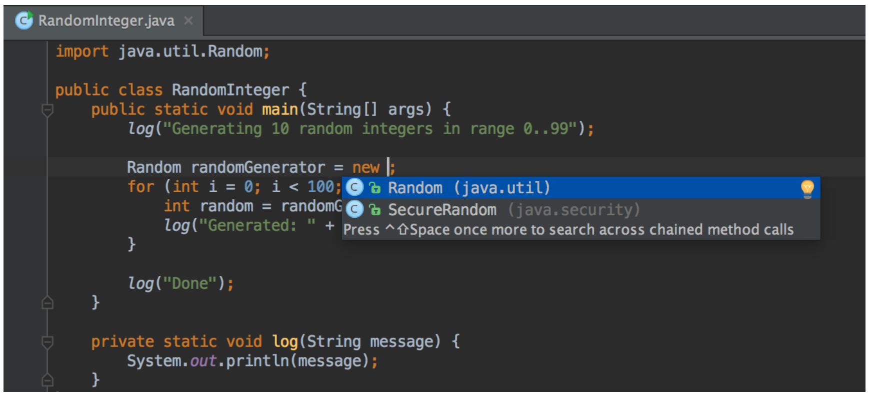
Integrated Development Environments (IDEs) are software applications allow facilitate the way through which developers write and analyze their code. In order to accomplish this, these applications have a plethora of features integrated into them. These features are described below, through the example of IntelliJ IDEA, an IDE for Java developers [Int].

3.1.1 Smart Completion

Smart completion offers the developers a popup with the most relevant symbols that are applicable in the current context of what they are writing, allowing for faster and more efficient programming. These suggestions are ordered by most frequent classes and packages used to least frequent, where the most frequent are the ones at the top of the list [Int].

3.1.2 Chain Completion

A deeper suggestion than Smart Completion, as Chain Completion searches for symbols that are applicable in the current context via methods or getters. Figures 3.1 and 3.2 present examples of smart and chain completion in order to further illustrate the differente between these features [Int].

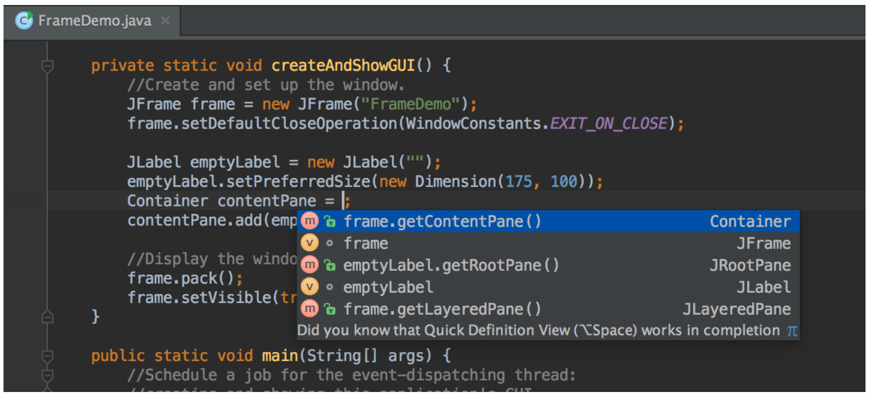


RandomInteger.java

```
import java.util.Random;

public class RandomInteger {
    public static void main(String[] args) {
        log("Generating 10 random integers in range 0..99");

        Random randomGenerator = new |;
        for (int i = 0; i < 100;
            int random = randomG
            log("Generated: " +
        }

        log("Done");
    }

    private static void log(String message) {
        System.out.println(message);
    }
```

```
Random (java.util)
SecureRandom (java.security)
Press ^⇧Space once more to search across chained method calls
```

Figure 3.1: Example of Smart Completion on the IntelliJ IDE [Int]



FrameDemo.java

```
    private static void createAndShowGUI() {
        //Create and set up the window.
        JFrame frame = new JFrame("FrameDemo");
        frame.setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);

        JLabel emptyLabel = new JLabel("");
        emptyLabel.setPreferredSize(new Dimension(175, 100));
        Container contentPane = |;
        contentPane.add(emp

        //Display the windo
        frame.pack();
        frame.setVisible(tr
    }

    public static void main(String[] args) {
        //Schedule a job for the event-dispatching thread:
```

```
frame.getContentPane()          Container
frame                              JFrame
emptyLabel.getRootPane()        JRootPane
emptyLabel                         JLabel
frame.getLayeredPane()       JLayeredPane
Did you know that Quick Definition View (⌥Space) works in completion
```

Figure 3.2: Example of Chain Completion on the IntelliJ IDE [Int]

3.1.3 Static Members Completion

Offers a list of symbols that match the input of the developer, in order to facilitate the use of static methods and constants. Whenever a user selects one the suggestions, the IDE automatically adds the code to import the necessary libraries [Int].

3.1.4 Cross-Language Refactorings

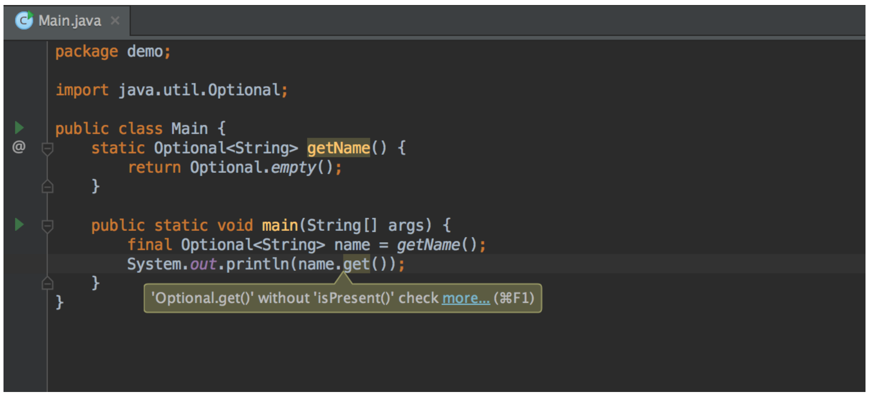
A refactoring mechanism that permits users to restructure their code without changing its behavior. IntelliJ facilitates this process allowing users to refactor whatever part of the code they want with the help of a popup menu [Int].

3.1.5 Duplicate Detection

IntelliJ is able to detect whenever a user writes code that has been written similarly previously. This is helpful to the user, as they now know may refactor their code by creating a function that performs what the duplicate code performs, rather than having duplicate code in their program [Int].

3.1.6 Quick-Fixes

This feature detects when users are about to make a mistake on their code, and showcases to them where the error is through the use of a small light-bulb icon and by underlining the code. An example of a quick fix is displayed in Figure 3.3 [Int].



```
package demo;

import java.util.Optional;

public class Main {
    static Optional<String> getName() {
        return Optional.empty();
    }

    public static void main(String[] args) {
        final Optional<String> name = getName();
        System.out.println(name.get());
    }
}
```

'Optional.get()' without 'isPresent()' check more... (⌘F1)

Figure 3.3: Example of a Quick Fix on the IntelliJ IDE [Int]

3.1.7 Version Control

IDEs are more than code editors, as they also include some built-in developer tools so that programmers have a bigger toolset, allowing them to work on a single application rather than having to multitask and switch through environments.

Version Control is one of these tools, as users are able to check their local version of the project and compare it to the remote version. Users are also able to fully use all features of version control systems such as Git [Int].

3.1.8 Test Runner and Coverage

IntelliJ is equipped with several tools that perform tests, including a test runner and coverage tool for Cucumber. These tools are able to execute tests for the project being analyzed and output the results of those tests to the user [Int].

3.1.9 Terminal

IntelliJ is equipped with a built-in terminal that users can access at any time in order to write any command that may be necessary without having to switch away from the working environment [Int].

3.1.10 Summary

It is clear that IDEs are very powerful coding platforms mainly due to the high support for developers and to the amount of different tools that they provide.

Cucumber is able to be integrated into several programming languages and different IDEs, but the support for the features stated previously is scarce. Most integrations in Cucumber are only able to provide syntax highlighting, code completion and snippets creation. Testing with Cucumber is also well supported, as most tools offer testing coverage outputs.

3.2 Visual Studio Tools

Visual Studio Code (VSCode)[1] is an open-source code editor that supports development operations such as debugging, task running, terminal usage and version control. It is one of the most popular code editors on the market.

VSCode allows users to install external extensions through the use of the Visual Studio marketplace[2]. With extensions, users are able to, among other things, support a programming language and support debugging a specific runtime. The tool developed in this dissertation fits in these two categories, since it will be able to support Gherkin syntax and be able to test or debug code in the developer's project.

The different extensions in the marketplace related to gherkin and testing that were searched and analyzed are described and detailed in the section below.

3.2.1 Cucumber (Gherkin) Full Support

This extension is the most popular Gherkin language supporter in the marketplace as it has a plethora of functionalities that assist in writing Gherkin and that add a language support for Gherkin features. These functionalities include [Cucd]:

- **Syntax highlighting**. Keywords will appear in different colors from regular text;
- **Step Autocompletion**. When writing keywords, the program suggests to the user what to write next;
- **Text Formatting**. Text will be properly indented, according to the hierarchy present in Gherkin features;
- **Basic Snippets support**. Snippets are templates that facilitate the writing of repeating code patterns, which in the case of Gherkin means that when a user types in the keyword `Feature`, for example, the extension automatically writes the rest of the pattern;
- **Tables formatting**. Tables appear in a more understandable and easier to read way.

While this extension is useful to users that want to write Gherkin scenarios, it does not assist in the testing aspects of Cucumber, nor does it provide feedback to the user when tests are made. There are many other extensions in the marketplace that support Gherkin syntax, but this one is the most complete.

[1] Visual Studio Code (Retrieved by: 08 January 2020)
[2] Visual Studio marketplace (Retrieved by: 08 January 2020)

At the time of writing this dissertation, the extension is currently in version v2.14.1.

3.2.2 Visual Studio Code Testing Extensions

Since VSCode is an editor that allows programmers to run their code, it also has a debugger incorporated. With the debugger, programmers are able to track variable values, memory information and other aspects of the project, but aren't able to receive feedback about successful or unsuccessful tests. As previously stated, there are several kinds of tests that can be performed, but in this dissertation we will focus on acceptance tests. As such, a search and analysis was conducted on the marketplace [3] to find testing extensions that facilitate the writing and analysis of acceptance tests.

There are several extensions that aim to facilitate the way programmers can test their code. Test Explorer UI [Tesa] is one of the most popular testing extensions and it allows users to test code written in Javascript, Python, C, C++ and a few others, but there is no mention of Gherkin or Acceptance Testing. There are several extensions available to perform tests on many programming languages, with some of the most popular being *Java Test* for Java, [4], *Mocha Test Adapter* for Javascript [5] and *ms python* for Python [6] but **none of them** support Gherkin or Acceptance Tests, meaning that it is an area that has much to explore.

The most relevant extension found was TestStory [Tesb] which is still in Beta as the time of writing this dissertation. This extension does not run the tests and show the results, but it does facilitate the way programmers are able to write acceptance tests that require variables.

3.3 Acceptance Testing Tools

3.3.1 Robot Framework

The Robot Framework is an open source automation framework that is usually used for acceptance testing. It has a keyword-driven testing approach and users can create new higher-level keywords from existing ones. This framework is independent from any operating system and application.

The framework has a modular architecture that can include self-made test libraries. Each file created using Robot may contain more than one test case. Whenever a test is executed, the framework must first parse the data, then utilize the keywords provided in order to interact with the system. This execution is done from the command line.

An example for testing if a login is valid is presented in Source 3.1 [Rob].

```
    *** Settings ***
Suite Setup        Open Browser To Login Page
Suite Teardown     Close Browser
```

[3] Visual Studio marketplace (Retrieved by: 10 January 2020)
[4] Java Test for Java (Retrieved by: 10 January 2020)
[5] Mocha Test Adapter (Retrieved by: 10 January 2020)
[6] ms python (Retrieved by: 10 January 2020)

```
Test Setup        Go To Login Page
Test Template     Login With Invalid Credentials Should Fail
Resource          resource.txt

*** Test Cases ***                User Name      Password
Invalid Username                  invalid        ${VALID PASSWORD}
Invalid Password                  ${VALID USER}  invalid
Invalid Username And Password     invalid        whatever
Empty Username                    ${EMPTY}       ${VALID PASSWORD}
Empty Password                    ${VALID USER}  ${EMPTY}
Empty Username And Password       ${EMPTY}       ${EMPTY}

*** Keywords ***
Login With Invalid Credentials Should Fail
    [Arguments]    ${username}    ${password}
    Input Username    ${username}
    Input Password    ${password}
    Submit Credentials
    Login Should Have Failed

Login Should Have Failed
    Location Should Be    ${ERROR URL}
    Title Should Be    Error Page
```

Source 3.1: Code for a test file that verifies if a login is valid [Rob]

In this example, there are three main sections. Settings allows users to define the paths for libraries and auxiliary files, test cases is where the test cases are defined and in the keywords section is where users can define custom keywords from pre-existing keywords.

