Bachelor Thesis
in the bachelor program
Information Management Automotive
at University of Applied Sciences Neu-Ulm

Development and Evaluation of a Behaviour-Driven Development Testing Concept

1st examiner:	Prof. Dr. Thomas Bauer
2nd examiner:	Meija Lohivina
company contact:	Dr. Maximilian Fürst

Author: Tobias Tögel (Enrolment number: 268560)

Topic received:	07.11.2022
Date of submission:	23.02.2023

Abstract

This thesis develops and evaluates a Behaviour-Driven Development (BDD) testing concept. The goal of this thesis is to answer the research question: Can tests in the style of BDD tests be implemented to test the functionality of existing applications? The resulting artefact is implemented in the environment of an exemplary project at eXXcellent solutions GmbH.

The approach of BDD has the advantages of encouraging collaboration across different roles, establishing a common understanding of what features to implement and creating up-to-date living documentation (Smart, 2015, Chapter 1.3.3). BDD is intended to be used in the development of new features. However, applications with existing functionality could benefit from the advantages of BDD.

To answer the research question, design science research activities were performed to develop and evaluate a BDD testing concept. Expert interviews were conducted to collect the requirements for the concept. The evaluation of the concept includes constant evaluation during the development process and feedback from project experts.

The developed concept consists of a test setup and a test process. The findings of the research activities show that BDD testing can be used to test the functionalities of existing applications. The test setup creates a test environment in which the BDD tests can be executed, and the test process identifies features and test scenarios that can be tested.

Future work in this field could validate and evaluate the concept in a long-term setting and apply the testing concept to other business applications. Further research could also analyse if the concept can be used for an introduction to BDD.

Table of Contents

Abstract II

Table of Contents III

List of Figures V

List of Abbreviations VI

1. Introduction 1

1.1. Motivation 1

1.2. Thesis Overview 2

2. Theoretical Background 3

2.1. Software Testing 3

2.1.1. Motivation 3

2.1.2. Classification 3

2.1.3. Principles 4

2.1.4. Levels 5

2.1.5. Frameworks 7

2.1.6. Process 8

2.1.7. Test Code Quality 9

2.2. Behaviour-Driven Development 10

2.2.1. Motivation 10

2.2.2. Test-Driven Development 10

2.2.3. Overview 11

2.2.4. Identifying Business Goals 12

2.2.5. Describing Features with Examples 12

2.2.6. Creating Executable Specifications 12

2.2.7. Gherkin Syntax 14

2.2.8. Test Implementation with Cucumber 16

2.2.9. Using BDD Tools as Documentation 18

2.2.10. Documenting and Testing Legacy Applications 20

2.3. Literature Review 21

2.4. Research Gap and Research Question 22

3. Concept Development 23

3.1. Research Methods 23

3.1.1. Design Science 23

3.1.2. Expert Interviews 24

3.2. Interview Execution 25

3.3. Interview Summary 29

3.4. Requirements 30

3.5. Test Setup 31

3.6. Test Process 32

4. Concept Implementation 34

4.1. Test Setup 34

4.2. Test Process 35

4.3. CI Integration 39

5. Concept Evaluation 41

5.1. Design Cycle Evaluation 41

5.2. Relevance Cycle Evaluation 42

6. Conclusion 45

6.1. Summary 45

6.2. Limitations 45

6.3. Lessons Learned 46

6.4. Future Work 46

References VII

Appendix X

List of Figures

Figure 1: Global spending on IT services (Gartner, Inc., 2023). 1

Figure 2: Optimal amount of testing (Patton, 2013, Chapter I.3). 4

Figure 3: Testing pyramid (Aniche, 2022, Chapter 1.4.4). 7

Figure 4: Iterative test process (Spillner & Linz, 2021, Chapter 2.3). 8

Figure 5: BDD activities (Smart, 2015, Chapter 1.3.3). 11

Figure 6: High-level and low-level executable specifications. 13

Figure 7: Gherkin feature "Guess The Word" (SmartBear Software, n.d.f). 14

Figure 8: Gherkin scenario outline “eating” (SmartBear Software, n.d.f). 15

Figure 9: Gherkin feature "Is it Friday yet?" (SmartBear Software, n.d.a). 16

Figure 10: Cucumber step definitions "Is it Friday yet?" (SmartBear Software, n.d.a). 17

Figure 11: Cucumber test logs (SmartBear Software, n.d.a). 18

Figure 12: Cucumber test report. 19

Figure 13: A three cycle view of design science research (Hevner, 2007). 24

Figure 14: MIS architecture (eXXcellent Solutions, n.d.). 27

Figure 15: Test setup concept. 31

Figure 16: Test process concept. 32

Figure 17: Setup of the development environment. 34

Figure 18: Download test scenario (Appendix C). 36

Figure 19: Download test scenario flow 38

Figure 20: Azure DevOps CI environment. 40

Figure 21: Non-BDD download test scenario. 43

List of Abbreviations

BDD	Behaviour-Driven Development
CD	Continuous Deployment
CI	Continuous Integration
IDR	Identifier Resolver Service
MIS	Machine Interface Service
OPC UA	Open Platform Communications Unified Architecture
PCS	Process Control System
TDD	Test-Driven Development
VCA	Vision Council of America

1. Introduction

1.1. Motivation

The continuous growth in the IT service sector (Figure 1) illustrates the sheer amount of money spent on developing IT products. There are virtually no devices, machines, or systems that are not partially or entirely controlled by software (Spillner & Linz, 2021).

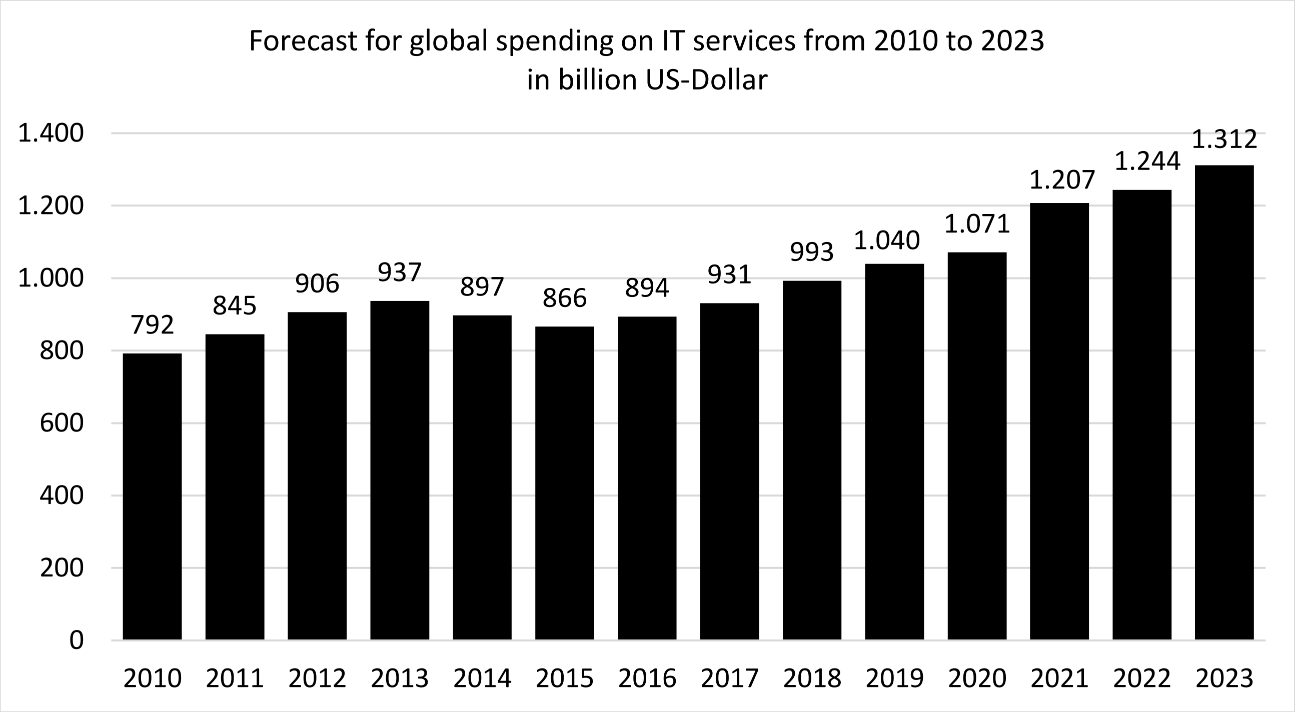


Figure 1: Global spending on IT services (Gartner, Inc., 2023).

In “Software Testing Foundations”, Spillner and Linz argue that the smooth running of countless companies today relies largely on the reliability of software systems that control major processes or individual activities. Quality has therefore become a crucial factor for the success of products and companies in technical and commercial software (Spillner & Linz, 2021). Companies constantly invest in their own development skills and improved system quality. One way to achieve these objectives is to introduce systematic software evaluation and testing procedures (Spillner & Linz, 2021).

There are many ways in which testing activities can be managed and carried out in practice. One approach is called Behaviour-Driven development (BDD). It is a set of software engineering practices that help teams build and deliver more valuable and higher-quality software (Smart, 2015, Chapter 1.3). The goal of BDD is to encourage collaboration across different roles within software development projects and to create a common understanding of what features should be implemented (Smart, 2015, Chapter 1.3.3). The process of BDD ultimately creates executable test scenarios that are easily readable and understandable while also testing the behaviour of the software to ensure that it is working correctly (SmartBear Software, n.d.b).

BDD is intended to be used in the development of new software features (SmartBear Software, n.d.b). The way how BDD works creates many benefits for software projects. Because of this, the question arises if already existing applications can also benefit from BDD tests. This thesis aims to develop a testing concept to test the functionalities of existing applications and to implement this concept for an exemplary software project to analyse the results of the concept implementation.

1.2. Thesis Overview

This thesis is divided into multiple chapters, each addressing a different aspect of the research. The next chapter lays out the theoretical background for developing the testing concept. This includes an overview of software testing theory and an explanation of the BDD testing approach. The background theory is then used to analyse the current state of research in the field and to formulate the research question for this thesis. The third chapter details the research methods and outlines the requirements for the development of the testing concept. Afterwards, the testing concept is developed and introduced. The fourth chapter presents the results of implementing the testing concept within the context of an exemplary project. A comprehensive evaluation of the testing concept is carried out after the implementation. The study's conclusion summarises the main findings and acknowledges its limitations. It also offers suggestions for future research to continue advancing in this field.

2. Theoretical Background

This chapter explains the theoretical background of this thesis. The first section includes the basics of software testing to show the motivation and processes behind testing activities. The subsequent section explains BDD, which serves as a foundation for creating a testing concept based on it.

2.1. Software Testing

2.1.1. Motivation

Software testing is an integral part of the software development lifecycle. Software testing is performed to identify and remedy software failures as well as to fulfil quality requirements (Spillner & Linz, 2021, Chapter 2.1). Similar to other industrially manufactured products, software has to meet certain quality requirements to be successfully delivered (Spillner & Linz, 2021, Chapter 2.1). Failure to fulfil these requirements can even have negative outcomes resulting in high costs and time efforts to fix quality problems as well as damage to the company's reputation (Spillner & Linz, 2021, Chapter 2.1).

2.1.2. Classification

Testing efforts can be divided into static and dynamic testing. Static tests can be performed on any kind of work product that is related to the product's development (Spillner & Linz, 2021, Chapter 4). Static tests can be reviews that analyse system documentation, specification, or system code (Spillner & Linz, 2021, Chapter 4). On the other hand, dynamic tests test the software by running it on a computer (Spillner & Linz, 2021, Chapter 5).

Tests can also be classified as functional and non-functional tests. Functional tests include techniques and methods used to test the test object's observable input and output behaviour and test whether or not the system is working according to its functional requirements (Spillner & Linz, 2021, Chapter 3.5).

Non-functional tests focus on how well a system fulfils its functional behaviour. This includes checking system characteristics like response times, behaviour under load with parallel users, data security, simplicity of use or the use of different configurations like languages or operating systems (Spillner & Linz, 2021, Chapter 3.5.2).

This thesis focuses on creating a dynamic, functional testing concept.

2.1.3. Principles

In order to be effective in the process of software testing, there are several principles that should be considered. These principles help to avoid pitfalls of testing and should give general guidance for the development of the testing concept.

First, testing shows the presence of defects, not their absence (Spillner & Linz, 2021, Chapter 2.1.6). This means that even though there appear to be no failures during testing, it does not mean that the software is free of faults, no matter how much effort was put into the tests. (Spillner & Linz, 2021, Chapter 2.1.6)

Secondly, exhaustive testing is impossible (Aniche, 2022, Chapter 1.3.1). Usually, projects do not have the resources to test every single aspect of the software, even with unlimited resources (Aniche, 2022, Chapter 1.3.1). Complex software would have a very high number of possible test cases. Every single combination of input parameters could lead to a different scenario that could be tested (Patton, 2013, Chapter I.3). That is why it is essential to know what to test and what not to test (Aniche, 2022, Chapter 1.2.6).

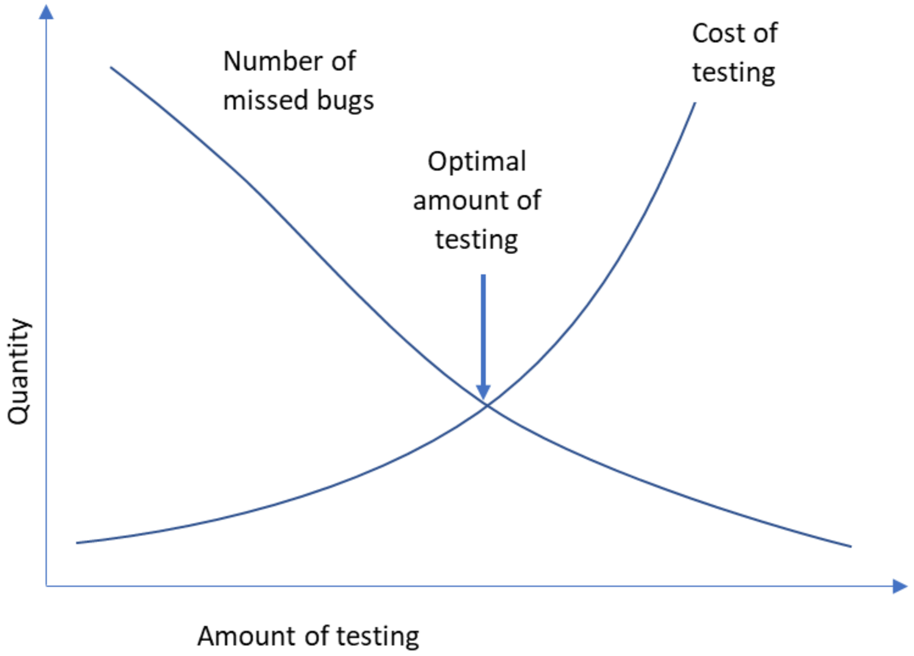


Figure 2: Optimal amount of testing (Patton, 2013, Chapter I.3).

Figure 2 shows that a low amount of testing can lead to a high number of bugs in the system, and on the other side, a high amount of testing results in a low number of bugs but an exponential increase in costs. The optimal amount of testing is in the middle where both curves meet because, on the one side, the costs for testing are not too high, but there are still enough tests to ensure the quality of the software (Patton, 2013, Chapter I.3)

Furthermore, defects happen in some places more than in others (Aniche, 2022, Chapter 1.3.4). Defects are not distributed evenly within the system but can be clustered in certain places. Concentrating on these places can help identify multiple defects. (Spillner & Linz, 2021, Chapter 2.1.6)

Testing is context-depended (Spillner & Linz, 2021, Chapter 2.1.6). Every software has its use and purpose, which is why testing efforts have to be specifically designed with its context in mind (Spillner & Linz, 2021, Chapter 2.1.6).

The pesticide paradox tries to explain that with time, tests become less effective. If tests are repeated on an unchanged system, they will not find new defects (Spillner & Linz, 2021, Chapter 2.1.6). Modifying existing tests or adding new tests can reveal new defects in previously untested parts of the system (Spillner & Linz, 2021, Chapter 2.1.6).

Testing early in the development lifecycle helps find and fix errors early on (Spillner & Linz, 2021, Chapter 2.1.5). In later stages, they might require more effort to be fixed, which leads to higher costs (Spillner & Linz, 2021, Chapter 2.1.6). The specialist knowledge of testers can also help guide the processes of writing requirements for the system and creating the system design, pointing out certain difficulties that can arise during testing phases (Spillner & Linz, 2021, Chapter 2.1.5). Additionally, early collaboration between developers and testers can help in understanding the developed code and in finding appropriate ways to test it. (Spillner & Linz, 2021, Chapter 2.1.5).

2.1.4. Levels

Usually, software systems consist of several subsystems and components that work together. Testing these subsystems and components can be achieved at different levels within the software architecture (Spillner & Linz, 2021, Chapter 3.4).

Unit tests are executed at the lowest level of the system and test single units of the software. A unit can consist of a method, class, or multiple classes that represent a single feature within the system (Aniche, 2022, Chapter 1.4.1). These features are tested in isolation from other parts of the system to make sure that there are no external influences and to identify that defects come from within this unit (Spillner & Linz, 2021, Chapter 3.4). Unit tests are white-box tests which means that the developer has access to the source code and uses their knowledge of the internal structure of the unit to design the tests (Spillner & Linz, 2021, Chapter 3.4.1). On the other side, black-box tests like integration and system tests focus on the input and output behaviour of test objects without knowledge of their internal structure (Spillner & Linz, 2021, Chapter 5.1).

The goal of integration tests is to test multiple units of a system together and to check if they are interacting correctly (Spillner & Linz, 2021, Chapter 3.4.2). Integration tests can be used to test the integration of internal units of the system as well as the integration of external systems. Testing the integration of external systems is called a system integration test (Spillner & Linz, 2021, Chapter 3.4.2). Compared to unit tests, it is more difficult to write integration tests because they require additional effort, like setting up a database and putting it into the right state to test the integration (Aniche, 2022, Chapter 1.4.2).

System tests check if the complete system fulfils its requirements (Spillner & Linz, 2021, Chapter 3.4.3). They are executed from the customer or end-users point of view (Spillner & Linz, 2021, Chapter 3.4.3). This is done by checking user scenarios of the finished system in an environment that resembles the system's production environment as closely as possible (Spillner & Linz, 2021, Chapter 3.4.3).

Each test level has its advantages and disadvantages. It is essential to know which test level to use for what part of the system and how many resources to invest into each testing level (Aniche, 2022, Chapter 1.4.4).

An automated test suite can execute unit, integration, and system tests (Aniche, 2022, Chapter 1.4.6). The advantage of this automation is that the test runs much faster compared to testing everything manually (Patton, 2013, Chapter IV.15). The time that is saved can be used for planning new tests (Patton, 2013, Chapter IV.15). Another advantage is that automated tests are more precise and do not introduce small mistakes that can happen with manual testing (Patton, 2013, Chapter IV.15).

However, it is still recommended to include manual testing efforts in the testing activities. Software testers use manual testing to explore the system they are testing and to find additional things they can test (Aniche, 2022, Chapter 1.4.6).

Introduced by Cohn in Succeeding with Agile, 2009 and adapted by Aniche, 2022, the different test levels are illustrated according to the testing pyramid (Figure 3).

Unit tests are the easiest to create and require the least time to execute (Aniche, 2022, Chapter 1.4.4). Integration tests are more complex to create and require more time to execute (Aniche, 2022, Chapter 1.4.4). At the top of the pyramid are system and manual tests, which are the most complex to create (Aniche, 2022, Chapter 1.4.4). It is impossible to test the entire system at the system or manual test level. This is why critical parts of the system should be identified, and certain tests should be prioritised (Aniche, 2022, Chapter 1.4.6).

