Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2019/2020

Zadání bakalářské práce

Student: **Muzikář Martin**
Program: Informační technologie
Název: **Nástroj pro efektivní správu testů webových aplikací**
Tool for Effective Management of Web Application Tests
Kategorie: Uživatelská rozhraní

Zadání:

1. Seznamte se s aktuálními technologiemi pro testování webových aplikací. Prostudujte využívané metody pro tvoření nových a udržování již existujících testů.
2. Vyberte vhodnou technologii a navrhněte celkový postup, dílčí metody a aplikaci pro efektivní správu a editaci testů webových aplikací. Při návrhu reflektujte potřeby uživatelů.
3. Implementujte navrženou aplikaci pro vhodnou platformu podle potřeb uživatelů (aplikace využívající TUI, desktop aplikace, webová aplikace, plugin pro IDE) s využitím relevantních dostupných technologií a vhodných knihoven.
4. Vyhodnoťte vlastnosti výsledného řešení na základě experimentů s reálným využitím aplikace.
5. Prezentujte klíčové vlastnosti řešení formou plakátu a krátkého videa.

Literatura:

- Semmy Purewal. *Learning Web App Development: Build Quickly with Proven JavaScript Techniques.* O'Reilly Media, Inc., 2014. ISBN: 9781449370190.
- Steve Krug. *Don't make me think, revisited: a common sense approach to web usability.* San Francisco: New Riders, ISBN 978-0321965516.
- Dále dle pokynu vedoucího.

Pro udělení zápočtu za první semestr je požadováno:

- Body 1, 2 a částečně bod 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Beran Vítězslav, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2019
Datum odevzdání: 28. května 2020
Datum schválení: 1. listopadu 2019

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

TOOL FOR EFFECTIVE MANAGEMENT OF WEB APPLICATION TESTS

NÁSTROJ PRO EFEKTIVNÍ SPRÁVU TESTŮ WEBOVÝCH APLIKACÍ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **AUTOR PRÁCE**	**MARTIN MUZIKÁŘ**
SUPERVISOR **VEDOUCÍ PRÁCE**	**VÍTĚZSLAV BERAN, Ing., Ph.D.**

BRNO 2020

Abstract

The goal of this thesis is to create a tool that allows testers to see the progress of the test they are currently developing on the fly. Hence, to eliminate an issue of constantly re-running tests which take a long time to execute.

The problem was solved by creating a program which modifies the test suite logic by using Java Instrumentation, enabling it to run test steps as the user needs and designing GUI that is able to communicate with previously mentioned modified test suite.

As a result the process of creating tests was significantly sped up (testers were able to create a test with an unknown test suite in terms of minutes), all users that participated in testing were fond of this approach and would adapt to this workflow.

The primary result is pinpointing that things can be handled more efficiently, with proper tools testers can significantly speed up their workflow and also make the introduction process for newcomers easier.

Abstrakt

Cílem této práce je vytvořit nástroj, který interpretuje testy během jejich vývoje a výzkum ohledně dopadů interaktivního testování na uživatele.

Problém byl vyřešen vytvořením programu, který upraví existující testovací sady pomocí Java Instrumentace a poskytne uživatelské prostředí pro manipulaci s upravenou testovací sadou.

Výsledkem bylo zejména zrychlení procesu vytváření testů (při uživatelském testování byli uživatelé schopni napsat test během minut, bez přechozí znalosti testovací sady). Všichni uživatelé kteří podstoupili uživatelské testování vyjádřili zájem o použití nástroje při běžné práci.

Hlavním výsledkem této práce je poukázání na neefektivitu současných nástrojů, analýzu potřeb uživatelů a návrh, vytvoření a otestování nového nástroje pro vyhovění těmto potřebám.

Keywords

java instrumentation, behavior driven testing, UI testing, react, monaco editor, selenium, web application testing

Klíčová slova

java instrumentace, behavior driven testing, testování UI, react, monaco editor, selenium, testování webových aplikací

Reference

MUZIKÁŘ, Martin. *Tool for Effective Management of Web Application Tests*. Brno, 2020. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vítězslav Beran, Ing., Ph.D.

Tool for Effective Management of Web Application Tests

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Vítězslava Berana, Ing., Ph.D. Další informace mi poskytli Tomáš Sýkora, Dongni Wang and Sarahjane Clark. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Martin Muzikář
May 27, 2020

Acknowledgements

My gratitude belongs to everyone who helped me with this thesis. Thank you Mr. Beran for being my supervisor, giving me a unique viewpoint on any presented issue, sometimes challenging my views to make sure I always stood by my decisions.

The Fuse QE team in Red Hat, namely Tomáš Sýkora, for being my consultant inside the company setting, for many hours of brainstorming and providing me with contacts for technical consultations. Dongni Wang and Sarahjane Clark for consulting my designs and user test ideas, I was able to get feedback even from discussing with them and the user tests were able to provide useful data.

Users participating in user tests, namely Alice Rum, Aneta Čadová, Štefan Vereš, and the rest of the team who provided me with constructive feedback and their support.

Contents

1 Introduction **2**

2 Related theory (Software verification, Java Instrumentation, Monaco editor) **3**
- 2.1 Software verification 3
- 2.2 Java Instrumentation 7
- 2.3 Web application components 12
- 2.4 Existing solutions 17

3 Solution draft for making BDT more interactive **20**
- 3.1 Current approach to testing 20
- 3.2 Solutions for user issues 21
- 3.3 Designing the application 23
- 3.4 Verifying designs 27
- 3.5 Application architecture 29
- 3.6 Backend API 32

4 Implementation of interactive BDT tool **37**
- 4.1 Agent implementation 37
- 4.2 Front-end implementation 41
- 4.3 User testing 44
- 4.4 Plans for the future 44

5 Conclusion **46**

Bibliography **48**

A Backend experiments **49**

B Gherkin monarch definition **51**

Chapter 1

Introduction

Verification of software is an important part of the software development process, so it only makes sense that there are many different tools and frameworks that make the verification part easier. There are many different approaches to software verification, the aim of this thesis is not to create the best and the only testing solution to be, because frankly that is not possible.

The aim is to observe one specific approach to verification which is Behavior Driven Testing (BDT for short) and creating a tool to save testers work with manual tasks. You can imagine the result of this thesis as an Integrated Development Environment specifically created for Behavior Driven Testing development.

In this thesis current implementations of tools for BDT will be shown and analyzed. You will get to know basics of program instrumentation and how it is done in the JVM, a deep-dive into the Cucumber framework (which is a BDT implementation) and BDT as a whole, what are its benefits and why is it even used. A design of a tool that runs tests as they are written, its implementation and user testing.

This thesis was made with real input from quality engineers who use Behavior Driven Testing, the analyzed problems and proposed solutions were designed with feedback from this team. Same team also participated in user testing to measure improvements provided by this approach.

First some context will be provided to analyze user needs properly in Chapters 2.1 and 2.4. For the implementation part familiarity with Java Instrumentation 2.2 which is used to modify program's behavior. And to implement an interface for users to interact with using the React library and Monaco editor library 2.3.

Prior to implementing a solution it needs to drafted first. In this thesis the current testing situation is analyzed 3.1, issues in the approach are pinpointed 3.2. With the issues in mind the application is designed 3.3 and a proposal for user testing is made 4.3. And to make the application work as a whole the architecture 3.5 and back-end API 3.6 are also proposed.

Following the solution draft my notes from implementing the application are split into agent implementation 4.1, implementing the user interface 4.2, testing the interface on actual users 4.3 and analyzing the results and making plans for the future of this project 4.4.

Chapter 2

Related theory (Software verification, Java Instrumentation, Monaco editor)

2.1 Software verification

This sub-chapter provides an overview of what are different terms in the area of software verification, or the testing process. The most essential topic of this section is Behavior Driven Testing as it is going to be referenced throughout the thesis. Other methodologies are mentioned as they were an inspiration while designing the whole idea. Software verification is a complex subject to describe on a few pages. As the team's focus is system testing where the application is tested as a whole, unlike unit or integration testing which covers only a portion of the project or communication between parts of the project.

Information in this chapter is primarily sourced from experienced quality engineers and [4].

Regresssion testing

Regression testing is a process of re-running tests after a change in the product to make sure that change didn't break anything. If a change appears and a test results in a failure, testers can see early if a specific feature is not working correctly.

Changes can also introduce wanted failures which don't necessarily mean a feature of tested application doesn't work as expected. Occasionally changes to the test itself are needed.

Integration testing

Integration testing is a process where all parts of the tested application are tested as one complex system. Much like black-box testing, tester interacts with the application and observes outputs as a user would.

Approaches to system testing

When it comes to testing an application where user interacts with the Graphical User Interface several approaches exist to facilitate the verification. The most popular or influential to interactive testing approach are described in the following chapter.

Manual testing

A tester (person responsible for writing and maintaining the tests of a web application) can prepare test scenarios and execute them manually each time there's a need for new results. The benefit of the manual approach is its simplicity, no need for programming knowledge and the fact that UI changes don't really affect the tests. Testers are able to see even the tiniest problems in the application. On the other hand it is really cumbersome to do the same tasks all the time, especially as the application is gaining complexity.

Interactive testing

A more advanced approach is automating the manual tests. There are tools which are able to record user's interaction within a web browser and perform such interactions on demand. However tools like this are usually not capable of executing code, if you need to check the state of a database for example or do any custom logic, those tools will likely not have all the features needed for such tasks. Also a UI change in the application could leave the whole test suite in a broken state which is not desirable.

The most used Open source tool is Selenium IDE [1]. It is made by the Selenium organization which also created Selenium, the biggest browser automation tool. It is simple to understand but provides more advanced features closer to programming. Big advantage of this tool is in being a browser plugin, apart from being easy to install, it does not require setting up environment to run the tests.

Why is automation needed?

Manual and interactive testing are usually used for smaller projects or early prototypes because they are easy to set up and execute. When it comes to bigger projects or fact changing applications, re-recording or rewriting the same test usually leads to tester burnout. According to the Continuous Delivery methodology acceptance tests should be automated which brings several benefits to the entire team [6]:

- Developers get feedback faster.
- Reduced workload on testers.
- Testers can focus on higher-level activities.
- All acceptance tests together create a well base for finding regressions in the software.

Testing User Interface

Because system testing tests an application as a whole that means that the User Interface must be covered by the tests as well. Different platforms provide different solutions but in design most all similar to Selenium described bellow.

Selenium

Selenium [3] is the most popular tool to automate Web applications, it has a big community support, language bindings to most popular languages.

Selenium provides an interface called `WebDriver` which is a language independent program, that accepts commands and instructs a browser to execute such commands. The browsers are backed by `drivers` which handle the communication between Selenium and the browser. A driver exists for each major browser in the market. Having a communication layer between developers and the drivers allows the same code to run on different browsers.

[1] https://selenium.dev/selenium-ide/

The main entry-point for interacting with the automated browser is through a `WebDriver` instance. In the following examples the instance is called `webdriver`.

Locating elements

Elements are located in Selenium by using *selectors*, there are different types of selectors, most commonly CSS[2], XPath[3] and name selectors are used. With a selector an element or a list of elements on an opened page can be found using

```
WebElement element = driver.findElement(By.name("username"));
//or
List<WebElement> element = driver.findElements(By.tagname("h2"));
```

This code snippet tries to find an element where an attribute `name` has value `username`, a proxy object is returned, that provides methods such as `exists` to check if the element was found, or the `is` method that allows to check for predefined conditions such as „element is visible", „element is clickable". If the WebElement exists, it allows for interactions using `sendKeys`, `click`, `clear` methods. WebElements also allow to search in their children by using the selectors, consider following example:

```
WebElement form = driver.findElement(By.id("login-form"));
if (form.exists()) {
    form.findElement(By.name("username")).sendKeys("admin");
    form.findElement(By.name("password")).sendKeys("password");
    form.findElement(By.cssSelector("input.primary[type=\"submit\"]")).click();
}
```

Model based testing

A more flexible and maintainable approach to this would be to use *models* that represent different pages. Which is the main idea behind **model based testing**. Creating abstractions of different pages is a good practise for several reasons:

- better maintainability - if particular UI aspect changes, just rewrite the model and all existing tests should stay the same
- easier collaboration - it is easier to understand a model of a page, than to look at a bunch of selectors and trying to find them in the DOM
- re-usability of already written components

This could be the final solution for most testers but there is still another level of abstraction that can be used. Each interaction with the application should be describable by one short sentence, such as „Click on the Submit button" which provides more abstraction from the code, and stakeholders of product are able to read or even create new test scenarios without any need for knowledge of programming language. This approach is called Behavior Driven Testing (BDT for short).

[2] https://www.w3schools.com/css/css_selectors.asp

[3] https://www.w3schools.com/xml/xpath_intro.asp

Behavior Driven Testing

Behavior Driven Testing [1] is an agile practise that allows all stakeholders to communicate clearly and collaborate in a better fashion on a product feature and every-ones expectation for said feature.