3.3.2 Gauge

Free and open source framework for writing and running acceptance tests. According to its documentation, some of its main functionalities include [Gau]:

- Consistent cross platform/language support for writing test code;
- A modular architecture with plugins support;
- Support of data driven execution and external data sources;
- Help the creation of maintainable test suites;
- Support for Visual Studio Code and IntelliJ IDEA.

However, in this tool the tests are written in Markdown, as it does not support domain specific languages such as Gherkin. Instead of feature files, Gauge uses specifications together with keywords that complete the Markdown syntax. Writing a test with Gauge includes naming the specification, creating one or more scenarios related to that specification that can be associated with tags and creating one or more steps, which represent the executable components of the specification, for each scenario. An example of this syntax is presented in Source 3.2 [Gau].

```
# Search the internet

## Look for cakes
* Goto Google's home page
* Search for "Cup Cakes"
```

Source 3.2: Example of a scenario written for the Gauge framework [Gau]

This syntax is very flexible as it also supports Data-Driven Testing, meaning that users can use Markdown tables and external Comma-Separated Values files for the tests.

3.4 Live Programming tools

In general, programming editors are visual interfaces. Over the last few decades, these editors have become increasingly more interactive and nowadays most developers use an IDE to write code. Most of the visual feedback displayed to the programmers includes color highlights, different fonts for keywords and visual assistance when navigating through code, like mini-maps. This type of features is ignored by the interpreter, as its main purpose is to assist the user in better interpreting and organizing his code. Live coding has the possibility of expanding this concept even further, as it not only presents this editing feedback to the programmer, but also presents visual feedback of the results of the code written during the editing phase, as it is being edited. Some relevant live programming tools are detailed below [MGCW10].

3.4.1 LiveMTBD

LiveMTBD (Live Model Transformation By Demonstration) is an extension from Model Transformation By Demonstration, which is a plugin to the Generic Eclipse Modeling Systems. Its main goal is to simplify the implementation of model transformation tasks through the generation of model transformation patterns after analyzing and inferring from user-demonstrated behavior and to enable end-users in realizing their desired model transformation tasks [SGW$^+$11].

The solution created uses a demonstration-based technique with three live features. The main idea is that users will be asked to use and apply changes to a template, rather than manually writing model transformation rules. A pattern is generated through the recording and inference engine that captures user operations and converts them into a pattern, which is then executed in any model instance, in order to repeatedly carry out the desired transformation process.

This implementation allows for three new capabilities that improve the creation and use of models in the MTBD framework. **Live demonstration**, through the use of a recording system that continuously records all editing operations done in the editor, so that users will be able to specify their desired editing activities by reflecting on their editing history. **Live Sharing** so that users may share patterns with other users using the tool, benefiting communication and collaboration between each other. **Live matching** to facilitate the way users can find and use similar patterns, which are patterns that satisfy the same precondition.

Even though is approach is meant for Model-Driven Engineering, it still showcases that Live Programming is becoming an increasingly popular tool amongst programmers [SGW+11].

3.4.2 NaturalMash

NaturalMash is a live mashup tools that combines model-driven Web engineering with natural language processing techniques in order to provide immediate feedback to the users by showing them the resulting mashup as its being edited. NaturalMash seeks to help users without programming knowledge in creating useful and feature-rich mashups with minimal prior learning. Mashups are a type of Web Applications which can be reused and that are focused on content and functionality [AP13, AP14, AP12].

NaturalMash is based on the What You See Is What You Get (WYSIWYG) interface and on structured natural language programming. The environment of this tool consists of four main elements [AP13, AP14]:

- **Visual field**, a visual field implementing the WYSIWYG interface, which displays the design and preview of the mashup being edited;
- **Text field**, where the natural language description of the mashup can be edited;
- **Component dock**, which displays the list of components used in the mashup;
- **Stack**, containing the component library and the mashups created by the users.

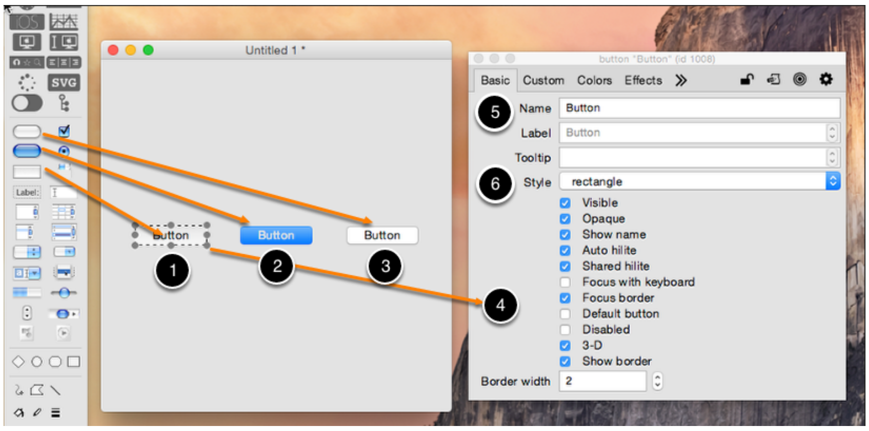
With the use of WYSIWYG, the most recent version of the mashup being edited is provided in the visual field, allowing the users to manipulate the interface directly and observe its changes in real time. By interacting with widgets, users are also able to more easily understand and program in natural language, because when clicking on a widget the corresponding natural language description is added to the text field area [AP14]. NaturalMash supports synchronized multi-perspective interaction, meaning that the component dock, text field and visual field are always synchronized with each user interaction. Programming by Demonstration is also present in this tool, as visual suggestions are presented to the user during the editing phase [AP13].

3.4.3 LiveCode

LiveCode allows a user to create native applications for iOS, Android, Windows, Mac, Linux, Server and the Web from the same code. It is a development environment that integrates user-interface design with writing and testing code. In order to create a program using LiveCode, users have widgets and libraries at their disposal to facilitate their work. With these features, users can drag and drop, resize and customize the properties of the objects placed on the interface [SSS^{+}18, Liv].

The code written is in an English-like programming language, making it easy to learn and understand for non-programming users. LiveCode is constantly waiting for an input from the user, whether it is a click on a button or typing in a text field. Whenever an interaction occurs, LiveCode sends a message and the user chooses the messages that they want to respond to, e.g., when a user types in a text field, LiveCode sends a message to the text field and the user may place code on the text field to describe what is should do when it got clicked or when it received text [Hol12, Liv].

LiveCode is object based, meaning that all applications created are based on objects. Users should start by designing the layout of the interface before they write code. In figure 3.4, an example is shown of a user adding a button to the interface. The user can choose through different buttons and customize them [Liv].



Name: Button
Label: Button
Tooltip:
Style: rectangle
✓ Visible
✓ Opaque
✓ Show name
✓ Auto hilite
✓ Shared hilite
☐ Focus with keyboard
✓ Focus border
☐ Default button
☐ Disabled
✓ 3-D
✓ Show border
Border width: 2

Figure 3.4: Example of a User Adding a Button [Liv]

This tool does not allow users to write tests.

3.5 Cucumber-Related Tools

3.5.1 Cucumber Eclipse

Cucumber-Eclipse is an Eclipse plugin for Cucumber. According to its documentation, the main features of this tool are [Cucc]:

Lambda Expression support for Cucumber-Java 8. Lambda Expressions are supported as of release version 0.0.20 of Cucumber-Eclipse and are used to create Java functions that belong to any class and without any name. Lambda expressions are passed as an argument for the keyword `Given`. Source 3.3 displays an example of how a step definition can be integrated with a lambda expression [cuca].

```java
public class ShoppingStepsDef implements En {

private int budget = 0;

public ShoppingStepsDef() {
    Given("I have (\\d+) in my wallet", (Integer money) -> budget = money);

    When("I buy .* with (\\d+)", (Integer price) -> budget -= price);

    Then("I should have (\\d+) in my wallet", (Integer finalBudget) ->
      assertEquals(budget, finalBudget.intValue()));
    }
}
```

Source 3.3: Step definition using a lambda expression in Cucumber-Java 8 [cuca]

Content Assistance for Feature File. In Eclipse, content assistance provides developers with popup windows that have features such as code completion suggestions. The developers may then select the best option, according to the current context. Cucumber-Eclipse supports this assistance in feature files, as it has suggestions for list of keywords and predefined-steps.

Syntax Highlighting. Syntax highlighting is a feature that displays keywords in source code in different colors and fonts. Cucumber-Eclipse supports this feature for the syntax of Gherkin.

New Step Definition File Wizard. A wizard is a popup interface with dialog boxes that helps users navigate through multiple steps. In Cucumber-Eclipse, whenever a user decides to create a new Step, he may do this with the assistance of a wizard, making this process much easier [Cucc].

3.5.2 Gwen

Gwen is an open-source dynamic interpreter that focuses on automating front end web tests and repetitive online processes with the feature specifications of Gherkin. This interpreter reads Gherkin features and transforms them into web automation instructions. Using this interpreter it is possible to transform features such as the one written in Source 3.4 [Gwe]:

```
    Feature: Google search

Scenario: Lucky Google search
    Given I have Google in my browser
     When I do a search for "Gwen automation"
     Then I should find a Gwen page
```

Source 3.4: An example of a Feature written in Gherkin for the Gwen open-source interpreter [Gwe].

Into the following automation tasks present in Source 3.5:

```
Feature: Google search meta (automation glue)

@StepDef
Scenario: I have Google in my browser
    Given I start a new browser
     When I navigate to "http://www.google.com"
     Then the page title should be "Google"

@StepDef
Scenario: I do a search for "<query>"
    Given the search field can be located by name "q"
     When I enter "$<query>" in the search field
     Then the page title should contain "$<query>"

@StepDef
Scenario: I should find a Gwen page
    Given link 1 can be located by css selector ".r > a"
     When I click link 1
     Then the current URL should match regex ".+[G|g]wen.*"
```

Source 3.5: Automation task generated from the feature in Source 3.3

This is done through the use of `Meta Features`, which are a mechanism for removing redundancy, as they separate declarative from imperative specifications. They establish locators and step definitions to match with the steps in the Gherkin feature files, allowing the Gwen interpreter to convert them into browser operations.

In the above example, the interpreter creates @StepDef scenarios, which allow users to compose custom steps and create procedures that can optionally accept parameters. These scenarios are declared the same way as normal scenarios [Gwe].

The Gwen intepreter supports Chrome [7], Firefox [8], Safari [9] and Edge [10].

3.5.3 Cucumber Query Language

Cucumber Query Language is a DSL, written in Ruby, for querying a Cucumber test suite. Its main purpose is to show summarized data or reports of a test suite. This DSL can query for any attribute available in the models.

An example of its usage, taken from the official documentation is written in Source 3.6 [Cuce]:

[7]Google Chrome (Retrieved by: 14 January 2020)
[8]Firefox (Retrieved by: 14 January 2020)
[9]Safari (Retrieved by: 14 January 2020)
[10]Microsoft Edge (Retrieved by: 14 January 2020)

```
cql_repo.query do
    select name, source_line
    from features
end
```

Source 3.6: Ruby example taken from the official documentation of Cucumber Query Language [Cuce]

This sample returns the results as a list of attribute maps. The query sample above may return results such as the ones displayed in Source 3.7:

```
[{'name' => 'Feature 1', 'source_line' => 1},
 {'name' => 'Feature 2', 'source_line' => 3},
 {'name' => 'Feature 3', 'source_line' => 10}]
```

Source 3.7: Result array of the request written in Source 3.6

This tool does not have a user interface, as users are required to use the command line to execute the queries. By using this tool users are able to analyze the performance of their test suite, but it does not offer any live support nor does it integrate with any code editor or IDE [Cuce].

3.6 Visual Programming Tools

Learning how to program using conventional programming languages is a difficult and time consuming process. By using visual programming tools, this difficulty can be reduced. The concept of visual programming has evolved throughout the years, but most authors define it the same as Brad Myers did 1986, by stating that its a programming paradigm in which two or more dimensions are used.

Computer programs are presented in a one-dimensional textual form, so it is not the optimized way in which our brains can interpret information, as the human visual information processing and visual systems are optimized for dealing with multi-dimensional data in our daily lives. Most people also use schematics and diagrams to visualize what they are thinking in order to better understand concepts and organize thoughts, and visual programming allows users to more easily express those thoughts. Thus, visual programming provides a way for people to have a better experience with programming, as these systems have demonstrated that non-programmers can create fairly complex programs with little training [Mye90, BM95, CFS14, DLF$^+$20, LDAF19].