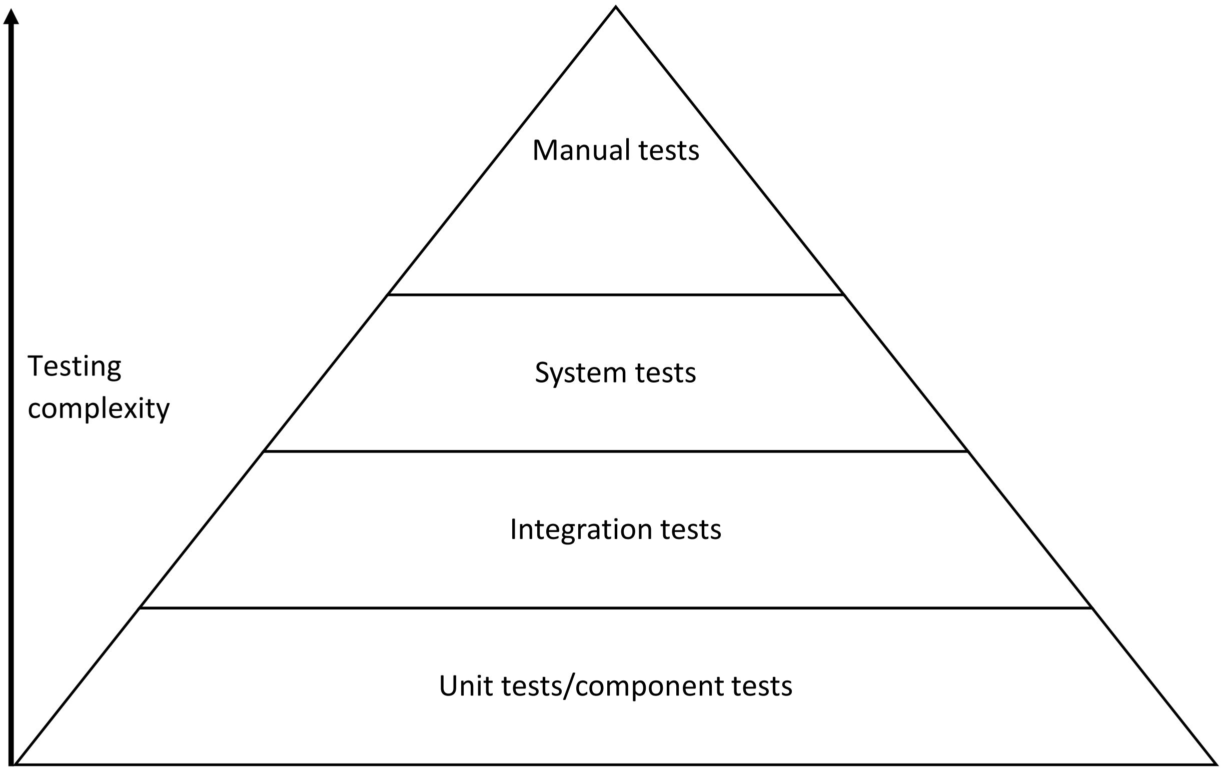


Figure 3: Testing pyramid (Aniche, 2022, Chapter 1.4.4).

2.1.5. Frameworks

As mentioned before, system tests are usually executed in a testing environment that resembles the system's production environment (Spillner & Linz, 2021, Chapter 3.4.3). Lower-level tests, including unit and integration tests, are usually not executable independently (Spillner & Linz, 2021, Chapter 5). They can be created and executed with the help of testing frameworks (Spillner & Linz, 2021, Chapter 5). One example is the JUnit testing framework that uses the Java programming language (Spillner & Linz, 2021, Chapter 7.1.4).

If tests depend on other parts of the system, testers can use mocks and stubs to simulate external dependencies (Aniche, 2022, Chapter 6.1). This preserves the test isolation of unit tests or focuses on the integration of integration tests. Mockito is a popular Java framework that provides the functionality to implement stubs and mocks in the test code (Aniche, 2022, Chapter 6.2). This functionality gives the tester more control over the tests and increases the speed of test executions because tests do not have to wait on processes from other dependencies (Aniche, 2022, Chapter 6).

Stubs provide hard-coded answers to function calls that are performed during the test. They do not have a working implementation (Aniche, 2022, Chapter 6.1.2). Developers can, for example, return a hard-coded list of objects when a stubbed method is called (Aniche, 2022, Chapter 6.1.3).

Mock objects return hard-coded data similar to stubs (Aniche, 2022, Chapter 6.1.4). Mock objects also save interactions that were made with them (Aniche, 2022, Chapter 6.1.4). Afterwards, the tester can verify how often a method was called if, e.g. the correct behaviour is only to call a method once (Aniche, 2022, Chapter 6.1.4).

2.1.6. Process

Besides executing the actual tests on the system, the testing process includes several other activities (Spillner & Linz, 2021, Chapter 2.1). Depending on the development lifecycle, these activities are either completed sequentially after each other with certain overlaps or iteratively (Spillner & Linz, 2021, Chapter 2.3). Software development is often done in an iterative way, meaning that these testing activities take place continuously (Spillner & Linz, 2021, Chapter 2.3). Figure 4 shows an example of an iterative process.

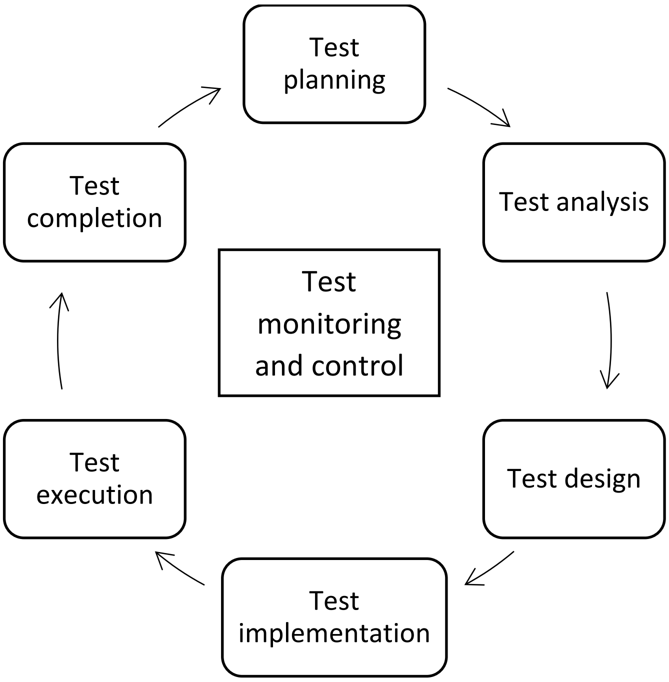


Figure 4: Iterative test process (Spillner & Linz, 2021, Chapter 2.3).

During test planning, testers create a plan that defines test objects, quality characteristics, testing objectives, activities to verify test results, and the time required to perform test activities (Spillner & Linz, 2021, Chapter 2.3.1).

Monitoring and controlling involve the observation of testing activities in comparison to the planned activities (Spillner & Linz, 2021, Chapter 2.3.2). Discrepancies are reported (Spillner & Linz, 2021, Chapter 2.3.2).

Test analysis determines what needs to be tested. The test basis, i.e. the system under test, is examined to identify testable features (Spillner & Linz, 2021, Chapter 2.3.3).

Test design activities determine how the system will be tested (Spillner & Linz, 2021, 2.3.4). This includes creating test cases (Spillner & Linz, 2021, Chapter 2.3.4). Automated test cases are implemented and executed with the help of the testing frameworks (Spillner & Linz, 2021, Chapter 2.3.5). To complete the testing activities, all planned activities should be finished at the end of the project cycle or iteration. Data collected during the test execution is evaluated, and defects are reported (Spillner & Linz, 2021, Chapter 2.3.7).

Iterative development models require every component to have reusable tests that can be repeated for each new increment (Spillner & Linz, 2021, Chapter 3.2). This way, testers have to spend less time repeating the tests when they rerun automatically (Smart, 2015, Chapter 6). Rerunning tests is also called regression testing and verifies that bugs that were found were fixed and no new bugs were introduced (Patton, 2013, Chapter IV.15). Automated tests can be used for every new build in the iterative development process (Spillner & Linz, 2021, Chapter 3.2).

New software versions can be released faster and more reliably when running these automatic tests during continuous integration or continuous deployment processes (Smart, 2015, Chapter 6). In continuous integration (CI), new code changes are followed by automatic building and testing of the system (Smart, 2015, Chapter 6). This way, the project always has a fully integrated and tested system running (Spillner & Linz, 2021, Chapter 3.2). Continuous deployment (CD) extends continuous integration by deploying the system in the production environment after a successful test run (Spillner & Linz, 2021, Chapter 3.2). CI and CD processes require high confidence in the automated test suite (Smart, 2015, Chapter 6).

2.1.7. Test Code Quality

Similar to the production code of a system, testers should put effort into writing high-quality test code (Aniche, 2022, Chapter 10). Best practices for software testing include writing tests that are fast, independent, isolated, repeatable, readable (Aniche, 2022, Chapter 10).

Tests that execute fast give faster feedback to the tester, reducing the waiting time (Aniche, 2022, Chapter 10.1.1). Tests can be accelerated by using stubs and mocks, by testing slower parts of the code separately from fast pieces of code or by executing slower tests less often (Aniche, 2022, Chapter 10.1.1).

Independent and isolated tests only test a single functionality or behaviour of the system with each test case (Aniche, 2022, Chapter 10.1.2). Tests should not depend on the results of other tests, and there should be no difference in the results when running the tests isolated or at the same time with other tests (Aniche, 2022, Chapter 10.1.2).

Repeatable tests have the same result every time they are executed (Aniche, 2022, Chapter 10.1.4). Having different test results without changing the system indicates that there are issues within the system or within the tests (Aniche, 2022, Chapter 10.1.4).

Test code with good readability simplifies the understanding of test code and reduces reading times, especially for other developers (Aniche, 2022, Chapter 10.1.9).

2.2. Behaviour-Driven Development

2.2.1. Motivation

As discussed in the previous chapter, software quality is integral to the software development lifecycle. Applications that are poorly designed or lack well-written and automated tests can be buggy, hard to maintain, hard to change, and hard to scale (Smart, 2015, Chapter 1.2.1). It can also lead to developers having to spend more time fixing bugs rather than working on new features (Smart, 2015, Chapter 1.2.1). However, high-quality software in itself is not enough to guarantee success because the software also has to benefit the users and business stakeholders (Smart, 2015, Chapter 1.2.1). If the goals are not clear, it is easy to deliver functional features that are of little use to the users (Smart, 2015, Chapter 1.2.1). BDD is a set of software engineering practices that help teams build and deliver more valuable and higher-quality software (Smart, 2015, Chapter 1.3).

2.2.2. Test-Driven Development

BDD was introduced in 2006 by Dan North to help teach the agile practice of “Test-Driven Development” (TDD) (North, 2006). The following gives a brief introduction to TDD to understand where BDD originated from:

In TDD, developers implement features by writing failing tests that describe or specify features (Smart, 2015, Chapter 1.3.1). Afterwards, they write the feature implementation to make the tests pass (Smart, 2015, Chapter 10.1). Passing the test means that the functionality has been successfully implemented (Smart, 2015, Chapter 10.1). The developer can then refactor the code for further improvements (Smart, 2015, Chapter 1.3.1). TDD has the advantage that Developers think more about the code they will write before the implementation (Smart, 2015, Chapter 10.1). This tends to result in higher-quality code that is easier to change and maintain (Smart, 2015, Chapter 10.1). However, TDD is not easy to learn, and developers can have difficulty knowing where to start and what tests to write next (Smart, 2015, Chapter 1.3.1). With TDD, developers can also get to detail oriented with specific testing methods and lose focus on the business goals they are trying to implement (Smart, 2015, Chapter 1.3.1).

2.2.3. Overview

BDD tries to eliminate these challenges by focusing on the behaviour of the software (North, 2006) and encouraging collaboration across roles with constant conversations between business analysts, developers, testers, and users (Smart, 2015, Chapter 1.3.3). This should build a common understanding of what features to create and enhance the agile process (Smart, 2015, Chapter 1.3.3; SmartBear Software, n.d.b).

The first step in BDD is identifying business goals and features that will help deliver these goals (Smart, 2015, Chapter 1.3.3). These Features are then illustrated with the help of concrete examples and automated in the form of executable specifications (Smart, 2015, Chapter 1.3.3). Executable specifications validate the software functionality and provide technical and functional documentation (Smart, 2015, Chapter 1.3.3). Figure 5 illustrates the iterative development process of BDD.

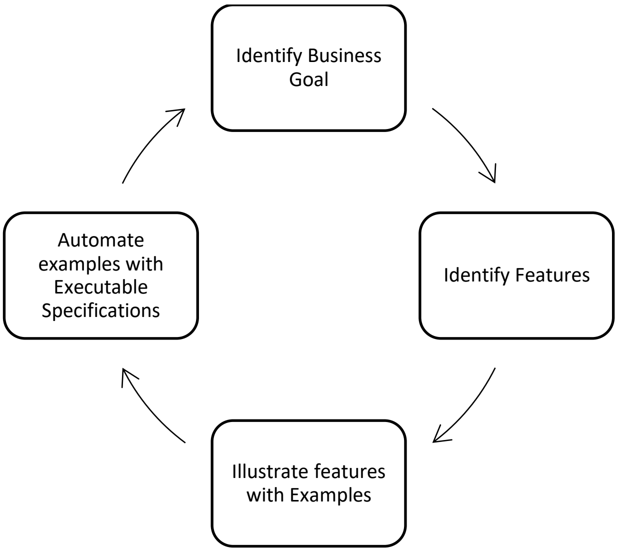

Identify Business Goal
Identify Features
Illustrate features with Examples
Automate examples with Executable Specifications

Figure 5: BDD activities (Smart, 2015, Chapter 1.3.3).

Pereira et al. studied the benefits and challenges of BDD. The study was conducted on BDD-experienced IT professionals. The results show the positive impacts of using BDD. According to the study, BDD helped improve communication and collaboration between roles in the software development project as well as having the advantage of up-to-date documentation. Another study by Nascimento et al. investigated BDD in active learning environments with software engineering students. They also found that BDD leads to better collaboration in the team and a better understanding of the client's needs. Challenges in using BDD included a lack of experience from the developers and the need for a greater commitment from the business and development team.

2.2.4. Identifying Business Goals

To identify the business goal of the software, it can be helpful to answer the questions of why the software is being built, how it will deliver value to the organisation, how the project will affect its stakeholders and what high-level capabilities it should provide to achieve the business goals more effectively (Smart, 2015, Chapter 3). Business goals can usually be grouped into the categories of increasing revenue, reducing costs, protecting revenue, and avoiding future costs (Smart, 2015, Chapter 3.4.2).

2.2.5. Describing Features with Examples

After defining the business goals, the next step in the BDD process is to identify and describe features that help achieve these business goals (Smart, 2015, Chapter 4). A feature is a piece of software functionality that will be valuable for the users (Smart, 2015, Chapter 4.1.1). In BDD, each feature is clarified with the help of concrete examples (Smart, 2015, Chapter 4). The so-called "Three amigos", consisting of a developer, tester, business analyst or product owner, get together and discuss the features trying to find examples (Smart, 2015, Chapter 4.5). This has the advantage that the developer can point out technical considerations for the feature development, the tester can point out certain test scenarios, and the business analyst can judge the relevance and value of scenarios (Smart, 2015, Chapter 4.5). Examples make it easier for everyone to understand a feature, and it helps build a shared understanding of the features that are being implemented (Smart, 2015, Chapter 4.5).

2.2.6. Creating Executable Specifications

Once the examples are found, they can be turned into executable specifications that describe the system's behaviour (Smart, 2015, Chapter 5). Executable specifications consist of different levels of granularity (see Figure 6).

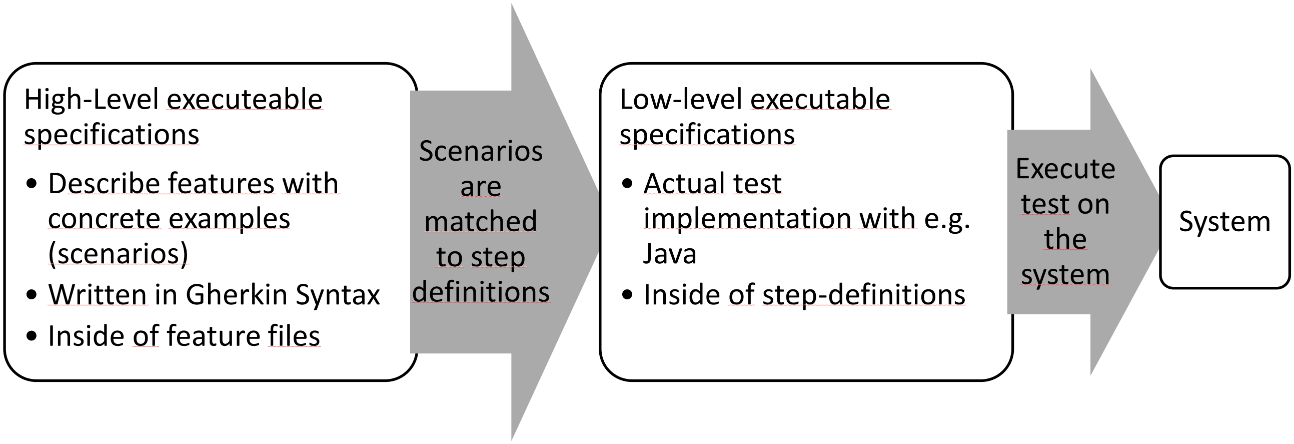

High-Level executeable specifications
• Describe features with concrete examples (scenarios)
• Written in Gherkin Syntax
• Inside of feature files
Scenarios are matched to step definitions
Low-level executable specifications
• Actual test implementation with e.g. Java
• Inside of step-definitions
Execute test on the system
System

Figure 6: High-level and low-level executable specifications.

High-level executable specifications are written in the native language of the stakeholders and structured with the “Given When Then” steps of the Gherkin syntax (Smart, 2015, Chapter 5.1). They are grouped by features inside “feature files” (Smart, 2015, Chapter 5.2). Each feature consists of multiple scenarios that illustrate the system’s behaviour with examples (Smart, 2015, Chapter 5.5). A system with multiple features will have multiple feature files. These specifications can be composed in collaboration by the “three amigos”, which again improves the common understanding (Smart, 2015, Chapter 4.5).

The high-level executable specifications are linked to low-level executable specifications that execute tests on the system (Smart, 2015, Chapter 1.1). The test code is written inside of so-called step definitions (Smart, 2015, Chapter 6). BDD test frameworks like Cucumber read and execute the high-level test scenarios written in the Gherkin syntax and match them to their corresponding step definitions to verify the application’s behaviour (Smart, 2015, Chapter 5.1). Different BDD test frameworks exist for different programming languages like Cucumber for Java, SpecFlow for .NET or Behave for Python (Smart, 2015, Chapter 5).

During the development, automated BDD tests help developers discover the behaviour they need to implement for each feature (Smart, 2015, Chapter 2). Similar to TDD, these tests first do not pass because the behaviour is not yet implemented (Smart, 2015, Chapter 2.4.2). The developer needs to implement the feature and make the tests pass, indicating that the implementation is completed (Smart, 2015, Chapter 2.4.2).

2.2.7. Gherkin Syntax

The Gherkin documentation by SmartBear shows an example of a feature file (Figure 7).

```
Feature: Guess the word

  # The first example has two steps
  Scenario: Maker starts a game
    When the Maker starts a game
    Then the Maker waits for a Breaker to join

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

Figure 7: Gherkin feature "Guess The Word" (SmartBear Software, n.d.f).

Gherkin uses certain keywords to give structure and meaning to the high-level executable specifications (Smart, 2015, Chapter 5.1; SmartBear Software, n.d.f). This includes the following keywords:

- **Feature**: Provides a high-level description of a software feature and groups together scenarios related to the feature (SmartBear Software, n.d.f). The feature keyword must be the first keyword in a Gherkin document, followed by a colon and a short text that describes the feature (SmartBear Software, n.d.f). Free-from text can be added underneath the keyword to add more description (SmartBear Software, n.d.f). Cucumber ignores this during the runtime (SmartBear Software, n.d.f). Descriptions can also be added underneath Scenario, Background, and Scenario Outline keywords (SmartBear Software, n.d.f).
- **Scenario**: Scenario illustrates a concrete example of a feature containing a list of Given, When, Then steps (SmartBear Software, n.d.f). It should summarise the example in a short and declarative sentence (Smart, 2015, Chapter 5.5.2).
- **Given**: Describes the preconditions of the test and puts the application into the correct pre-test state (Smart, 2015, Chapter 5.2.3). This includes creating required test data or, e.g. logging into a web application (Smart, 2015, Chapter 5.2.3). Preconditions should only be directly related to the scenario (Smart, 2015, Chapter 5.2.3).
- **When**: Describes an event or action (SmartBear Software, n.d.f).
- **Then**: Describes an expected outcome or result. It should use an assertion to compare the actual outcome to the expected outcome (SmartBear Software, n.d.f).

- **And/But**: And and But can be used to extend any of the steps of a scenario (Smart, 2015, Chapter 5.2.4).
- **Background**: Given steps that repeat in multiple scenarios of a feature can be grouped under the background keyword before the first scenario (SmartBear Software, n.d.f). In the test execution, it is run before each scenario (SmartBear Software, n.d.f).
- **Scenario outline**: Scenario outline is the same as Scenario but it is used to run the same scenario multiple times with different values from the Examples table (SmartBear Software, n.d.f).
- **Examples**: A scenario outline must contain one or more examples sections (SmartBear Software, n.d.f). The scenario outline is run for each row in the examples section (SmartBear Software, n.d.f):

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

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

Figure 8: Gherkin scenario outline “eating” (SmartBear Software, n.d.f).

In the example of Figure 8, the scenario called “eating” runs two times. Cucumber replaces the parameters in angle brackets with the values from the table (SmartBear Software, n.d.f). The “Given” step creates the initial state of cucumbers. The “When” step represents the action of eating cucumbers, and the “Then” represents the expected outcome. The test passes when the actual outcome of the test is equal to the expected outcome of the “Then” step (SmartBear Software, n.d.f).

Additional Keywords in Gherkin include “Doc Strings” (“””) that can be used to pass data with multiple lines into “Given, When, Then” steps (SmartBear Software, n.d.f). Comments in Gherkin can be written with the # symbol at the start of a new line (SmartBear Software, n.d.f). The | symbol is used to create data tables like in the example above (SmartBear Software, n.d.f).

When writing scenarios, Smart, 2015 suggests not putting too much detail into “Given” steps and only focusing on the business context. Instead of writing, “Given that an admin account is set up in the database and I am logged in as an admin”, write “Given I am logged in as an administrator” (Smart, 2015, Chapter 5.4.1). Another suggestion about writing “When" steps is to describe what it does without details about how it does it (Smart, 2015, Chapter 5.4.2). Instead of writing a “When” step for every input field in a user registration, it can be simplified to “When the user submits their registration” (Smart, 2015, Chapter 5.4.2).

2.2.8. Test Implementation with Cucumber

As mentioned before, high-level executable specifications define how the software should behave with the help of examples (Smart, 2015, Chapter 4). Tools like Cucumber can be used to implement automated tests based on the feature files (Smart, 2015, Chapter 5). Cucumber cannot implement these tests on its own. Tests have to be implemented by a developer (Smart, 2015, Chapter 6.1).

The test implementation is done separately, e.g. with Java code in so-called step definitions. When executing a Gherkin step, Cucumber will look for a matching step definition to execute (SmartBear Software, n.d.i).

Figure 9 shows the connection of a Gherkin scenario with the test implementation in Java. The Gherkin scenario shows the expected system behaviour that is written before the production code implementation:

```
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario Outline: Today is or is not Friday
    Given today is "<day>"
    When I ask whether it's Friday yet
    Then I should be told "<answer>"

  Examples:
    | day            | answer |
    | Friday         | TGIF   |
    | Sunday         | Nope   |
    | anything else! | Nope   |
```

Figure 9: Gherkin feature "Is it Friday yet?" (SmartBear Software, n.d.a).

Step definitions implement executable Cucumber tests written in Java:

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.When;
import io.cucumber.java.en.Then;
import static org.junit.jupiter.api.Assertions.*;

class IsItFriday {
   static String isItFriday(String today) {
      return "Friday".equals(today) ? "TGIF" : "Nope";
   }
}

public class Stepdefs {
   private String today;
   private String actualAnswer;

   @Given("today is {string}")
   public void today_is(String today) {
      this.today = today;
   }

   @When("I ask whether it's Friday yet")
   public void i_ask_whether_it_s_Friday_yet() {
      actualAnswer = IsItFriday.isItFriday(today);
   }

   @Then("I should be told {string}")
   public void i_should_be_told(String expectedAnswer) {
      assertEquals(expectedAnswer, actualAnswer);
   }
}
```

Figure 10: Cucumber step definitions "Is it Friday yet?" (SmartBear Software, n.d.a).