I had a liberty of having a discussion with an agile practitioner that overlooks a huge team of developers and testers, in his own words BDT is a tool that should encourage communication and discussion.

Separate tests are called *scenarios*, each test you would normally program is a scenario that corresponds to a bigger *feature*. For example in a calculator, you might have a feature of `Square root` and different scenarios could be:

- Square root of odd numbers
- Square root of even numbers
- Square root of zero
- Square root of negative numbers

A test suite can consist of many different feature files with a ton of scenarios in each feature file. Scenarios consist of *steps*, steps are the interactions with the application. For programmers it is simply an abstraction for a method, each step has a pattern and arguments. There is a best practise in place that steps pattern should be self-descriptive, but since we use our natural language to define the steps, it is not always achievable to have the same meaning to everyone.

For further explanation, let's create a sample feature file for the square root example

```
Feature: Square root
    Background: Calculator running & clean state
        Given calculator application is running
        And I press "CE" button

    Scenario: square root of~odd number
        When I input 7 into the~calculator
        And I press "Square root" button
        Then calculator should display 2.6457

    ...
```

As you see, some prepositions are used before actually using a step, these are also best practises and aren't enforced by any shape, way or form by BDT. But to quickly explain:

- Given - preparing the initial state of the application
- When - describing an action or an event
- Then - describing an expected outcome
- And, But - used only for ease of reading, instead of having multiple When, usually it's When, And, And...

Also a *Background* block is used, this block is run before each scenario in a feature file. Though it is recommended to use it rarely to keep the scenarios easy to understand.

2.2 Java Instrumentation

Java Instrumentation allows developers to modify certain classes at runtime. By utilising this the test suite can be transformed to behave in a certain way, without any need to modify exiting code. This is a great benefit as it makes setting up the environment require little to no changes in existing code.

Instrumentation is a service that allows Java classes to be modified at runtime. This is done via „Java agents“. This might be confused with Reflection which is an introspection tool, it allows to read metadata of class and in sense of modification is not really flexible, it is commonly used for serialization and accessing private methods and fields of classes. But doesn't support modifying the behavior of a class, the only modifications allowed are sort of a communication layer. What is different about Java Instrumentation is you still get access to all the information you'd get with Reflection but you are allowed to change the actual bytecode of the class, so you can add fields and methods, modify existing behavior of methods, create classes at runtime.

This chapter's main sources of information were [8] and [5] which both go into detail of how exactly the JVM operates [8] and how the transformations are performed [5].

Java Agents

A Java Agent can be described as a program modifying the bytecode. It is specified as a JAR file that contains required attributes in Manifest and is attached to a program running on the JVM.

An agent can be attached while starting the program or dynamically when the program is already running, although attaching agent after start (from the codebase) is up to specific JVM implementation to be supported. If a JVM implementation supports attaching agents at runtime, the loading behavior is unspecified. At the time of writing, it is not a part of the JVM specification to allow modifying already loaded classes which might cause issues as the agent might attempt to modify a class that was already loaded.

Creating a basic agent

Similar to any other Java application, an agent needs to have an entrypoint specified. Among other properties, the agent entrypoint is specified in the Manifest file.

- Agent-Class - fully classified name of the class used as an entrypoint when attaching an agent **at runtime**.
- Premain-Class - fully classified name of the class used as an entrypoint when attaching an agent **at startup**.
- Boot-Class-Path - a list of paths to be searched by the boot classloader, any agent libraries should be added here.
- Can-Redefine-Classes - ability of the agent to redefine classes, `boolean`
- Can-Retransform-Classes - ability of the agent to retransform classes, `boolean`
- Can-Set-Native-Method-Prefix - ability of the agent to the native method prefix, `boolean`

Java 8 was used in all examples, the specification might have changed in newer versions.

Retransformation is a process which uses the registered class transformers, the changes can be additive and one class might be modified by more than one agent. When class is redefined, the current implementation is overwritten by a new one. The class redefinition process takes into consideration if a class transformer is marked as Can-Redefine-Classes and applies the transformations according to this values. Understanding the process in not in the scope of this thesis, more information is available in the ClassFileTransformer documentation[4].

An entrypoint class must contain a method with any of these signatures:

```
public static void premain(String args, Instrumentation instrumentation);
public static void premain(String args);
public static void agentmain(String args, Instrumentation instrumentation);
public static void agentmain(String args);
```

The methods are called in the order as they are written, if a method with Instrumentation argument is found it gets called. The premain method is called when the agent was attached at startup, agentmain if the agent was attached at runtime. Both premain and agentmain methods can exist in the same class or can be in a different classes.

Java Bytecode

Contrary to other popular programming languages, such as C, C++ or Rust, output of compiling Java code is called bytecode. This bytecode is loaded into a virtual machine, with its own instruction set where the individual bytes from the bytecode are operation codes. The operations also take arguments as we are used to from assembler languages, but detailed knowledge of the Java bytecode is not needed to understand Java Instrumentation and hence is not required by this thesis. What is more important is being familiar with the general class structure and how the JVM operates.

Class structure

Since Java Instrumentation allows us to only modify classes, we need to understand what makes a class, how are classes stored, how exactly are they loaded and what does modifying the class mean.

Each class is stored in one file containing data of the `ClassFile` structure.

```
ClassFile {
u4             magic;
u2             minor_version;
u2             major_version;
u2             constant_pool_count;
cp_info        constant_pool[constant_pool_count-1];
u2             access_flags;
u2             this_class;
u2             super_class;
u2             interfaces_count;
u2             interfaces[interfaces_count];
u2             fields_count;
field_info     fields[fields_count];
u2             methods_count;
```

[4] https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/ClassFileTransformer.html

```
    method_info    methods[methods_count];
    u2             attributes_count;
    attribute_info attributes[attributes_count];
}
```

Snippet directly taken from [8]

First three fields are used for validating the contents of the file and aren't important to cover here.

Next we can see a field named `constant_pool` and `constant_pool_count`, this constant pool is an array of values that are bigger than one byte, so each operation can stay at minimal size and also it avoids duplication of those values. There are many possible types of the constant data in a class, but it's important to mention that class references, method references etc. are also stored here.

Access rights to the class are stored in the `access_flags` field. Classes referencing the class and its super class are stored in the `this_class` and `super_class` fields. Essentially the constant type class is just a fully qualified name of the class that the JVM has to resolve. To illustrate better, JDK comes with a handy tool called `javap` that makes the class' byte structure into a human readable structure.

```
  public class mmuzikar.Stepdefs
  minor version: 0
  major version: 52
  flags: (0x0021) ACC_PUBLIC, ACC_SUPER
  this_class: #1                          // mmuzikar/Stepdefs
  super_class: #3                         // java/lang/Object
  interfaces: 0, fields: 1, methods: 8, attributes: 3
Constant pool:
    #1 = Class              #2            // mmuzikar/Stepdefs
    #2 = Utf8               mmuzikar/Stepdefs
    #3 = Class              #4            // java/lang/Object
    #4 = Utf8               java/lang/Object
    ...
   #10 = Methodref          #11.#13       // j...Class.getName:()Lj...String;
   #11 = Class              #12           // java/lang/Class
   #12 = Utf8               java/lang/Class
   #13 = NameAndType        #14:#15       // getName:()Ljava/lang/String;
```

In this structure, you can see the header of a class and a specific parts of the constant pool, as you can see `this_class` and `super_class` are just references to the constant pool, where we can find the fully qualified names of the classes. We can see that somewhere in the class a method `Class#getName` is called and it doesn't take any parameters and returns an object of type `java/lang/String`. Understanding the signatures and everything presented here is not crucial, but being familiar with existence of constant pool and what it's used for is important when dealing with Instrumentation.

Type Signature	Java Type
Z	boolean
B	byte
C	char
I	int
J	long
F	float
D	double
L fully-qualified-class;	fully-qualified-class
[type	type[]
(arg-types)ret-type	method type

Table 2.1: JVM signature table

JVM signature types

Converting JVM signature to the ones we are used from Java with this useful table from the JNI (Java Native Interface) documentation[5]:

As an example the method `long f (int n, String s, int[] arr);` has the signature `(ILjava/lang/String;[I)J`

Class Transformations

Java by default allows us to modify classes in just one way, to register a `ClassTransformer` in Java Agent startup method. The `ClassTrasformer` is an interface with just one method and that is

```
byte[] transform(ClassLoader loader,
                 String className,
                 Class<?> classBeingRedefined,
                 ProtectionDomain protectionDomain,
                 byte[] classfileBuffer)
                   throws IllegalClassFormatException
```

Where we get many parameters we are only able to perform basic reflection tasks with `classBeingRedefined` and the compiled class bytecode in `classfileBuffer` which can be modified and returned in the function.

This means that the agent should be able to read and write valid bytecode which requires developers to be well versed in this. That is a hard and error prone way to create agents which was the reason for 2 awesome libraries which enable more users to write instrumentation code easier and in a safer way.

Frameworks

There are way more than 2 framework for dealing with manipulating bytecode but ASM [6] and Bytebuddy [7] are the most commonly used, Mixins[8] library is worth at least mentioning as it looks like the easiest library to actually develop with.

[5] https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/types.html
[6] https://asm.ow2.io/
[7] https://bytebuddy.net/
[8] https://github.com/SpongePowered/Mixin

ASM

ASM is an all purpose Java bytecode manipulation and analysis framework. It can be used to modify existing classes or to dynamically generate classes, directly in binary form. ASM provides common bytecode transformations and analysis algorithms from which custom complex transformations and code analysis tools can be built [9].

ASM is even used in the OpenJDK, Groovy and Kotlin compilers which speaks a lot about its maturity. Although it deals only with bytecode instructions, it makes the process easier thanks to utilising the Visitor pattern and handling manual tasks such as recalculating the offsets of instructions or dealing with the constant frames or calculating size of stack for methods.

Bytebuddy

Bytebuddy is a code generation and manipulation library for creating and modifying Java classes during the runtime of a Java application and without the help of a compiler [10].

Bytebuddy does not aim to compete with ASM, rather it strides towards abstracting as much bytecode instructions as possible, it utilises complete Domain Specific Language to create new and modify existing classes.

[9]Description taken from project page: https://asm.ow2.io/

[10]Description taken from project page https://bytebuddy.net/

2.3 Web application components

The most user friendly approach is usually to build a Graphical User Interface (GUI from now on) for your application which will also be needed in this thesis. There are countless approaches and resources to building a general GUI application which is the reason this section will cover basics of web application development and more detailed description into the Monaco Editor library as it is widely known by being taken from the Visual Studio Code editor but the documentation is lacking in many aspects.

Web application development

Web applications are used commonly for their universal appeal on any platform, developers don't have to concern themselves with operating system specifics. It is not a surprise that the Web application frameworks and libraries are one of the most rapidly growing projects.

Typical web application consists of HTML to define the contents, CSS to define visual side of the content and Javascript to make the websites interactive and functional. When it comes to creating more complex applications, there are many different libraries and frameworks that aim to simplify the process.

Because the React library was used to implement the application it is going to be described. Other notable mentions are Vue [11], Ember [12], Angular [13].

React

React [2] is a Javascript library for building interactive user interfaces in a declarative way. Philosophy of React is that application is made up of web components which manage their own state and react to changes in the state.

A web component receives `props` from its parent. Props are data coming from an upper layer. A component can keep track of its `state` which is a structure that is manipulated from only inside the component.

React uses virtual DOM technology to keep track of any changes to the web elements, this process is called reconciliation. When a change is detected in the virtual DOM, only then is a redraw of the specific web component actually called in the browser view. This enables the declarative functionality of React and saves performance.

Monaco editor

Monaco editor is a library that uses the editor of popular editor Visual Studio Code[14], it supports syntax highlighting, autocompletion, adding custom commands and widgets, multiple cursors, and everything else modern code editors support. Although it might seem as developers can use the documentation of Visual Studio Code to use as reference when using Monaco Editor - apart from basic examples, the documentation [15] is a list of defined types where some functions have a small description.

Monaco editor installation

Unlike most Javascript libraries, Monaco Editor needs to be loaded in a proper way into the website as it uses `Webworkers` to offload non UI-critical tasks, Monaco Editor comes with its own loader or according to most claims other loaders will work as well. An example

[11] https://vuejs.org/
[12] https://emberjs.com/
[13] https://angular.io/
[14] https://code.visualstudio.com/
[15] https://microsoft.github.io/monaco-editor/api/modules/monaco.editor.html

repository [16] exists to showcase basics of how different loaders can be used. It is not necessary to understand how webworkers work and why is it required to load them, as Monaco Editor abstracts those concepts from developers. After the loaders are set up correctly, a editor can be created with calling a Javascript function on an HTMLElement to use as a parent.