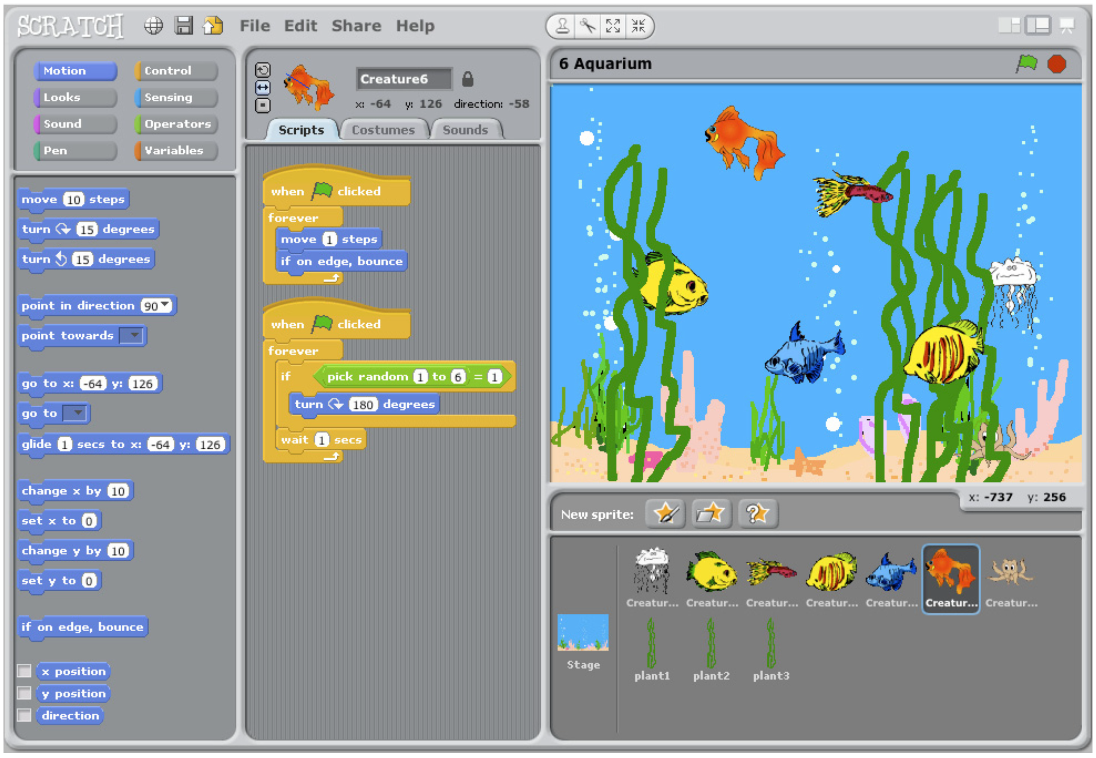
Visual Programming tools transform blocks of code into source code, which is then executed within the compiler. This concept of transforming a block code into executable code is a similar process to what has been developed in this dissertation, since users will be able to type Gherkin steps and observe it being lively transformed into JavaScript or TypeScript code.

3.6.1 Scratch

Scratch is a visual programming educational platform that lets users create interactive and media-rich projects, developed by the Massachusetts Institute of Technology mainly designed for children and teenagers from ages 8 to 16.

One of the main goals of Scratch is to be able to introduce programming to people with no programming experience. As such, Scratch is mainly used in educational environments, such as schools, after-school clubs, libraries and community centers. This goal lead to the way that Scratch was designed. The design was motivated by the needs and interests of the target audience and consists of a single-window user interface layout with a minimal command set, which is also designed in a way to influence the target audience's approach to error handling.

Using Scratch, users are able to import or create images and sounds using a built-in paint tool and sound recorder. Scratch projects contain sprites representing 2D characters and objects, and a stage background. The programming is performed by placing command blocks in the user interface, which are used to control the elements of the Scratch projects.



```
when [flag] clicked
forever
    move 1 steps
    if on edge, bounce

when [flag] clicked
forever
    if pick random 1 to 6 = 1
        turn ↻ 180 degrees
    wait 1 secs
```

Figure 3.5: The Scratch user interface [MRR+10].

The Scratch user interface is represented in Figure 3.5. The left pane is a command palette filled with buttons used to select one or more categories; the middle pane has three folder tabs used to view and edit the sprite, the costumes and the sounds with the use of programming blocks; the bottom-right pane contains the thumbnails of all sprites present in the project and the upper-right pane is used to visualize the results. Scratch scripts are built by snapping blocks together, which end up representing statements, expressions and control structures.

Scratch is also a live development tool, as there is no edit/run mode distinction and it doesn't require any type of compilation. Users are free to experiment with the interface and observe the results immediately, even being able to change parameters and add blocks to a script while it's running. This approach encourages users to tinker with the tool, making them active learners. Scratch highlights the current scripts being run with a white border so that users are able to receive feedback during execution and to understand which scripts are being run. If the script has an error, it will have a red border instead of a white one [RMMH$^+$09, MRR$^+$10, scr].

Users become quick learners when using Scratch due to the simple interface and the ability to generate executable code from code blocks provided by the tool.

3.6.2 Unreal Engine

Unreal Engine is a video game engine first developed in 1998 by Epic Games, revealed with the launch of the first-person shooter *Unreal* video-game. When first introduced this engine was able to combine rendering, collision detection, scripting, file management networking, visibility and artificial intelligence, and later became popular due to engine's architecture, which had a modular design, and the implementation of the UnrealScript scripting language, based on C#, which allowed users to create modifications easily. Throughout the years, video game engines have become incrementally more powerful, leading to bigger and more realistic video games. The current version of Unreal Engine is Unreal Engine 4, which was released in 2014.

Visual programming is present in Unreal Engine when the user wants to implement blueprints, which are a powerful tool in Unreal. Blueprints are instances created by users which allow them to group together objects with other objects, triggers, collisions and lights and that can be reused multiple times throughout development. Blueprints allow users to make modifications to one of the instances and have them automatically be transferred to every instance in the project, which is particularly useful when changing values in the lights or collisions. Users are also able to add scripts, actions and variables to the blueprints, which is an action lacking in similar tools in other game engines [Sha14, San16].

Thanks to the use of visual programming and blueprints, users are able to drag and drop blocks of objects, variables and states, which can then be connected to form graphs, functions and macros. This is done so that users can more easily implement different blueprint types, such as [unr]:

- Level designs, to be able to manipulate and reference Actors within the level and also manage level-related systems such as checkpoints;
- Classes for interactive assets such as buttons, doors, switches and collectible items;
- Creating a playable game character and control its camera behavior and animations, together with implementing input events for the mouse, controller and touch screens;
- Creating a Heads-Up Display.

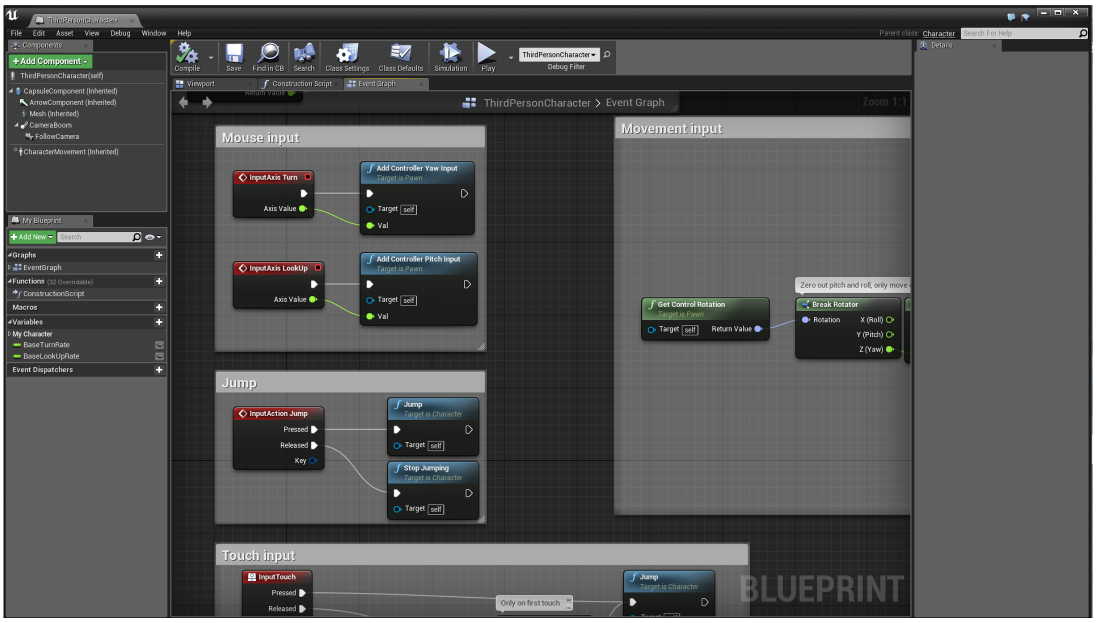


Figure 3.6: The Unreal Engine blueprint user interface [unr].

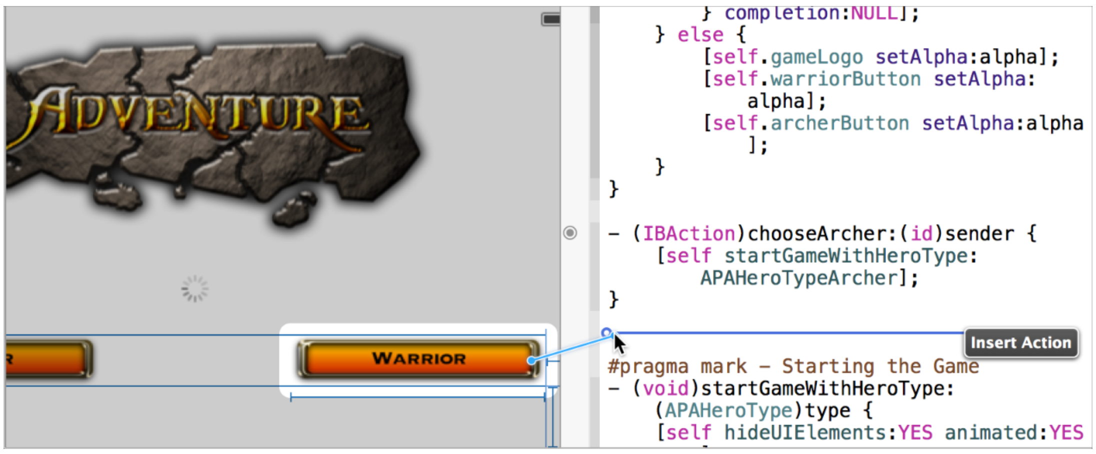
3.6.3 XCode

Xcode is an IDE in which programmers are able to develop for iOS. The `Xcode` term is used in two different ways as it serves as the name of the application and the name of a suite of utilities that accompany the application. The current version is Xcode 12.

Xcode projects follow a structure similar to other IDEs, where projects are composed by the source files, written by the user, which are compiled by the application, by `.storyboard` and `.xib` files, which are graphically expressing interface objects meant to be instantiated whenever the app runs, by resources such as audio and images, by the settings instructions interpreted when the app is built and by any frameworks required by the code.

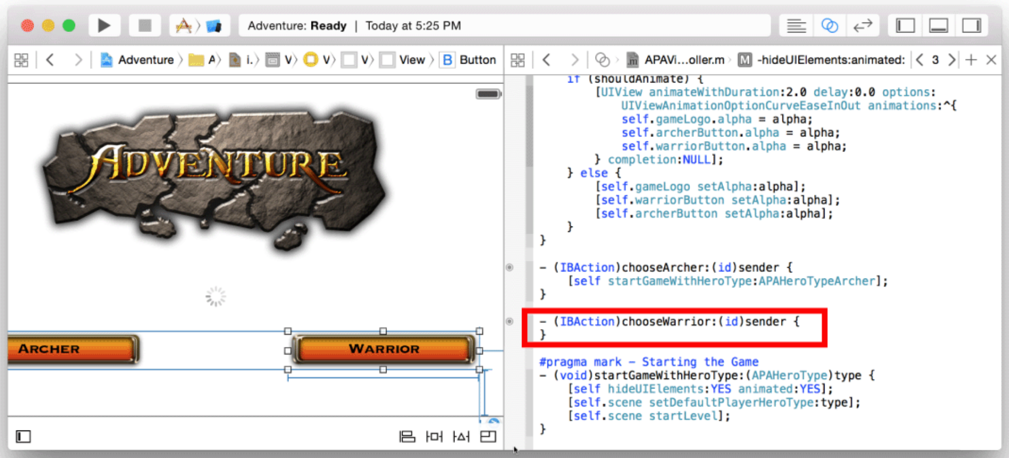
This IDE contains an interface builder, which is a way by which users are able to visually create the user interface of their project application and connect the interface objects with their code. The code communicates with the interface objects via action and outlet connections. An action connection is used is used for sending messages from a control, which is a user interface object with instant actions or visible results whenever the user manipulates an object, to the code. An outlet connection is used to send a message from the code the interface object, which can either be a control or any other object. These objects are defined in a storyboard or on a `xib` file.