In figure 10, the text inside the “Given, When, Then” annotations match the steps of the Gherkin feature file. This tells Cucumber what method to execute for each step of this scenario. The “Given” step is matched to the “Given” annotation of the step definition, which then executes the today_is() method. This is also the case for the “When” & “Then” steps which execute I_ask_whether_it_is_Friday_yet() as the second and i_should_be_told() as the third method. Gherkin steps can pass values as input parameters to the step definitions, which are received through curly brackets in the annotation and passed to the method’s input.

At first, when the scenario is executed, it fails because there is no test implementation. In that case, Cucumber provides code snippets for each step definition that can be used as a basis for the test implementation (SmartBear Software, n.d.a). Scenarios that fail due to incorrect behaviour of the system can be identified through the Cucumber logs, which would throw errors, e.g. java.lang.AssertionError: expected:<Nope> but was:<null> (SmartBear Software, n.d.a). This assertion error tells the developer that the actual outcome “null” was different from the expected outcome “Nope”.

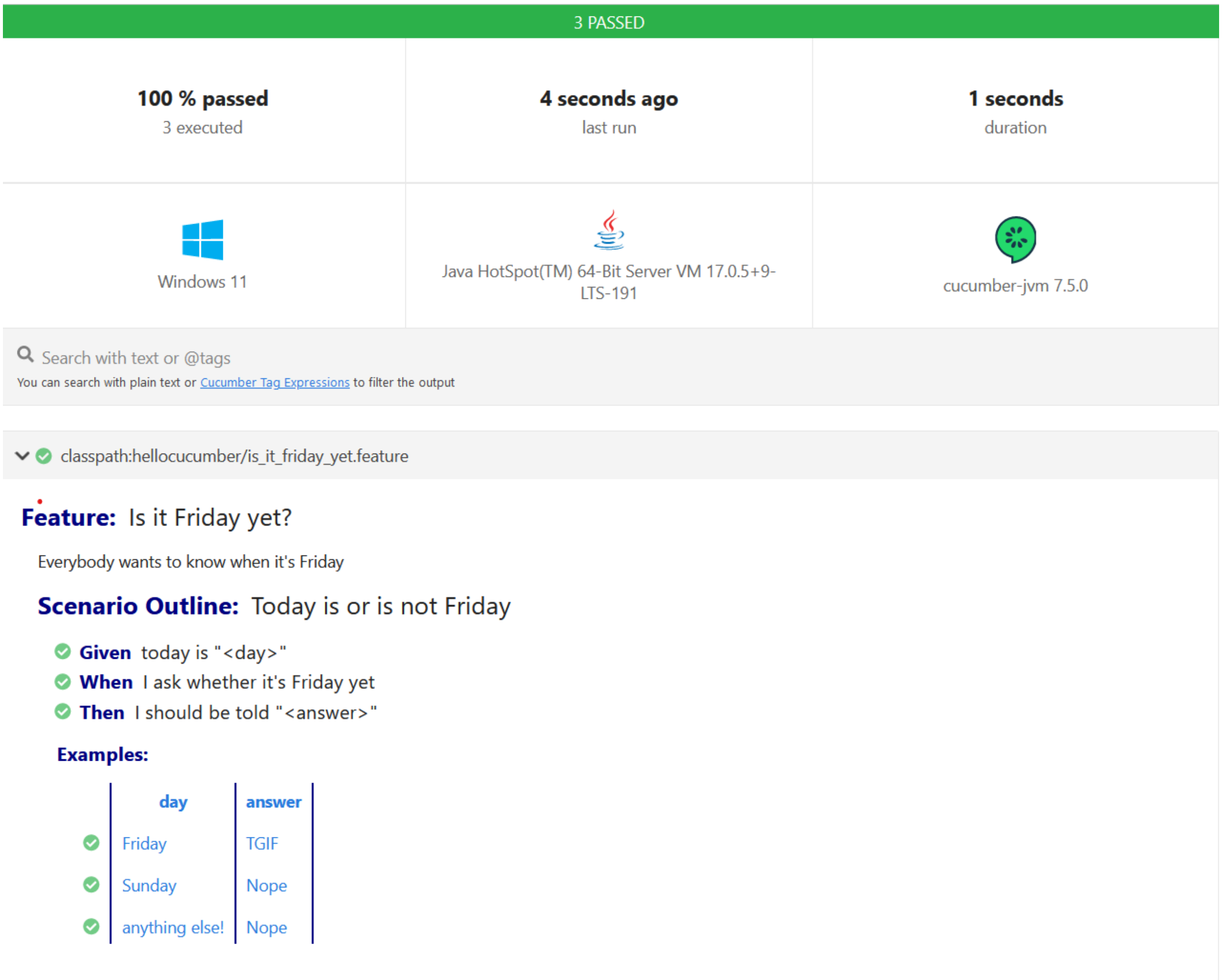
Figure 11 shows the complete logs of successful tests. The scenario is executed three times with different values from each row of the table. Each scenario has three steps resulting in a total of 9 steps being executed:

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running hellocucumber.RunCucumberTest
Feature: Is it Friday yet?
  Everybody wants to know when it's Friday

  Scenario Outline: Today is or is not Friday # hellocucumber/is_it_friday_yet.feature:4
    Given today is "<day>"
    When I ask whether it's Friday yet
    Then I should be told "<answer>"

    Examples:

  Scenario Outline: Today is or is not Friday # hellocucumber/is_it_friday_yet.feature:11
    Given today is "Friday"                 # Stepdefs.today_is(String)
    When I ask whether it's Friday yet       # Stepdefs.i_ask_whether_it_s_Friday_yet()
    Then I should be told "TGIF"             # Stepdefs.i_should_be_told(String)

  Scenario Outline: Today is or is not Friday # hellocucumber/is_it_friday_yet.feature:12
    Given today is "Sunday"                 # Stepdefs.today_is(String)
    When I ask whether it's Friday yet       # Stepdefs.i_ask_whether_it_s_Friday_yet()
    Then I should be told "Nope"             # Stepdefs.i_should_be_told(String)

  Scenario Outline: Today is or is not Friday # hellocucumber/is_it_friday_yet.feature:13
    Given today is "anything else!"         # Stepdefs.today_is(String)
    When I ask whether it's Friday yet       # Stepdefs.i_ask_whether_it_s_Friday_yet()
    Then I should be told "Nope"             # Stepdefs.i_should_be_told(String)