Syntax highlighting

Monaco comes with its own way to specify language syntax in Javascript/JSON, on the contrary VS Code supports other ways. It is called **Monarch** and it works as following:

- Syntax definition is a self-contained Javascript object, supporting only data.
- Syntax rules are defined in a tokenizer field.
- Syntax rules can use groups of tokens defined in the root object.
- Rules can be pushed on a stack.
- a set of rules is used based on what is on top of the rule stack.
- a rule is defined as a list, where [<pattern>, <action>, [<next>]].
- a pattern can either be a regular expression, or a token group defined in the object (referred to as „@<name-of-group>“.
- An action can be a string that holds a category of matched token.
- Next is an optional value if after matching this token a different set of rules should be used, for example [„//*“, „comment“, „commentBody“] would classify „//*“ as a comment and push „commentBody“ on the stack, so the tokenizer now uses that set of rules.

An example of the Cucumber syntax written in Monarch can be seen in Appendix B.

There are more advanced features of monarch that allow matching brackets etc., those are documented on the Monarch website [17].

Adding custom functionality

The editor proposes two ways of interacting with the editor or the contents of the editor which are **commands** and **actions**. There is no explanation provided about what is the difference between those two approaches and what should they be used for. But there are some differences and non-written rules when it comes to using other Monaco components.

The simplest way to differentiate actions and commands would be: commands are more lightweight, they are registered as a function with an optional key binding and optional preconditions. When it comes to actions, they require an id, can be used from the context and command menu. It can be confusing because what Monaco calls command menu shows what developers register as actions.

When it comes to other features such as CodeLens, if an interaction is desired with it a command id needs to be supplied, and if it is desired to have the functionality of actions such as visibility in command menu, an action can be registered that call `editor#executeCommand()` with the `command id`.

[16] https://github.com/microsoft/monaco-editor-samples
[17] https://microsoft.github.io/monaco-editor/monarch.html

Codelens

Codelens is a mechanism where custom text can appear above specified lines of code, that can provide additional info to users. A `command id` can be supplied together with the Codelens object to execute a command when the Codelens is clicked on. An important detail when it comes to executing commands from Codelens is how the arguments are passed to the command.

```
{
    range: range,
    id: "Run-Step-CodeLens",
    command: {
        id: runStepId,
        title: "Run Step",
        tooltip: "Runs the~step",
        arguments: [model, range.startLineNumber]
    }
}
```

In this code snippet, a Codelens object is created that is tied to arbitrary range `range` (which contains start and end line numbers and start and end columns). On click it executes a command with id `runStepId` and passes arguments `model and range.startLineNumber` to the command. But the command by default receives one argument which is named context in the API documentation but the meaning of what the variable contains is not explained anywhere. To make the command work with Codelens defined above, it needs to be registered as

```
//0 as first argument because no key bind is wanted
runStepId = editor.addCommand(0, (ctx, model, lineNum) => {
    commandLogic(model, lineNum);
});
```

Decorating text

Most code editors provide a way to provide feedback, the most famous example is a red squiggly underline of typos or incorrect code. The way to add custom line decorations in Monaco is by calling `editor#deltaDecorations`, this method expects 2 parameters, first one is the old parameters and second one is a list of `monaco.editor.IModelDeltaDecoration` which contains fields range and options, range determines the range the decoration is applied to and options specifies the CSS class names used and other options such as if the decoration should be placed in the margin (which is next to the line numbers). Important note when decorating lines is the first parameter, in the example it is just empty list, but that causes wrong behavior when editing the text. The old decorations are returned when calling `editor#deltaDecorations`. Consider following snippet:

```
var decorations;
...
function decorateText(lines) {
    decorations = editor.deltaDecorations(decorations || [], lines.map((i) => ({
        range: new monaco.Range(i, 1, i, 1),
        options: {
            isWholeLine: true,
            inlineClassName: 'squiggly-error'
        }
    });
}
```

In the snippet a `lines` variable contains all line numbers with syntax errors which get 'squiggly-error' CSS class applied to them, notice that decorations are being set to the return value of the function and are passed when calling the function next time, that is important for consistent text decorations.

Custom widgets

There are different ways of adding an HTML element into the editor, the different ways can be see in an example called „Listening to mouse events" [18] where different widgets are used to demonstrate listening to mouse events.

In summary, Monaco editor provides these ways of customizing the editor:

- Viewzones - allows to specify a custom DOM node for a set number of lines, effectively adding content between lines.
- Content widgets - allows to create a DOM node in the editor, this node is considered as part of the content of the editor, so it scrolls along with the text.
- Overlay widgets - allows to create a DOM node overlaying the editor, node is not affected by scrolling.
- Codelens - allow to display clickable text above a line of code.
- Glyph margins - allows to apply custom decorations to the gutter area, right next to the line numbers.

Working with these specified widgets differs in terms of flexibility, the first 3 items require developers to add the DOM node on their own, so those are the most flexible methods. Codelens allows specifying a command id which is executed when the text is clicked on, in terms of flexibility of manipulating the text, the whole editor content is passed to the function providing the lenses, in that terms there are no limitations. Glyph margins are more decorative, so the only control provided is the line number and the class name, but since Monaco editor allows custom listeners for key and mouse events, that functionality can be provided as well.

Custom autocomplete

Providing custom suggestions while user is typing in the editor is really easy in the Monaco editor, and it is well covered by an example [19]. A suggestion provider can be registered by

[18] https://microsoft.github.io/monaco-editor/playground.html#interacting-with-the-editor-listening-to-mouse-events

[19] https://microsoft.github.io/monaco-editor/playground.html#extending-language-services-completion-provider-example

calling function
monaco.languages.registerCompletionItemProvider(<languageId>, <completionProvider>), where languageId is an id of the language the provider is going to be registered for ('json' or 'java' as an example).

completionProvider is an instance of type monaco.languages.CompletionItemProvider which requires functions

- provideCompletionItems(model, position, context, token)
- resolveCompletionItem(model, position, item, token)

Where both functions return ProviderResult<CompletionList> which is a type alias for CompletionList or a Promise<CompletionList>. CompletionList contains two fields: incomplete and suggestions, where incomplete can be set to true if there is data missing in the items which are resolved by calling the resolveCompletionItem with each item as the third parameter.

When a function is computationally demanding, it is recommended to provide just the most basic information such as labels in the provideCompletionItems function and then add the information on per item basis in resolveCompletionItem function.

The CompletionItem type[20] has a lot of fields to document, but the basic fields are:

- *label* - an id and by default a display text in the suggestions, the id is important to provide for the resolveCompletionItem
- *insertText* - the text that is going to be inserted, can use placeholders to allow users TAB into specific places
- *kind* - the type of the item, such as variable, function, symbol, etc.
- *detail* - short documentation of the item, is displayed above the text when it was inserted.
- *documentation* - documentation of the item, that is displayed on demand
- *insertTextRules* - rules that are applied when this completion is used (this is used mainly to allow using placeholders in the insertText)

Accessing the content

The content of the editor is stored in Models, a model represents one tab open in the editor. A model can be created by calling monaco.editor.createModel(<value>, <language>, <uri>), where value is the initial text, language is optional and determines the suggestion providers and the syntax highlighting used, uri is optional as well, if no file path is specified, Monaco just keeps the models in memory.

Since the editor is made primarily for editing code, there are many functions that facilitate searching for patterns in the model, navigating through lines, applying edits to allow undo and much more[21].

[20] https://github.com/Microsoft/monaco-editor/blob/master/monaco.d.ts#L5242
[21] https://microsoft.github.io/monaco-editor/api/interfaces/monaco.editor.itextmodel.html

2.4 Existing solutions

The closest things to existing solutions are various plugins to text editors and IDEs that highlight the syntax for describing different testing scenarios, some tools also provide additional functionality.

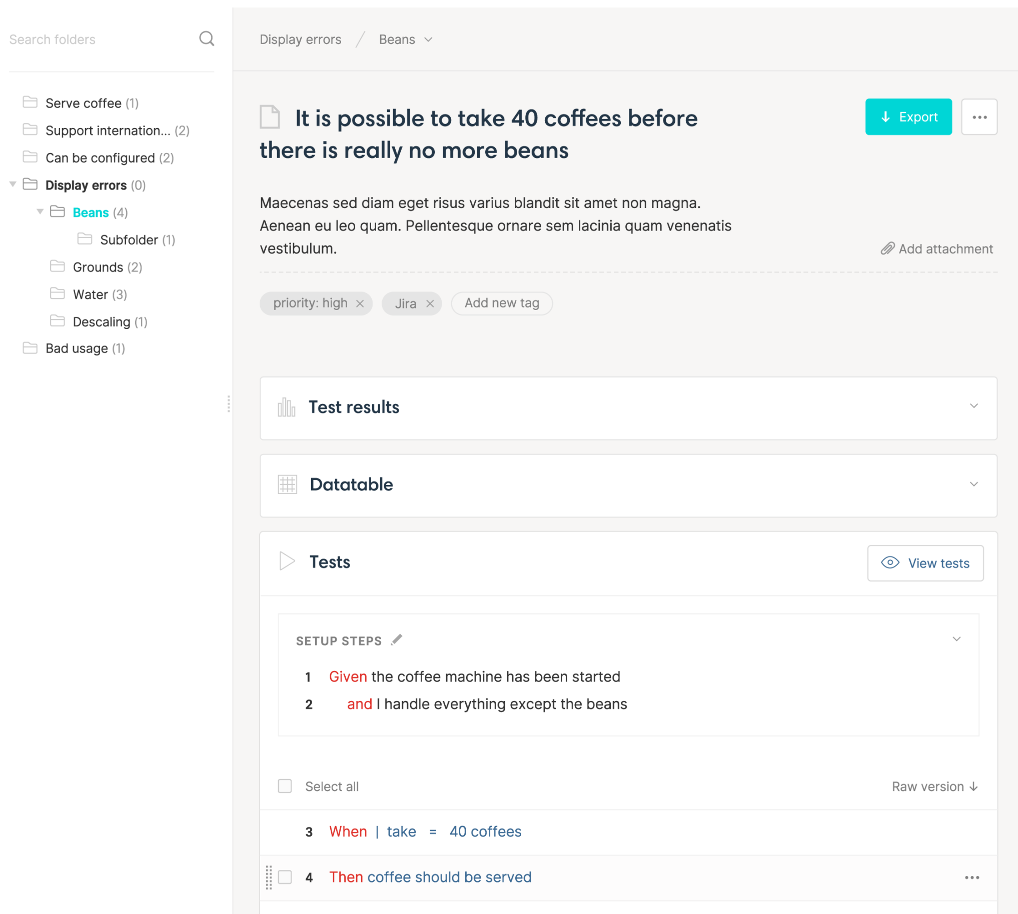
These tools are described and analyzed because Cucumber Studio provides the most functionality and the Intellij IDEA plugin is the most used by the user base. At the time of writing Cucumber studio was the most sophisticated tool. Other tools exist such as Cuke Test [22] and plugins for text editors Visual Studio Code [23], Atom [24].

Cucumber Studio

Cucumber Studio is an in-house tool created by the team behind Cucumber which is a popular Behavior Driven Testing framework. It provides more functionality than other tools.

This tool was definitely created as a tool for for everyone in the software creation process, there is no code visible to the users.

Figure 2.1: Cucumber studio test scenario example[25]



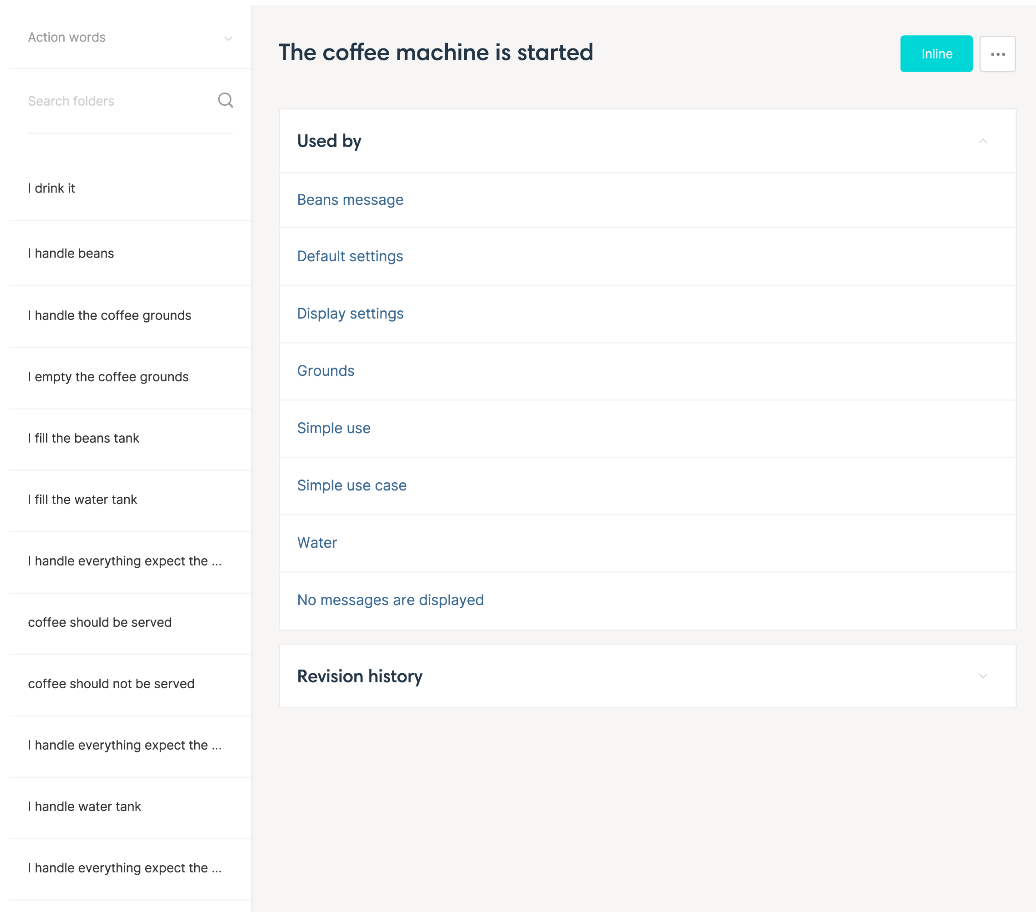
In the picture you can see an example of one test scenario, it is a good representation for stakeholders of a product, but from the testers perspective, it could be improved. Also

[22] http://cuketest.com/
[23] https://marketplace.visualstudio.com/items?itemName=alexkrechik.cucumberautocomplete
[24] https://atom.io/packages/cucumber

the Cucumber Studio has quite unusual definitions in terms of BDT terminology as there are actions and action words and results. Overall I found it confusing and too convoluted. As a tester I am writing actions or steps, whether it is to do a test setup or to do a verification.