A control should send a message to the code whenever the user activates it. Users are able to insert action messages into the code by opening the Assistant Editor, which splits the Xcode interface screen into two components, the storyboard editor and the code editor. Users are then able to use a drag and drop technique, i.e., the Control-drag method, by dragging from the control in the Interface Builder into the implementation file and Xcode will automatically open a popup for the user to personalize. Figure 3.7 displays this process. After editing the information required,



```
        } completion:NULL];
    } else {
        [self.gameLogo setAlpha:alpha];
        [self.warriorButton setAlpha:
            alpha];
        [self.archerButton setAlpha:alpha
            ];
    }
}

- (IBAction)chooseArcher:(id)sender {
    [self startGameWithHeroType:
        APAHeroTypeArcher];
}

#pragma mark - Starting the Game
- (void)startGameWithHeroType:
    (APAHeroType)type {
    [self hideUIElements:YES animated:YES
```

Figure 3.7: A user using the Control-drag method to implement an action message in their code [xco].



```
    if (shouldAnimate) {
        [UIView animateWithDuration:2.0 delay:0.0 options:
            UIViewAnimationOptionCurveEaseInOut animations:^{
            self.gameLogo.alpha = alpha;
            self.archerButton.alpha = alpha;
            self.warriorButton.alpha = alpha;
        } completion:NULL];
    } else {
        [self.gameLogo setAlpha:alpha];
        [self.warriorButton setAlpha:alpha];
        [self.archerButton setAlpha:alpha];
    }
}

- (IBAction)chooseArcher:(id)sender {
    [self startGameWithHeroType:APAHeroTypeArcher];
}

- (IBAction)chooseWarrior:(id)sender {
}

#pragma mark - Starting the Game
- (void)startGameWithHeroType:(APAHeroType)type {
    [self hideUIElements:YES animated:YES];
    [self.scene setDefaultPlayerHeroType:type];
    [self.scene startLevel];
}
```

Adventure: Ready | Today at 5:25 PM
Adventure
View
Button
APAVi...oller.m
-hideUIElements:animated:
3
ADVENTURE
ARCHER
WARRIOR

Figure 3.8: The final result after the action message implementation [xco].

Xcode will then automatically implement code onto the source file. The process to creating a communication between the code and an outlet is very similar to creating an action message. Figure 3.8 displays the final result after the Xcode implementation [And12, xco, Neu13, KN13].

3.7 Summary

This chapter presented the most relevant tools in the areas of focus. The main features of an IDE were discussed and some extensions for VSCode were presented. Visual Programming was also introduced, since the concept of using tools and applications to automatically transform code from one implementation into another is often encountered in this type of programming.

There is an inherent lack of live feedback when it comes to acceptance testing as tools that integrate Cucumber and Gherkin do not provide live testing feedback to the user. Programmers lose a large amount of time when writing tests due to the traditional edit-compile-run cycle, and the introduction of live feedback technologies could mitigate this problem.

There also exists a lack of VSCode extensions that focus on assisting the user in writing Gherkin steps and creating tests automatically. Currently, users have to manually translate every Gherkin feature step into JavaScript test files and manually write every step, even if they are very similar to ones that have been previously written. This process can be resource and time consuming during the development stage. Adding to this lack of user assistance is also the factor of VSCode having a lack of extensions that provide feedback on whether the tests associated with the Gherkin scenarios are successful or not. By having to manually run the tests every time a change is done within the files, the process can become tedious and repetitive.

Chapter 4

Problem Definition

4.1 Current Issues

The Visual Studio marketplace [1] is filled with several extensions that aim to assist the user with Gherkin feature files, but there are still some problems to be solved. These problems include:

- **Lack of assistance in writing Gherkin**. While writing Gherkin steps, extensions allow users to use the snippet functionality present in their IDE. However, users get little to no assistance when trying to write steps based on ones that have been written previously or steps based on tests in the test file. This functionality would be particularly useful when writing steps that are very similar to a previous one, but with a small change in one of the conditions. Having to write the steps manually can be a repetitive and tedious task, so by using the autocomplete provided by Visual Studio Code, the time spent writing steps can be reduced, improving the user's experience while writing Gherkin..

- **Manually having to transform Gherkin into tests**. There are currently no extensions that allow users to transform Gherkin steps into JavaScript or TypeScript code while they are editing the feature document. After writing the Gherkin steps, users will then have to manually transform every Gherkin step that has been written into a JavaScript test and then type it into the test file. This process can be time consuming.

- **Lack of an extension attached testing output**. After writing the Gherkin steps and corresponding JavaScript tests, users would often need to switch from their current workspace

[1] Visual Studio Marketplace (Retrieved by: 25 June 2020)

and manually type in the tests they would want to execute. This is also a process which can become repetitive and time consuming.

4.2 Motivation

Given the problems featured above, the main goal of this dissertation is to facilitate the way in which users are able to write Gherkin and to improve the feedback they get when testing. Since manually creating tests based on Gherkin steps can be a repetitive and tiresome task, the extension aims for improving the user's experience by turning this into an automatic process. Also, replacing the need to manually switch from the development environment to the testing environment in order verify the results of the tests that have been written with automatically displaying the results of the test to the user whenever the contents are saved is also a way to help achieve our goal.

4.3 Problem Statement

Currently, manually typing in every Gherkin step and having them translated into JavaScript and typed in the test file is a time consuming and tedious process. There also exists a lack of extensions which allow users to visualize the results of their tests without the need to switch from their current workspace.

4.4 Proposal

With the motivation and problems stated above in mind, we decided to create a Visual Studio Extension with features such as syntax highlighting, snippet creation, autocomplete and keyword replacement, while using an architecture that allows for easy adaptations to other IDEs. Adding to this, the extension should also be able to automatically transform steps written in Gherkin into tests written in JavaScript or TypeScript and write them into the correct test file, which should be done by associating each feature file with the corresponding test file. After writing scenario steps the users could then also use the extension to run the cucumber framework in order to assess and verify how well each test is performing. Cucumber is able to show through an output if all scenarios passed and which scenarios have failed, when applicable.

4.4.1 Desiderata

Desiderata is a latin term used when referring to something that is wanted. In this document it is used to present the actions that a user should be able to perform, based on the proposal written in 4.4. Those actions are:

D1: Use snippets in order to be able to automatically input main keywords and corresponding templates;

D2: Use Visual Studio Code's autocomplete function to automatically type in Gherkin tests that have been written previously in the feature file;

D3: Use Visual Studio Code's autocomplete function to automatically type in Gherkin tests that are based on functions written in the JavaScript or TypeScript test files associated with the feature file;

D4: Automatically create JavaScript or TypeScript test files associated to the Gherkin feature file while writing Gherkin steps;

D5: Automatically create JavaScript or TypeScript tests based on the Gherkin steps that have been written, while editing the Gherkin feature file;

D6: Use a command to automatically replace `examples` headers in Gherkin steps that match their corresponding values;

D7: Be able to view the results of the tests that have been written and executed after saving the feature file;

D8: Be able to edit the path where the JavaScript or TypeScript tests are automatically generated;

D9: Be able to control what tests are executed when saving the feature file;

D10: Be able to safely edit and delete feature steps without affecting the corresponding tests;

Chapter 5

Work Developed

5.1 Language Server Protocol

The Language Server helps in implementing features which are traditionally more tiresome to implement. When developing features such as autocomplete or GOTO definitions, this work must be repeated several times, as each feature requires multiple APIs.

According to the Microsoft Visual Studio Documentation, the factors that lead to the implementation of are threefold [lspa].

- Visual Studio Code has a `Node.js` runtime, while Language Servers are usually implemented in their native programming languages;
- When using a Language Server, language features can be resource intensive as they may require several operations that could incur significant CPU and memory usage. This is unsustainable because VS Code's performance cannot be affected by those operations;
- Integrating multiple language toolings with multiple code editors is heavy work. Language toolings need to adapt to code editors with multiple APIs and the code editors need to expect more than a uniform API from the language toolings. In order to implement support for M languages in N code editors it is required the work of N * M.

The Language Server Protocol was specified by Microsoft. It allows for standardized communication between language tooling and code editor, making it so that Language Servers can be implemented in any language and run in their process. This not only allows for performance improvements when communicating with the Language Server Protocol, but also to easily integrate language toolings with multiple LSP-compliant code editors and any LSP-compliant code editors can easily pick up multiple LSP-compliant language toolings [lspb, lspa].

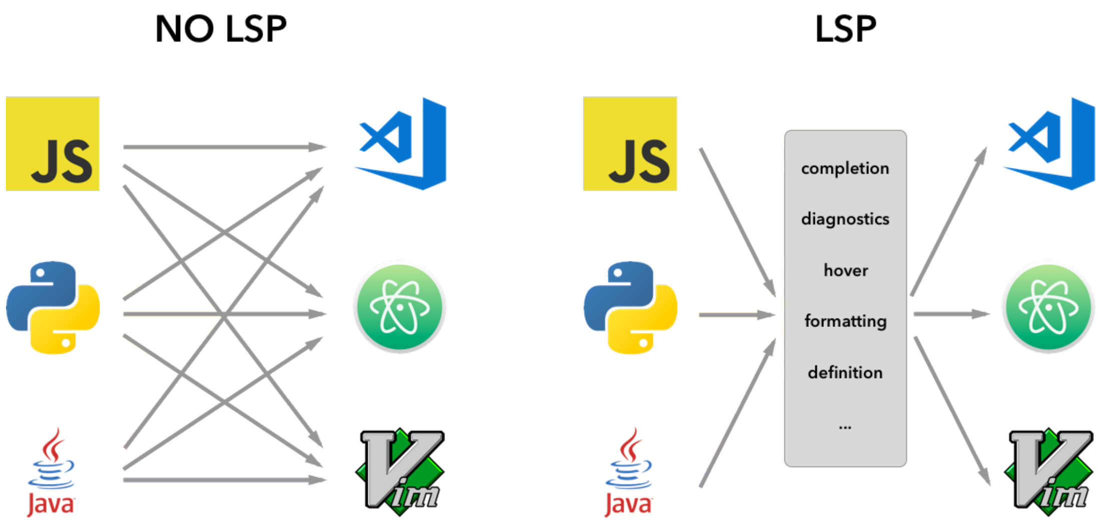


Figure 5.1: Diagram displaying the differences in implementation with and without the use of a Language Server Protocol. Without the LSP, a `python` language extension must be implemented three different ways so as to be be incorporated into code editors such as `Visual Studio Code`, `Atom` or `Vim`. The same happens with JavaScript or Java. When using an LSP, one implementation for every language is able to be performed in multiple code editors [lspb].

As demonstrated in Figure 5.1, without the use of the LSP one language tooling had to implement multiple versions in order to be able to be compatible with multiple code editors, while the LSP offers a standardized way for one language tooling to be compatible with multiple code editors.

5.2 Extension Features

5.2.1 Snippets

The extension has several features designed to assist the user in writing Gherkin feature files.

Snippets allow users to quickly insert repeated code or boilerplates onto their text. This extension has several snippets in order to provide quality of life improvements for the users. The most relevant snippets present in the extension are present in Table 5.1 (p. 43).

Whenever a user is editing a feature file and he types in one of the prefixes mentioned above, they are able to automatically paste the corresponding body, which allows for faster step creation. The users have snippets available for all Gherkin keywords, so as to help as best as possible.

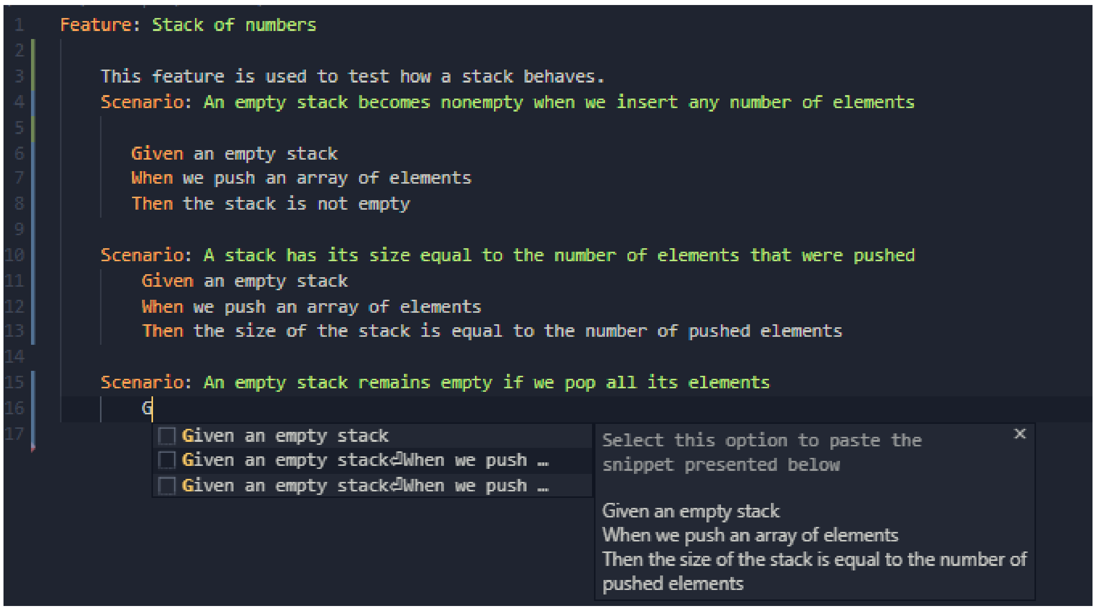
5.2.2 Autocomplete

Another important feature available in the extension is autocomplete. There are two main types of autocomplete available in the extension. One of them is the possibility of instantly pasting a step that was written previously, displayed on Figure 5.2 (p. 43).