3 Scenarios (3 passed)
9 Steps (9 passed)
0m0.255s
```

Figure 11: Cucumber test logs (SmartBear Software, n.d.a).

2.2.9. Using BDD Tools as Documentation

Besides guiding the developer to build the right behaviour and having automated tests, BDD tools like Cucumber can also be used as functional and technical documentation with the help of test reports in formats like HTML or JSON (SmartBear Software, n.d.h). Figure 12 illustrates a Cucumber report created for the “Is it Friday” feature.



3 PASSED

100 % passed
3 executed

4 seconds ago
last run

1 seconds
duration

Windows 11

Java HotSpot(TM) 64-Bit Server VM 17.0.5+9-LTS-191

cucumber-jvm 7.5.0

Search with text or @tags
You can search with plain text or Cucumber Tag Expressions to filter the output

✓ classpath:hellocucumber/is_it_friday_yet.feature

Feature: Is it Friday yet?

Everybody wants to know when it's Friday

Scenario Outline: Today is or is not Friday

✓ **Given** today is "<day>"
✓ **When** I ask whether it's Friday yet
✓ **Then** I should be told "<answer>"

Examples:

	day	answer
✓	Friday	TGIF
✓	Sunday	Nope
✓	anything else!	Nope

Figure 12: Cucumber test report.

The test report shows features and scenarios that were executed, including the values for each scenario. It can contain multiple features that are expandable and collapsible for better readability. There is a general overview of the percentage of tests that have passed, some details about the system it was executed on, and the execution time. This visualisation serves as documentation about what the system is expected to do and if the system performs these operations correctly (Smart, 2015, Chapter 11.1). This kind of documentation is called living documentation which provides several uses (Smart, 2015, Chapter 11).

Living documentation can be created by running build processes on a CI server using build tools like Maven, Ant or Rake to execute Cucumber tests (SmartBear Software, n.d.c). Cucumber will provide an exit code to the CI server as information on whether tests are failing (SmartBear Software, n.d.c). Running the tests for every build means that the living documentation is always up-to-date, and the developer can react to failing test scenarios (SmartBear Software, n.d.g). Some

CI platforms provide plugins for Cucumber reports for further visualisation (SmartBear Software, n.d.g).

Living documentation creates a fast and efficient feedback loop (Smart, 2015, Chapter 11.1). Stakeholders who participated in the conversation that specified these scenarios can see the test results of the scenarios (Smart, 2015, Chapter 11.1).

Living documentation can be used to display feature readiness (Smart, 2015, Chapter 11.2). Since some stakeholders are more interested in what features are ready to be deployed to the production system rather than what individual tests fail (Smart, 2015, Chapter 11.2).

Lastly, living documentation can help new team members understand the system's behaviour through examples that describe the system's features (Smart, 2015, Chapter 11.1). Additionally, the implementation of unit or integration tests can also serve as low-level technical documentation (Smart, 2015, Chapter 11.6). Developers should be able to read and understand this technical documentation in the form of well-written and readable code (Smart, 2015, Chapter 11.6).

2.2.10. Documenting and Testing Legacy Applications

The previous contents of this chapter showed the BDD process. The iterative process started from the ground up by defining business goals and features, finding examples for the features, and automating these examples with executable specifications. They help developers to discover the implementation of the new behaviour while providing automated tests and living documentation.

In "BDD in action", Smart argues that legacy applications that are still in use may lack technical or functional documentation and sometimes only have a few unit and integration tests (Smart, 2015, Chapter 11.6.2). The lack of automated tests makes it harder and riskier to deliver new features or to fix bugs (Smart, 2015, Chapter 11.6.2). In contrast to using BDD tests to discover the implementation of system behaviour, Smart describes the strategy of adding high-level acceptance tests to the system (Smart, 2015, Chapter 11.6.2). This helps describe and document the behaviour of the existing system and is useful as a regression test to ensure that the system behaves the same way when new features are introduced (Smart, 2015, Chapter 11.6.2). In "The Cucumber for Java Book", Rose proposes a similar approach by adding characterisation tests to test and understand the current behaviour of the system.

2.3. Literature Review

This section gives an overview of the literature research and review that was conducted to find the current state of research related to the topics of this thesis. The IEEE and ACM Digital library databases were searched to find recent studies. The O'Reilly learning platform provided access to computer science literature that was used to build the theoretical background of this thesis.

In "BDD in Action", Smart provides an extensive summary of the complete BDD process that includes the theory of BDD as well as hands-on examples on how to apply BDD in practice. In the foreword of the book, Dan North, the inventor of BDD, calls "BDD in Action" a comprehensive book about BDD and the first time every piece of BDD was put into a cohesive whole (Smart, 2015). This important status was also reflected during the literature research as many other works related to BDD referenced "BDD in Action". "BDD in Action" shows concrete examples that use different BDD testing frameworks but still provides a summary of BDD that can be understood without any specific framework. In general, the BDD process is independent of the testing framework and usually only differs in the test implementation between the different testing frameworks. The difference in the test implementation includes the programming language in which the tests are written and additional functionalities or other peculiarities of the framework.

In an article from 2006 in the software development magazine Better Software, Dan North (North, 2006) first mentions the term BDD and explains his intentions behind creating BDD. He explains the benefits of the BDD approach and gives examples on how to write tests the BDD way with the testing framework JBehave that he created for it. He also introduces other terminology that is used in BDD. This introduction serves as a basis for other works in the field of BDD and the creation of BDD testing frameworks.

Various books detail BDD testing frameworks and the test implementation with their respective programming language. This includes "The Cucumber for Java book". It explains the fundamentals of BDD and the Cucumber framework and gives examples of how to apply BDD with Cucumber tests that are written in Java. Other tool-specific BDD books include "Behavior-Driven Development with Cucumber", "Practical Test Automation: Learn to Use Jasmine, RSpec, and Cucumber Effectively for Your TDD and BDD" or "The rspec Book" (Chelimsky et al., 2010; Lawrence & Rayner, 2019; Matsinopoulos, 2020). The specific tools can be chosen based on the project's requirements and context. These books provide background knowledge and practical examples that help developers understand BDD and their testing framework.

Grey literature, like the official Cucumber documentation, lays out in-depth technical documentation of the framework. It contains documentation on installing Cucumber, setting up a Cucumber testing environment, writing Cucumber tests and other technical documentation related to Cucumber functionalities or the Cucumber test development.

Research papers from Pereira et al., 2018 and Nascimento et al., 2020 studied the benefits and challenges of BDD. Pereira et al., 2018 conducted semi-structured interviews with 24 IT professionals that had practical BDD experience. Based on that, the study's results showed improved communication and collaboration in the project. Additionally, the participants reported the benefits of having living documentation that is always up-to-date. Similar results were found in the study by Nascimento et al., 2020. The study conducted an expert panel with 28 active-learning experts. Results showed an improved collaboration within the project. Additionally, it found challenges for less experienced developers that are unfamiliar with BDD practices and testing culture. The studies were conducted on relatively small groups of people, which is why more studies might be necessary to verify their findings.

2.4. Research Gap and Research Question

The literature research showed that there are many books and other resources that describe the motivations and processes behind BDD and how software can be developed and tested with the help of BDD testing frameworks. These books mainly explain the BDD process by building new applications and new functionalities from scratch.

Rose et al., 2015 & Smart, 2015 shortly touch on testing legacy applications by writing automated BDD tests to benefit from the test automation and living documentation of BDD testing frameworks. However, not only legacy applications but all types of software projects could benefit from this additional test automation and living documentation. The literature research did not find any research or concrete concepts about testing existing software functionality with BDD tests. That is why the following research question and sub-question was derived for this thesis:

Can tests in the style of BDD tests be implemented to test the functionality of existing applications, and what potential does it have? How do these BDD tests compare to other kinds of tests?

This thesis aims to develop and evaluate a testing concept to answer these research questions. The resulting artefact is implemented for a business application at eXXcellent solutions GmbH and serves as an exemplary project.

3. Concept Development

To reach the goal of developing and evaluating a testing concept and to answer the research questions, this chapter identifies the requirements and develops the concept based on these requirements. Before the concept can be developed, the test object and environment must be analysed to get a clear picture of what the concept is developed for and what aspects must be considered in the development. The research methods used in this thesis, namely design science research and expert interviews, were partly used to identify the requirements. This chapter will also explain and summarise these research methods.

3.1. Research Methods

3.1.1. Design Science

The fundamental principle of design science research is that knowledge and understanding of a design problem and its solution are acquired in the building, application, and analysis of an artefact (Recker, 2021, Chapter 5.3). An artefact is defined as something that is constructed by humans, as opposed to occurring naturally (Recker, 2021, Chapter 5.3). In design science as a research activity, the artefacts are created to improve on existing solutions to problems or perhaps provide a first solution to a problem (Recker, 2021, Chapter 5.3).

Figure 13 shows the design science framework by Hevner. At its core, design science focuses on the two research activities of building and evaluating artefacts as part of the design cycle (Recker, 2021, Chapter 5.3). It is important to maintain a balance between efforts spent in constructing and evaluating the evolving design artefact (Hevner, 2007). The result of the design cycle of this thesis is the testing concept that tests functionalities of existing applications. The evaluation of the finished artefact is part of chapter 5.

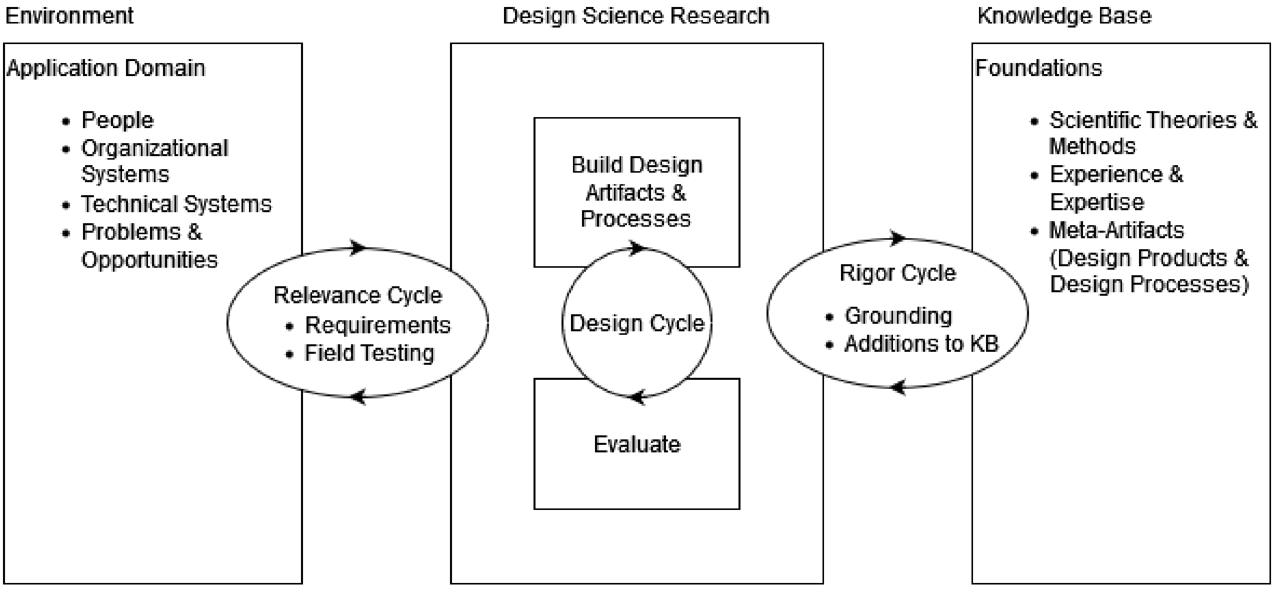


Figure 13: A three cycle view of design science research (Hevner, 2007).

The relevance cycle links design science research to the environment, i.e. the context in which the artefact is developed, including people and organisational systems (Hevner, 2007). It provides the requirements for the research upon which the evaluation is based (Hevner et al., 2004). In addition to the design cycle's evaluation, the design science research output must be returned to the environment for evaluation in the application domain (Hevner, 2007). The environment, as well as the requirements, are identified and analysed in more detail with the help of expert interviews later in this chapter. The finished artefact is evaluated by verifying its functionality and evaluating it in the context of the environment as part of chapter 5.

Design science draws through the rigor cycle from a vast knowledge base of scientific theories and engineering methods that provide the foundations for rigorous design science research (Hevner, 2007). The Literature research from the previous chapter serves as the knowledge base.

As a result of design science research, the rigor cycle contributes extensions to the original theories, new meta-artefacts and experiences from performing the research to the knowledge base (Hevner, 2007). Extensions to the original theories and new meta-artefacts are communicated through the results of this work.

3.1.2. Expert Interviews

Expert interviews were conducted to accurately describe the environment and identify requirements for the design science research. Interviews were done with experts from the project environment to gain valuable insights into the project. The interview is a qualitative research method (Recker, 2021, Chapter 5.2). This is the most suitable research method for this kind of data collection. It is performed between the researcher and key informants whose position in a research setting gives them specialist knowledge about other people, processes, events or phenomena that are relevant to the research (Recker, 2021, Chapter 5.2). Interviews have the advantage of targeting a selected topic, and the interviewer can ask follow-up questions to steer the conversations in the desired area of interest (Recker, 2021, Chapter 5.2). Furthermore, the flexibility of interviews can also be used to explore additional research questions if they arise (Recker, 2021, Chapter 5.2).

In general, there are three different types of interviews. On one side, the structured interview strictly follows pre-planned sets of questions (Recker, 2021, Chapter 5.2). On the other side, unstructured interviews do not use preconceived protocols or sequences and are more like open-ended conversations (Recker, 2021, Chapter 5.2). In between lays the semi-structured interview, which profits from the advantages of both structured and unstructured interviews (Recker, 2021, Chapter 5.2). Questions formulated before the interview guide the interview and serve as a structure (Recker, 2021, Chapter 5.2). Semi-structured interviews are more flexible than structured interviews (Recker, 2021, Chapter 5.2). The researcher can ask new questions during the interview reacting to responses from the interviewee (Recker, 2021, Chapter 5.2).

3.2. Interview Execution

Interviews were performed in the form of semi-structured interviews. They were carried out four times with different technical stakeholders, as the main area of interest were requirements related to technical aspects. The interviews were held as online meetings from which the audio was recorded to simplify the process of writing and summarising the interview protocols[1]. The interview language was chosen to be German, which is the native language of the interviewees and should prevent language barriers between interviewer and interviewee.

Each interview was done with an identical structure (Appendix A) to get answers to the same questions from each stakeholder and potentially see different perspectives. Each interview had certain variations in answers that were given as well as follow-up questions to these answers since semi-structured interviews are flexible and do not follow a strict structure. However, the interview structure helped keep the focus on important questions in case it steered in other directions.

[1] The audio recordings are part of the digital submission of this thesis except for the fourth interview, which could not be recorded but instead was protocolled during the interview (see Appendix B)

Each interview began with a short introduction explaining the goal of the interview and giving necessary background information before the actual questions. The interviews continued with introductory questions that focused on the software development experience, educational background, project-specific experience, and overall goal of the project. These questions were followed by the main questions that were asked to identify requirements and other research-critical or project-specific information.

The interviewees included three software developers and one software architect with educational backgrounds in computer science and mathematics[2]. At the point of the interview, person 2 had 17 years of experience in backend development (Person 2, 2022). Persons 3 and 4 had six years of experience in frontend and backend development (Person 3, 2022; Person 4, 2022). The experience of person 1 amounted to 3 years of experience (Person 1, 2022). Persons 1 and 4 have been part of the project for 2,5 years (Person 1, 2022; Person 4, 2022). Persons 2 and 3 have around 1,25 years of experience in the project (Person 2, 2022; Person 3, 2022).

The project's software is developed by eXXcellent solutions in close collaboration with the customer, who is also part of the project team. The relevant software for this thesis is called machine interface service (MIS). It serves as an interface between physical production machines and the rest of the system landscape that manages and processes data from the production machines (see Figure 14). Depending on which protocol the machine supports, it communicates with MIS either through the Open Platform Communications Unified Architecture (OPC UA) or Vision Council of America (VCA) protocols (Person 2, 2022). The requests are processed by MIS, which sends requests to the rest of the system landscape in the form of REST requests (Person 1, 2022). MIS only processes and does not persist data (Person 1, 2022). This is done by third-party services of the system landscape (Person 1, 2022).

[2] The interviewees are referred to as persons 1, 2, 3 & 4 in no particular order.

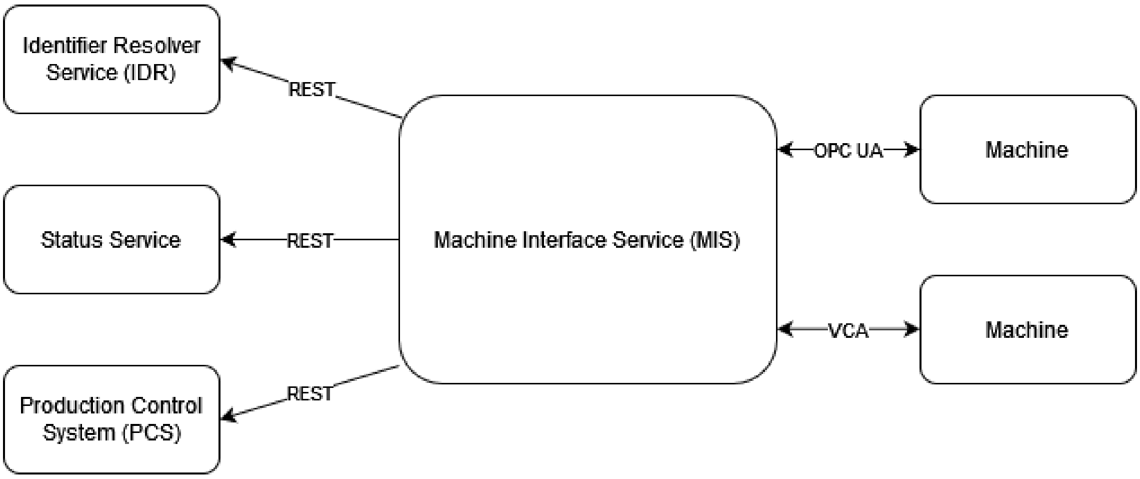

Identifier Resolver Service (IDR)
REST
Status Service
REST
Production Control System (PCS)
REST
Machine Interface Service (MIS)
OPC UA
Machine
VCA
Machine

Figure 14: MIS architecture (eXXcellent Solutions, n.d.).

In the productive environment, MIS is only used by machines and, therefore, does not require any user interface that a person might use. For development and testing purposes, the development team created a command line tool called VCA client that simulates a machine that supports the VCA protocol (Person 2, 2022). Developers can use this command line tool to send requests to MIS, which is running, e.g. on the local machine of the developer (Person 2, 2022).

The machine can send so-called download and upload requests to MIS (Person 1, 2022). In the production process, products pass through various production steps (Person 1, 2022). When the product arrives at a machine, it sends a download request to MIS (Person 1, 2022). MIS then requests certain identifiers of this product from the identifier resolver service (IDR) (eXXcellent Solutions, n.d.). Once MIS receives the identifier for the product, it can validate that the product is in the correct state with a request to the status service (eXXcellent Solutions, n.d.). This is necessary because the product might not be in the correct state, which should prevent the machine from processing these products (eXXcellent Solutions, n.d.). If the product is in the correct state, MIS requests data from the Production Control System (PCS) and returns this data to the machine (eXXcellent Solutions, n.d.). This data contains key-value pairs (labels) that are used as parameters for the production process. (EXXcellent Solutions, n.d.).

After the machine has processed the product, it sends an upload request to MIS, including the data that should be uploaded (Person 1, 2022). Similar to the download process, the MIS once again sends requests to the IDR and status service to validate that it is allowed to upload data (eXXcellent Solutions, n.d.). If it receives the expected responses to allow the upload, it uploads the data to PCS, where it is persisted (eXXcellent Solutions, n.d.).

Additionally, there is a special variant of the download process that behaves differently from the previously mentioned download process. In this variant, MIS uses its internal cache as an alternative data source (eXXcellent Solutions, n.d.). The cache is filled with data that is sent through Apache Kafka (eXXcellent Solutions, n.d.). Afterwards, the download can be triggered, and MIS reads the data from the internal cache without sending requests to the external services (eXXcellent Solutions, n.d.).

The project is not managed in a completely agile or sequential way (Person 2, 2022). Because the customer is part of the project team, requirements for new functionalities directly come from the product owner on the customer's side (Person 2, 2022). Those requirements are divided into work packages that get refined and then implemented by the developers (Person 2, 2022). After completing an implementation, the developer creates a pull request that is then peer-reviewed by another developer (Person 1, 2022). Once the pull request is approved, it gets merged into the main branch and is part of the next release cycle (Person 1, 2022). Each merge triggers a pipeline that executes automated unit and integration tests (Person 1, 2022).

Developers write unit and integration tests as part of the implementation of new features (Person 2, 2022). Unit tests test the functionalities of small units like classes or services (Person 2, 2022). Functionalities are tested in isolation from the rest of the system (Person 2, 2022). Certain parts of the system are mocked if it is impossible to test the unit in complete isolation (Person 2, 2022). Integration tests are used to test the integration level of the system (Person 2, 2022). Integration tests execute broader functionalities of the system, including the integration with external systems IDR, status service and PCS, which are mocked in these test scenarios (Person 3, 2022).

In addition to the automated tests, manual tests are performed before each release to test the system's behaviour, similar to how it is used in the productive environment (Person 2, 2022).

The participants were asked whether they saw potential in using BDD tests with a testing framework like Cucumber in the current project environment[3] (see Appendix A, question 9). The answers showed that they did see some potential, like the aspect of living documentation of BDD tests that document the system's behaviour in a way that is easy to understand and is always up-to-date (Person 1, 2022; Person 2, 2022; Person 3, 2022). This is especially interesting for the project because certain functionalities and scenarios of the system can change and are difficult to

[3] The participants had no previous knowledge of BDD or the cucumber framework. Answers were given based on the brief explanation at the beginning of the interview.

document (Person 1, 2022). Person 1 mentioned that this could help non-technical stakeholders like the product owner to understand better what is currently tested and give input to what scenarios are currently missing, which could be very helpful (Person 1, 2022).

Person 2 also saw potential in writing BDD tests for existing functionality because specifications of the functionalities are often written in the early stages of the project and are not adjusted once there are changes. This makes it harder for future implementations that depend on up-to-date specifications, which could be achieved with BDD tests (Person 2, 2022).

Test automation by the test framework was also mentioned as a potential advantage to keep manual test efforts low and to quickly receive feedback from test runs (Person 2, 2022).

The interviewees were asked which test-level implementation they thought was most suitable for the test concept (see Appendix A). Integration tests were regarded as the most suitable test level for test implementations of BDD tests (Person 1, 2022; Person 2, 2022; Person 4, 2022).

3.3. Interview Summary

With the help of expert interviews, it was possible to gather valuable information about the project environment. The following summarises insights that were gained.

The first insight concerns the relevant behaviour and test scenarios of MIS. Core behaviour that can be tested is the upload and download functionality. A third possible behaviour is the download functionality, where MIS reads from its cache. For each of these functionalities, certain scenarios could be tested to ensure the correct behaviour of the system. Explanations of project experts and internal documentation of MIS can give more details about how MIS is supposed to behave in different scenarios. MIS upload and download processes can be either successful or unsuccessful. In case of success, MIS returns a list of key-value pairs (labels) with the status 0 (eXXcellent Solutions, n.d.). In case it is unsuccessful, it returns other status codes that are defined in the internal documentation.

Secondly, MIS is written in Java and can be built with the maven build tool (Person 2, 2022). When it comes to writing the implementation of the test concept, there is a variety of different BDD frameworks. Choosing the Cucumber framework would make sense because it is available in Java (SmartBear Software, n.d.d). Cucumber tests can be executed either from the command line or with maven, which automatically executes all test scenarios (SmartBear Software, n.d.e). Using the test framework would require a test setup to write test implementations and execute tests on the system.

Furthermore, the most suitable test level for the test concept is most likely the integration test level. This was the opinion of the project experts and is also the most feasible option in the context of this thesis. Integration tests could be performed as black box test that does not require any knowledge of the code or other internal workings of the system, which would require much time to analyse and understand. Triggering and observing system behaviour simply requires knowledge of the system's overall functionality. Integration tests could focus on the integration of MIS with the production machine. It also creates a relatively small number of tests. When writing tests on the unit level, the number of tests might be too high, and they would test low-level functionality which does not reflect the core behaviour of the system.

The next insight relates to third-party systems MIS uses to send and request data during download and upload processes. Persons 1 and 2 argued that the implementation of integration tests could focus on the behaviour of the MIS itself, which means that external services and their responses can be mocked (Person 1, 2022; Person 2, 2022).

Another insight concerns triggering functionality of the MIS. This can be done by using the already existing command line tool known as VCA client, which simulates a machine that supports the VCA protocol (Person 2, 2022). The VCA client would have to be used by the tests to trigger the functionality of MIS.

One potential of the test concept that was identified during the interviews was improved visibility and readability. Cucumber tests can be integrated into CI pipelines (SmartBear Software, n.d.c). By executing the cucumber tests with a CI pipeline, the project team would have a central place to access the test results and the cucumber reports, which serve as living documentation. To have the documentation up to date, the tests, i.e. the CI pipeline, must be executed at regular intervals.

3.4. Requirements

Based on the insights of the project-specific environment, the following are generalised requirements for the development of the BDD testing concept.

Requirement 1: Identify features and scenarios that can be used to test system behaviour.

Requirement 2: Choose a test framework and create a test setup to write the test implementations of the identified features and test scenarios.

Requirement 3: Create tests on the integration level. That way, core functionalities are tested and do not require internal knowledge of software code.

Requirement 4: Create mocks of certain parts of the system. This includes everything that does not belong to the scope of the behaviour that is tested.

Requirement 5: Ensure that the test can execute or trigger the system's behaviour.

Requirement 6: Integrate and execute the tests in a CI pipeline to generate living documentation. Execute the tests at regular intervals to always keep them up to date.

3.5. Test Setup

The following concept for a test setup was developed to answer the research question of whether BDD tests can be implemented to test the functionalities of existing applications. Figure 15 shows the concept outline that satisfies the previously established requirements.

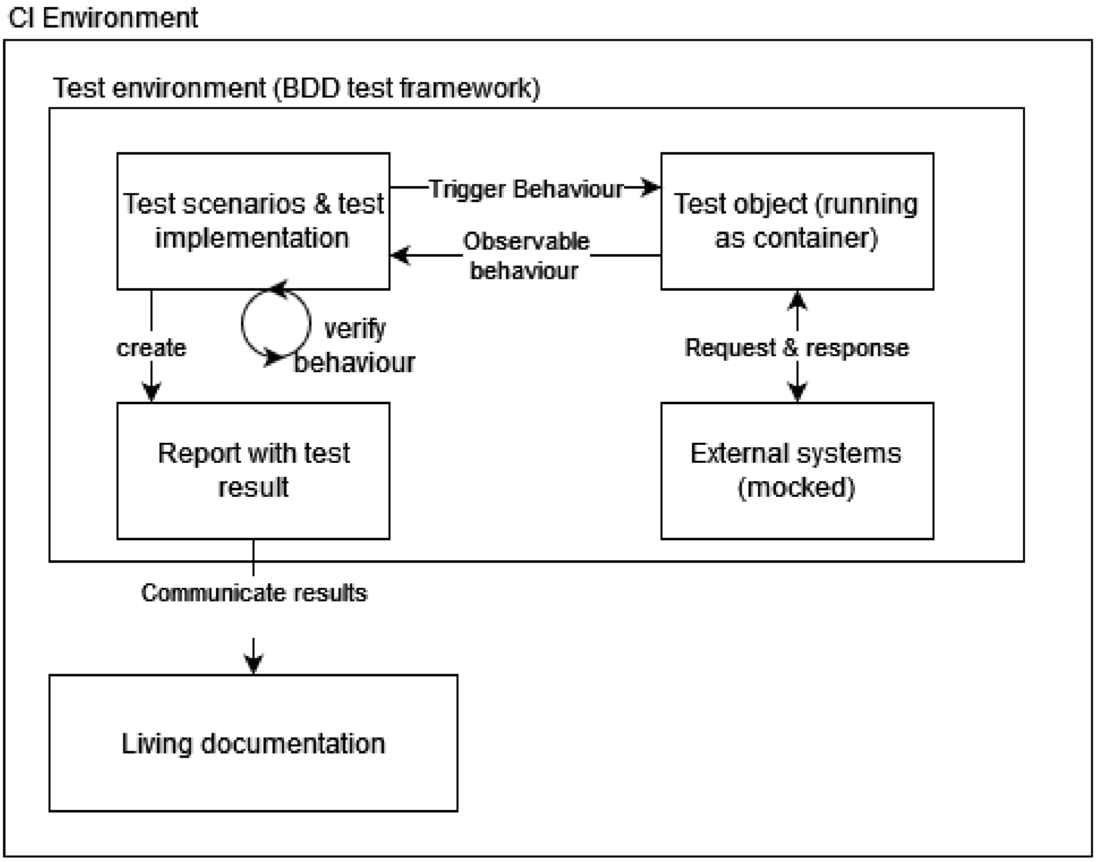


Figure 15: Test setup concept.

The setup is created outside, and independently of the software it is testing. To achieve this, there needs to be a way to run the test object during the test execution. One way this can be done is by running it containerised with the Testcontainers library. Testcontainers runs throwaway instances of anything that can be run in a docker container (Richard North, n.d.).

The setup uses a BDD testing framework which serves as the test environment. The test scenarios and test implementations trigger the test object's behaviour inside the test environment. If external systems exist, the test object communicates with them, receiving mocked responses in return. While the test object runs, it executes its behaviour and the test implementation can observe this behaviour. The test verifies the behaviour and creates a test report that visualises the test outcome. The results are then communicated through the living documentation of BDD tests. The test environment is part of a CI environment where it can be executed, and project members can access living documentation.

3.6. Test Process

Implementing the requirements and the test setup without any systematic process would be inefficient. The original process of BDD can be adapted to allow project teams to iteratively create BDD tests for existing software functionality (Figure 16). The process can be initiated when the project team wants to ensure the correct behaviour of the system. According to the pesticide paradox from chapter 2.1.3, tests become less effective with time and need to be modified or extended with new tests. Adding new tests can uncover bugs that were not found previously. The test concept could also be implemented as a kind of regression test before the team refactors existing code or when it wants to implement new features that could alter the way how the system is currently working.

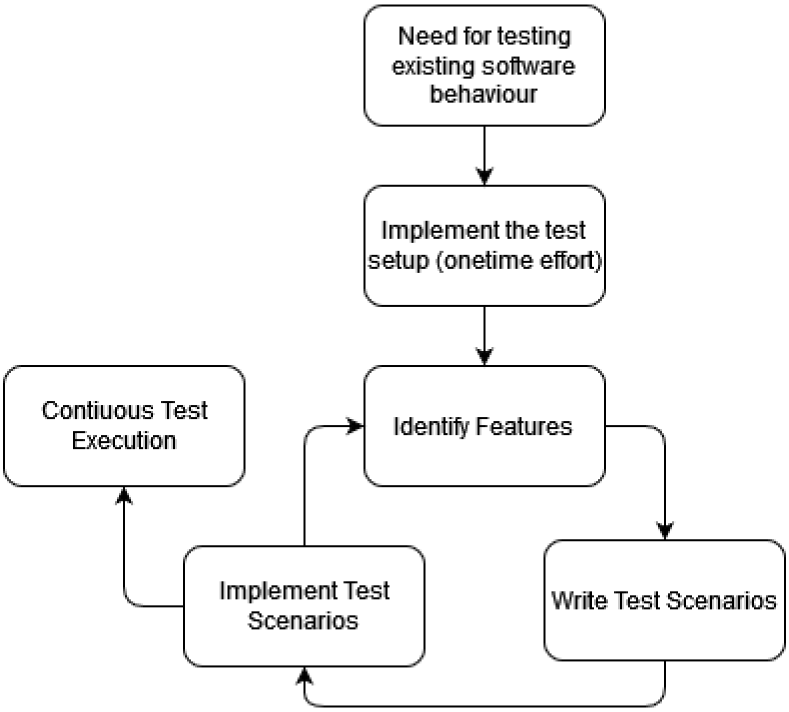

Need for testing existing software behaviour
Implement the test setup (onetime effort)
Contiuous Test Execution
Identify Features
Implement Test Scenarios
Write Test Scenarios

Figure 16: Test process concept.

After the process is initiated, the developer sets up the test environment according to Figure 15. This is a one-time effort that is only necessary to be completed once.

Following the completion of the test setup, the project team identifies features that should be tested. After identifying the features, they are illustrated with test scenarios that show the behaviour of the software with concrete examples. Both steps are part of regular meetings with technical and non-technical project members like business analysts, developers, and testers. This will create a conversation on how a feature is supposed to behave and include different perspectives that project members might have. Just like with BDD, this creates a common understanding that supports the process of defining features and test scenarios.

Afterwards, the developer can start writing the corresponding test implementation for the test scenarios. Developers write the necessary implementation until the tests pass and represent the correct behaviour of the system. Development tasks of these test implementations can be added to the development backlog and completed simultaneously, besides other development tasks that the development team is working on.

Once test scenarios have been implemented, they should be continually executed. Tests can be executed at certain points in the development process, for example, before every new software release. This has the advantage that the team can verify that the release does not create issues with the existing features. Additionally, the test suite can also be executed on demand when the team sees the need for it.

Once the process is finished for one feature, it repeats with the identification of the next feature that needs to be tested to ensure the correct behaviour of the system.

4. Concept Implementation

Following the concepts of the test setup and test process that were established in the previous chapter, this chapter focuses on the implementation[4] of the test concept.

4.1. Test Setup

Applying the test concept in the project environment was started by creating the test setup illustrated in Figure 15. To have all necessary parts of the test setup working together properly, the setup of the test environment was done by sequentially adding the necessary modules and verifying that they work together (Figure 17).

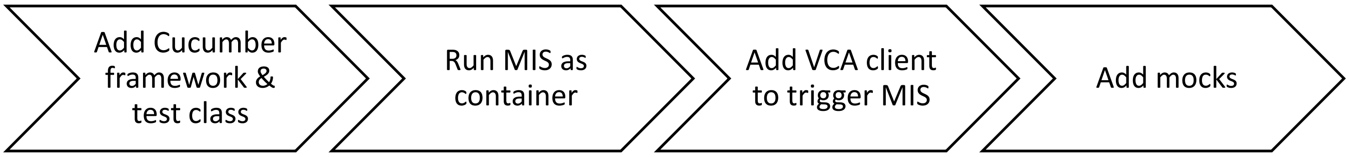

Add Cucumber framework & test class
Run MIS as container
Add VCA client to trigger MIS
Add mocks

Figure 17: Setup of the development environment.

First, the Cucumber framework was added as a maven dependency to the project, and the Cucumber JUnit Class RunCucumberTest.Java (Appendix G) was put into the folder src/test/java. Secondly, a temporary test class was created to test the MIS container start-up. Running MIS as a container is possible because it is deployed to the production environment like that (eXXcellent Solutions, n.d.). Adding and starting the container required setting MIS-specific configuration parameters. Once it ran, the VCA client was added as an external dependency. The VCA client is a command line tool that simulates a production machine and can be used to test MIS behaviour. Once this worked, mocks of external services were created, and it was verified that MIS received and processed these responses as intended. The external services include the services IDR, status service and PCS that were mentioned as part of the expert interviews in chapter 3.2. In the test setup, the API mocking tool WireMock was used to mock the external services.

[4] Test classes and files created during the concept implementation are part of the digitally submitted repository. Relevant files are CommonStepDefinitions.Java, DownloadStepDefinitions.Java, KafkaStepDefinitions.Java, UploadStepDefinitions.Java, mis_download.feature, mis_upload.feature, mis_kafka.feature (Appendix C-F). All other files are either configurations or helper classes provided by Cucumber or the project environment.

4.2. Test Process

The implementation time window took place during five weeks. The test setup required approximately two weeks. Because of limited implementation time and limited availability of team members, the suggested process of defining features and test scenarios was executed at a smaller scale that did not include all necessary participants.

Regular meetings were held to identify and define features and scenarios. These meetings included the developer, who wrote the test implementation and at least one other project member who represented the remaining roles of the project environment. In these meetings, existing features were discussed, and test scenarios with concrete examples were created.

The first iteration defined and implemented tests for MIS's download feature and was followed by repeating the process for the remaining features. To reiterate what features are tested, how the test environment is set up, and how the test execution works, the following briefly summarises these aspects.

Testable features:

- Download feature: Production machines send download requests to MIS before a production step starts. MIS sends requests to external services and receives data that it sends back to the machine. Data contains parameters for the production step.
- Download feature that reads from the internal cache of MIS: MIS cache is filled with data that is sent through Apache Kafka. The production machine sends download requests to MIS before a production step starts. MIS reads the data from the internal cache without sending requests to the external services. MIS sends data back to the machine.
- Upload feature: Production machines send upload requests to the MIS after completing the production step. MIS sends requests to external services and uploads data from the production step to the external service PCS where it is persisted.

Test setup:

The environment of the test setup is created with the Cucumber framework. Cucumber requires "feature files" that contain test scenarios for each feature and test implementations that are written in the form of step definitions inside Java test classes. Inside the test classes, MIS is started containerised with testcontainers and running throughout the test execution. MIS cannot be triggered directly, which is why the test implementation uses the VCA client to trigger the behaviour of MIS. External services IDR, status service and PCS are mocked with WireMock and return responses according to the test scenarios.

Test execution:

The test execution first starts the MIS container. The scenarios then trigger MIS behaviour with the VCA client. Depending on the feature and scenario, MIS sends requests to external systems and returns data to the VCA client. Based on the data that was returned and the behaviour that was observed during the test execution, the system behaviour is verified. A test report is generated to show the result of the tests.

For each feature, a Cucumber feature file was added to the src/test/java/resources/cucumberFeatureFiles folder and filled with test scenarios written according to the Gherkin reference by Cucumber. The file was named to reflect the behaviour, e.g. mis_download.feature.

Figure 18 shows the scenario “Successful download left & right”.