But a great feature of this tool is this feature of showing where a step is used, providing users with other scenarios which use that step as an example. This is a common practise while writing tests to see what other scenarios use and modify to suite their needs.

Figure 2.2: Cucumber studio step „Used by“ page[26]



Cucumber plugin for Intellij IDEA

While doing research in the team of quality engineers, every one of them stated they used this plugin as their main driver for writing BDT Scenarios, main reasons for that are:

- the team writes the test logic in Java which means most of them are using Intellij IDEA anyway.
- By far this plugin has the best support for scanning dependencies of test suite for any steps to provide autocomplete.
- It provides „jump to definition“ feature which the testers use when investigating any issue with scenarios.

When asked what they missed, common requests were (in order):

- a breakpoint functionality for scenarios.
- a hierarchical view of test suite features and tags.
- Way to run specific test scenario while it is open in the editor.

The following requirements are the basis for the designed tool: **make the work less convoluted**, don't force tester to look for information - **provide step definition list and search**, allow testers to analyze the application under test state - add code execution functionality. If the goals were to be summarized into one short goal it would be: Provide an Integrated/Interactive Development Environment for Behavior Driven Testing.

Chapter 3

Solution draft for making BDT more interactive

The overarching goal of this tool is to save time for testers and make their jobs easier. To know how and if time can be saved, an analysis of the current workflow needs to be done in order to propose a better solution.

3.1 Current approach to testing

Let's say there's a new feature in a web app that needs to be tested. First the new feature needs to go through a **hands-on phase**, where tester observes what the feature brings to the application and if there is anything present in the test suite they can use for testing this feature. This hands-on phase usually takes day or two, depending on the size of the feature.

Then a **happy path** scenario gets created which means writing the scenario for the feature as it was designed to get used. This phase is hard to guess the average time as you need to take into account if the required logic is written, so let's say an hour for one scenario for a new feature when all the logic is already written in the test suite, however if new logic needs to be written, that can take up to 3 or more days.

How is a scenario written?
Most testers start with copying other scenario, then they open the application and start performing the actions manually and trying to find steps that perform those actions as well.

When a happy path scenario or a base scenario (a minimal working scenario) is created, testers usually start copying that scenario and modifying it, when they modify it and feel like it is correct, they just run the test and observe the results. And keep in mind these tests take time to execute, average time of one test execution is roughly 4 minutes [1].

Issues in current approach
The biggest issue in the current approach is everything takes too long, especially when copying and modifying tests most testers describe that phase as „trial and error“ where they often face issues with typos or other logic issues.

Note that it is not really easy to see what exactly is going on in a running UI tests, you don't see a cursor, you just see a browser that either performs the actions on its own

[1] This information is approximated from the test suite used by the quality engineering team. At the time of writing there were in total 682 test scenarios and execution time in total was around 2 days and 4 hours. Resulting in average time per test scenario 4.8 minutes.

or stalls, it doesn't really tell you what is wrong, it might be waiting for some condition, or you wrote a CSS selector wrong, but you will learn that only when the test fails and the browser window is closed.

It is possible to save the HTML and a screenshot of the page, but all users told me that it is often not enough to see what the issue is.

Another issue is, it's quite easy to miss steps that are already defined, this leads to each tester defining their own steps which then brings unnecessary complexity into the test suite and later makes maintenance harder.

Initial feedback

Primary users are people skilled enough to write test scenarios and the logic behind it, so quality engineers. However Behavior Driven Testing is made so not only testers are capable of understanding the test scenarios, it is meant to be a common ground between technical people and others such as project managers, technical writers, even software developers who are not familiar with the test suite, can easily read what is tested for a specific feature.

This application is suited mainly for the testers but could theoretically be used by other groups of people.

the user needs

Although users are now capable to work just fine enough, that doesn't mean there isn't a room for improvement. I obtained a feedback from testers in various phases of familiarity with the testing process. Main complaints that turned into the list of the user needs were:

- difficulty to familiarize themselves with tests and steps for features not written by them
- lacking options of debugging the tests, the process is still code centered
- while debugging, there's no easy option to just pause the test execution
- no warnings regarding typos or incorrectly used steps

When asked what they missed, common requests were (in order):

- a breakpoint functionality for scenarios.
- a hierarchical view of test suite features and tags.
- Way to run specific test scenario while it is open in the editor.

3.2 Solutions for user issues

First of all let's recapitulate the main requirements for the tool and sum up our options. What is required:

- fast feedback - tester should always know what is happening
- information - tester should be able to get enough information from the application to make proper decisions based on that information
- safety - any wrong action should not sacrifice tester's progress, they should feel safe to experiment

- ease of use - application should be easy enough even to be used as a macro tool, if testers have automation written for specific scenarios they should be able to use the automation
- easy to setup - minimal work should be needed to start the application when a „compatible“ test suite already exists, it should be as easy as starting the tests with a different parameter

What needs to be changed to provide this functionality?
As it was described this functionality is not provided by default in any standard test suite implementation. The test suite functionality must change in order to allow any other requirement to even be addressed.

Ideal test suite functionality
Before any changes the test suite starts executing any specified tests and reports the different results in a desired way (web page, text document). Creating tests interactively requires the test suite to execute testers actions and wait for commands. First action to implement this tool will be to create a test suite that initializes and waits for any commands from the user, turning it into an interpreter effectively.

After the test suite is able to respond to commands a way for users to exchange information with the test suite is required. The first and easiest information to provide is to list all the step definitions from the running test suite.

How will it be made interactive?
Second step to making the testing process interactive is allowing testers to execute a step. This step should be executed like it was read from a feature file, all parameters must be extracted correctly and passed to the testing function.

When the step fails the test suite should report an error but not fail and exit. And if such report was to happen, then testers need to see it. Which means the reporting interface (either output streams or different means in term of internal test suite reporting) must be provided to the tester in the application.

In this state the application can be considered a minimal working product, although it does not allow creating a whole test scenario in its entirety a significant amount of time can be saved and the safety and fast feedback goals were met.

Providing more information
Since the application under test is running as the test is written more information can be provided the user. A mechanism to allow providing custom information from the application under test must be provided.

Creating complete test scenarios
When executing the tests the application should keep track of the steps executed to make a test scenario from them. To add to the ease of use, the test scenario and feature should be also possible while writing the steps.

Summary
To allow proper working of the tool, the test suite must be able to execute commands on demand and an application must be designed around this feature.

Implementing the solutions

To address all the needs from previous section, it is apparent that my solution will consist of two different components communicating with each other.

Java Instrumentation is going to be used for manipulating existing test suites to provide required behavior. Meaning a program will exist that will modify the behavior of the test suite and provide an API for communicating with the test suite. This is further described in section 3.5

The second component is a User Interface which means that first of all it needs to be designed and verified with users, further described in section 3.3 and 3.4. And for this specific project a working compatible test suite needed to exist for any user testing. It was required because the users needed to see what they are testing and the results of their actions. The details for implementing the User Interfaces are in section 3.5 where data structures used for inner and outer communication are used.

3.3 Designing the application

Platform

Platform is an important fact to keep in mind while designing UI but due to the nature of this tool, it is not required to take phones or touchscreens into consideration, since testers do their jobs on laptops and write using a keyboard. So platform is going to be just a desktop application.

My first idea was to implement just a terminal interface, since writing the steps is quite similar to typing commands into terminal. Modern terminals are able to provide graphics enhancements and handle mouse input quite well but with the planned complexity of this project, sooner or later the terminal interface would become cluttered. Also it is not desired to make people learn to use the application, it is meant to save time, so spending time learning keyboard shortcuts and navigation in the application would be counter productive.

A great idea was to implement the fronted as a part of an IDE, so no new tool is needed. Therefore, the final platform was **web application** which is able to run in the browser, integrates well into editors like VS Code and thanks to Electron can be made into an actual standalone application.

Mockups

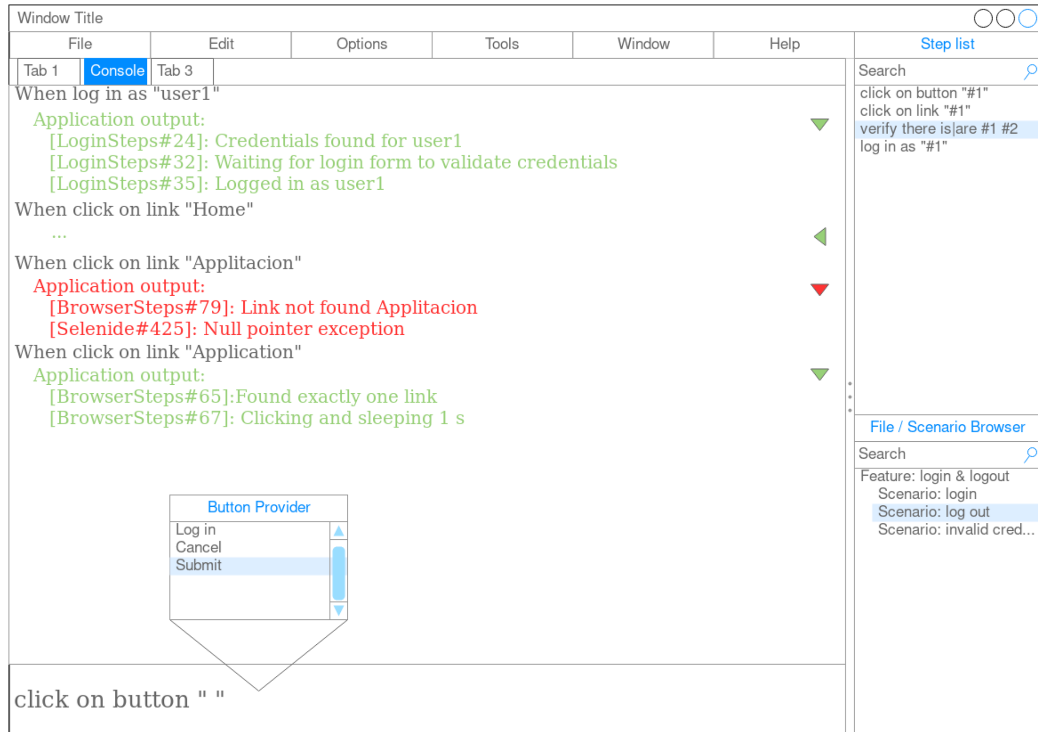
As stated before, the first prototypes used a shell-like interface which worked quite well but other designs started to come to mind while working on mockups. What if there was no differentiation between typing steps and executing them? When user writes the step in an editor, it just executes. What if there was no editor present in the UI? And the steps were presented just as cards neatly formatted as it will be in the resulting scenario.

Let's focus on the two different approaches, the *Shell* and *Editor* designs to evaluate which would be a better fit.