PREFIX	BODY
!f	Feature: Feature name Feature Description
!s	Scenario: Scenario name Given Type your precondition(s) here When Type your event(s) here Then Type your expected result(s) here
Examples	Examples: | Header 1 | Header 2 | Header 3 | | Value 1 | Value 2 | Value 3 |
!g	Given Type your precondition(s) here
!w	When Type your event(s) here
!t	Then Type your expected result(s) here

Table 5.1: The main snippets used in the extension. The prefixes represent the input the user has to type in and the body represents the content that is posted in the text document.

Many times, when a user is editing a feature file they want to reuse a step that was written previously but with a small change. With the use of autocomplete, they don't need to rewrite the entire step. By starting to write the keyword 'Given' a popup will appear suggesting to the user all the steps that were written previously. The user may then scroll through all the options and select one according to their needs.



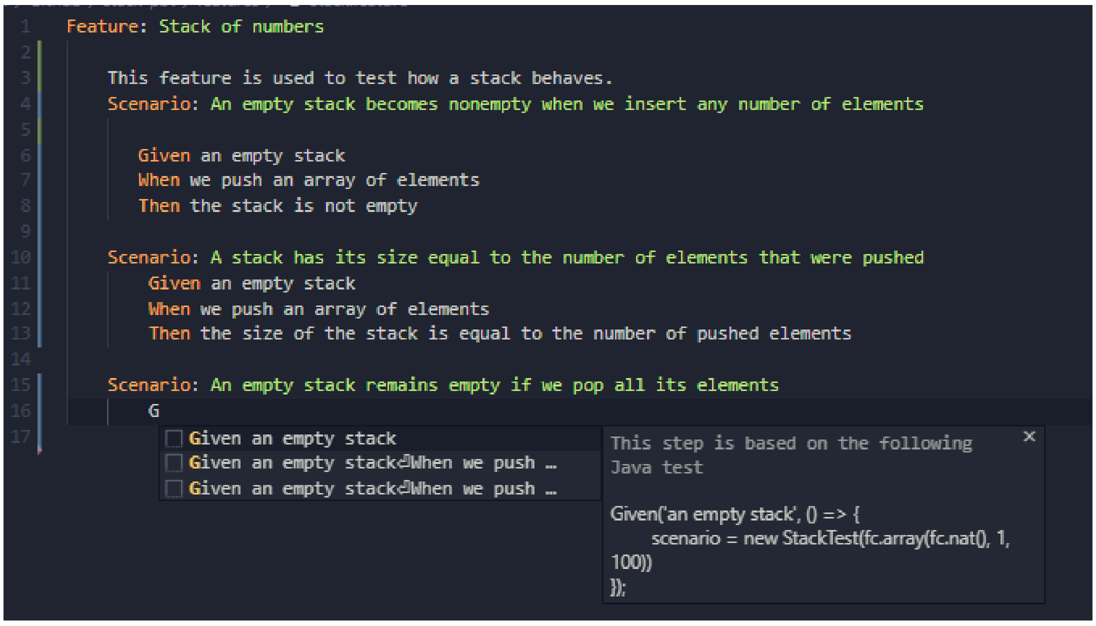
```
Feature: Stack of numbers

    This feature is used to test how a stack behaves.
    Scenario: An empty stack becomes nonempty when we insert any number of elements

       Given an empty stack
       When we push an array of elements
       Then the stack is not empty

    Scenario: A stack has its size equal to the number of elements that were pushed
        Given an empty stack
        When we push an array of elements
        Then the size of the stack is equal to the number of pushed elements

    Scenario: An empty stack remains empty if we pop all its elements
        G
```

```
Given an empty stack
Given an empty stack↵When we push …
Given an empty stack↵When we push …
```

```
Select this option to paste the
snippet presented below

Given an empty stack
When we push an array of elements
Then the size of the stack is equal to the number of
pushed elements
```

Figure 5.2: Autocomplete for writing a full step that has been previously written. When the user starts writing a step, every step that includes what has been written will be available to be selected.

The other type of autocomplete, displayed on Figure 5.3 is the possibility of writing a step based on a test written in the JavaScript or TypeScript test file. When editing the feature file, the extension has information about the corresponding JavaScript or TypeScript test file. The extension parses the file and extracts information about the steps. Then, when users start writing the beginning of a `Given`, `When`, `And`, `Then` or `But` step, the extension will suggest the steps that are written in the test file and that match the step being written.



```
Feature: Stack of numbers

    This feature is used to test how a stack behaves.
    Scenario: An empty stack becomes nonempty when we insert any number of elements

        Given an empty stack
        When we push an array of elements
        Then the stack is not empty

    Scenario: A stack has its size equal to the number of elements that were pushed
        Given an empty stack
        When we push an array of elements
        Then the size of the stack is equal to the number of pushed elements

    Scenario: An empty stack remains empty if we pop all its elements
        G
```

```
Given an empty stack
Given an empty stack⏎When we push …
Given an empty stack⏎When we push …
```

```
This step is based on the following
Java test

Given('an empty stack', () => {
    scenario = new StackTest(fc.array(fc.nat(), 1,
100))
});
```

Figure 5.3: Autocomplete for writing a step that has been written in the test file. When the user starts writing a step, every step that matches what has been written will be available to be selected.

5.2.3 Storing JavaScript and TypeScript Tests

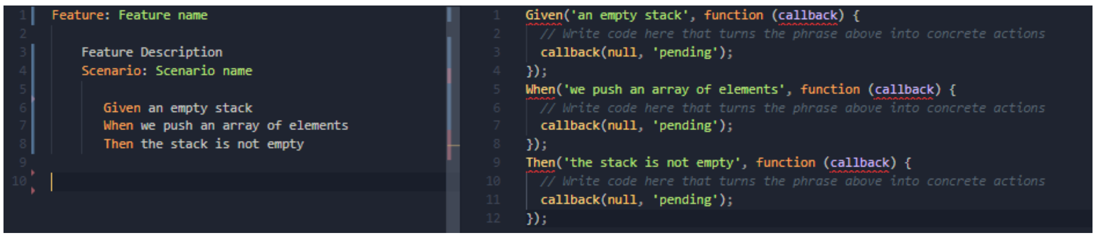
One of the main components of the extension is the ability to automatically translate Gherkin steps into JavaScript or TypeScript tests, depending on the preference of the user.

Whenever a user finishes writing a full gherkin step, i.e., composed of at least a `Given`, a `When` and a `Then` step, the extension selects the text that is in front of each step and creates a JavaScript/TypeScript test based on that step and on a predetermined template. The template for the tests is as follows:

```
"Template('lorem ipsum', function (callback) {
    // Write code here that turns the phrase above into concrete actions
    callback(null, 'pending')
});
```

Source 5.1: JavaScript code template for the tests that are automatically transformed from Gherkin.

The `Template` is replaced with the step keyword and `lorem ipsum` is replaced with the step definition. The created test is written into a file with the same name as the feature file but with a JavaScript or TypeScript termination, depending on the preference of the user. An example of this process is showed on Figure 5.4.



```
Feature: Feature name

    Feature Description
    Scenario: Scenario name

        Given an empty stack
        When we push an array of elements
        Then the stack is not empty

```

```
Given('an empty stack', function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback(null, 'pending');
});
When('we push an array of elements', function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback(null, 'pending');
});
Then('the stack is not empty', function (callback) {
  // Write code here that turns the phrase above into concrete actions
  callback(null, 'pending');
});
```

Figure 5.4: After writing the test on the left side of the image, the extension will automatically write the code on the right onto a test file. That test file will have the same name as the feature file and it may be a TypeScript or JavaScript file, depending on the preferences of the user.

In the settings of the extension, the user is able to write the path in which they want to store the test files. The user is also able to select a checkbox that defines whether or not they want the extension to automatically write the tests onto the test file. This way, if the user wants to write some scenarios with no JS/TS correspondent they are also able to do it. The main objective of these two settings is to give the user as much control as possible over the extension so as to not have any undesirable outcomes.

5.2.4 Header Replacement

The keyword `Examples` is used when a user wants to have variables in the scenario steps. This extension not only supports writing examples with a compatible snippet but also allows for automatically replacing the headers in places where they are compatible. The way this works is as follows:

In the extension's settings there is a configurable array of strings titled `regex` where the user is able to type in regular expressions. These regular expressions will be used in order to find matches in the feature file. There is already a predefined regex defined in the extension that searches for any integer in the file. For instance, if a user has a regex that searches for e-mail strings, then he may write the example in Source 5.2.

```
Examples:
    | <int_0> | <string_0> |
    | 1 | test@email.com |
```

Source 5.2: A user written Example with `<int_0>` and `<string_0>` as headers and `1` and `test@email.com` as their corresponding values.

After writing this, the user may send a formatting request within Visual Studio Code (Format Document in the command prompt) and the extension will match each value, `1` and test@email.com in this case, with a regex. After finding the matches, the extension will map the value and corresponding header with a regex and search for words in the feature file that match each regex. When these matches are found, the extension simply switches each match in the feature file with the corresponding header. For clarification purposes, if a user has the scenario written in 5.3 and they decide to send a formatting request with the examples written in Source 5.2, then the scenario will be automatically replaced with the code in Source 5.4.

```
    Given another.email@email.com as an e-mail address
    When the number of users is below 10
    Then an account with that e-mail address can be created
```

Source 5.3: Example of a scenario that can be written in a feature file.

```
    Given <string_0> as an e-mail address
    When the number of users is below <int_0>
    Then an account with that e-mail address can be created
```

Source 5.4: Resulting code after the user uses the formatting command that will replace the scenario written in Source 5.3.

5.2.5 Cucumber testing output

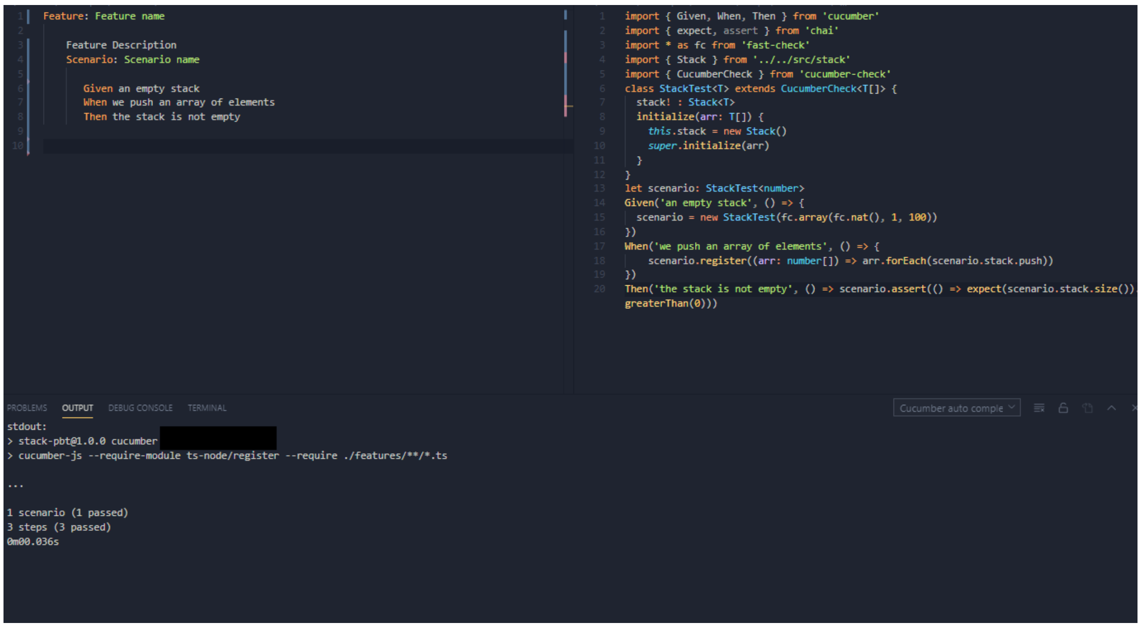
This extension allows users to run cucumber scripts, with the objective of viewing the results of the scenarios that they have written.

In order to accomplish this, the user must first define the command to run and the path where it should be run in the extension settings. After defining this, whenever a user saves the document that they are editing, the extension will show the output of the command. This feature is showcased in Figure 5.5 (p. 47).

5.2.6 Diagnostics

Diagnostics are a form of providing visual output to the user and to inform them of possible warnings and errors. Visual Studio Code presents these diagnostics in the form of a squiggly line underneath the part of the code that triggers a diagnostic.

In this extension, a warning is showed to the user in the form of a yellow squiggly line under the step whenever a user writes a step in the wrong order, e.g., writing a `Given` step after a `When`. An example of a diagnostic is showcased in Figure 5.6 (p. 47).