```
Feature: MIS Download
  Background:
    Given all external services are available
  Scenario: Successful download Left & Right
    Given external services respond with:
| service        | http method  | url | response body  | status        |
| idr            | get          | […] | […]            | 200           |
| idr            | get          | […] | […]            | 200           |
| status service| post          | […] | […]            | 200           |
| pcs            | get          | […] | […]            | 200           |
| pcs            | get          | […] | […]            | 200           |
  When a download with jobId "123" is started
  Then actual result should have the same number of entries as expected result
"download_results_left_right.txt"
  And requested labels and values should be equal to expected labels and values
      from "download_results_left_right.txt"
```

Figure 18: Download test scenario (Appendix C).

The scenarios start by setting up external services as mocks inside the “Given” steps. After making the mocks available, they are prepared to send responses when MIS sends requests. Depending on the scenario, mocks return different response bodies or status codes.

The “When” step triggers download or upload processes via the VCA client. “Then” steps differ between upload and download scenarios. In download scenarios, they compare the number and value of key-value pairs (labels) returned by MIS to the VCA client. In the upload scenarios, MIS does not return any key-value pairs (labels) that can be compared. Because of this, “Then” steps verify requests that MIS made to the external services. It includes verifying the number of requests and the contents of the request.

The mis_kafka feature requires an additional “Given” step that sends data to the internal cache of MIS. Triggering and validating the download behaviour of this feature is done identically to the regular download process.

Defining a test scenario in the feature file results in a failing test. To make the test pass, it must be implemented in step definitions of a Java test class. The implementation was done according to the Cucumber reference explained in chapter 2.4.5. Step definition classes were organised inside the src/test/java/de/excellent/stepDefinitions folder. The class was again named to reflect the tested behaviour, e.g. DownloadStepDefinitions.Java.

After implementing tests for the first feature, the process was repeated for the remaining features. Once multiple features were implemented, certain parts of test scenarios and test code could be refactored and improved. Some of the features have identical steps in their scenarios which is why the CommonStepDefinitions.Java class was introduced to reduce code duplication. Cucumber matches the test scenario with its corresponding step definitions, which can be located in multiple Java classes. For instance, “Given” steps of the scenario “Successful download left & right” are inside the class CommonStepDefinitions.Java. “When” and “Then” steps are part of DownloadStepDefinitions.Java.

Figure 19 shows the order in which Cucumber executes the download feature test scenarios.

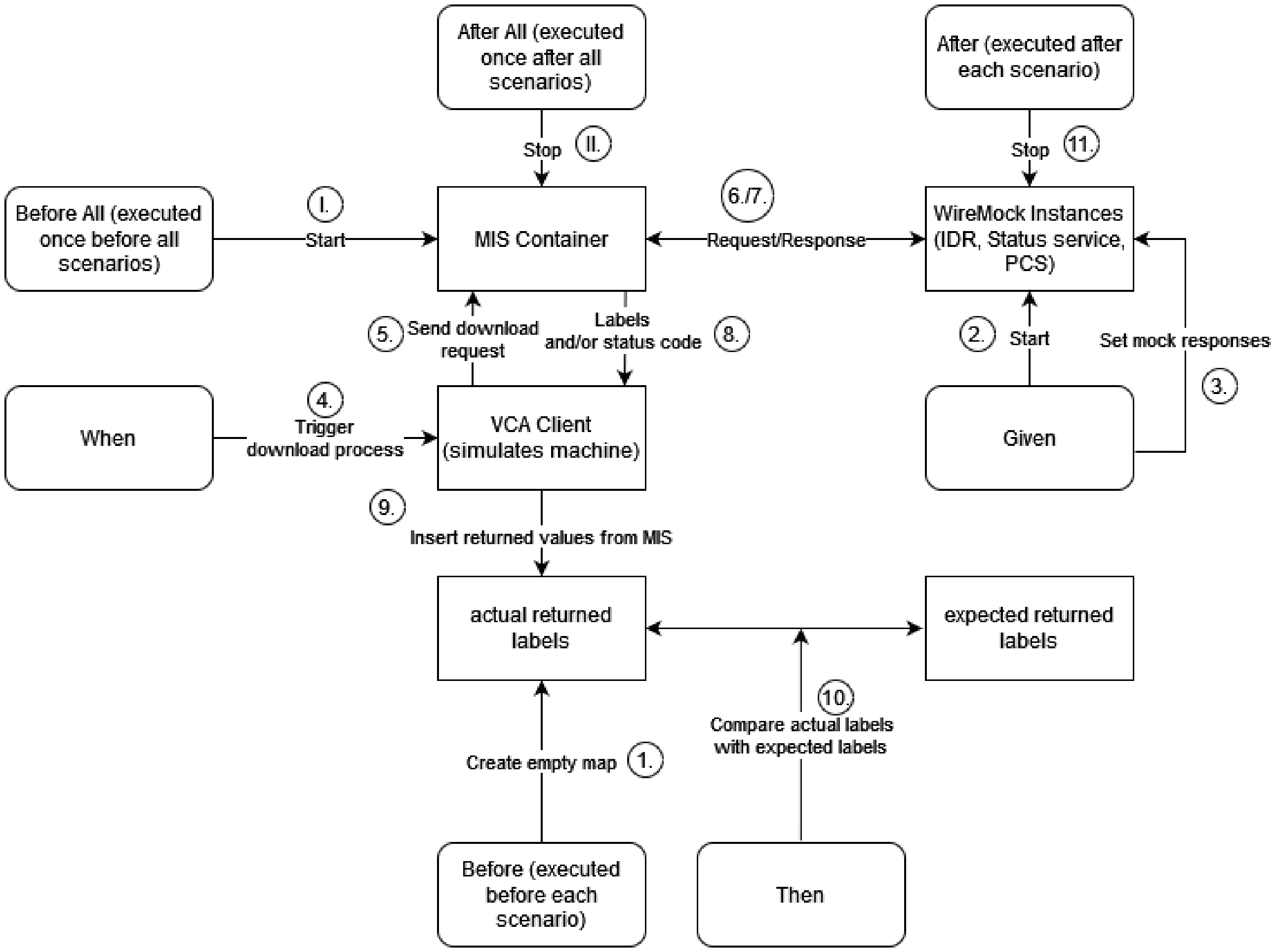

After All (executed once after all scenarios)
Stop
II.
After (executed after each scenario)
Stop
11.
Before All (executed once before all scenarios)
I.
Start
MIS Container
6./7.
Request/Response
WireMock Instances (IDR, Status service, PCS)
5.
Send download request
Labels and/or status code
8.
2.
Start
Set mock responses
3.
When
4.
Trigger download process
VCA Client (simulates machine)
Given
9.
Insert returned values from MIS
actual returned labels
expected returned labels
10.
Compare actual labels with expected labels
Create empty map
1.
Before (executed before each scenario)
Then

Figure 19: Download test scenario flow

The execution begins with the "Before All" lifecycle hook. It starts the MIS container with the method call misTestContainer.start() and is followed by "Before", which creates an empty map. The second step starts the WireMock instances of IDR, Status service, and PCS and sets mock responses. Step 4 triggers the download request of MIS with the VCA client's "run" method and passes the reference of the map: vcaMachine.run(commonStepDefinitions.getReturnedLabels()). MIS sends and receives mock requests in steps 6 and 7. MIS returns key-value pairs (labels) and a status code to the VCA clients, which is inserted into the map that was created in step 1. Cucumber compares the actual test result to the expected test result in step 10 with the following assertions:

```
assertThat(commonStepDefinitions.getReturnedLabels()).hasSameSizeAs(expectedResult)
```

```
assertThat(commonStepDefinitions.getReturnedLabels()).containsAllEntriesOf(expectedResult)
```

The test run is finalised by the lifecycle hooks "After" and "After All", which stop the WireMock instances and the MIS container.

4.3. CI Integration

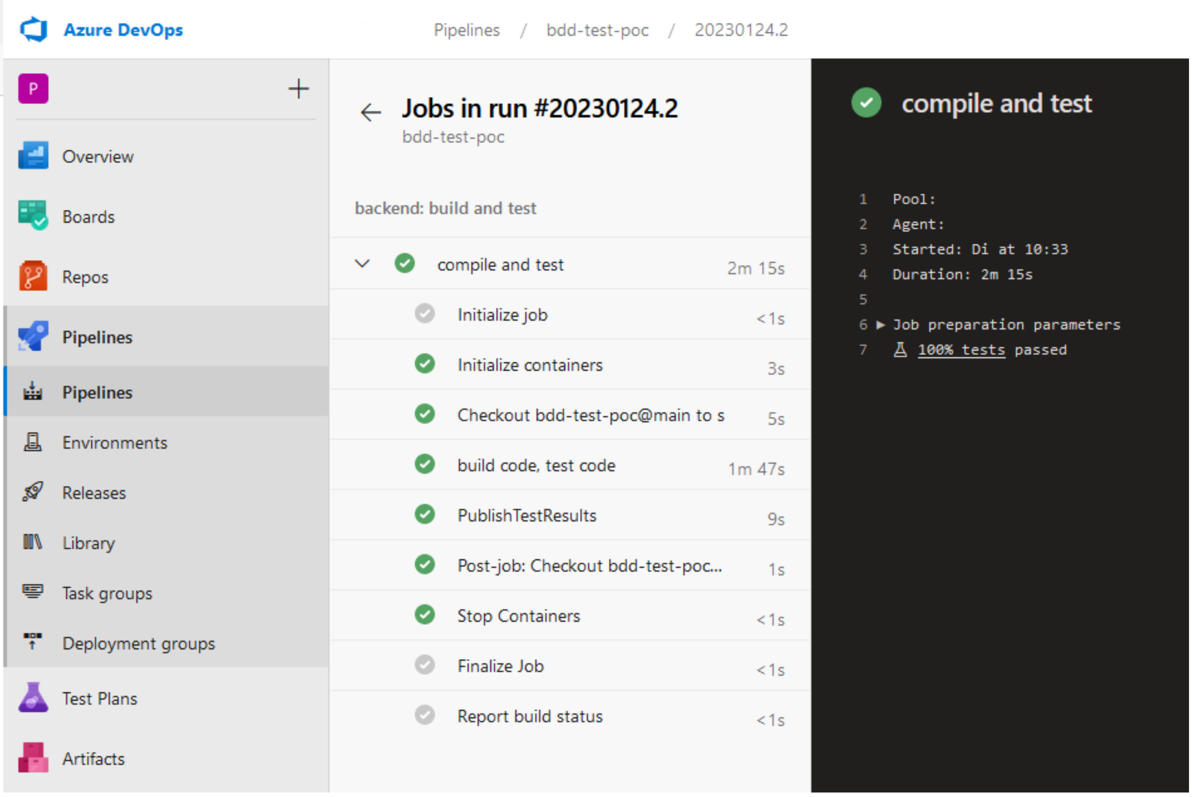
According to Figure 15, the goal is to run the test concept inside a CI environment. The project environment of this thesis uses Azure DevOps for software projects like MIS. Azure DevOps provides CI environments where pipelines can be used to build, test and deliver applications (Microsoft Corporation, 2023b). The project uses YAML pipelines for its applications. YAML pipelines follow a special syntax to customise the pipeline (Microsoft Corporation, 2023c).

Azure DevOps is where the remote repository of the test concept is located. To run the test concept inside a CI pipeline of Azure, the repository needs to contain a .yml file which the pipeline uses to trigger build, test and delivery processes (Microsoft Corporation, 2023a).

A pipeline can comprise one or more stages that describe the CI/CD process (Microsoft Corporation, 2023c). A stage is a way of organising jobs in a pipeline (Microsoft Corporation, 2022). Each stage can have one or more jobs, and each job contains one or more steps (Microsoft Corporation, 2022). Steps can be a task or a script and are the smallest building block of a pipeline (Microsoft Corporation, 2022).

In the case of the test concept implementation, the “bash” step is used to run a script. This script contains maven commands which start the execution of the Cucumber framework.

Figure 20 shows an overview of a successful pipeline run, testing all features and scenarios of the test implementation. In this example, all scenarios are passing. The actual test execution is part of the “build code, test code” step, which took 1 minute and 47 seconds to complete. The following step, “PublishTestResults”, generates a report of the test execution. Steps before and after are automatically created by the Azure pipeline. The CI pipeline can be run before new software releases or on demand by project members.



Jobs in run #20230124.2

bdd-test-poc

backend: build and test

✓ compile and test	2m 15s
Initialize job	<1s
✓ Initialize containers	3s
✓ Checkout bdd-test-poc@main to s	5s
✓ build code, test code	1m 47s
✓ PublishTestResults	9s
✓ Post-job: Checkout bdd-test-poc...	1s
✓ Stop Containers	<1s
Finalize Job	<1s
Report build status	<1s

✓ compile and test

```
Pool:
Agent:
Started: Di at 10:33
Duration: 2m 15s

▶ Job preparation parameters
100% tests passed
```

Figure 20: Azure DevOps CI environment.

Appendix H shows a test report that is automatically generated by Cucumber during the test execution and saved in the target folder of the Java project inside of the CI environment. This report serves as the living documentation. The report shows the test results with the number of successful and unsuccessful test scenarios. It gives an overview of all features that are documented with the help of test scenarios and the corresponding “Given, When, Then” steps that were defined in the feature files. It gives detailed examples for every test scenario that can help project members understand the system’s functionalities. In case of failure, developers can quickly spot what step has failed. The report could be integrated into the CI/CD process and moved by an automation tool from the target folder to a central place where the whole team can access it.

5. Concept Evaluation

The design science research method chosen for this thesis suggests two evaluation cycles. The evaluation of the design cycle takes place constantly by evaluating the design process of the artefact. During the concept development, this was done by constantly comparing the developed concept with the concept requirements and verifying that the requirements were met. As a final evaluation, the finished concept is once more compared with the requirements. The concrete implementation of the concept also serves as proof that the design cycle was successful.

The evaluation of the relevance cycle is where the artefact is returned to the environment and evaluated in the application domain. This was done by presenting the finished concept to project experts of the environment and by asking for feedback that can be used to evaluate the concept.

5.1. Design Cycle Evaluation

Chapter 4 shows the concrete implementation of the concept for the exemplary application MIS that is being developed by eXXcellent solutions in close collaboration with the customer. Overall, the implementation showed that the concept could successfully be used for its purpose. As a more detailed evaluation, the following goes through the requirements of chapter 3.4 and examines whether the individual requirement is fulfilled.

Requirement 1: Identify features and scenarios that can be used to test system behaviour. The concept fulfils this requirement. One step of the suggested test process is identifying and defining features and test scenarios. During the implementation, this was done in conversations with project participants. The process successfully identified different features and test scenarios.

Requirement 2: Choose a test framework and create a test setup to write the test implementations of the identified features and test scenarios. The test environment was set up according to the test setup concept by sequentially adding each module of the test setup. In the end, the finished test setup was working, and test scenarios could be executed.

Requirement 3: Create tests on the integration level. That way, core functionalities are tested and do not require internal knowledge of software code. Creating the test on the integration level had the intended effect of not needing to know the internal code of the system and only focusing on the core behaviour of the system.

Requirement 4: Create mocks of certain parts of the system. This includes everything that does not belong to the scope of the behaviour that is tested. This requirement was fulfilled in the concept implementation by setting up and using mocks of external systems that did not belong to the scope of the tested behaviour.

Requirement 5: Ensure the test can execute or trigger the system's behaviour. Using the test setup and running the test object as a container made it possible to execute and trigger the system's behaviour. In the implementation, this also required adding the additional VCA client dependency to trigger the behaviour.

Requirement 6: Integrate and execute the tests in a CI pipeline to generate living documentation. Execute the tests at regular intervals to always keep them up to date. The implementation shows that the integration and execution of the test concept in a CI pipeline are working successfully. The CI pipeline can be executed regularly to keep the living documentation up-to-date and verify that new features do not introduce bugs to existing features.

5.2. Relevance Cycle Evaluation

The evaluation of the relevance cycle was done by presenting the concept to three project members who have expert knowledge in the project environment. The presentation started with a short introduction of the general concept and continued with a more detailed presentation and demonstration of the finished concept implementation. Based on this, the experts were asked to give feedback on the concept and implementation.

The project experts mentioned that the major benefit of the testing concept is the increased readability and understandability of the cucumber test scenarios (Personal communication, 24.01.2023). Test scenarios are written in an understandable language and illustrate the system's behaviour with concrete examples. The implemented test scenarios include the documentation of requests and responses of the external systems. Previously there was no detailed documentation about this which is now visible at one glance (Personal communication, 24.01.2023).

Figure 21 shows an existing, non-BDD style integration test that tests one scenario of MIS's download feature. From the functional point of view, it is similar to download scenarios of the concept's implementation. Comparing these two types of tests with each other highlights the advantages of BDD-style tests. In Figure 21, it is much harder to read and understand, e.g. how exactly mocks respond or how the test is asserted.