Shell design

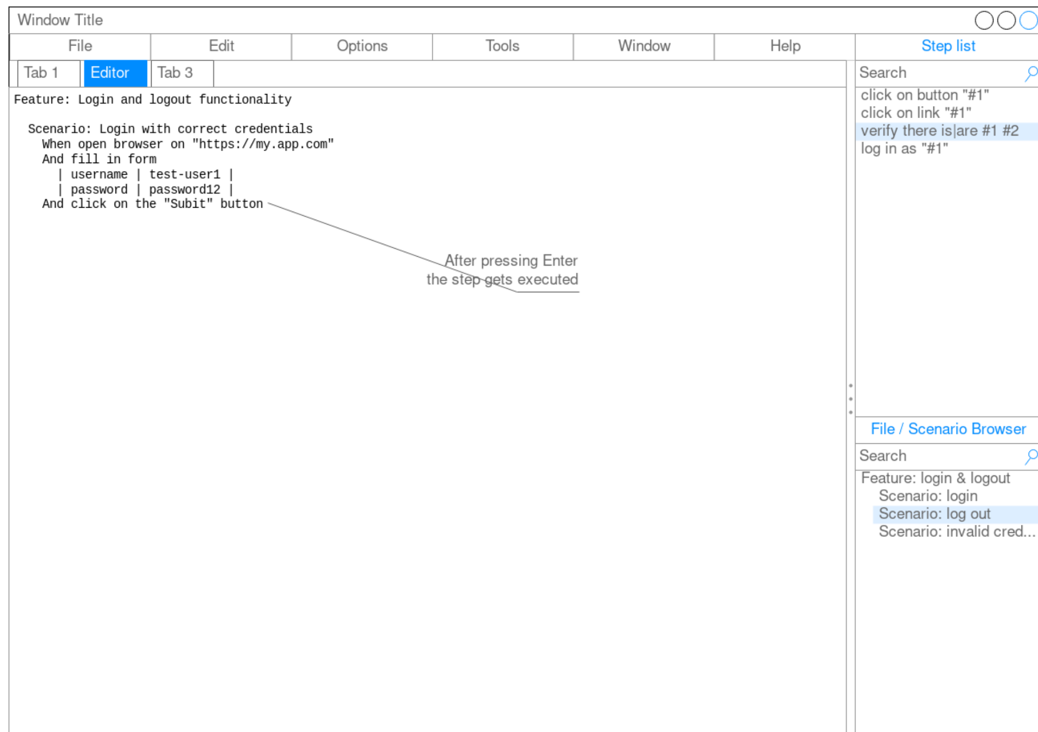
In figure 3.1 you can see, there is one main element at the bottom, where user writes the steps they want executed and at the top they see the status of executed steps and potentially the output. On the right there is a list of all available steps and other tools, this panel would more or less stay the same between the two designs. A great advantage of this design is that it's just simple for developing but also for understanding, since it's

Figure 3.1: Shell design mockup



so close to the terminal interface, many users should be able to just pickup the tool and understand what is going on where. Also it's potentially more expandable when taking into consideration the ability to write code in the input panel. A disadvantage is that this isn't the natural process of writing scenarios, some commands will just become actions nowhere to be documented. Or the output could very quickly become unreadable and the written scenario can be lost. On the other hand this might be what we need for more advanced features like discarding failed steps or debugging.

Figure 3.2: Editor design mockup



Editor design

The editor design (figure 3.2) is different by the entire application just being an editor window (with maybe the side panels), and all the actions user want performed would be accessible from that editor. The greatest advantage is the seamless experience it would provide, the tester would write test scenarios as they are used to writing and the scenario would be executed as they are typing. But that can also be a disadvantage. I can see more possibilities in confusing the user with technically more advanced actions, such as copy pasting, moving and removing steps. If the only consideration was executing steps as they are being written, this approach is the best fit, but for the more advanced purposes could be really confusing for the user to understand how exactly should these actions be performed and also leave the scenario in progress intact.

Decision between designs

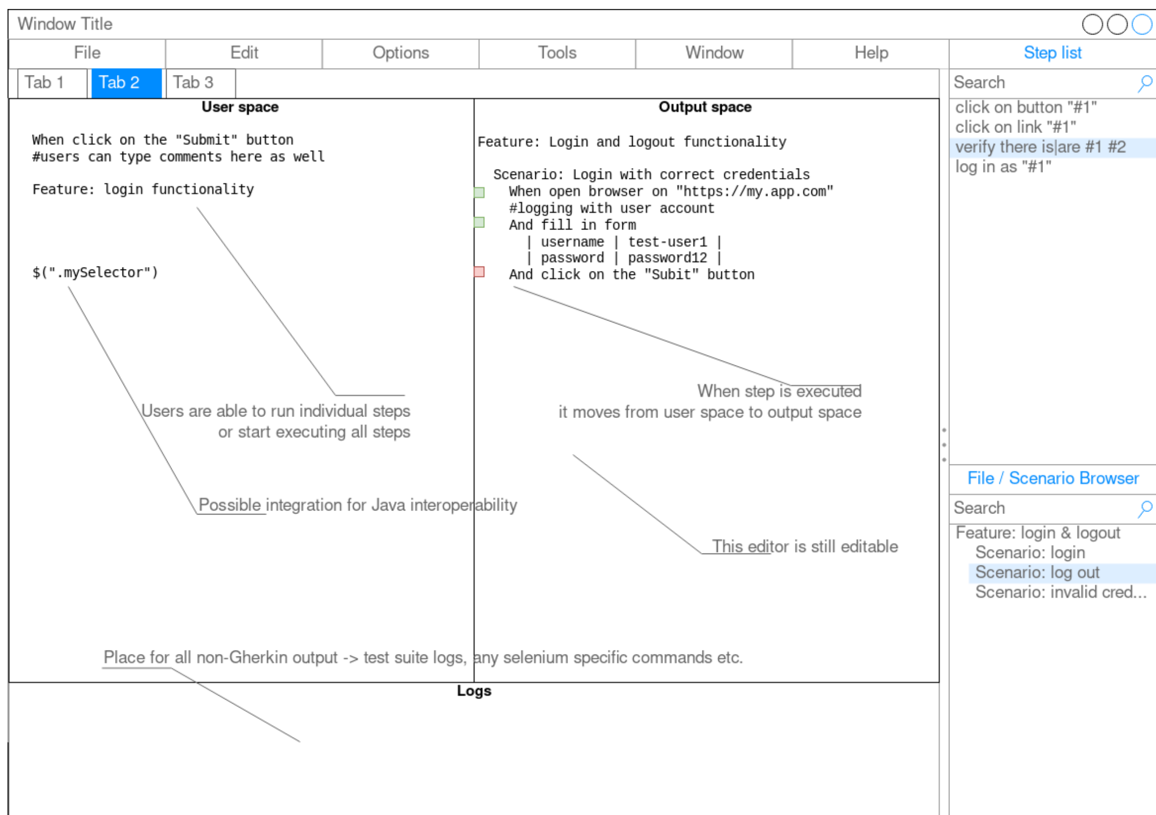
The most important factor in deciding what the final design is going to look like was of course the convenience of the users. So apart from my own thoughts on the design, I asked the users to see if there are any strong feelings about either of those designs.

I collected great feedback for both designs, but to sum it up these were the main takeouts from asking around: The editor design - overall felt better to the users, but everyone asked „How do I edit it? How do I edit and run/not run the step?“, also users wanted to be able to see the resulting feature file, but still see the history of their actions. The shell design - the overall feelings of the users was that it'd be good to do a quick experiment or just use a few steps, but they didn't even expect being able to edit the file, or write comments. It was just too complicated, and there wasn't much space to show users the possible actions they can perform.

Based on this feedback, I looked very hard for already existing solutions, to see how others overcame the issues I am facing and to my surprise I didn't get it all wrong, but I just needed to merge both designs together to resemble typical What You See Is What You Get editor. Which meant dividing the user space into two main areas, the user workplace, where they will write the test scenarios, name their scenarios etc. and the other where they see results of their actions.

Side-by-side editor design

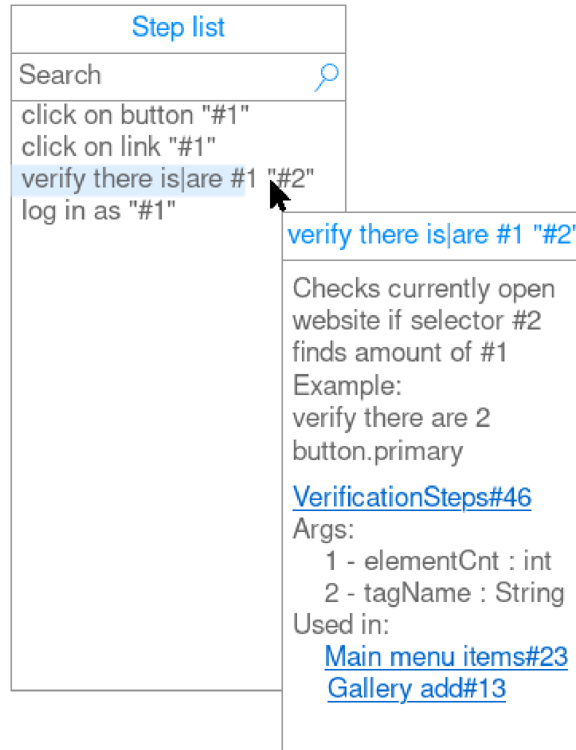
Figure 3.3: Side-by-side editor design mockup



As you can see there are inspirations from both designs, but it's overall neater. Gives a clear separation of where the working focus should be. This also makes developing

the application easier, since there are many open source text editor libraries, that already have their great implementation of syntax highlighting and autocompletion.

Figure 3.4: Context view of a step definition



3.4 Verifying designs

In this section, the issues/improvements are going to be mentioned and what was the design decision to not make it a problem anymore, or at least make handling that problem for testers easier.

Too many step definitions, defined in a bunch of places
The easiest solution is to allow testers to search through all the step definitions and provide autocomplete for when they are writing the test scenarios, so they don't have to remember the exact pattern, just to know when they type a part of it they can find it in the listing. What could improve this design is ability to add a hierarchy to the searching component, maybe allowing users to have a hierarchy like in the classes, or to be able to tag the step definitions, for example have „account" steps although they might not contain the word account. An example can be the **Step list** component in figure 3.3.

Long recoveries from mistakes
As this was the issue that inspired the idea, this was the priority during design process. The application encourages experimentation with different step definitions because they are all available to the user when they find them and if the execution of the step fails, they can simply manually revert any progress, observe the logs and still continue in writing the test scenario.

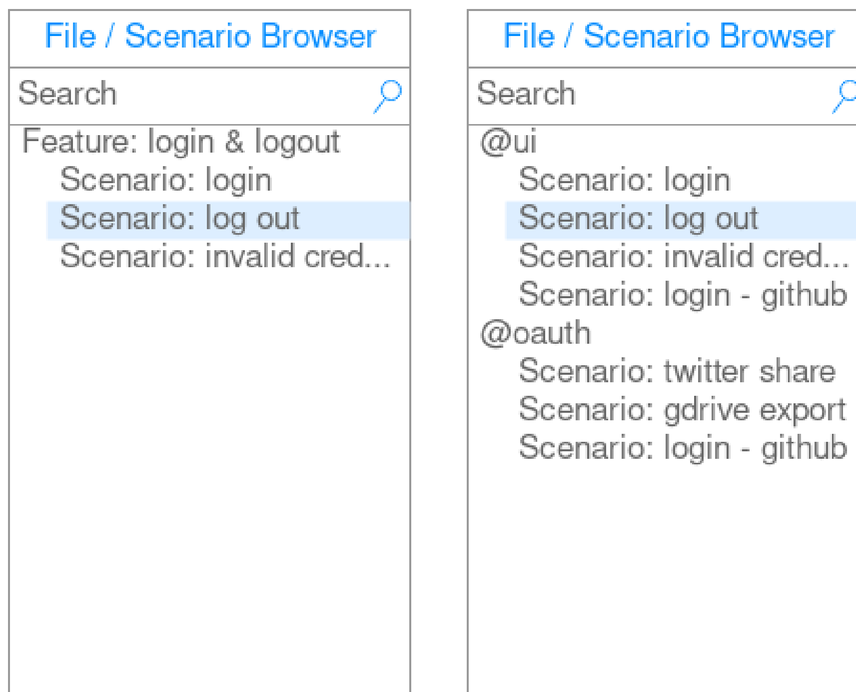
What did they mean in this step definition
Most testers would rather make informed choices while testing which comes with its own set of challenges. A short sentence in form of a step definition is never going to be enough. A solution for this is to provide context when tester is looking at a step definition, a proposed solution is depicted in figure 3.4

This design was presented during the user testing process to all testers, it covers all the the context „clues“ they commonly used to figure out how to use a step definition, most of them looked in already existing scenarios which is covered. Then they resort to code, where they look if any of the used methods are documented, if not they resort to reading the code. All of those methods are covered by the component, and users especially appreciated that documentation of step definition is going to be more visible here than in code and it **would encourage them to write documentation** for their own step definitions.

Does such test exist? What tags do I use?
Due to the test scenarios being stored in a file per feature, it sometimes happen that overlapping scenarios are in one feature file and that leaves room for similar or equivalent scenarios due to testers inability to find it in a given feature file.

Test scenarios and features can be identified and grouped together using tags, the usual workflow is one unique tag per scenario, one unique tag for feature and then tags for common functionality, such as „`@ui`“, „`@oauth`“, etc. When testers were asked about how they use the common tags, none of them had clear answer. They were proposed with design depicted in figure 3.5

Figure 3.5: Scenario/tag explorer widget



The focus of this widget was to hide the notion of files to ease up the navigation in test suite. Scenarios will still remain in separate feature files, but at least while researching testers are not bothered by navigating in files and folders.