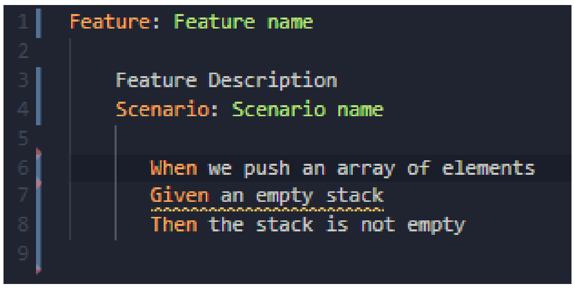


```
Feature: Feature name

    Feature Description
    Scenario: Scenario name

        Given an empty stack
        When we push an array of elements
        Then the stack is not empty
```

```
import { Given, When, Then } from 'cucumber'
import { expect, assert } from 'chai'
import * as fc from 'fast-check'
import { Stack } from '../../src/stack'
import { CucumberCheck } from 'cucumber-check'
class StackTest<T> extends CucumberCheck<T[]> {
  stack! : Stack<T>
  initialize(arr: T[]) {
    this.stack = new Stack()
    super.initialize(arr)
  }
}
let scenario: StackTest<number>
Given('an empty stack', () => {
  scenario = new StackTest(fc.array(fc.nat(), 1, 100))
})
When('we push an array of elements', () => {
    scenario.register((arr: number[]) => arr.forEach(scenario.stack.push))
})
Then('the stack is not empty', () => scenario.assert(() => expect(scenario.stack.size()).
greaterThan(0)))
```

```
stdout:
> stack-pbt@1.0.0 cucumber [redacted]
> cucumber-js --require-module ts-node/register --require ./features/**/*.ts

...

1 scenario (1 passed)
3 steps (3 passed)
0m00.036s
```

Figure 5.5: After writing the Gherkin tests presented on the left side of the image, the extension will automatically create tests, which can then be edited onto tests like the ones presented on the right side of the image. After this process, the user may then save the feature file, which will automatically run the cucumber command that has been stored and will present the results to the user.



```
Feature: Feature name

    Feature Description
    Scenario: Scenario name

        When we push an array of elements
        Given an empty stack
        Then the stack is not empty
```

Figure 5.6: A warning diagnostic is presented to the user whenever a step is written in the wrong order.

If the user hovers over the line, an information window will appear displaying a message that the step order is switched.

5.2.7 Extension Settings

The extension has six customizable settings, where four of them have already been described previously. Those settings are:

- A checkbox that defines if the extension is allowed to automatically write to the test file;
- A string to define the path from where to run a Node Package Manager (npm) command;

- A string to define the npm command to be run in order to test the scenarios;
- A string to define the path where to store the test file that is automatically generated.

One of the remaining settings is an array of strings where a user can type in customized parameter types to be replaced when writing the tests in the JavaScript/TypeScript file. The extension is implemented with a default parameter type, where all integers are switched to `{int}` in the test file. This means that the step written in Source 5.5 will, before being stored in the test file, be translated by default into the code written in Source 5.6.

```
Given a stack with 2 elements
```

Source 5.5: Example of a Given stack with an integer written in the precondition.

```
Given a stack with {int} elements
```

Source 5.6: Final result of the code written in Source 5.5 after being transformed

Users are able to add more parameter types by adding a string that follows the structure `regex, word to replace`. As an example, if a user inputs the string `red|blue, color` this means that wherever there is a match for the regular expression `red|blue` will be replaced with `color`.

The other remaining setting is an customizable array of strings, where the user is able to add regular expressions to be matched with the `Examples` values. The extension will then sort through the array in order to retrieve all the regular expressions, so that when the user sends the Formatting Request detailed in subsection 5.2.4 the extension will have the information about what regular expressions to search for in the feature document.

Chapter 6

Implementation

6.1 Extension Architecture

The extension is divided into two main components: the client and the server.

All messages exchanged between client and server are formed by a header and a content part, which are separated by '\r\n'. The header part consists of multiple header fields which are comprised of a name and a value and that follow the structure conformed to the HTTP Semantic. As of the time of writing this dissertation, the Language Server Protocol supports the header fields represented in Table 6.1.

HEADER FIELD NAME	VALUE TYPE	DESCRIPTION
Content-Length	number	The length of the content part in bytes. This header is required
Content-Type	string	The mime type of the content part. Defaults to application/vscode-jsonrpc;charset=ut

Table 6.1: Header fields sent from the client to the server, with their corresponding value type and description.

The content part contains the actual content of the message and uses JSON-RPC [1] to describe requests, responses and notifications. The content encoding is done using the charset provided in `Content-Type` field mentioned above. If no charset is provided it defaults to `utf-8`, which is the only encoding supported as of the time of writing.

[1]JSON-RPC (Retrieved by: 25 June 2020)

The client and server use request messages and notifications to communicate. Request messages are used to describe a request between the client and server and every processed request must send a response back to the sender of the request. A request message is composed of a request id, the method to be invoked and, optionally, the method's parameters. A Response message is sent as a result of the request. If the request has been successful, the response is null. A Notification message works the same way as an event and does not require a response. A notification can send as content a method to be invoked and, optionally, the notification's parameters.

The first request sent from the client to the server is the Initialize request. Any request sent that is received by the server before the initialize request should receive an error response and any notification received before the initialize request should be dropped unless it's an exit notification, in which case it will allow the exit of a server. The client must not send any additional requests or notifications until it has received an `InitializeResult` from the server. Similarly, the server is also not allowed to send requests or notifications until it has responded with an `InitializeResult`, with the exception of the notifications `window/showMessage`, `window/logMessage`, `telemetry/event` and `window/showMessageRequest`.

When the extension is activated, the client is created using an id, a name, the server options and the client options. The server options have run and debug executables. The client options contains information regarding the type of files that are to be analysed by the extension, which in this case are the feature files. The client also creates a system file watcher, responsible for notifying the server about file changes.

The server upon initialization is responsible for receiving requests sent from the client, handling those events and returning the results to the client. When initializing, the server sets multiple variables to true, defining its behavior towards certain events. These variables include:

- `textDocumentSync` is used to determine what information is sent when a text document opened by the user is edited. The extension currently has it set to send the full text existing in the document;

- `completionProvider` is used to determine whether the extension is compatible of providing completion requests or not;

- `documentFormattingProvider` is used to determine whether the extension is able to provide document formatting support or not;

- `documentRangeFormattingProvider` to determine whether the extension supports formatting while typing or not.

The communication between the editor and the language server during a regular coding routine is displayed in Figure 6.1 (p. 51).

After finishing the initial communication, the server may now receive the `onDidOpenTextDocument` notification from the client, which is signalled whenever a user opens a feature file.

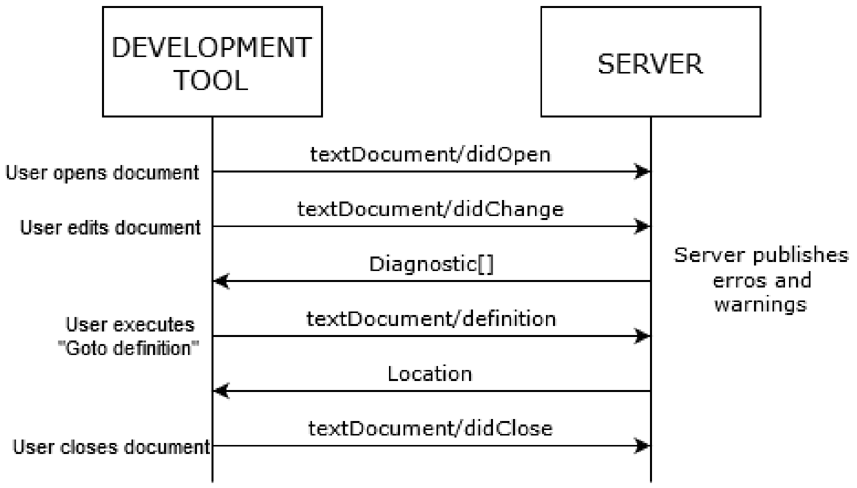


Figure 6.1: Typical communication between Development Tool (client) and the Language Server. A `textDocument/didOpen` notification was sent, signalling that the user has opened a text document. The client then sends a `textDocument/didChange` notification, signalling that the document has been edited, which is followed by a `textDcoument/definition` notification from the server. A `textDocument/definition` request is sent from the client, indicating that the user has executed a `Goto Definition`, followed by a `textDocument/definition` response from the user. Finally, the client sends a `textDocument/didClose` notification, signalling that the user has closed the document. Image adapted from [lspa].

This notification must not be sent more than once before sending the corresponding close notification. This notification's method is `textDocument/didOpen` and the parameter is one of type `DidOpenTextDocumentParams`, which contains the text document that was opened [lspa, hyp].

6.2 Feature Requests and Communication

6.2.1 Snippets

The extension contains a JSON file containing all information regarding the snippets. The file is written with the structure demonstrated in Source 6.1.

```
[
    "title": {
        "prefix": "prefix",
        "body": [
            "body"
        ]
    }
]
```

Source 6.1: Code snippet displaying how the JSON file is structured.

The title is used to identify the snippet, the prefix is the text to be written by the user that activates the snippet and the body is the text, which may have one or multiple lines, that is pasted onto the document text being edited by the user. There are a total of thirteen snippets, as there is at least one for each Gherkin keyword.

Since the snippets are stored client side, it is not necessary to send requests nor to receive information from the server [lspa].

The snippets present in this extension have been adapted from the extension entitled Cucumber (Gherkin) Full Support [2], which is available in the Visual Studio Code marketplace [3].

6.2.2 Autocomplete

In order for the extension to support autocomplete, the client side of the application needs to send a Completion Request and a Completion Item Resolve Request to the server. These are two way communications, as the client expects an answer when sending the request. The Completion Request request is sent whenever a user selects a completion item in the interface and it is done in order to compute completion items at given cursor position and presented via the `IntelliSense` interface existent in VS Code. Whenever this request is sent by the client, the server responds with an array filled with all relevant Completion Items, which has been defined on the handler for the Completion Item Resolve Request.

When handling the Completion Item Resolve Request, the server receives a Completion Item as a parameter. This Completion Item contains three parameters which are then analyzed and changed according to the current context. Those parameters are `data`, which is used as an identifier, `detail` which is the title of the completion item and `documentation` which is the description of the item. After updating those parameters, the handler will then send the updated completion item back to the client and display the correct information to the user [lspa].

The communication between client and server is represented in Figure 6.2.

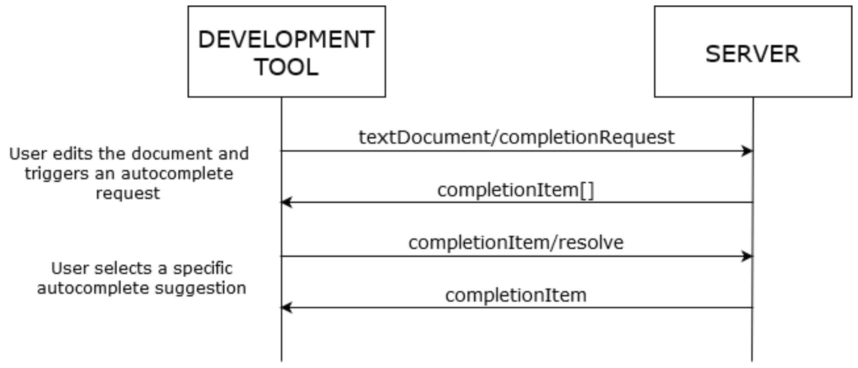

DEVELOPMENT TOOL
SERVER
User edits the document and triggers an autocomplete request
textDocument/completionRequest
completionItem[]
User selects a specific autocomplete suggestion
completionItem/resolve
completionItem

Figure 6.2: The communication between client and server when sending autocomplete requests. The `textDocument/completionRequest` request is answered with a `completionItem[]` and the `completionItem/resolve` is answered with a `completionItem`.

[2]Cucumber (Gherkin) Full Support (Retrieved by: 27 June 2020)
[3]Visual Studio Marketplace (Retrieved by: 27 June 2020)

6.2.3 Storing JavaScript and TypeScript Tests

Whenever a user edits the text document, the client signals the server with a notification signalling that the user has changed the document's text. This is a one way communication, as the client does not expect a response when sending the notification. Before being able to send this request and being able to change a text document, the client must first claim ownership of its content using the `textDocument/didOpen` notification, which has been previously explained in section 6.1.

The server-side handler for this request receives a parameter of the class `DidChangeTextDocumentParams`, which contains two main attributes: `textDocument`, containing the Uniform Resource Identifier of the file that has been edited and `contentChanges`, which contain the contents of the file that has been changed by the user. This handler is the one responsible for sending diagnostics to the client and for everything related with creating and storing the JavaScript/TypeScript test files [lspa].

The communication between client and server is demonstrated in Figure 6.3.

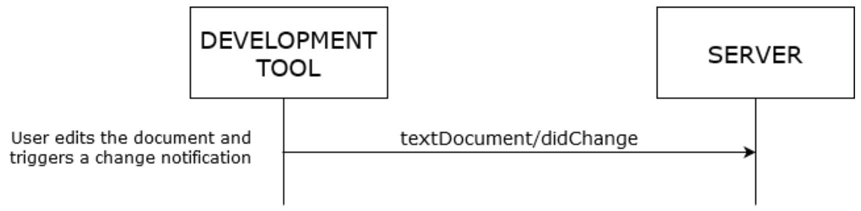


Figure 6.3: Diagram representing the communication between the client (development tool) and the server, whenever the client edits the feature file and triggers a `textDocument/didChange` notification.