```
void download_right_concave(String locationOfCsv) throws IOException,
ExecutionException, InterruptedException {
        List<CSVRecord> records = getCsvRecords(locationOfCsv);
        addLabelsToServer(getMachine1(), records);
        mockIdentifierResolverService("R");
        mockStatusCheckService(false);
        mockGetVcaValues(records, false);
        establishMachineConnection();

        simulatedMachineSetupService.startDownload(getMachine1(),
"Slot_1", JobIdentifierType.RFID, "123", "R");
        waitForDownload(getMachine1(), "Slot_1");

        assertDownload(getMachine1(), "Slot_1", records, Map.of("STATUS",
"0", "TrayPosition", "R", "_JOBRFID", "123"));
    }
```

Figure 21: Non-BDD download test scenario.

Furthermore, the project experts saw the advantage of integrating the tests in a CI environment (Personal communication, 24.01.2023). With CI integration, the test and the living documentation are located in a central place to which every project member has access to. Everyone can run tests, verify the system's behaviour, and read the living documentation to understand how the system works (Personal communication, 24.01.2023).

On the other side, the project experts questioned whether the advantages will hold true when it is continually used in practice (Personal communication, 24.01.2023). There was the concern of maintainability, practicability, and extendibility. Long-term usage would, for instance, show the practicability of using the tests and if the test scenarios can easily be extended with other features (Personal communication, 24.01.2023). In terms of maintainability, it was questioned whether significant changes in the system would result in extensive refactoring efforts in the test implementation or if the test implementation would not be affected (Personal communication, 24.01.2023).

The project experts also argued that continuing the usage of the concept generally results in additional work and effort by having to implement test scenarios for the existing features. This additional effort decreases time resources previously available for other development tasks (Personal communication, 24.01.2023). Further experience in developing and using the test concept would be necessary to see if the advantages of the concept outweigh the additional efforts.

Another question that is still open is whether the concept is beneficial when using it in other project environments or if it is only practical in the context of the MIS environment.

Lastly, the project experts gave the feedback that it would be interesting to see if the test concept could realise system and unit-level tests (Personal communication, 24.01.2023). Tests on the system level would be interesting because they could cover complete user scenarios of the system. Tests on the unit level would be interesting because they sometimes lack good documentation and it can take some time to understand test scenarios when unit tests have to be changed or fixed. Having unit tests that are better documented and easier to understand could benefit the development process (Personal communication, 24.01.2023).

6. Conclusion

6.1. Summary

In conclusion, this thesis shows the result of design science research activities that developed and evaluated a BDD testing concept. The developed concept was implemented for an exemplary software project to answer the research question of whether BDD tests can be used to test the functionalities of existing applications. The concept development was based on the requirements of the system environment of the exemplary project at eXXcellent solutions. The evaluation was performed by implementing the concept for the exemplary project and verifying if the requirements were fulfilled. Additionally, feedback from project experts was used to evaluate the artefact in the application domain. The concept was developed so that it can generally be applied to other types of existing business applications.

The findings of the research activities show that BDD test can be used to test the functionalities of existing applications. The concept that was developed consists of a test setup and test process. With the test setup, it was possible to create a test environment in which the BDD tests could be executed and test the behaviour of the test object. By following the test process that adapts the already existing BDD process, it was possible to identify features and test scenarios and write executable tests. Combining these two artefacts and implementing them in the project environment of the MIS system showed the positive result of the research question.

6.2. Limitations

There are several limitations in the context of this thesis. The concept was developed based on the requirements that were identified in the environment of the MIS system. Even though it was formulated in a generalised way that could be transferred to other project environments, it is tailored closely to the MIS's requirements. A limiting factor, therefore, is that MIS was the only test object and has not yet been tested with other test objects. As mentioned in chapter 2.1.3, testing is context-dependent, so there will always be differences between the concrete test implementations for different test objects.

Another limitation is the minimal evaluation activities in the project environment. The concept implementation and the feedback of the project experts serve as a basic evaluation and provide an initial assessment. Long-term usage of the concept in the project environment would be necessary to analyse and evaluate the concept in more depth and consider aspects that have not been identified in the current evaluation.

Limitations in the test process include that identifying and defining test scenarios was executed at a smaller scale without every project participant that was suggested in the test concept.

Another limitation concerns the regular execution of the test implementation during the release cycle of the application. Because of the time constraint, it was impossible to evaluate these aspects, which would need to be further analysed to get a holistic evaluation of the concept.

6.3. Lessons Learned

Apart from answering the research question, there were additional learnings that were gained during the implementation.

Besides using the test concept to identify potential bugs that are introduced by new features, tests can also identify bugs in existing functionalities. The test scenarios helped guide the test implementation and uncovered minor bugs in the existing functionality that otherwise were not found before the implementation.

Implementing the test concept was an excellent introduction to how the exemplary system works. The core functionality had to be thoroughly analysed, and the test scenarios for the various features had to be implemented. By the end of the implementation, it was much more apparent how the system works compared to the beginning when the features were explained in theory.

Regarding the usage of the Cucumber framework, it was learned that step definitions could be reused between different features. Cucumber automatically finds the correct step definition for each step of the feature files. The Java test class does not have to be specified. This means that features can be split between multiple test classes, and common test classes can be introduced to reduce code duplication. “@Before(All)” and “@After(All)” lifecycle hooks of the Cucumber framework proved to be very useful in the preparation and finalisation of the test process. Cucumber plugins that are available in development environments like IntelliJ can be used to execute and debug single features or test scenarios, which helps to find errors in the test implementation and saves time not having to wait for the execution of the whole test suite. Finally, configuring Cucumber was impossible according to its documentation suggesting cucumber.properties files. Alternatively, the configuration was possible with “@ConfigurationParameter(...)” annotations in the RunCucumberTest.Java class.

6.4. Future Work

Future work could validate and evaluate the concept in a long-term setting and gather additional information and experience necessary for an in-depth evaluation. Furthermore, future work could apply the testing concept to other business applications to validate whether the concept works in other environments. Additional research would also be necessary to examine whether the concept can be applied to other test levels and if this concept can be used to replace or only complement already existing tests.

Lastly, BDD teams create new features and test scenarios if there is new system behaviour to implement. Testing existing functionality with BDD tests might end once the team has identified and tested all relevant features. If projects use the concept and do not already follow the BDD process, they could transition to using the BDD process for implementing new functionalities. At this point, the team will already be somewhat familiar with BDD, and it will create a more uniform test suite. Further research could analyse if this transition makes sense and if the concept could be used to introduce teams without BDD experience to BDD.

References

Aniche, M. (2022): Effective software testing: A developer's guide. Shelter Island, NY: Manning Publications.

Chelimsky, D.; Astels, D.; Helmkamp, B.; North, D.; Dennis, Z., & Hellesoy, A. (2010): The RSpec book: Behaviour driven development with Rspec. Raleigh, NC: Pragmatic Bookshelf.

eXXcellent Solutions (n.d.): Machine Data Interface Service Documentation (02.02.2023).

Gartner, Inc. (2023): Forecast for global spending on IT services from 2010 to 2023 in billion US-Dollar. https://de-statista-com.ezproxy.hnu.de/statistik/daten/studie/184781/umfrage/weltweite-ausgaben-fuer-it-services/ (03.02.2023).

Hevner, A. R. (2007). A three cycle view of design science research. Scandinavian Journal of Information Systems, Vol. 19, pp. 87-92.

Hevner, A. R.; March, S. T.; Park, J., & Ram, S. (2004). Design science in information systems research. MIS Quarterly, pp. 75–105.

Lawrence, R., & Rayner, P. (2019): Behavior-Driven Development with Cucumber: Better Collaboration for Better Software. Boston: Addison-Wesley Professional.

Matsinopoulos, P. (2020): Practical Test Automation: Learn to Use Jasmine, RSpec, and Cucumber Effectively for Your TDD and BDD. Berkeley, CA: Apress.

Microsoft Corporation (2022): Key concepts for new Azure Pipelines users. https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/key-pipelines-concepts?view=azure-devops (02.02.2023).

Microsoft Corporation (2023a): Customize your pipeline - Azure Pipelines. https://learn.microsoft.com/en-us/azure/devops/pipelines/customize-pipeline?view=azure-devops (02.02.2023).

Microsoft Corporation (2023b): What is Azure Pipelines? https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/what-is-azure-pipelines?view=azure-devops (02.02.2023).

Microsoft Corporation (2023c): YAML schema reference. https://learn.microsoft.com/en-us/azure/devops/pipelines/yaml-schema/?view=azure-pipelines (02.02.2023).

Nascimento, N.; Santos, A. R.; Sales, A., & Chanin, R. (2020). Behavior-Driven Development: An Expert Panel to Evaluate Benefits and Challenges. In *SBES '20,* Proceedings of the XXXIV Brazilian Symposium on Software Engineering, pp. 41–46. New York, NY, USA: Association for Computing Machinery.

North, D. (2006). Behavior modification. Better Software, Vol. 8.

Patton, R. (2013): Software Testing (2nd ed.). Indianapolis, Indiana: Sams.

Pereira, L.; Sharp, H.; Souza, C. de; Oliveira, G.; Marczak, S., & Bastos, R. (2018). Behavior-driven development benefits and challenges: Reports from an Industrial Study. In XP '18 Companion: 19th International Conference on Agile Software Development, Porto Portugal.

Person 1 (06.12.2022). Interview by T. Tögel.

Person 2 (06.12.2022). Interview by T. Tögel.

Person 3 (07.12.2022). Interview by T. Tögel.

Person 4 (08.12.2022). Interview by T. Tögel.

Recker, J. (2021): Scientific research in information systems: A beginner's guide (2nd edition). Cham, Switzerland: Springer.

Richard North (n.d.): Testcontainers for Java. https://www.testcontainers.org (19.02.2023).

Rose, S.; Wynne, M., & Hellesoy, A. (2015): The cucumber for Java book: Behaviour-driven development for testers and developers. Raleigh, NC: The Pragmatic Bookshelf.

Smart, J. F. (2015): Bdd in action: Behavior-driven development for the whole software lifecycle. Shelter Island, NY: Manning Publications.

SmartBear Software (n.d.a): 10 Minute Tutorial - Cucumber Documentation. https://cucumber.io/docs/guides/10-minute-tutorial/?lang=java (10.02.2023).

SmartBear Software (n.d.b): Behaviour-Driven Development - Cucumber Documentation. https://cucumber.io/docs/bdd/ (02.02.2023).

SmartBear Software (n.d.c): Continuous Build - Cucumber Documentation. https://cucumber.io/docs/guides/continuous-integration/ (10.02.2023).

SmartBear Software (n.d.d): Cucumber Reference - Cucumber Documentation. https://cucumber.io/docs/cucumber/api/?lang=java (02.02.2023).

SmartBear Software (n.d.e): Cucumber-JVM - Cucumber Documentation. https://cucumber.io/docs/installation/java/ (02.02.2023).

SmartBear Software (n.d.f): Gherkin Reference - Cucumber Documentation. https://cucumber.io/docs/gherkin/reference/ (02.02.2023).

SmartBear Software (n.d.g): Living Documentation Plugin | Cucumber for JIRA. https://cucumber.io/living-documentation/ (02.02.2023).

SmartBear Software (n.d.h): Reporting - Cucumber Documentation. https://cucumber.io/docs/cucumber/reporting/?lang=java (10.02.2023).

SmartBear Software (n.d.i): Step Definitions - Cucumber Documentation. https://cucumber.io/docs/cucumber/step-definitions/?lang=java (10.02.2023).

Spillner, A., & Linz, T. (2021): Software testing foundations: A study guide for the certified tester exam (5th edition). Heidelberg: dpunkt.verlag.

Appendix

Appendix A: Interview Structure

Introduction:

Short description of the topic

Goal of this interview

Introductory Questions:

1. How big is your software development experience? What is your software development background?
2. How long have you been part of the project and what is your role?
3. What is the goal of the project and how is it structured (Participants & technically)?
4. How do you rate the relevance of software testing? Is it more like a chore or do you think it supports the development process?

Main Questions:

5. What is the MIS System? What is it used for and how does it work?
6. How is MIS structured?
7. How does the development process look like?
8. What is the current testing process? What kind of software tests currently exist?
9. Do you see potential in using BDD tests/Cucumber in the current environment? If yes, what potential? (e.g. in terms of software quality or project success)
10. Can the project benefit from test automation & living documentation?
11. What tests/test levels are most suitable for this test implementation?
12. What kind of challenges do you see for test implementations (Considering the MIS environment, setting up the tests and project dependencies)
13. What kind of problems could arise from the introducing of cucumber tests?
14. Are there any other aspects that should be considered?

Appendix B: Interview Protocol Person 4

Einstiegsfragen

1. **Wie groß ist deine Erfahrung in der Softwareentwicklung? / Was ist dein Background?**

Entwickler seit ca. 6 Jahren. Hauptsächlich Java und JavaScript

2. **Wie lange arbeitest du bereits in dem aktuellen Projekt und was ist deine Rolle?**

Seit Beginn des Projekts bzw. Produkts als Softwarearchitekt.

3. **Was ist das Ziel des Projektes und wie ist es aufgebaut (Personell & technisch)?**

(Frage ausgelassen, da in vorherigen Interviews bereits ausführlich beantwortet wurde)

4. **Wie bewertest du die Relevanz von Software Tests? Findest du es eher störend oder hilfreich?**

Tests sind sehr wichtig. Es ist aber wichtig, genau zu spezifizieren, welche Tests gemeint sind. Wichtig ist, hier möglichst viel zu automatisieren. Natürlich ist es auch ein gewisser Aufwand, die Tests zu schreiben. Die andere Seite ist, dass Tests auch nutzlos sein können.

Hauptfragen

5. **Was ist das MIS System? Wie funktioniert MIS /Für was wird MIS eingesetzt?**

MIS ist eine Schnittstelle für alle Maschinen aus der Produktion. In der Vergangenheit gab es eine Vielzahl an verschiedenen Schnittstellen, z.B. per HTTP oder seriell. MIS stellt hier eine zentrale Schnittstelle bereit und unterstützt OPCUA oder VCA (Protokolle über, die die Maschine mit MIS kommuniziert). Die Maschine muss einer der beiden Protokolle unterstützen.

6. **Wie ist MIS aufgebaut?**

MIS ist ein Teil eines größeren Systems. MIS ist per Events und REST an andere Systeme angebunden. Geschrieben ist MIS in Java (Quarkus Framework).

7. **Wie sieht der Entwicklungsprozess aus?**

(Frage ausgelassen, da in vorherigen Interviews bereits ausführlich beantwortet wurde)

8. **Wie wird aktuell getestet? Welche Arten von Tests gibt es aktuell im MIS Projekt?**

Intern weiß ich das nicht. Ansonsten gibt es Tests direkt an einer Maschine, die das gesamte System testen. Dabei wird nicht nur MIS getestet, sondern das ganze System inklusive der eigentlichen Maschine. Aktuell gibt es sehr viele Tests an der Maschine, da immer neue Maschinen dazu kommen. Das wird aber insgesamt weniger.

9. **Siehts du Potential im Einsatz von BDD/Cucumber Tests in der aktuellen Testlandschaft? Wenn ja welches? (z.B. bezüglich Softwarequalität oder Projekterfolg)**

Cucumber Tests gibt es in einem anderen Bereich schon. Wird wahrscheinlich schwierig einen funktionalen Test mit der Maschine abzubilden.

10. **Kann das Projekt von den Vorteilen der Testautomatisierung & Lebender Dokumentation profitieren?**

(Frage aufgrund des Interviewverlaufs ausgelassen)

11. **Welche Test (-szenarien) /Test Stufen eignen sich am besten für eine Implementierung?**

Eher Unit oder Integrationstest. Z.B. Test der Kommunikation zwischen MIS und Maschine.

Wichtig ist, dass sich MIS immer identisch verhält. Das könnte man durchaus testen. Bei Testen von Maschinenanfragen testen welchen Status geliefert wird, ob Dokument richtig erzeugt wird… Müsste man im Detail dann nochmal klären.

12. **Ist es irrelevant welche Maschine die Daten anfragt?**

Ja, solange niemand Sonderfälle fordert. Es sollte auch egal sein, ob eine Maschine mit OPCUA oder VCA die Daten von MIS anfragt bzw. schickt.

13. **Welche Schwierigkeiten kann es bei der Testimplementierung geben (Bezogen auf das MIS Umfeld)? Worauf muss man achten? (Test Setup, Setup von dependencies)**

(Frage aufgrund des Interviewverlaufs ausgelassen)

14. **Gibt es weitere Aspekte die man beachten sollte?**

Wichtig wäre noch eine gute aber kurze Dokumentation. Schön wäre mit Hilfe der Testfälle eine kurze Doku zu schreiben, die dann das Verhalten der Maschine beschreibt. Keine detaillierte Beschreibung zur BA, sondern eher eine Art Nutzeranleitung.

Appendix C: Download Feature File & Step Definitions (mis_download.feature & DownloadStepDefinitions.Java)

```
Feature: MIS Download

  Background:
    Given all external services are available

  Scenario: Successful download right side, left side 404 status
    Given external services respond with:
      | service        | http method | url                                                     | response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=R.*                           | {"globalOrderId":
10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "R"}]}
| 200    |
      | idr            | get         | /items/byJobId/123\\?side=L.*                           | {}
| 404    |
      | status service | post        | /validationList                                         | {"isValid": true,
"statusId": 1234, "sf": false}
| 200    |
      | pcs            | get         | /pcs/rest/vcaValues/byItem/10000000426621/lens/72383/.* |
{"SURFMT":["1;R;side;10;20;30;40;slope;low;many"],"SURFMTDZX":["1;2"],"SURFMTDZY":["3;4"],"SURFMTDZXY":["1"],"SURFMTZZ":[
"1;2;3"], "CIRC":["127"], "CRIB":["56"], "CYL":["1.5"]} | 200    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result "download_results_right.txt"
    And requested labels and values should be equal to expected labels and values from "download_results_right.txt"

  Scenario: Successful download Left & Right
    Given external services respond with:
      | service        | http method | url                                                     | response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.*                           | {"globalOrderId":
10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "L"}]}
| 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.*                           | {"globalOrderId":
10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side": "R"}]}
| 200    |
      | status service | post        | /validationList                                         | {"isValid": true,
"statusId": 1234, "sf": false}
| 200    |
      | pcs            | get         | /pcs/rest/vcaValues/byItem/10000000426621/lens/72383/.* |
{"SURFMT":["1;L;side;10;22;30;40;slope;low;many"],"SURFMTDZX":["1;4"],"SURFMTDZY":["3;4"],"SURFMTDZXY":["1"],"SURFMTZZ":[
"1;2;3"], "CIRC":["128"], "CRIB":["65"], "CYL":["1.6"]} | 200    |
      | pcs              | get           | /pcs/rest/vcaValues/byItem/10000000426621/lens/72384/.* |
{"SURFMT":["1;R;side;10;20;30;40;slope;low;many"],"SURFMTDZX":["1;2"],"SURFMTDZY":["1;4"],"SURFMTDZXY":["1"],"SURFMTZZ":[
"1;2;3"], "CIRC":["127"], "CRIB":["56"], "CYL":["1.5"]} | 200    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result "download_results_left_right.txt"
    And requested labels and values should be equal to expected labels and values from "download_results_left_right.txt"

  Scenario: IDR service returns status 500
    Given external services respond with:
      | service | http method | url                          | response body | status |
      | idr     | get         | /items/byJobId/123\\?side=.* | {}            | 500    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result "idr_not_available.txt"
    And requested labels and values should be equal to expected labels and values from "idr_not_available.txt"

  Scenario: IDR returns 404 status for left and right
    Given external services respond with:
      | service | http method | url                  | response body | status |
      | idr     | get         | /items/byJobId/123.* | {}            | 404    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result "idr_404_response.txt"
    And requested labels and values should be equal to expected labels and values from "idr_404_response.txt"

  Scenario: Idr returns invalid jobId (returns wrong side for L & R)
    Given external services respond with:
      | service | http method | url                           | response body
| status |
      | idr     | get         | /items/byJobId/123\\?side=R.* | {"globalOrderId": 10000000426621, "globalPairId":
10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "L"}]} | 200    |
      | idr     | get         | /items/byJobId/123\\?side=L.* | {"globalOrderId": 10000000426621, "globalPairId":
10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side": "R"}]} | 200    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result "idr_returns_invalid_job_identifier.txt"
    And requested labels and values should be equal to expected labels and values from
"idr_returns_invalid_job_identifier.txt"