The tag exploring widget was seen as a nice to have feature, it would be nice to know what tests are executed with a given tag and to be able to see all defined tags (apart from the unique per scenario ones) to potentially add to a scenario or a feature.

User testing proposal
For verifying the designs in an ideal world a prototype could be made for each mockup and test those prototypes on at least 3 users with different experience in this area. It is important to have a working prototype for testing, because part of the testing is to see how users react to the fact they are working in real time environment. To save time of developers and testers, a decision for the best design can be made and implement working prototype for that design.

When it comes to hands-on testing of the application, several conditions should apply:

- Users should be all of different experience.
- Users should **not** be familiar with the test suite.
- Users should be presented with a more abstract goal, such as „Test login functionality“, to encourage exploration.
- Users should be encouraged to follow the think-aloud protocol, to gain the most feedback from them.
- Users should encounter a case where a step they just wrote didn't success, to observe their reaction.
- Observe where the user is looking for information, how are they finding new step definitions?
- If users show signs of struggle, or being lost, ask them what are they looking for and where would they expect to find it.

A list of recommended questions at the end of the hands-on session:

- Would you adapt this workflow?
- If you are hesitating to adapt - what are you missing?
- Did you feel your job was made easier? Did something interfere in your work?
- How do you write scenarios now? Where do you look for step definitions? Do you look for documentation somewhere?

The user testing plan was pilot tested on 2 user experience engineers and used knowledge from Handbook of Usability Testing [7]. Feedback from users is used throughout this thesis and the main results of this testing are described in 4.3.

3.5 Application architecture

First a short explanation of how the test suite is structured is needed to understand what needs to change.

A standard BDT test suite consists of code and test scenarios. The code provides **step definitions** that are used in scenarios as **steps**. Main difference is that step definition is

like a declaration of a step, it has a pattern and an action. On the other hand step refers to a usage of step definition, like a function call it can have parameters that are defined in the pattern.

When the test suite is started, the code is loaded and step definitions in the code are registered, then the test suite runner determines which scenarios are to be run and then loads said scenarios and runs them. When a scenario is run, it means that line by line a step is read, gets matched against registered step definitions and if a match is found, the action is run; if a match is not found, then that scenario fails.

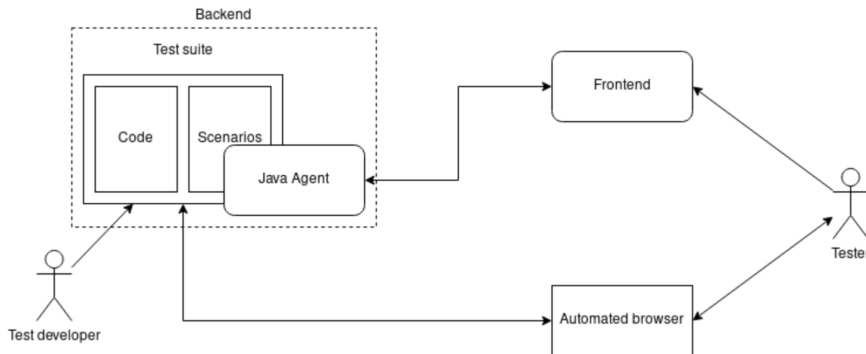
Since the purpose of test runners is to run tests, running steps on demand and making test suite wait for supplied steps is not out of the box functionality of any test framework. This means that either a new test runner needs to be written, or the currently used test runner needs to be modified.

The process of starting and executing tests by the test suite is visualized in figure ??.

Writing a new test runner isn't the best choice, because it required a lot of effort both from person developing this application to testers as there will be 2 test runners which in the worst case could introduce issues such as some dependency incompatibilities and more importantly different runtime behaviors. So introducing a new component was required to change specific parts of the test suite to allow required behavior. I will refer to this component as **the Agent**.

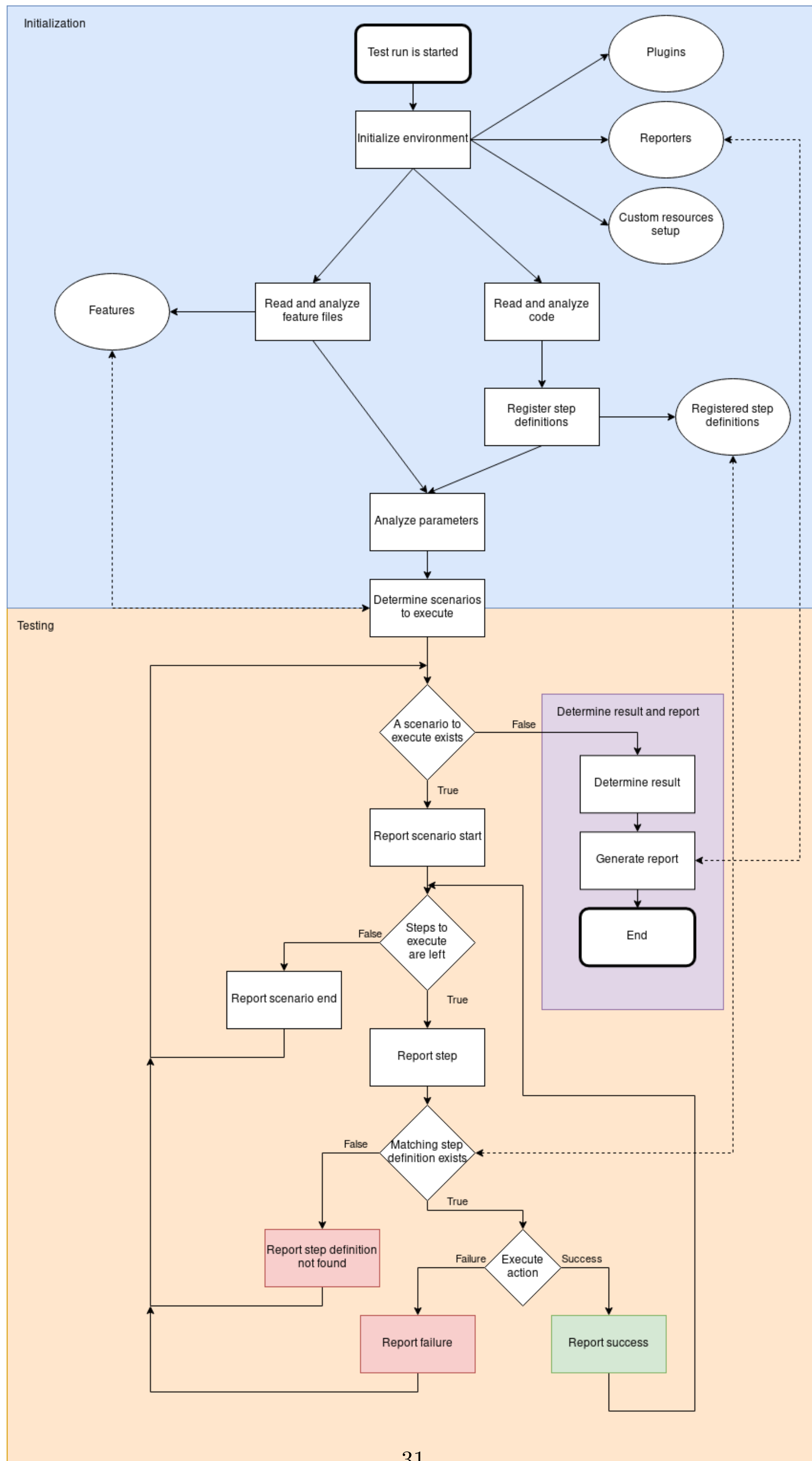
The architecture is visualized in the figure 3.6.

Figure 3.6: Application infrastructure



Together the agent and the test suite makes a back-end which allows for needed functions such as list registered step definitions and run a step. Testers will then use the **front-end** to utilize those functions.

Figure 3.7: Test suite lifecycle



3.6 Backend API

A universal proposal for all test suites (not just Java and Cucumber) is to modify the runner code right before the execution starts and open an HTTP server to allow communication with any tool. HTTP server is recommended for its simplicity, it is easy to communicate with an HTTP server through any language/framework. And specifically in case of Java an HTTP server client and server are already included in the standard library, as using libraries is not as easy an universal as it can be (more about this in Implementation 4).

To be able to provide all functionality to the tool, several things are required from the test suite:

- step definitions - for listing in application and running;
- all features - for opening scenarios, reading the tags, the „used in" context section
- in more advanced cases - type registry

Back-end functionality

The back-end is going to resemble a RESTful service, with following paths, methods and data shapes:

Data shapes

These following data shapes are going to be used in the RESTful service as data types, values ending with ? denote optional value.

```
StepDefinition = {
    id: integer, //< autogenerated id for~the~step definition
    pattern: string, //< regular expression source to match the~step definition
    location?: Location,
    documentation?: string,
    arguments?: Argument[]
}
Location = {
    filename: string, //< source filename
    lineNumber: integer //< source line number
}
Argument = {
    suggestionProvider?: string, //< a~unique name for~a~suggestion provider
    type: string, //< data type of~the~argument
    name?: string
}
CompileStepDefinition = {
    pattern: string, //< regular expression pattern for~the~step definition
    code: string //< code in a~programming language that is executed when this step def
}
CompileResult = {
    stackTrace?: string, //< output of~the~compiler
    failed: boolean //< result of~the~operation,
}
SuggestionRequest = {
```

```
    name: string, //< name of~the~suggestion provider, can be obtained from argument da
    value: string, //< the~step currently written
    position: integer //< which argument is the~suggestion for
}
Logs = {
    stdout: string, //< the~test suite logs to standard output stream
    stderr: string  //< the~test suite logs to standard error stream
}
```

Paths

Paths are used in a REST service to differentiate between the different types or services provided. When a path contains a `{name}` that means the `name` will be used as a parameter.

/steps
Methods:

- `GET : StepDefinition[]` - returns a list of all step definitions registered. If providing list of all step definitions with all values is computationally expensive (for example loading documentation), a list with only required fields is returned.

/step/{id} Methods:

- `GET : id -> StepDefinition` - finds a step definition with the same id as in the parameter and returns a complete `StepDefinition`, *can take longer time to compute.*

- `POST : body=CompileStepDefinition -> StepDefinition | CompileResult` - creates a new step definition in the test suite, if compilation and adding the step definition is successful, 201 : StepDefinition is returned, otherwise 406 : CompileResult is returned.

- `PATCH : id, body=CompileStepDefinition -> StepDefinition | CompileResult` - replaces step definition with id `id` in the test suite, if compilation and replacing the step definition is successful, 201 : StepDefinition is returned, otherwise 406 : CompileResult is returned.

/run Methods:

- `GET : body=string -> string` - body is considered as one step, test suite runs the step as if it was just read from a feature file, executes it and returns the log output in the response with code `200`, if there was an error while executing the step, `400` with the log output in the body is returned.

/logs Methods:

- `GET : Logs` - the outputs kept for sending to the application, the application is polling for any logs, if log is once read, it is cleared. If no new output occurred in a given stream, then empty string is set for that field.

/suggestion Methods:

- `POST : body=SuggestionRequest -> string[]` - invokes the suggestion provider for given name with parameters from body, all suggested values from the `SuggestionProvider` are returned.

/scenarios Methods:

- `GET : string[]` - returns paths of all feature files present in the test suite.

/scenario/ Methods:

- `POST : body=string -> string` - returns contents of a feature file indicated by the path present in the request body.

Suggestion mechanism

The suggestion provider mechanism is meant for step definitions to be able to provide more context while the application is running. It is desired to make it really easy to provide suggestions for a step argument. The most convenient way for developers is to provide an annotation `@Suggestions(<class>)`, the annotation can be used either for a function parameter or as a class annotation which makes all arguments of the type of annotated class use that suggestion provider, if the BDT framework allows to automatically convert step argument to custom data types.

the class used as value of the annotation needs to implement following interface:

```
public interface ISuggestionProvider {

    List<Object> provide(String step);
    List<Object> provide(String step, int arg);

}
```

If context is needed, the `step` variable will have value of the step that is already written, but the value might not match any step definition as it uses fuzzy searching. Finding the right position of the argument is a complex issue due to the pattern being a regular expression. But it is safe to ignore the `arg` function, as by default that function just delegates to the function with just one argument.

Predefined classes such as `EnumSuggestionProvider` can be a part of the agent library if it can be implemented in a generic way.

Communication proposal

Communication is going to follow the RESTful contract described above, an authentication mechanism should be implemented, as when the service provides code execution features, it will be really easy to leak information such as any credential info, execute code on the hosting machine and instruct the controlled browser to access malicious websites.

Client is meant to be polling the server, so more than one connection at the time might occur even when the application is used by only one user.

Front-end proposal

Front-end should reflect the side-by-side design mentioned above in figure 3.3, it is highly recommended to use a code editor library.

The functionality for executing the different inputs inside the input editor are classes implementing the `IService` interface which are all registered inside `Services` singleton class. Each service accepts the whole contents of the input editor as `Model`, this is required because some services might need to read more than one line of text to execute the command correctly. Due to the asynchronous nature of particular services, there must be a messaging mechanism in place for the service to send updates.