6.2.4 Header Replacement

This feature requires the server to change the text of the document currently being edited by the user. In order to accomplish this, the server needs to send a `TextEdit` array. `TextEdit` is a namespace belonging to the vscode-languageserver-types node module that supports replacing, inserting and deleting blocks of text present in the text document. In order to send the `TextEdit`, the server needs to utilize the handler for the formatting requests received, as it is the only function that supports and allows changing the text on the client side.

As previously mentioned in subsection 5.2.4, a formatting request is sent when the user utilizes the command *Format Document* in the command window. The method of the request message that is sent to the server is `textDocument/formatting` and the parameter sent is of the type `DocumentFormattingParams`, which is a class that has two parameters: the text of the document to format and the format options. After analyzing and editing all information necessary, the handler sends a response containing the `TextEdit` array, making this request a two way communication, as the client expects an answer from the server. The handler responsible for dealing with this notification is `onDocumentFormatting` [lspa].

The communication between client and server is demonstrated in Figure 6.4 (p. 54).

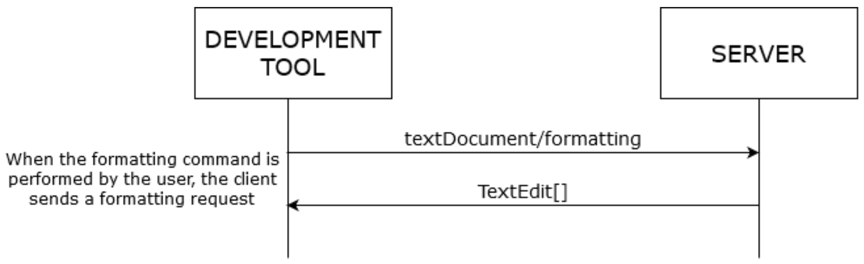

DEVELOPMENT TOOL
SERVER
When the formatting command is performed by the user, the client sends a formatting request
textDocument/formatting
TextEdit[]

Figure 6.4: Diagram of the communication between client and server during a formatting request. The server responds to the `textDocument/formatting` with `TextEdit[]`.

6.2.5 Cucumber Testing Output

Whenever the user saves the text document, the client sends a notification to the server with the method `textDocument/didSave` and a parameter of type `DidSaveTextDocumentParams` which is a class containing the document and, optionally, the content of the document. This notification doesn't expect an answer back from the server, making it a one way communication.

The handler responsible for dealing with this notification on the server side of the extension is `onDidSaveTextDocument`. This handler initiates a child using the `child_process` node module and executes the npm script that has been written in the settings of the extension by the user. It then displays the results on the output of the VS Code window, so that the user can see them [lspa].

The communication between client and server is demonstrated in Figure 6.5.

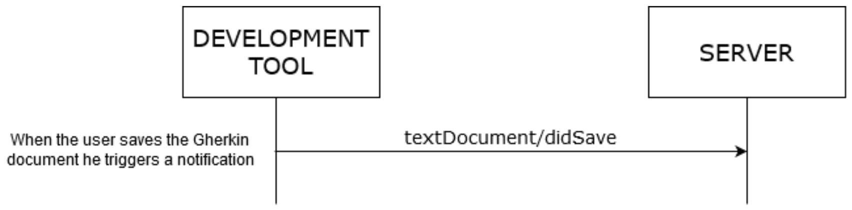

DEVELOPMENT TOOL
SERVER
When the user saves the Gherkin document he triggers a notification
textDocument/didSave

Figure 6.5: Diagram representation of the communication between client and server when a `textDocument/didSave` request is sent.

6.2.6 Diagnostics

Diagnostics are handled on the server side of the extension. Whenever the user changes the feature document, the extension will search for any Gherkin code that has been written out of order, like in the example shown in Figure 6.6 (p. 55). After finding any occurrence of this problem, the server will create a `Diagnostic` with the following structure:

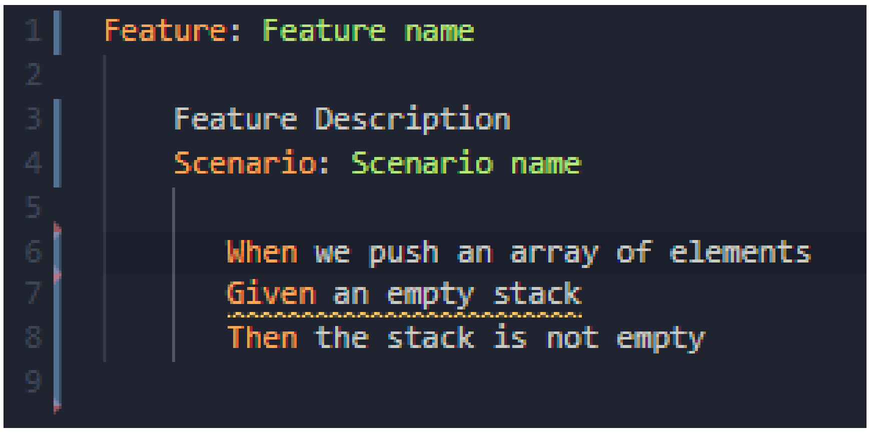
```
{
    severity: DiagnosticSeverity.Warning,
    range: {
        start: firstCharacterOfStep,
        end: lastCharacterOfStep
    },
    message: 'Step order is switched',
    source: id of extension
}
```

Source 6.2: JavaScript code displaying how diagnostics are structured.

There are three different types of diagnostic: Error, Warning and Information. This extension only utilizes diagnostics of the type Warning, as shown above in the severity parameter of the diagnostic. The diagnostic will display the warning under the step that is switched with the message that the order is switched

Diagnostics notifications are sent from the server to the client to signal results of validation runs. It is the responsibility of the server to clear all diagnostics, if necessary. According to the Language Server Protocol documentation, VS Code servers follow a rule to generate diagnostics [lspa]:

> If a language is single file only (for example HTML) then diagnostics are cleared by the server when the file is closed.
>
> If a language has a project system (for example C#) diagnostics are not cleared when a file closes. When a project is opened all diagnostics for all files are recomputed (or read from a cache).



```
Feature: Feature name

    Feature Description
    Scenario: Scenario name

        When we push an array of elements
        Given an empty stack
        Then the stack is not empty
```

Figure 6.6: A diagnostic warning is displayed to the user whenever the steps order is switched. In this case the `Given` step is written after the `When` step, which should be avoided.

Whenever the client document changes, it is the server's responsibility to re-calculate all diagnostics and send them to the client. If there are no diagnostics then the client must send an empty array so that all diagnostics may be cleared. The last pushed diagnostics will always replace the old ones, as there is no merges happening on the client side [lspa].

The communication between client and server is demonstrated in Figure 6.7.

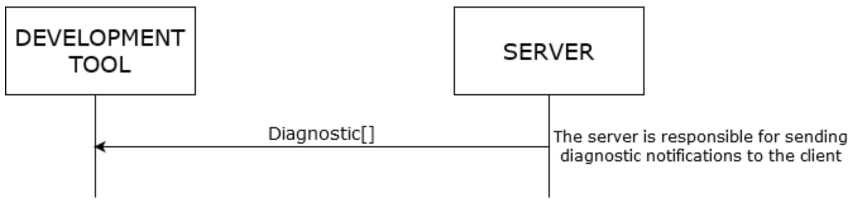

DEVELOPMENT TOOL
SERVER
Diagnostic[]
The server is responsible for sending diagnostic notifications to the client

Figure 6.7: A diagram displaying the communication between client and server whenever a diagnostic notification is pushed to the client.

6.2.7 Extension Settings

When the user changes the settings of the extension, the client will send a request to the server with the method `workspace/didChangeConfiguration` and a parameter of type `DidChange ConfigurationParams`, which contains information regarding the changed settings. This is a one way communication as the client doesn't expect a response from the server.

The handler for this request on the server is `onDidChangeConfiguration` which analyses the information that is in the parameter received and stores all useful data. These settings will then be used by other functions in the extension [lspa].

The communication between client and server is demonstrated in Figure 6.8.

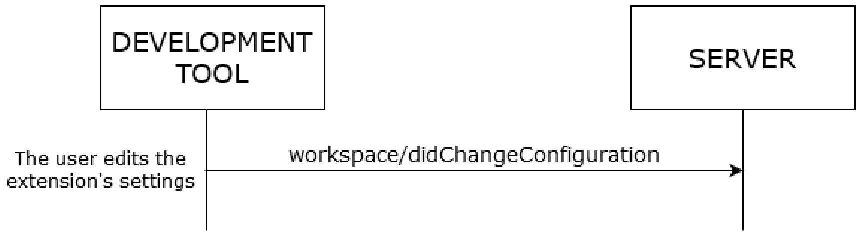

DEVELOPMENT TOOL
SERVER
The user edits the extension's settings
workspace/didChangeConfiguration

Figure 6.8: A diagram displaying the communication between client and server whenever the client sends a `workspace/didChangeConfiguration` to the server, which occurs whenever the client performs a change on the extension's settings.

Chapter 7

Conclusions and Future Work

7.1 Contributions

Based on what has been proposed in 4.4.1 detailed below is what has been accomplished with the development of this dissertation.

The extension has several snippets available to the user, making it so that users are able to successfully and automatically input main keywords and the corresponding templates when they type in one of the snippet's prefixes;

The extension is equipped with an autocomplete functionality, which allows users are able to automatically write Gherkin steps that have been written previously. The steps are sorted by the order in which they have been written. With autocomplete, users are also able to automatically write in steps based on the tests.

When the users are editing the Gherkin feature file, the extension is able to automatically create a test file with the same name as the Gherkin feature file. By editing the extension's settings, the user is also able to type in the path where they wish to create and store the tests.

Gherkin steps that are repeated throughout the document are only written to the file once.

```
Scenario: An empty stack becomes nonempty when we insert any number
of elements
    Given an empty stack
    When we push an array of elements
    Then the stack is not empty
```

Source 7.1: Feature code snippet for a scenario that tests if an empty stack will become not empty when the users push an array of elements into it.

```
Scenario: A stack has its size equal to the number of elements
that were pushed
    Given an empty stack
    When we push an array of elements
    Then the size of the stack is equal to the number of pushed
    elements
```

Source 7.2: Similar feature code snippet to the one written in Source 7.1 with the difference that the expected results is that the size of the stack is equal to the number of pushed elements.

If a user has the scenario written in Source 7.1 and they want to write a scenario such as the one displayed in 7.2 it can be observed that the `Given` and `When` steps are exactly the same in both scenarios. This means that when writing those scenarios, both `Given` steps will be associated with the same Given test in the test file and both `When` steps will be associated with the same `When` test, whereas each `Then` step will have its own `Then` test.

Users are also able to edit tests that have been previously written in the feature file. By editing previously written steps, the step definition in the corresponding test is also edited. This also means that all code written inside the test function will not be changed, as that would be destroying progress done by the user.

One important aspect to notice about the extension is that deleting scenarios in the feature file doesn't necessarily mean that the corresponding test in the test file is also deleted. When the scenario step is deleted, the extension will search for the corresponding test and verify if it has been altered by the user, i.e., if it is different from the template for that particular step. If the test has been altered, then the extension will keep that test in the test file. If the test has not been altered, then it is assumed that it is safe to delete. This way, the user knows that the progress done is safe to keep and that they are free to experiment within the feature file while maintaining the tests.

The extension has a customized formatting command, so that users are able to automatically replace specific values and regular expressions with the corresponding `Examples` headers so as to be able to more efficiently implement Gherkin steps. This is a process that can be done multiple times and the headers will never interfere with another, which means that, for example, an integer that belongs to a header name will not match with a regex that searches for integers. After replacing the values with the headers, the corresponding test in the test file will also be edited without interfering with what the user has already written for that test. This makes it so that the user is free to edit the examples and the steps without the need to worry about rolling back progress.

Users are also able to view the results of the tests they have written, with the use of the extension settings. When saving the document, the extension provides an information message to the user stating that the tests are being run and displays the command and results on the output window of Visual Studio Code.

Based on what was proposed in subsection 4.4.1, the extension satisfies all user needs with the exception of allowing the users to run a single scenario step, which is a feature mentioned in section 7.3.

7.2 Conclusions

Throughout this dissertation, we aimed to create an extension for Visual Studio Code that assisted , while allowing for an easy adaptation for other Integrated Development Environments through the use of the Language Server Protocol. This extension would assist users in writing Gherkin steps, assist users in transforming those steps into JavaScript or TypeScript tests and provide feedback to the users of how the tests are performing while they are editing their files.

A state of the art analysis was performed so as to analyze the current tools available in the areas focused on during this dissertation. This analysis was required so that we could analyze the flaws in the current tools, the missing features and what could differentiate our extension from others available online.

During this work, in order to be able to start the implementation phase, we began by exploring the Language Server Protocol documentation to analyse its architecture and how communications between the client and server were performed. This was fundamental so that we could understand how to build the extension and how to be able to fully take advantage of the features provided by the protocol.

The extension was completed with most of the features implemented, as shown in the 7.1. It is not perfect, as there is still room for it to grow with new features to implement, which are showcased in 7.3.