  Scenario: Invalid JobId
    When a download with jobId "" is started
    Then actual result should have the same number of entries as expected result "invalid_job_identifier.txt"
    And requested labels and values should be equal to expected labels and values from "invalid_job_identifier.txt"
  Scenario: PCS  returns status 500
    Given external services respond with:
      | service        | http method | url                           | response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.* | {"globalOrderId": 10000000426621, "globalPairId":
10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "L"}]} | 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.* | {"globalOrderId": 10000000426621, "globalPairId":
10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side": "R"}]} | 200    |
      | status service | post        | /validationList               | {"isValid": true, "statusId": 1234, "sf": false}
| 200    |
      | pcs            | get         | /pcs/rest/vcaValues/byItem/.* | {}
| 500    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result "pcs_not_available.txt"
    And requested labels and values should be equal to expected labels and values from "pcs_not_available.txt"

  Scenario: PCS returns 404 status for one side
    Given external services respond with:
      | service        | http method | url                                                     | response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.*                           | {"globalOrderId":
10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "L"}]}
| 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.*                           | {"globalOrderId":
10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side": "R"}]}
| 200    |
      | status service | post        | /validationList                                         | {"isValid": true,
"statusId": 1234, "sf": false}
| 200    |
      | pcs            | get         | /pcs/rest/vcaValues/byItem/10000000426621/lens/72384/.* |
{"SURFMT":["1;R;side;10;20;30;40;slope;low;many"],"SURFMTDZX":["1;2"],"SURFMTDZY":["3;4"],"SURFMTDZXY":["1"],"SURFMTZZ":[
"1;2;3"], "CIRC":["127"], "CRIB":["56"], "CYL":["1.5"]} | 200    |
      | pcs            | get         | /pcs/rest/vcaValues/byItem/10000000426621/lens/72383/.* | {}
| 404    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result "download_results_right.txt"
    And requested labels and values should be equal to expected labels and values from "download_results_right.txt"

  Scenario: Status service returns status 500
    Given external services respond with:
      | service        | http method | url                  | response body
| status |
      | idr            | get         | /items/byJobId/123.* | {"globalOrderId": 10000000426621, "globalPairId":
10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "R"}]} | 200    |
      | status service | post        | /validationList      | {}
| 500    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result
"status_service_not_available_or_wrong_status.txt"
    And requested labels and values should be equal to expected labels and values from
"status_service_not_available_or_wrong_status.txt"

  Scenario: Product is in wrong status
    Given external services respond with:
      | service        | http method | url                           | response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.* | {"globalOrderId": 10000000426621, "globalPairId":
10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "L"}]} | 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.* | {"globalOrderId": 10000000426621, "globalPairId":
10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "R"}]} | 200    |
      | status service | post        | /validationList               | {"isValid": false}
| 200    |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result
"status_service_not_available_or_wrong_status.txt"
    And requested labels and values should be equal to expected labels and values from
"status_service_not_available_or_wrong_status.txt"
```

```java
package de.exxcellent.stepDefinitions;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.jetbrains.annotations.NotNull;
import vcaclient.VcaMachine;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.HashMap;
import java.util.Map;
import static org.assertj.core.api.Assertions.assertThat;

public class DownloadStepDefinitions {

CommonStepDefinitions commonStepDefinitions = new CommonStepDefinitions();

@When("a download with jobId {string} is started")
public void a_download_is_started(String jobId) {
VcaMachine vcaMachine = new        VcaMachine(CommonStepDefinitions.misTestContainer.getMappedPort(8776),
CommonStepDefinitions.misTestContainer.getHost(), true, false, jobId, "CYL", null, null);
vcaMachine.run(commonStepDefinitions.getReturnedLabels());
}

@Then("actual result should have the same number of entries as expected result {string}")
public void actual_result_should_have_the_same_number_of_entries_as_expected_result(String fileName) throws IOException {
Map<String, String> expectedResult = loadExpectedResult(fileName);
assertThat(commonStepDefinitions.getReturnedLabels()).hasSameSizeAs(expectedResult);
}

@Then("requested labels and values should be equal to expected labels and values from {string}")
public void requested_labels_should_be_equal_to_expected_labels_from(String fileName)throws IOException {
Map<String, String> expectedResult = loadExpectedResult(fileName);
assertThat(commonStepDefinitions.getReturnedLabels()).containsAllEntriesOf(expectedResult );
}
```

```java
@NotNull
private Map<String, String> loadExpectedResult(String fileName) throws IOException {
  Map<String, String> expectedResult = new HashMap<>();
  Path path = FileSystems.getDefault().getPath("src/test/resources/expectedResults", fileName);
  try (BufferedReader br = new BufferedReader(new FileReader(path.toUri().getPath()))) {
    String line;
    while ((line = br.readLine()) != null) {
      String key = line.substring(0, line.indexOf("="));
      String value = line.substring(line.indexOf("=") + 1);
      expectedResult.put(key, value);
      }
   }
  return expectedResult;
}
```

Appendix D: Kafka Download Feature File & Step Definitions (mis_kafka.feature & KafkaStepDefinitions.Java)

```
Feature: MIS Kafka

  Background:
    #External services only needed to verify no requests were made by mis/Idr service necessary for second scenario
    Given all external services are available

  Scenario: Read machine request with _JOBFSVPackagingIDR from cache
    #MIS directly checks item in cache with custom identifier
    Given following machine request is sent to Kafka:
      | identifier                                                   | identifierType     | globalMachineIdentifier |
statusId   | vcaData                                                                                          |
      | 2405403518508422156424999104I6Z4Z91000116040210410000385416 | LENS_SERIAL_NUMBER | LoadTestMachine         | IQC-
VL-L-3 | ["_JOBFSVPackagingID=2405403518508422156424999104I6Z4Z91000116040210410000385416","CYL=-3.75","SPH=-4.25"]] |
    When a download with jobId "123" and custom Identifier
"_JOBFsvPackingID=2405403518508422156424999104I6Z4Z91000116040210410000385416" is started
    Then actual result should have the same number of entries as expected result
"download_read_machine_request_with_PackagingSFID.txt"
    And requested labels and values should be equal to expected labels and values from
"download_read_machine_request_with_PackagingSFID.txt"
    And there should be requests for:
      | service        | number |
      | idr            | 0      |
      | status service | 0      |
      | pcs            | 0      |

  Scenario: Read machine request with FrameId from cache
    #MIS starts download with jobId, doesn't find item in cache, requests data from Idr, receives response with FrameId,
checks cache again with FrameId and finds entry
    Given external services respond with:
      | service | http method | url                            | response body
| status |
      | idr     | get         | /items/byJobId/123\\?side=R.* | {"globalOrderId": 1, "globalPairId": 100, "items":
[{"itemId": "63395", "type": "FRAME", "side": "R"}]} | 200    |
      | idr     | get         | /items/byJobId/123\\?side=L.* | {"globalOrderId": 1, "globalPairId": 100, "items":
[{"itemId": "63396", "type": "FRAME", "side": "L"}]} | 200    |
    And following machine request is sent to Kafka:
      | identifier | identifierType | globalMachineIdentifier | statusId   | vcaData
|
      | 63395      | FRAME_ITEM     | LoadTestMachine         | OQC-VL-L-7 |
["_JOBGlobalPairID=100008435","AX=0.00","CYL=0.00","SPH=-2.00"] |
      | 63396       | FRAME_ITEM     | LoadTestMachine         | OQC-VL-L-7 |
["_JOBGlobalPairID=100008435","AX=0.00","CYL=0.00","SPH=-2.00"] |
    When a download with jobId "123" is started
    Then actual result should have the same number of entries as expected result
"download_read_machine_request_with_FrameId.txt"
    And requested labels and values should be equal to expected labels and values from
"download_read_machine_request_with_FrameId.txt"
    And there should be requests for:
      | service        | number |
      | idr            | 2      |
      | status service | 0      |
      | pcs            | 0      |
```

```java
package de.exxcellent.stepDefinitions;
import com.github.tomakehurst.wiremock.WireMockServer;
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import de.exxcellent.KafkaHelper;
import vcaclient.VcaMachine;
import java.util.List;
import java.util.Map;
import static com.github.tomakehurst.wiremock.client.WireMock.*;

public class KafkaStepDefinitions {
CommonStepDefinitions commonStepDefinitions = new CommonStepDefinitions();

@Given("following machine request is sent to Kafka:")
public void machine_request_is_sent_to_kafka(List<Map<String, String>> dataTable) {
String schemaRegistryUrl = "http://" + CommonStepDefinitions.schemaRegistry.getHost() + ":" +
CommonStepDefinitions.schemaRegistry.getMappedPort(8081);
KafkaHelper kafkaHelper = new KafkaHelper(CommonStepDefinitions.kafka.getBootstrapServers(), schemaRegistryUrl, "quantum-
machine-data-interface-microservice");
for (Map<String, String> entry : dataTable) {
final MachineRequest machineRequest = KafkaHelper.createMachineRequest(entry.get("identifier"),
entry.get("identifierType"), entry.get("globalMachineIdentifier"), entry.get("statusId"), entry.get("vcaData"));
kafkaHelper.sendDataToKafka(machineRequest);
 }
}
@When("a download with jobId {string} and custom Identifier {string} is started")
public void request_to_kafka_is_sent(String jobId, String customIdentifier) {
VcaMachine vcaMachine = new VcaMachine(CommonStepDefinitions.misTestContainer.getMappedPort(8776),
CommonStepDefinitions.misTestContainer.getHost(), true, false, jobId, "CYL", customIdentifier, null);
vcaMachine.run(commonStepDefinitions.getReturnedLabels());
}
@Then("there should be no requests to any external service")
public void there_Should_Be_No_Requests_To_Any_Service() {
commonStepDefinitions.getIdrWireMockServer().verify(0, anyRequestedFor(anyUrl()));
commonStepDefinitions.getStatusServiceWireMockServer().verify(0, anyRequestedFor(anyUrl()));
commonStepDefinitions.getPcsWireMockServer().verify(0, anyRequestedFor(anyUrl()));
}
@Then("there should be requests for:")
public void there_Should_Be_No_Requests_To_Pcs_Status(List<Map<String, String>> dataTable) {
for (Map<String, String> entry : dataTable) {
WireMockServer wireMockServer = commonStepDefinitions.getWireMockServer(entry);
wireMockServer.verify(Integer.parseInt(entry.get("number")), anyRequestedFor(anyUrl()));
  }
 }
}
```

Appendix E: Upload Feature File & Step Definitions (mis_upload.feature & UploadStepDefinitions.Java)

```
Feature: MIS Upload

  Background:
    Given all external services are available

  Scenario: Successful upload left & right
    Given external services respond with:
      | service        | http method | url                                                                    |
response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.*                                          |
{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side":
"L"}]} | 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.*                                          |
{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side":
"R"}]} | 200    |
      | status service | post        | /validationList                                                        |
{"isValid": true, "statusId": 1234, "sf": false}
| 200    |
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument.*                                |
{"_id":"123456"}
| 200    |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false\\?source=LoadTestMachine.* |
{}
| 200    |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false\\?source=LoadTestMachine.* |
{}
| 200    |
    When an upload with jobId "123" is started
    Then following requests should be sent:
      | service | http method | url                                                            | request
count | request body
|
      | pcs     | post        | /pcs/rest/documentStorage/vcaDocument                          | 1
| {"statusId":"1234","processType":"UPLOAD","fileName":"LoadTestMachine-10000000426621-L-
72383","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine
","type":"GLOBAL_MACHINE_IDENTIFIER"},{"value":"10000000426621","type":"GLOBAL_PAIR_ID"},{"value":"10000000426621","type"
:"GLOBAL_ORDER_ID"}],"items":[{"itemId":72383,"type":"LENS","side":"L","globalOrderItem":null}]} |
      | pcs     | post        | /pcs/rest/documentStorage/vcaDocument                          | 1
| {"statusId":"1234","processType":"UPLOAD","fileName":"LoadTestMachine-10000000426621-R-
72384","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine
","type":"GLOBAL_MACHINE_IDENTIFIER"},{"value":"10000000426621","type":"GLOBAL_PAIR_ID"},{"value":"10000000426621","type"
:"GLOBAL_ORDER_ID"}],"items":[{"itemId":72384,"type":"LENS","side":"R","globalOrderItem":null}]} |
      | pcs     | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine | 1
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '128.57' ], "CYL": [ '22' ], "CIRC": [
'128.56' ], "DO": [ 'B' ], "SURFMT": [ '1;L;side;10;22;30;40;slope;low;many' ], "SURFMTDZY": [ '3;4' ], "SURFMTDZX": [
'1;4' ]}
|
      | pcs     | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine | 1
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '124' ], "CYL": [ '22' ], "CIRC": [
'223' ], "DO": [ 'B' ], "SURFMT": [ '1;R;side;10;20;30;40;slope;low;many' ], "SURFMTDZY": [ '1;4' ], "SURFMTDZX": [ '1;2'
]}
|
    And number of items uploaded in documents should be 2

  Scenario: Successful upload right side, left side 404 status
    Given external services respond with:
      | service          | http method | url                                                                            |
response body
| status |
      | idr              | get         | /items/byJobId/123\\?side=L.*                                                  |
{}
| 404    |
      | idr              | get         | /items/byJobId/123\\?side=R.*                                                  |
{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side":
"R"}]} | 200    |
      | status service | post        | /validationList                                                                |
{"isValid": true, "statusId": 1234, "sf": false}
| 200    |
      | pcs              | post        | /pcs/rest/documentStorage/vcaDocument.*                                        |
{"_id":"123456"}
| 200    |
      | pcs              | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false\\?source=LoadTestMachine.* |
{}
| 200    |
      | pcs              | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false\\?source=LoadTestMachine.* |
{}
| 200    |
    When an upload with jobId "123" is started
    Then following requests should be sent:
      | service | http method | url                                                                         | request
```

```
count | request body
|
      | pcs     | post        | /pcs/rest/documentStorage/vcaDocument                                      | 1
| {"statusId":"1234","processType":"UPLOAD","fileName":"LoadTestMachine-10000000426621-R-
72384","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine
","type":"GLOBAL_MACHINE_IDENTIFIER"},{"value":"10000000426621","type":"GLOBAL_PAIR_ID"},{"value":"10000000426621","type"
:"GLOBAL_ORDER_ID"}],"items":[{"itemId":72384,"type":"LENS","side":"R","globalOrderItem":null}]} |
      | pcs     | post        | /pcs/rest/documentStorage/vcaDocument                                      | 1
| {"statusId":"1","processType":"UPLOAD","fileName":"LoadTestMachine-null-null-
null","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine"
,"type":"GLOBAL_MACHINE_IDENTIFIER"}],"items":[]}
|
      | pcs     | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine | 0
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '128.57' ], "CYL": [ '22' ], "CIRC": [
'128.56' ], "DO": [ 'B' ], "SURFMT": [ '1;L;side;10;22;30;40;slope;low;many' ], "SURFMTDZY": [ '3;4' ], "SURFMTDZX": [
'1;4' ]}
|
      | pcs     | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine | 1
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '124' ], "CYL": [ '22' ], "CIRC": [
'223' ], "DO": [ 'B' ], "SURFMT": [ '1;R;side;10;20;30;40;slope;low;many' ], "SURFMTDZY": [ '1;4' ], "SURFMTDZX": [ '1;2'
]}
|
    And number of items uploaded in documents should be 1

  Scenario: IDR returns 404 status for left and right
    Given external services respond with:
      | service | http method | url                                     | response body     | status |
      | idr     | get         | /items/byJobId/123\\?side=L.*           | {}                | 404    |
      | idr     | get         | /items/byJobId/123\\?side=R.*           | {}                | 404    |
      | pcs     | post        | /pcs/rest/documentStorage/vcaDocument.* | {"_id":"123456"}  | 200    |
    When an upload with jobId "123" is started
    Then following requests should be sent:
      | service        | http method | url                                                                        |
request count | request body
|
      | status service | post        | /validationList                                                            | 0
| {}
|
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument                                      | 2
| {"statusId":"1","processType":"UPLOAD","fileName":"LoadTestMachine-null-null-
null","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine"
,"type":"GLOBAL_MACHINE_IDENTIFIER"}],"items":[]} |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine | 0
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '128.57' ], "CYL": [ '22' ], "CIRC": [
'128.56' ], "DO": [ 'B' ], "SURFMT": [ '1;L;side;10;22;30;40;slope;low;many' ], "SURFMTDZY": [ '3;4' ], "SURFMTDZX": [
'1;4' ]}  |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine | 0
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '124' ], "CYL": [ '22' ], "CIRC": [
'223' ], "DO": [ 'B' ], "SURFMT": [ '1;R;side;10;20;30;40;slope;low;many' ], "SURFMTDZY": [ '1;4' ], "SURFMTDZX": [ '1;2'
]}         |
    And number of items uploaded in documents should be 0

  Scenario: Idr service returns status 500
    Given external services respond with:
      | service | http method | url                                  | response body    | status |
      | idr     | get         | /items/byJobId/123\\?side=.*          | {}               | 500    |
      | pcs     | post        | /pcs/rest/documentStorage/vcaDocument | {"_id":"123456"} | 200    |
    When an upload with jobId "123" is started
    Then following requests should be sent:
      | service        | http method | url                                                                         |
request count | request body
|
      | status service | post        | /validationList                                                             | 0
| {}
|
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument                                       | 2
| {"statusId":"1","processType":"UPLOAD","fileName":"LoadTestMachine-null-null-
null","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine"
,"type":"GLOBAL_MACHINE_IDENTIFIER"}],"items":[]} |
      | pcs            | post        | /pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine | 0
| {}
|
      | pcs            | post        | /pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine | 0
| {}
|
    And number of items uploaded in documents should be 0

  Scenario: Idr returns invalid jobId (returns wrong side for L & R)
    Given external services respond with:
      | service        | http method | url                                                                             |
response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.*                                                   |
{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side":
```

```
"R"}]} | 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.*                                                   |
{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side":
"L"}]} | 200    |
      | status service | post        | /validationList                                                                 |
{"isValid": true, "statusId": 1234, "sf": false}
| 200    |
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument.*                                         |
{"_id":"123456"}
| 200    |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false\\?source=LoadTestMachine.* |
{}
| 200    |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false\\?source=LoadTestMachine.* |
{}
| 200    |
    When an upload with jobId "123" is started
    Then following requests should be sent:
      | service        | http method | url                                                                        |
request count | request body
|
      | status service | post        | /validationList                                                            | 0
| {}
|
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument                                      | 2
| {"statusId":"1","processType":"UPLOAD","fileName":"LoadTestMachine-null-null-
null","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine"
,"type":"GLOBAL_MACHINE_IDENTIFIER"}],"items":[]} |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine | 0
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '128.57' ], "CYL": [ '22' ], "CIRC": [
'128.56' ], "DO": [ 'B' ], "SURFMT": [ '1;L;side;10;22;30;40;slope;low;many' ], "SURFMTDZY": [ '3;4' ], "SURFMTDZX": [
'1;4' ]}  |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine | 0
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '124' ], "CYL": [ '22' ], "CIRC": [
'223' ], "DO": [ 'B' ], "SURFMT": [ '1;R;side;10;20;30;40;slope;low;many' ], "SURFMTDZY": [ '1;4' ], "SURFMTDZX": [ '1;2'
]}        |
    And number of items uploaded in documents should be 0

  Scenario: Invalid JobId
    When an upload with jobId "" is started
    Then upload should not be processed
  Scenario: Status service returns status 500
    Given external services respond with:
      | service        | http method | url                                   | response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.*         | {"globalOrderId": 10000000426621,
"globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "L"}]}   | 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.*         | {"globalOrderId": 10000000426621,
"globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side": "R"}]}{} | 200    |
      | status service | post        | /validationList                       | {}
| 500    |
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument | {"_id":"123456"}
| 200    |
    When an upload with jobId "123" is started
    Then following requests should be sent:
      | service        | http method | url                                                                        |
request count | request body
|
      | status service | post        | /validationList                                                            | 0
| {}
|
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument                                      | 2
| {"statusId":"1","processType":"UPLOAD","fileName":"LoadTestMachine-10000000426621-null-
null","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine"
,"type":"GLOBAL_MACHINE_IDENTIFIER"},{"value":"10000000426621","type":"GLOBAL_PAIR_ID"},{"value":"10000000426621","type":
"GLOBAL_ORDER_ID"}],"items":[]} |
      | pcs            | post        | /pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine | 0
| {}
|
      | pcs            | post        | /pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine | 0
| {}
|
    And number of items uploaded in documents should be 0