The following contract applies for an `IService` implementation:

- Each service has its own messaging channel.
- Each services payload contains the status of execution and optionally data with a data shape previously declared by the service.
- `canHandle(line:string):boolean` - receives a line from the model, returns `true` if this service is capable of handling such input, otherwise returns `false`. No additional validation is implemented in this method.
- `handle(model:Model, lineNum:integer):Status` - starts consuming input from the model on position `lineNum`, after a line is consumed, it should no longer be a part of the model. Returns the status of execution (`SUCCESS, RUNNING, FAILURE`). Any other data or updates should be send through the messaging channel.
- `provideSuggestions(model:Model, position:Position):SuggestionData[]` - provides autocomplete items to the editor, returns empty list if none are or available. The suggestions can range from context-aware to constants such as Cucumber keywords, for added context the `Position` type contains column number. `SuggestionData` is abstract type - depending on the editor library.

The `Services` class is responsible for keeping track of all registered services, finding the right match for an input and delegating the execution. The UI Widgets subscribe to concrete services' messaging channels and update their data and UI accordingly.

Application lifecycle
When the application starts up, first call to back-end should always be getting all registered step definitions. When all step definitions are displayed in the UI, the basic functionality of the editor can work while the application requests details for more information in the background.

An anonymous file is always created at start-up, user is able to start writing steps or other control sequences into the input editor. When an execution command is fired (keybind pressed or a widget is clicked), the application calls its internal service registry to determine which registry is able to satisfy the request. When the request is served, the line is removed and output is written to according place. The place for service output is chosen as: if output belongs to feature file, then it is added into the output editor, otherwise it is added into the logs view.

When the export button is pressed, any not-edited and failed steps are removed from the output editor and a save file dialog is shown and file is saved in selected location.

Possible extensions
Some functionality was left out from the proposal due to its complexity and being dependant on a single technology, because they are not required for testers to be able to use this

application, they will be described here abstractly keeping in mind there probably will be deviations due to specific technologies.

Executing code in the application

This functionality is dependant on the language used by the test suite. Expecting it is possible to execute code from text, it almost a must for an „exposing“ mechanism. When a class/-variable/function are exposed, they are usable in the context of the code, for example: `ScriptExecutor.register(<name>, <reference>)` or `@Expose <Type> <variable>` so users are able to use the code of the test suite for debugging.

Debugging the test suite

If a code execution functionality is in place, debugging can be achieved by making it possible to load a scenario in the application. A breakpoint mechanism, where if the test is currently executing and either there is a breakpoint at the current line or the previous step returned `Failure` status, the execution is stopped and testers are allowed to inspect the application state and execute code if needed.

Chapter 4

Implementation of interactive BDT tool

The main idea of this thesis is to allow testers see currently written tests being executed on-the-fly. Meaning the first implementation step is to create a test runner that allows this. Since no test runners provide this behavior and writing a test runner only for this behavior can cause conflicts in existing test suites and adds complexity, the implemented solution will change existing test runner instead of creating one (further described in Section 4.1).

After the test suite is able to listen to commands and execute testing steps on demand, the user interface will be implemented and gradually tested on users to seek feedback and make improvements (further described in Sections 4.2, 4.3).

This chapter relies heavily on the solution design, short reminder of commonly used words and their meaning:

- test suite - code and test cases together in one project
- scenario - one test case written in Gherkin syntax (readable by everyone)
- agent - program responsible for modifying test suite behavior to suite the application needs
- back-end - test suite modified with agent - ready to communicate with the front-end
- front-end - web application that allows testers to run test parts on demand

4.1 Agent implementation

Creating back-end from the test suite was technically the hardest obstacle of this thesis, as at the time of writing the thesis, to my knowledge, there was not any project to allow users to write their BDT scenarios in real-time. And the information sources were scarce. The process that lead me to implementing the agent is implemented in Appendinx A

Java Agent is defined as a jar file where the MANIFEST file contains either a `Premain-Class` or `Agent-Class` attributes which contain the fully qualified name of a class with a class that has a proper signature specified in the Java Instrumentation documentation: `public static void premain(String arg, Instrumentation instr)`.

Java agents get loaded before any code from the application gets loaded, so it solves the issue of redefining already loaded classes.

Redefining Test runner

In order to provide the wanted behavior, the most important step is to redefine the currently used Test runner class to do all the required initialization, but to stop it from executing tests.

First it is necessary to know what method is in need of modification, if it is desired to replace the code or add code before or after the current code, or in specific cases it could be also required to insert code in a certain section of a method, using local variables, but this is not luckily the case for most cases.

Short version of default runner:

```
public class Cucumber extends ParentRunner<FeatureRunner> {

    public Cucumber(Class clazz) {
        loadStepDefinitions();
        loadScenarios();
        prepareScenariosToRun();
    }

    @Override
    protected Statement childrenInvoker(RunNotifier notifier) {
        return () -> runScenarios();
    }

    private void addChildren(List<CucumberFeature> cucumberFeatures) {
        createRunnersForScenarios(cucumberFeatures);
    }
}
```

At a glance, it is understandable that all the needed initialization happens in the constructor and tests start executing in the `childrenInvoker` method. Now it is only needed to understand what the code does and what is needed for it to do so it serves the purpose of the application.

The `Statement` return type is simply a method that gets called to execute the tests, since it is not required to run any tests, the default behavior can be to simply rewrite this implementation.

How to rewrite implementation?

To make the job easier, I decided to use the Bytebuddy library, with Bytebuddy-agent additional library which allows developers to easily create `ClassTransformers` without the need of direct manipulation of bytecode. It also uses Domain Specific Language to create the Transformers so the code stays readable.

```
public static void premain(String args, Instrumentation instrumentation) {
    new AgentBuilder.Default()
        .with(AgentBuilder.TypeStrategy.Default.REDEFINE)
        .with(AgentBuilder.InjectionStrategy.UsingReflection.INSTANCE)
        .type(ElementMatchers.named("cucumber.api.junit.Cucumber"))
         //.with(AgentBuilder.Listener.StreamWriting.toSystemError())
        .transform((builder, typeDescription, classLoader, module) -> {
            return builder.method(ElementMatchers.named("childrenInvoker"))
```

```
                .intercept(MethodDelegation.to(CucumberInterceptor.class));
        }).installOn(instrumentation);
}
```

Short explanation of the important lines:

- `TypeStrategy.Default.REDEFINE` - means to overwrite the code, `TypeStrategy.Default.REBASE` would create an inner class where the original implementation is kept
- `InjectionStrategy.UsingReflection.INSTANCE` - instruction how to inject newly created class into the classloader, this is the safest and easiest approach, other implementations allow using `Instrumentation` or the `Unsafe` class
- `ElementMatchers.named(„cucumber.api.junit.Cucumber")` - narrowing the classes to be rewritten, other methods of filtering classes exist such as: isDecorated, implements/extends type etc.
- `.with(AgentBuilder.Listener.StreamWriting.toSystemError())` - this is not really a documented feature, but it is almost a necesity to be able to debug Java agents, it logs classnames as they load, provides information whether the classes are transformed and logs useful error messages when an error happens
- `return builder.method(ElementMatchers.named(„childrenInvoker")) .intercept(MethodDelegation.to(CucumberInterceptor.class));` - find method named `childrenInvoker` and delegate the implementation to the `CucumberInterceptor` class

The `CucumberInterceptor` class contains a method with signature:

```
@RuntimeType
public static Statement childrenInvoker(RunNotifier notif, @This Object cucumber);
```

and ByteBuddy is implemented in a way where the best match for a method gets chosen as the intercepted method. It follows rules such as: argument counts and types, return type, name of method. The annotation `@RuntimeType` instructs ByteBuddy to try to cast the argument types while determining which method gets chosen as the interceptor and the `@This Object cucumber` parameters is a mechanism that passes the `this` reference of the original object to the intercepted methods.

Presented information related to Java Instrumentation and Bytebuddy were primarily sourced from an amazing conference talk [9].

In summary the childrenInvoker method does these following things:

- reads values from the `Cucumber` instance such as registered steps,
- register contexts for different server endpoints,
- starts an HTTP server and processes incoming requests.

The method still has to return value, and due to JUnit specific notions, it isn't possible to start the HTTP server in the returned statement, so I ended up returning a primitive closing function:

```
return new Statement() {
    @Override
    public void evaluate() throws Throwable {
        while(!"quit".equalsIgnoreCase(lastMessage)){
            Thread.sleep(1000);
        }
    }
};
```

Also important note is that it is not a good idea to use lambdas in any intercepting code, as lambdas are compiled to inner anonymous classes. Which often leads to issues as the different JVM implementations support this differently[1].

Initialization process
Before the HTTP server is opened it is needed to obtain the values to be provided by the back-end. Because there is a goal not to include any testing library code to avoid conflicts, the values are extracted dynamically. This process can be seen in classes `CucumberInterceptor` and `StepDefProcessor`, where the `StepDefinition` class is analyzed and values are extracted for two different versions of the Cucumber library. For a fully featured implementation, the `StepDefProcessor` should be able to obtain the required values from any object.

Opening the HTTP server
To avoid any more conflicts in the libraries the `com.sun.net.httpserver.HttpServer` was used as the HTTP server implementation because it is already included in the standard library. The HTTP server allows to register different contexts which allows a specific path to invoke a specific code, the contexts are registered in an enum called `Handlers`, where each member of holds an object that implements the `Handler` interface.

```
public interface Handler {

    void handle(HttpExchange exchange) throws IOException;

    default String getPath() {
        return "/" + getClass().getSimpleName()
            .toLowerCase().replace("handler", "");
    }
}
```

The implemented handlers the code provided with this thesis are

- `RUN_STEP(new RunStepHandler())` - runs a step provided in a POST request body
- `LIST_STEPS(new ListStepsHandler())` - provides a list of all step definitions
- `SUGGEST(new SuggestionHandler())` - provides suggestion for a registered provider
- `LOG(new LogHandler())` - provides access to the test suite logs

[1] the author of Bytebuddy himself doesn't advice to use lambdas: https://github.com/raphw/byte-buddy/issues/731#issuecomment-533068046

Allowing developers to provide custom suggestions when someone is using a step definition was high on my priority list because it well presents the benefit of interactive testing. To provide a suggestion for a parameter of a step definition, an annotation `@Suggestion` can be used with a value which holds a class implementing `ISuggestionProvider` interface.

```
public interface ISuggestionProvider {

    List<Object> provide(String step);
    default List<Object> provide(String step, int arg){
        return provide(step);
    }

}
```

Where only the provide method with just the String parameter has to be implemented. The step parameter contains currently written step in the application, if more context is required the arg parameter should contain which argument the suggestion was requested for.

4.2 Front-end implementation

Implementing front-end was more streamlined experience, it was implemented in the React framework using the Typescript language. For quite a long time the desired design to implement was the terminal design (see figure 3.1), but as more functionality was adding and more IDE-like features were adding, it became clear that design has some serious limitations.

Following libraries were used:

- flux[2] - provides functionality for unidirectional dataflow in React;
- react-grid-layout[3] - provides React components to create manipulable grid layouts;
- fuse[4] - fuzzy search library;
- monaco-editor[5] - fully featured code editor library;
- react-monaco-editor[6] - library adding a Monaco editor React component;

The project is bundled via Webpack, along with plugins `html-loader`, `html-webpack-plugin`, `monaco-editor-webpack-plugin`, `source-map-loader`.

Before using monaco library and typescript, I tried to implement a prototype UI just using React, it wasn't the best choice for implementation but I learned a lot about React while trying to implement the terminal design.

During development of the application in Javascript, there were many mistakes while naming props and state variables. Many mistakes were discovered while testing the application that needed to recompile the whole application and test again. Typescript made any

[2] https://facebook.github.io/flux/
[3] https://github.com/STRML/react-grid-layout
[4] https://fusejs.io/
[5] https://microsoft.github.io/monaco-editor/
[6] https://github.com/react-monaco-editor/react-monaco-editor

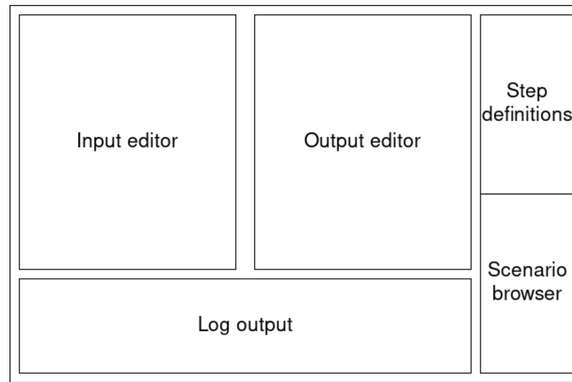


Figure 4.1: Overview of UI components

fails show up faster and the tooling and support in React is even better than when using only Javascript.