7.3 Future Work

The final product has some potential improvements, which can not be performed at this time due to time restrictions. These improvements include:

- Removing the need for the user to type in the command which allows cucumber to run all tests. Right now, the user is able to write in the settings a command, in the form of a string, whose output after running is displayed in the Visual Studio Code output window. This process can still be optimized, by having it be an automatic process without the need for the user to input the command. The way the user writes decides to run his current tests is a process with many possibilities, making it hard to predict. In order to automate this process we would need to determine a default way for the users to perform tests, which would require investigation and user tests.

- Removing the need for the user to paste the directory in which the command mentioned in the previous bullet point is run. Similarly to the test command, the directory is also a variable subject to many changes by the user. As of the time of writing this dissertation, the simplest solution is to require the user to type in the directory, so as to allow for more personalization and control from the user, which is a priority in this extension.

- Adding JavaScript and Typescript files to the interpretable files from the extension alongside feature files so that the performance of the extension can be improved. By doing this, the extension will no longer have to wait for the user to type in the feature file to update its internal data with information taken from the tests file.

- Conducting an empirical study in the future with the goal of evaluating how the extension performs with potential users and validating its performance.

References

[And12] Fritz F Anderson. *Xcode 4 unleashed*. Sams Publishing, 2012. Cited on p. 34.

[AO16] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016. Cited on pp. 1, 3, and 6.

[AP12] Saeed Aghaee and Cesare Pautasso. Englishmash: usability design for a natural mashup composition environment. In *International Conference on Web Engineering*, pages 109–120. Springer, 2012. Cited on p. 26.

[AP13] Saeed Aghaee and Cesare Pautasso. Live mashup tools: challenges and opportunities. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 1–4. IEEE, 2013. Cited on p. 26.

[AP14] Saeed Aghaee and Cesare Pautasso. End-user development of mashups with naturalmash. *Journal of Visual Languages & Computing*, 25(4):414–432, 2014. Cited on p. 26.

[ARC+19] Ademar Aguiar, André Restivo, Filipe Figueiredo Correia, Hugo Sereno Ferreira, and João Pedro Dias. Live software development: tightening the feedback loops. In *Proceedings of the Conference Companion of the 3rd International Conference on Art, Science, and Engineering of Programming*, pages 1–6, 2019. Cited on p. 15.

[bdd] Behavior-driven development (bdd) introduction. https://www.leapwork.com/blog/introduction-to-behavior-driven-development/. Accessed: 2020-07-01. Cited on p. 8.

[Bec03] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003. Cited on p. 2.

[Bet16] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Packt Publishing Ltd, 2016. Cited on p. 14.

[BF+14] Pierre Bourque, Richard E Fairley, et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014. Cited on p. 5.

[BM95] Margaret M Burnett and David W McIntyre. Visual programming. *COMPUTER-LOS ALAMITOS-*, 28:14–14, 1995. Cited on p. 30.

[CFS14] Omar Castro, Hugo Sereno Ferreira, and Tiago Boldt Sousa. Collaborative web platform for unix-based big data processing. In Yuhua Luo, editor, *Cooperative Design, Visualization, and Engineering*, pages 199–202, Cham, 2014. Springer International Publishing. Cited on p. 30.

[CPFA10] M. Cunha, A. C. R. Paiva, H. S. Ferreira, and R. Abreu. Pettool: A pattern-based gui testing tool. In *2010 2nd International Conference on Software Technology and Engineering*, volume 1, pages V1–202–V1–206, 2010. Cited on p. 6.

[cuca] Cucumber java 8 support. https://www.baeldung.com/cucumber-java-8-support. Accessed: 2020-07-01. Cited on pp. 27 and 28.

[Cucb] Cucumber documentation. https://cucumber.io/. Accessed: 2020-01-21. Cited on pp. 2, 3, 8, 9, 10, 11, 12, 13, and 14.

[Cucc] Cucumber-eclipse. https://github.com/cucumber/cucumber-eclipse. Accessed: 2020-02-06. Cited on pp. 27 and 28.

[Cucd] Cucumber full support extension. https://marketplace.visualstudio.com/items?itemName=alexkrechik.cucumberautocomplete. Accessed: 2020-01-22. Cited on pp. 3 and 22.

[Cuce] Cucumber query language. https://github.com/enkessler/cql. Accessed: 2020-02-06. Cited on pp. 29 and 30.

[DCPF18] J. P. Dias, F. Couto, A. C. R. Paiva, and H. S. Ferreira. A brief overview of existing tools for testing the internet-of-things. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 104–109, 2018. Cited on p. 6.

[DLF$^+$20] João Pedro Dias, Bruno Lima, João Pascoal Faria, André Restivo, and Hugo Sereno Ferreira. Visual self-healing modelling for reliable internet-of-things systems. In Valeria V. Krzhizhanovskaya, Gábor Závodszky, Michael H. Lees, Jack J. Dongarra, Peter M. A. Sloot, Sérgio Brissos, and João Teixeira, editors, *Computational Science – ICCS 2020*, pages 357–370, Cham, 2020. Springer International Publishing. Cited on p. 30.

[Fow10] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. Cited on p. 14.

[Gau] Gauge documentation overview. https://docs.gauge.org/overview.html. Accessed: 2020-01-31. Cited on pp. 24 and 25.

[Gho10] Debasish Ghosh. *DSLs in action*. Manning Publications Co., 2010. Cited on p. 14.

[Gol04] Kenneth G Goldman. Live software development with dynamic classes. 2004. Cited on p. 15.

[Gwe] Gwen. https://github.com/gwen-interpreter/gwen-web/. Accessed: 2020-02-06. Cited on pp. 28 and 29.

[HH08] Børge Haugset and Geir Kjetil Hanssen. Automated acceptance testing: A literature review and an industrial case study. In *Agile 2008 Conference*, pages 27–38. IEEE, 2008. Cited on p. 7.

[Hol12] Colin Holgate. *LiveCode Mobile Development Beginner's Guide*. Packt Publishing Ltd, 2012. Cited on p. 27.

[HVG13] Brian Hambling and Pauline Van Goethem. *User acceptance testing: a step-by-step guide*. BCS Learning & Development, 2013. Cited on p. 7.

[hyp] Hypertext transfer protocol. https://tools.ietf.org/html/rfc2616. Accessed: 2020-06-30. Cited on p. 51.

[Int] Intellij features. https://www.jetbrains.com/idea/features/. Accessed: 2020-02-06. Cited on pp. 19, 20, and 21.

[isq] International software testing qualifications board glossary. https://glossary.istqb.org/. Accessed: 2020-07-01. Cited on p. 6.

[KN13] Maurice Kelly and Joshua Nozzi. *Mastering Xcode: Develop and Design*. Peachpit Press, 2013. Cited on p. 34.

[LDAF19] Pedro Lourenço, João Dias, Ademar Aguiar, and Hugo Ferreira. Cloudcity: A live environment for the management of cloud infrastructures. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, ENASE 2019, page 27–36, Setubal, PRT, 2019. SCITEPRESS - Science and Technology Publications, Lda. Cited on p. 30.

[Liv] Livecode documentation. https://livecode.com/docs/9-5-0/introduction/welcome/. Accessed: 2020-01-31. Cited on p. 27.

[lspa] Language server protocol specification. https://microsoft.github.io/language-server-protocol/specifications/specification-current/. Accessed: 2020-06-30. Cited on pp. 41, 51, 52, 53, 54, 55, and 56.

[lspb] Language server extension guide. https://code.visualstudio.com/api/language-extensions/language-server-extension-guide. Accessed: 2020-06-20. Cited on pp. 41 and 42.

[MC02] Roy Miller and Christopher Collins. Acceptance testing. 05 2002. Cited on pp. 1, 2, and 7.

[McD13] Sean McDirmid. Usable live programming. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 53–62, 2013. Cited on p. 15.

[MGCW10] Alex McLean, Dave Griffiths, Nick Collins, and Geraint Wiggins. Visualisation of live code. *Electronic Visualisation and the Arts (EVA 2010)*, pages 26–30, 2010. Cited on p. 25.

[MRR$^+$10] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4):1–15, 2010. Cited on pp. 31 and 32.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011. Cited on p. 6.

[Mye90] Brad A Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990. Cited on p. 30.

[Neu13] Matt Neuburg. *iOS 7 Programming Fundamentals: Objective-c, xcode, and cocoa basics.* " O'Reilly Media, Inc.", 2013. Cited on p. 34.

[PSS+18] Lauriane Pereira, Helen Sharp, Cleidson Souza, Gabriel Oliveira, Sabrina Marczak, and Ricardo Bastos. Behavior-driven development benefits and challenges: reports from an industrial study. pages 1–4, 05 2018. Cited on pp. 2, 3, and 8.

[RAM16] André Restivo, Ademar Aguiar, and Ana Moreira. Incremental modular testing for aop. In *Proceedings of the 11th International Joint Conference on Software Technologies*, 2016. Cited on p. 6.

[RAM17] André Restivo, Ademar Aguiar, and Ana Moreira. An incremental approach to testing aop. In Enrique Cabello, Jorge Cardoso, André Ludwig, Leszek A. Maciaszek, and Marten van Sinderen, editors, *Software Technologies*, pages 309–331, Cham, 2017. Springer International Publishing. Cited on p. 6.

[RMMH+09] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009. Cited on p. 32.

[Rob] Robot framework. https://robotframework.org/. Accessed: 2020-02-06. Cited on pp. 23 and 24.

[San16] Andrew Sanders. *An introduction to Unreal engine 4.* CRC Press, 2016. Cited on p. 32.

[scr] Scratch website. https://scratch.mit.edu/. Accessed: 2020-06-30. Cited on p. 32.

[SGW+11] Yu Sun, Jeff Gray, Christoph Wienands, Michael Golm, and Jules White. A demonstration-based approach to support live transformations in a model editor. In *International Conference on Theory and Practice of Model Transformations*, pages 213–227. Springer, 2011. Cited on pp. 25 and 26.

[Sha14] Ryan Shah. *Mastering the Art of Unreal Engine 4-Blueprints.* Lulu. com, 2014. Cited on p. 32.

[SHD+18] Colin Snook, Thai Son Hoang, Dana Dghyam, Michael Butler, Tomas Fischer, Rupert Schlick, and Keming Wang. Behaviour-driven formal model development. In *International Conference on Formal Engineering Methods*, pages 21–36. Springer, 2018. Cited on pp. 7 and 8.

[She18] Mark Shead. *Starting Agile: Finding your Path.* Swift Word Publishing, 2018. Cited on p. 7.

[Som10] Ian Sommerville. *Software engineering*, volume 1. Addison-Wesley Publishing Company, 9 edition, 2010. Cited on p. 5.

[SPB+10] Ellinor Busemann Sokole, Anna Płachcínska, Alan Britten, EANM Physics Committee, et al. Acceptance testing for nuclear medicine instrumentation. *European journal of nuclear medicine and molecular imaging*, 37(3):672–681, 2010. Cited on p. 7.

[SPRG18] Pedro Silva, Ana CR Paiva, André Restivo, and Jorge Esparteiro Garcia. Automatic test case generation from usage information. In *2018 11th International Conference on the Quality of Information and Communications Technology (QUATIC)*, pages 268–271. IEEE, 2018. Cited on p. 6.

[SSS$^+$18] K Sanjana, G Sindhu, M Siri, R Sripriya, and AN Nandakumar. Power generation using livecode. In *3rd National Conference on Image Processing, Computing, Communication, Networking and Data Analytics*, page 19, 2018. Cited on p. 27.

[SWD12] Mathias Soeken, Robert Wille, and Rolf Drechsler. Assisted behavior driven development using natural language processing. In *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, pages 269–287. Springer, 2012. Cited on p. 2.

[swf] Software testing fundamentals. http://softwaretestingfundamentals.com/. Accessed: 2020-07-01. Cited on p. 6.

[Tan90] Steven L Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages & Computing*, 1(2):127–139, 1990. Cited on pp. 15 and 16.

[Tan13] Steven L Tanimoto. A perspective on the evolution of live programming. In *2013 1st International Workshop on Live Programming (LIVE)*, pages 31–34. IEEE, 2013. Cited on pp. 15 and 17.

[tdd] Test driven development with python - what are we doing with all these tests?, chapter 4. https://www.oreilly.com/library/view/test-driven-development-with/9781449365141/ch04.html. Accessed: 2020-07-01. Cited on p. 2.

[Tesa] Test explorer ui. https://marketplace.visualstudio.com/items?itemName=hbenl.vscode-test-explorer. Accessed: 2020-01-22. Cited on p. 23.

[Tesb] Test story. https://marketplace.visualstudio.com/items?itemName=grzesiek110.teststory. Accessed: 2020-01-22. Cited on p. 23.

[unr] Unreal engine blueprint documentation. https://docs.unrealengine.com/en-US/Engine/Blueprints/GettingStarted/index.html. Accessed: 2020-06-30. Cited on pp. 32 and 33.

[xco] Xcode documentation. https://developer.apple.com/documentation/xcode/. Accessed: 2020-07-01. Cited on p. 34.