  Scenario: Product is in wrong status
    Given external services respond with:
      | service        | http method | url                                                                            |
response body
| status |
      | idr            | get         | /items/byJobId/123\\?side=L.*                                                  |
{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side":
"L"}]} | 200    |
      | idr            | get         | /items/byJobId/123\\?side=R.*                                                  |
{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side":
"R"}]} | 200    |
      | status service | post        | /validationList                                                                 |
{"isValid": false}
| 200    |
      | pcs            | post        | /pcs/rest/documentStorage/vcaDocument.*                                         |
{"_id":"123456"}
| 200    |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false\\?source=LoadTestMachine.* |
{}
| 200    |
      | pcs            | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false\\?source=LoadTestMachine.* |
{}
| 200    |
    When an upload with jobId "123" is started
    Then following requests should be sent:
      | service | http method | url                                                                         | request
count | request body
|
      | pcs     | post        | /pcs/rest/documentStorage/vcaDocument                                       | 2
| {"statusId":"1","processType":"UPLOAD","fileName":"LoadTestMachine-10000000426621-null-
null","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine"
,"type":"GLOBAL_MACHINE_IDENTIFIER"},{"value":"10000000426621","type":"GLOBAL_PAIR_ID"},{"value":"10000000426621","type":
"GLOBAL_ORDER_ID"}],"items":[]} |
      | pcs     | put         | /pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine | 0
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '128.57' ], "CYL": [ '22' ], "CIRC": [
'128.56' ], "DO": [ 'B' ], "SURFMT": [ '1;L;side;10;22;30;40;slope;low;many' ], "SURFMTDZY": [ '3;4' ], "SURFMTDZX": [
'1;4' ]}
|
      | pcs     | put         | /pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine | 0
| {"STATUS": [ '0' ], "SURFMTZZ": [ '1;2;3' ], "SURFMTDZXY": [ '1' ], "CIRC3D": [ '124' ], "CYL": [ '22' ], "CIRC": [
'223' ], "DO": [ 'B' ], "SURFMT": [ '1;R;side;10;20;30;40;slope;low;many' ], "SURFMTDZY": [ '1;4' ], "SURFMTDZX": [ '1;2'
]}
|
    And number of items uploaded in documents should be 0
```

```
package de.exxcellent.stepDefinitions;
import com.github.tomakehurst.wiremock.stubbing.ServeEvent;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import vcaclient.VcaMachine;
import java.io.File;
import java.nio.file.FileSystems;
import java.nio.file.Path;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.TimeoutException;
import java.util.stream.Collectors;
import static com.github.tomakehurst.wiremock.client.WireMock.*;
import static de.exxcellent.stepDefinitions.CommonStepDefinitions.misTestContainer;
import static org.assertj.core.api.Assertions.assertThat;
import static org.junit.jupiter.api.Assertions.assertThrows;

public class UploadStepDefinitions {
CommonStepDefinitions commonStepDefinitions = new CommonStepDefinitions();
ScheduledThreadPoolExecutor delayedExecutor = new ScheduledThreadPoolExecutor(1);

@When("an upload with jobId {string} is started")
public void an_upload_with_job_id_is_started(String jobId) {
Path path = FileSystems.getDefault().getPath("src/test/resources", "upload_data.oma");
File file = new File(path.toUri().getPath());
VcaMachine vcaMachine = new VcaMachine(misTestContainer.getMappedPort(8776), misTestContainer.getHost(),false, true,
jobId, "", null, file);
vcaMachine.run(commonStepDefinitions.getReturnedLabels());
}

@Then("number of items uploaded in documents should be {int}")
public void number_Of_Items_Uploaded_In_Documents_Should_Be(Integer sizeOfItems) {
List<String> entryList = new ArrayList<>();
for (ServeEvent entry : commonStepDefinitions.getPcsWireMockServer().getServeEvents().getRequests()) {
 if (entry.getRequest().getUrl().contains("/pcs/rest/documentStorage/vcaDocument")) {
  String requestBody = entry.getRequest().getBodyAsString();
  int charToBeginningOfItemsObject = 8;
  int charToEndOfItemsObject = 2;
  String items = requestBody.substring(requestBody.indexOf("items") + charToBeginningOfItemsObject, requestBody.length()
  - charToEndOfItemsObject);
  if (items.length() > 0) {
  entryList.add(items);
}}}
assertThat(entryList).hasSize(sizeOfItems);
}

@Then("following requests should be sent:")
public void following_requests_should_be_sent(List<Map<String, String>> dataTable) throws ExecutionException,
InterruptedException, TimeoutException {
waitForRequests();
for (Map<String, String> entryExpected : dataTable) {
if (entryExpected.get("http method").equals("post")) {
commonStepDefinitions.getPcsWireMockServer().verify(Integer.parseInt(entryExpected.get("request count")),
postRequestedFor(urlEqualTo(entryExpected.get("url"))).withRequestBody(equalToJson(entryExpected.get("request
body"),true, true)));
} else if (entryExpected.get("http method").equals("put")) {
commonStepDefinitions.getPcsWireMockServer().verify(Integer.parseInt(entryExpected.get("request count")),
putRequestedFor(urlEqualTo(entryExpected.get("url"))).withRequestBody(equalToJson(entryExpected.get("request body"),true,
true)));
 }
}
for (ServeEvent entryActual : vcaDocumentRequests()) {
assertThat(entryActual.getRequest().getBodyAsString().contains("fileContent")).isTrue();
assertThat(entryActual.getRequest().getBodyAsString().contains("traceId")).isTrue();
 }
}
@Then("upload should not be processed")
public void upload_should_not_be_processed() {
assertThrows(TimeoutException.class, this::waitForRequests);
}

private void waitForRequests() throws InterruptedException, ExecutionException, TimeoutException {
for (int i = 0; i < 30; i++) {
 if (delayedExecutor.schedule(() -> vcaDocumentRequests().size() > 1, 100, TimeUnit.MILLISECONDS).get()) {
  return;
}}
throw new TimeoutException();
}
```

```
private List<ServeEvent> vcaDocumentRequests() {
return commonStepDefinitions.getPcsWireMockServer().getServeEvents().getRequests()
.stream()
.filter((requests) -> requests.getRequest().getUrl().contains("/pcs/rest/documentStorage/vcaDocument"))
.collect(Collectors.toList());
 }
}
```

Appendix F: Common Step Definitions (CommonStepDefinitions.Java)

```java
package de.exxcellent.stepDefinitions;
import com.github.tomakehurst.wiremock.WireMockServer;
import com.github.tomakehurst.wiremock.client.ResponseDefinitionBuilder;
import io.cucumber.java.After;
import io.cucumber.java.AfterAll;
import io.cucumber.java.Before;
import io.cucumber.java.BeforeAll;
import io.cucumber.java.en.Given;
import org.jetbrains.annotations.Nullable;
import org.testcontainers.Testcontainers;
import org.testcontainers.containers.GenericContainer;
import org.testcontainers.containers.KafkaContainer;
import org.testcontainers.containers.Network;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import static com.github.tomakehurst.wiremock.client.WireMock.*;
import static com.github.tomakehurst.wiremock.core.WireMockConfiguration.wireMockConfig;
import static org.assertj.core.api.Assertions.assertThat;

public class CommonStepDefinitions {
static WireMockServer idrWireMockServer;
static WireMockServer pcsWireMockServer;
static WireMockServer statusServiceWireMockServer;
static final Network network = Network.newNetwork();
static final KafkaContainer kafka = new
KafkaContainer(DockerImageNameConfiguration.KAFKA_IMAGE_NAME).withNetwork(network).withNetworkMode("bridge");

static final GenericContainer<?> schemaRegistry = new
GenericContainer<>(DockerImageNameConfiguration.SCHEMA_REGISTRY_IMAGE_NAME).withNetwork(network)
.withNetworkMode("bridge")
.withExposedPorts(8081)
.withNetworkAliases("test-schema-registry")
.withEnv("SCHEMA_REGISTRY_KAFKASTORE_BOOTSTRAP_SERVERS", kafka.getNetworkAliases().get(0) + ':' + 9092)
.withEnv("SCHEMA_REGISTRY_HOST_NAME", "test-schema-registry")
.withEnv("SCHEMA_REGISTRY_ACCESS_CONTROL_ALLOW_ORIGIN", "*")
.withEnv("SCHEMA_REGISTRY_ACCESS_CONTROL_ALLOW_METHODS", "GET,POST,PUT,OPTIONS");
public static final GenericContainer<?> misTestContainer = new
GenericContainer<>(DockerImageNameConfiguration.MIS_IMAGE_NAME)
```

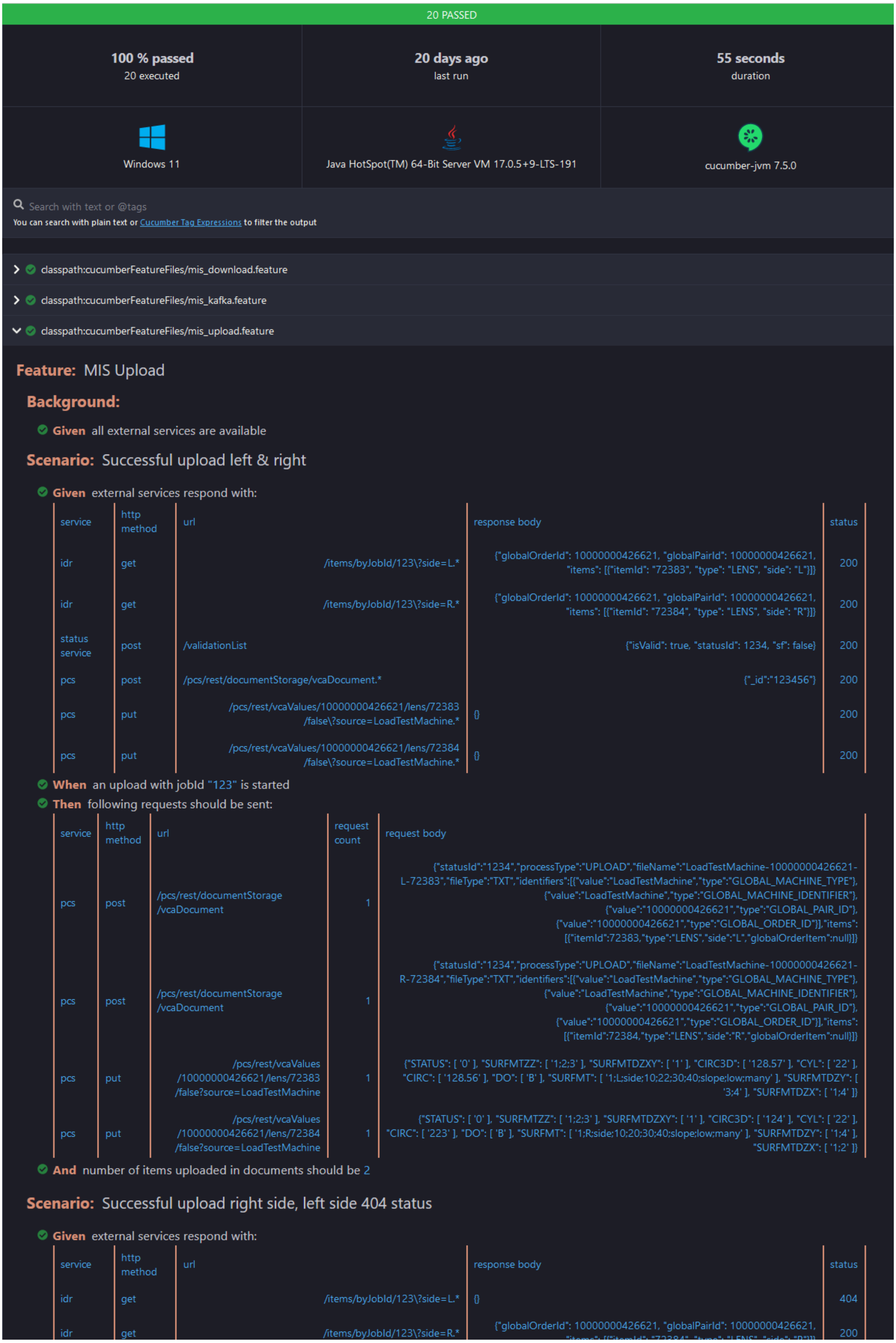
```
.withNetwork(network)
.withNetworkMode("bridge")
.withExposedPorts(8776)
.withEnv("IDENTIFIER_RESOLVER_REST_SERVICE_MP_REST_URL", "http://host.testcontainers.internal:8085")
.withEnv("QUANTUM_REST_SERVICE_MP_REST_URL", "http://host.testcontainers.internal:8080/pcs/rest")
.withEnv("STATUS_REST_SERVICE_MP_REST_URL", "http://host.testcontainers.internal:8084")
.withEnv("MACHINE_CONFIGURATION", MachineConfiguration.machineConfiguration)
.withEnv("KAFKA_BOOTSTRAP_SERVERS", kafka.getNetworkAliases().get(0) + ':' + 9092)
.withEnv("KAFKA_SCHEMA_REGISTRY_URL", "http://" + schemaRegistry.getNetworkAliases().get(0) + ':' + 8081);
private static Map<String, String> returnedLabels;
@BeforeAll
public static void beforeAll() {
misTestContainer.withAccessToHost(true);
Testcontainers.exposeHostPorts(8085);
Testcontainers.exposeHostPorts(8080);
Testcontainers.exposeHostPorts(8084);
try {
kafka.start();
schemaRegistry.start();
misTestContainer.start();
assertThat(kafka.isRunning()).isTrue();
assertThat(schemaRegistry.isRunning()).isTrue();
assertThat(misTestContainer.isRunning()).isTrue();
} catch (Exception e) {
throw new RuntimeException(e);}
}
@AfterAll
public static void afterAll() {
kafka.stop();
schemaRegistry.stop();
misTestContainer.stop();
}
@Before
public void before() {
returnedLabels = new HashMap<>();
}
@After
public void after() {
stopWireMocks();}

@Given("all external services are available")
public void all_external_services_are_available() {
```

```
startIdrWireMock();
startPcsWireMock();
startStatusServiceWireMock();
}

@Given("external services respond with:")
public void external_services_respond_with(List<Map<String, String>> dataTable) {
for (Map<String, String> entry : dataTable) {
WireMockServer wireMockServer = getWireMockServer(entry);
if (entry.get("http method").equals("post")) {
wireMockServer.stubFor(post(urlMatching(entry.get("url"))).willReturn(mockResponse(entry)));
} else if (entry.get("http method").equals("get")) {
wireMockServer.stubFor(get(urlMatching(entry.get("url"))).willReturn(mockResponse(entry)));
} else if (entry.get("http method").equals("put")) {
wireMockServer.stubFor(put(urlMatching(entry.get("url"))).willReturn(mockResponse(entry)));
}}}

private static ResponseDefinitionBuilder mockResponse(Map<String, String> entry) {
return aResponse()
.withHeader("Content-Type", "application/json")
.withBody(entry.get("response body"))
.withStatus(Integer.parseInt(entry.get("status")));
}

private void startStatusServiceWireMock() {
statusServiceWireMockServer = new WireMockServer(wireMockConfig().port(8084));
statusServiceWireMockServer.start();
}
private void startPcsWireMock() {
pcsWireMockServer = new WireMockServer(wireMockConfig().port(8080));
pcsWireMockServer.start();
}
private void startIdrWireMock() {
idrWireMockServer = new WireMockServer(wireMockConfig().port(8085));
idrWireMockServer.start();
}
private void stopWireMocks() {
if (idrWireMockServer != null) {
idrWireMockServer.stop();
}
if (pcsWireMockServer != null) {
pcsWireMockServer.stop();
}
if (statusServiceWireMockServer != null) {
statusServiceWireMockServer.stop();
}
}
@Nullable WireMockServer getWireMockServer(Map<String, String> entry) {
WireMockServer wireMockServer = null;
switch (entry.get("service")) {
case "idr":
wireMockServer = idrWireMockServer;
break;
case "pcs":
wireMockServer = pcsWireMockServer;
break;
case "status service":
wireMockServer = statusServiceWireMockServer;
break;
}
return wireMockServer;
}
Map<String, String> getReturnedLabels() {
return returnedLabels;
}
WireMockServer getPcsWireMockServer() {
return pcsWireMockServer;
}
WireMockServer getStatusServiceWireMockServer() {
return statusServiceWireMockServer;
}
WireMockServer getIdrWireMockServer() {
return idrWireMockServer;
}
}
```

Appendix G: Cucumber JUnit Test Class (RunCucumberTest.Java)

```
package de.exxcellent;
import org.junit.platform.suite.api.ConfigurationParameter;
import org.junit.platform.suite.api.IncludeEngines;
import org.junit.platform.suite.api.SelectClasspathResource;
import org.junit.platform.suite.api.Suite;
import static io.cucumber.junit.platform.engine.Constants.PLUGIN_PROPERTY_NAME;
import static
io.cucumber.junit.platform.engine.Constants.PLUGIN_PUBLISH_QUIET_PROPERTY_NAME;

@Suite
@IncludeEngines("cucumber")
@SelectClasspathResource("cucumberFeatureFiles")
@ConfigurationParameter(key = PLUGIN_PROPERTY_NAME, value = "pretty")
@ConfigurationParameter(key = PLUGIN_PROPERTY_NAME, value =
"html:target/cucumber-reports.html")
@ConfigurationParameter(key = PLUGIN_PUBLISH_QUIET_PROPERTY_NAME, value =
"true")
public class RunCucumberTest  {
}
```

Appendix H: Cucumber Test Report (cucumber-reports.html)



20 PASSED

100 % passed
20 executed

20 days ago
last run

55 seconds
duration

Windows 11

Java HotSpot(TM) 64-Bit Server VM 17.0.5+9-LTS-191

cucumber-jvm 7.5.0

Search with text or @tags
You can search with plain text or Cucumber Tag Expressions to filter the output

> ✓ classpath:cucumberFeatureFiles/mis_download.feature
> ✓ classpath:cucumberFeatureFiles/mis_kafka.feature
⌄ ✓ classpath:cucumberFeatureFiles/mis_upload.feature

Feature: MIS Upload

Background:

✓ **Given** all external services are available

Scenario: Successful upload left & right

✓ **Given** external services respond with:

service	http method	url	response body	status
idr	get	/items/byJobId/123\?side=L.*	{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72383", "type": "LENS", "side": "L"}]}	200
idr	get	/items/byJobId/123\?side=R.*	{"globalOrderId": 10000000426621, "globalPairId": 10000000426621, "items": [{"itemId": "72384", "type": "LENS", "side": "R"}]}	200
status service	post	/validationList	{"isValid": true, "statusId": 1234, "sf": false}	200
pcs	post	/pcs/rest/documentStorage/vcaDocument.*	{"_id":"123456"}	200
pcs	put	/pcs/rest/vcaValues/10000000426621/lens/72383/false\?source=LoadTestMachine.*	{}	200
pcs	put	/pcs/rest/vcaValues/10000000426621/lens/72384/false\?source=LoadTestMachine.*	{}	200

✓ **When** an upload with jobId "123" is started

✓ **Then** following requests should be sent:

service	http method	url	request count	request body
pcs	post	/pcs/rest/documentStorage/vcaDocument	1	{"statusId":"1234","processType":"UPLOAD","fileName":"LoadTestMachine-10000000426621-L-72383","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_IDENTIFIER"},{"value":"10000000426621","type":"GLOBAL_PAIR_ID"},{"value":"10000000426621","type":"GLOBAL_ORDER_ID"}],"items":[{"itemId":72383,"type":"LENS","side":"L","globalOrderItem":null}]}
pcs	post	/pcs/rest/documentStorage/vcaDocument	1	{"statusId":"1234","processType":"UPLOAD","fileName":"LoadTestMachine-10000000426621-R-72384","fileType":"TXT","identifiers":[{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_TYPE"},{"value":"LoadTestMachine","type":"GLOBAL_MACHINE_IDENTIFIER"},{"value":"10000000426621","type":"GLOBAL_PAIR_ID"},{"value":"10000000426621","type":"GLOBAL_ORDER_ID"}],"items":[{"itemId":72384,"type":"LENS","side":"R","globalOrderItem":null}]}
pcs	put	/pcs/rest/vcaValues/10000000426621/lens/72383/false?source=LoadTestMachine	1	{"STATUS": ['0'], "SURFMTZZ": ['1;2;3'], "SURFMTDZXY": ['1'], "CIRC3D": ['128.57'], "CYL": ['22'], "CIRC": ['128.56'], "DO": ['B'], "SURFMT": ['1;L;side;10;22;30;40;slope;low;many'], "SURFMTDZY": ['3;4'], "SURFMTDZX": ['1;4']}
pcs	put	/pcs/rest/vcaValues/10000000426621/lens/72384/false?source=LoadTestMachine	1	{"STATUS": ['0'], "SURFMTZZ": ['1;2;3'], "SURFMTDZXY": ['1'], "CIRC3D": ['124'], "CYL": ['22'], "CIRC": ['223'], "DO": ['B'], "SURFMT": ['1;R;side;10;20;30;40;slope;low;many'], "SURFMTDZY": ['1;4'], "SURFMTDZX": ['1;2']}

✓ **And** number of items uploaded in documents should be 2

Scenario: Successful upload right side, left side 404 status

✓ **Given** external services respond with:

service	http method	url	response body	status
idr	get	/items/byJobId/123\?side=L.*	{}	404
idr	get	/items/byJobId/123\?side=R.*	{"globalOrderId": 10000000426621, "globalPairId": 10000000426621,	200