The UI consists of a few components show in Figure 4.1:

The most important components are both of the editors, they provide most of the information to the user. The step definitions list allows users to search through the registered step definitions, allowing them to quickly familiarize themselves with the test suite. The logs output is used for reading the logs of the test suite. The Scenario browser should allow users to browse through the defined scenarios, features and tags.

Services

The different interactions are implemented in services and the service functionality is then used in the editor code. Because the services play an important role in the implementation, they need to be documented prior to describing the components.

A services' role is to be able to read a command in the input editor and execute it either by changing the inner application state or by using resources outside the application. A service extends following abstract class and implements the abstract methods.

```
export interface ServiceResult {
    status: ResultType,
    data?: any,
}
abstract class Service<T extends ServiceResult> {
    dispatcher:Dispatcher<T> = new Dispatcher();

    abstract canHandle(line: string): boolean;
    abstract handle(model: Model, from: number):ResultType;
    async provideSuggestions(model:Model,
                            position: Position,
                            context: CompletionContext) : Promise<CompletionItem[]>

    canHandleModel(model: Model, from: number): boolean {
        return this.canHandle(this.peek(model, from));
    }
}
```

Since the command can be executed from any line number, the service reads the editor content from the model. If the `canHandle` method returns true, meaning the command is recognized by this service and the service is able to execute it, the line is consumed and removed from the model (editor content). The command validation takes place in the `handle` method, where if the command is not valid a `ResultType.Failure` is returned. Otherwise if the error contains data or in case of asynchronous communication, the dispatcher is used. Every service has its own dispatcher and the UI components will register to the dispatchers to receive messages with data or changes in a command state.

If a service can provide suggestions to the editor autocomplete, the `provideSuggestions` method is used, the method receives the same parameters as when it is used by the monaco editor, if no suggestions are to be provided a resolved promise with an empty list will be returned.

A special `UnknownOpService` is implemented in case there is no service that can handle a command. All it does is consume the line it was invoked on.

The services existing in the implementation are

- Cucumber service - Communicates with the back-end, sends commands to execute the steps and provides suggestions if suggestion provider is registered.
- Comment service - Moves comments to output adding them to the final scenario
- Variable service - Allows users to set a variable to a certain value, used for naming scenarios and can be used to insert the value of variable into input editor

A legacy implementation of managers is also left in the code. This code was written before using monaco editor, it serves the main purpose of handling the communication with the back-end.

In the `editor` folder most of the implementation relating to the monaco editor resides. A monarch syntax definition had to be created for this project which is included in Appendix B. It is registered for both input and output browser.

Input editor functionality

The functionality of the input editor component is provided by the monaco-editor component with proper actions and commands registered. A Code lens provider and a suggestion provider is registered as well. All the functionality provided in the „editor“ folder in source code deals with interfacing with the monaco editor, the actual functionality is always provided by services.

Output editor functionality

The additional behavior for the output editor is defined in the component file itself. The additional features provided are adding margin decorations to executed steps, providing an export button and setting the scenario names from the variable service.

Step list

The Step list component is a simple list of all step definitions with a search function. The fuse.js library was used for the search functionality to provide results even with misspelled search text or only parts or the text.

Input console

```
Run Step
1  When open browser on "https://google.com"
Run Step
2  And click on link "Search"
```

Figure 4.2: Code lens example

4.3 User testing

In the first user testing round, my main goal was seeing if users understand the concept of the almost WYSIWG approach to the tool. Users were presented with a sample test suite, that they had seen for the first time, mainly to see if just presenting them with the steps is enough, or more context is needed.

I had 2 users with different familiarity of UI Testing to see if different skill sets affect their workflow. Users were presented with one overarching task: „Write a test scenario to perform a search in Google images“, where I presented them with minor tasks which they either did right or struggled with and we had a discussion about why did they struggle and what would help them out.

To sum up the results of the first testing round, I got an overwhelmingly positive feedback about the idea of the tool, both users wanted to adapt this workflow in their day to day jobs. The side by side editor idea did turn out to be fairly easy to grasp, the main issue that both testers expressed was that it was confusing at first to know where they are supposed to write and they would expect the editor to be read only. I also wanted to see how users are going to Submit actions, both were presented with keyboard shortcut `Ctrl+Enter` to submit, but there was also a button and a code lens.

And neither of those users didn't notice the CodeLens (see Figure 4.2) at all which is really good to know as I planned to implement other features by using CodeLens, but it just blends into the code too much.

In conclusion of first user testing, users would still appreciate more context about the steps, and small tweaks to the UI, such as disabling the output editor by default and making the test suite logs easier to read. This will all be addressed by later designs.

Impact on testers

Although the user tests were quite short (1 hour at max), every participant got used to working with this tool really fast and expressed interest into adapting this workflow. Many were sure it will save them significant amount of time (due to time constraints it was not possible to do a longer test).

4.4 Plans for the future

As it was mentioned several times, during the user testing every user expressed interest in this project. As well as other people from other parts of the team other than Quality Engineering, this project is going to be open sourced with a team forming around it.

Due to some outstanding issues it is not easy and convenient to set up with existing test suites due to mismatching library versions. The first task to bring this project to more people and get more contributors is to support most major versions of the Cucumber library.

After fixing this more structural issue the feedback from user testing can be addressed and new features implemented.

In regards to the solution design (Chapter 3) not every feature was implemented. The code execution feature was not implemented due to the lack of any libraries providing this functionality out of the box. The feature files are not read and provided by the agent, as well as the „Used in“ context section. The suggestion provider functionality is not context aware, as I was not able to make and algorithm determining if the caret is in place of an argument, so suggestions are provided as „<number>: <value>“ where number is position of the argument and value is the text to be inserted.

In hindsight most of these issues were caused by the complexity of the Cucumber framework and Java Instrumentation, if this were to be developed in a dynamic scripting language such as Python, the implementation could have been more advanced, but the solution draft should be universal.

Chapter 5

Conclusion

The aim of this thesis was to analyze a specific approach (Behavior Driven Testing) to software verification and to make this approach more user friendly, resulting in faster times of tests development and quicker on-boarding experience for newcomers getting familiar with new test suites.

To create a proposal of the solution I studied the libraries used for Behavior Driven Testing and ways to modify existing code, software verification with focus on Behavior Driven Testing and UI testing. A big resource for designing this environment was a team of quality engineers that use Behavior Driven Testing as a main tool for system testing.

A solution proposal addressed the main hindrances while testing was created after getting feedback from testers. All proposals were thoroughly discussed and reviewed with testers to verify the solution solves the issues. The proposed solution was to create an environment that introduces a fail fast environment and allows to execute tests as they are written, making the process more interactive and feel less like trial and error.

After the significant part of the system was implemented it was tested by users. The users were able to get familiar with a completely new test suite and create a test scenario under an hour. While it usually takes a few days to get familiar with a new test suite and creating a scenario usually takes around 3 hours without any prior preparation. More significantly all users favoured this approach to what they were used to, further proving the point of this thesis.

No specific data can be provided in regards to speeding up the workflow due to the small sample size and the implementation being unable to work with the existing test suite. However, it is safe to say that getting familiar with a test suite was significantly sped up. During the user testing sessions it was clear that biggest speed up comes from being notified about an error and still being able to interact with the application.

To make this approach feature-complete and available to general public there are still issues that need to be sorted out: making this tool compatible with more than one version of the used testing libraries, reading existing scenarios from the test suite and providing them in the UI and adding a debug functionality.

The idea of interactive test writing sparked interest among the quality engineering team and also among teams from different specializations. Software engineers showed the desire to collaborate, to bring their knowledge of the JVM and to make this tool usable for them as well. UX experts are interested in the changes this solution brings to the testing. Front-end engineers are eager to help out with the front-end implementation. This nicely shows what the Fuse agility practitioner said: „BDT is a collaboration and communication tool first, testing framework second.“ This opens up a lot of new opportunities how collaborate

further and allows to embrace the BDT concept of everyone should be able to read the tests and now also write them.

Bibliography

[1] *Cucumber documentation* [online]. [cit. 2020-4-10]. Available at: https://cucumber.io/docs/guides/overview/.

[2] *React JS for professionals* [online]. Goalkicker.com [cit. 2020-05-10]. Available at: https://books.goalkicker.com/ReactJSBook/.

[3] *Selenium documentation* [online]. [cit. 2020-4-20]. Available at: https://www.selenium.dev/documentation/en/.

[4] *What is Software Testing?* [online]. [cit. 2020-4-05]. Available at: https://www.guru99.com/software-testing-introduction-importance.html.

[5] Bruneton, E. *ASM 4.0 A Java bytecode engineering library* [online]. USA: asm.ow2.io, september 2011 [cit. 2020-02-11]. Available at: https://asm.ow2.io/asm4-guide.pdf.

[6] Humble, J. and Farley, D. *Continuous Delivery*. 1st ed. Wiley Publishing, Inc., 2011. ISBN 978–0–321–60191–9.

[7] Rubin, J. and Chisnell, D. *Handbook of Usability Testing*. 2nd ed. Wiley Publishing, Inc., 2008. ISBN 978-0-470-18548-3.

[8] Tim Lindholm, G. B. and Buckley, A. *The Java® Virtual Machine Specification* [online]. USA: Oracle.com, march 2015 [cit. 2020-02-20]. Available at: https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf.

[9] Winterhalter, R. The definitive guide to Java agents. In: JFokus. JFokus, February 2020. Available at: https://www.youtube.com/watch?v=oflzFGONG08.

Appendix A

Backend experiments

Prototype

Before I even started to research Instrumentation, the naive way was to create a new test runner and run the steps from the standard input. Creating a new runner is really simple, it requires to implement just one class, on the other hand any other configuration or setup had to be done for both runners and some other things didn't behave as they did with the original runner.

Also the *Surefire runner* starts the test process in a new process, so there is no direct access to the process input and output streams.

But as a prototype it showed that it is possible with a little effort and some hard coded workarounds it was possible to execute steps on demand. This was especially useful to get familiar the framework structures, I learned what is a test runner and how exactly does Cucumber work. Also realizing that the test runner spawns other processes came crucial in later parts.

Loading compiled classes

My first idea on how to avoid the issues from the prototype phase was to compile the testsuite and load the compiled classes in other process, which would be a way to read all step definitions and run them on demand. This would also resolve the issue of spawning other processes as there would be no test runner, or it would not be the main class that gets run.

This proved be to be even more difficult or even more error prone than the prototype version, as some dependencies were loaded at runtime and the worst part was making Selenium work, as it required many workarounds. Eventually this approach proved to be bad, but it still brought some positives. I learned to read classes and familiarized myself with the class loading process, also it proved well as a basic introduction to Java Bytecode, which was a good enough start for the next phase.

Modifying classes at runtime

Since both of these experiments so far proved that it is hard to setup the testsuite in another process, I started researching Java Instrumentation, which to put is simply is a way to change classes when an application is running.

This brought several benefits:

- No edge cases as in previous try should interrupt this method, if something is done at runtime it should all work as it does when executing tests.
- No need for the end user to know many things about the JVM architecture and require them to pre-compile all of their classes

- Any added dependencies or code at runtime won't have to be handled separately (this was the cause for Selenium in previous experiment)

This wasn't however without issues, the main issue was when the Instrumentation started, the main class of testsuite has this basic structure:

```
public class TestRunner {

    public TestRunner() {
        if (shouldRunInteractiveTool()){
            modifyClasses();
        }
    }

    @BeforeClass
    public void setup() {
        //Used for initialization generally - before any test starts
    }

    @AfterClass
    public void teardown() {
        //Used for cleanup after all tests are finished
    }

}
```

The initial idea was to start modifying the classes inside the `setup` method or in the constructor, however it is not possible to modify already loaded classes. There is a proposal and a prototype[1], but it hasn't made it to any JVM yet.

[1] JEP 159 - https://openjdk.java.net/jeps/159

Appendix B

Gherkin monarch definition

```
{
        defaultToken: ’invalid’,
        symbols: [’"’, "’"],
        tokenizer: {
            root: [
                [/#.*$/, ’comment’],
                [/@[\w\-]*/, ’annotation’],
                [/(?:Feature|Scenario|Background):/, ’keyword’, ’@description’],
                [/(?:Then|When|And|Given|But)/, ’keyword’, ’@step’],
                [/\|/, ’delimiter’, ’@table’],
                [/"""/, ’string’, ’@multilineString’]
            ],
            description: [
                [/.*/, ’identifier’, ’@pop’]
            ],
            table: [
                [/[^\|]/, ’string.table’],
                [/\|\s*$/, ’delimiter’, ’@pop’],
                [/\|/, ’delimiter’],
            ],
            step: [
                [/"[^"]*"$/, ’string’, ’@pop’],
                [/\S$/, ’identifier’, ’@pop’],
                [/\s$/, ’whitespace’, ’@pop’],
                [/"[^"]*"/, ’string’],
                [/\S/, ’identifier’],
                [/\s/, ’whitespace’]
            ],
            multilineString: [
                [/.*"""/, ’string’, ’@pop’],
                [/.*$/, ’string’],
            ]
        }
    }
```