TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Hans Hendrik Starkopf 201427IVSM

Behaviour-driven Specification Management and Execution System

Master's Thesis

Supervisor: Gert Kanter
PhD

Co-supervisor: Dietmar Pfahl
PhD

Tallinn 2023

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Hans Hendrik Starkopf 201427IVSM

KASUTAJA KÄITUMISEL PÕHINEVA SPETSIFIKATSIOONI HALDUS- JA KÄIVITUSSÜSTEEM

Magistritöö

Juhendaja: Gert Kanter
PhD

Kaasjuhendaja: Dietmar Pfahl
PhD

Tallinn 2023

Author’s Declaration of Originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Hans Hendrik Starkopf

18.05.2023

Abstract

Behaviour-driven Specification Management and Execution System

In agile methodologies, software requirements are initially captured informally, therefore automated acceptance testing relies on subjective interpretation of requirements by the developer. Behaviour-Driven Development (BDD) is focused on defining fine-grained specifications of the behaviour of the targeting system, in a way that can be automated. However, the current state of BDD tools relies on the usage of Domain Specific Languages (DSL), such as Gherkin. DSLs are connected to several usability and maintenance issues as the specification grows. Developers, who usually work with code base, where the source of truth for the DSL-based specification lives, would become responsible for interpreting and translating initial requirements into actionable scenarios and test code, introducing subjectivity and potentially altering the system's intent while organising test suites. As a result, maintenance issues and inconsistencies between requirements and their implementation may arise, leading to potential issues in the final software product.

In this thesis, a Design Science Research (DSR) methodology was utilised to propose and evaluate a process for creating and executing behaviour-driven software specifications, that doesn't rely on DSL. For that purpose, a new tool was developed to enable UI-based specification management that incorporates predefined granular executable user interactions for creating specifications.

The process was evaluated using a task management application as an example to specify and execute its requirements, using the new tool developed. Additionally, a single user testing session was conducted. The tool enabled to capture and modify software specifications, as well as execute without requiring any test code to be written. As a result, the new tool enabled central management of specifications, offered new capabilities to reduce specification maintenance issues, while minimising the subjectivity aspect of testing requirements. Despite the several advantages of the proposed approach, it introduced multiple new limitations.

The thesis is written in English and contains text on 65 pages, 10 chapters, 24 figures, 3 tables.

Annotatsioon

Kasutaja käitumisel põhineva spetsifikatsiooni haldus- ja käivitussüsteem

Agiilsetes tarkvaraarendusmeetodites kirjeldatakse tarkvara nõudeid mitteformaalsel viisil, seega automatiseeritud *acceptance testing* ehk vastuvõtutestimine põhineb arendaja subjektiivsel nõuete tõlgendusel testkoodiks. *Behaviour-Driven Development* (BDD) ehk käitumisel põhinev arendusmetoodika keskendub süsteemi kasutaja käitumist kirjeldava spetsifikatsiooni defineerimisele viisil, mida saab automatiseerida. Valdav osa BDD tööriistu aga sõltuvad domeenispetsiifiliste keelte (DSL), näiteks Gherkini, kasutamisest. DSL-e seostatakse aga mitmete kasutatavuse ja hooldusega seotud probleemidega. Kuna DSL failid asuvad valdavalt tarkvara koodibaasi lähedal, siis DSL-i põhise spetsifikatsiooni haldajateks saavad tihtipeale arendajad, kes vastutavad esialgsete nõuete tõlgendamise eest automatiseeritavateks stsenaariumideks ja testikoodideks. See hõlmab endas esialgsete nõuete subjektiivset tõlgendamist ning võib potentsiaalselt kaasa tuua süsteemi eesmärkide muutmist testide haldamisel. Tulemusena võivad tekkida hooldusprobleemid ning ebakõlad nõuete ja nende rakendamise vahel.

Selles töös kasutati Design Science Research (DSR) metoodikat, et pakkuda välja protsess käitumispõhiste tarkvara spetsifikatsioonide loomiseks ja täitmiseks, mis ei tugine DSL-il. Selle protsessi toetamiseks töötati välja uus tööriist, mis võimaldab spetsifikatsioone hallata läbi kasutajaliidese. Tööriist sisaldab eeldefineeritud kasutaja interaktsioone, mille abil spetsifikatsioonidesse automatiseeritavaid kasutusjuhte luua.

Protsessi hinnati kasutades näidisrakendust testitava süsteemina, millele loodi spetsifikatsioon kasutades väljatöötatud tööriista. Lisaks viidi läbi ühe osalejaga kasutajatestimine. Uus tööriist võimaldas spetsifikatsiooni luua, muuta ning käivitada ilma testkoodi kirjutamata, pakkus uusi haldusvõimalusi võrreldes tekstipõhiste DSL-idega, vähendades potentsiaalseid haldusprobleeme ning nõuete subjektiivset tõlgendamist. Vaatamata pakutud lähenemisviisi mitmetele eelistele, kehtestas see mitmeid uusi piiranguid.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 65 leheküljel, 10 peatükki, 24 joonist, 3 tabelit.

List of Abbreviations and Terms

API	Application Programming Interface
ARE	Agile Requirements Engineering
AT	Acceptance Testing
BDD	Behaviour-Driven Development
CLI	Command Line Interface
DSL	Domain Specific Language
ERD	Entity Relationship Diagram
HTTP	Hyper Text Transfer Protocol
IDE	Integrated Development Environment
RE	Requirements Engineering
SUD	System Under Development
SUT	System Under Test
TDD	Test-Driven Development
UI	User Interface

Table of Contents

1 Introduction **9**
1.1 Research motivation 10
1.2 Research goal 11

2 Background **12**
2.1 Requirements Engineering 12
2.1.1 Agile Requirements Engineering 13
2.2 Software Testing 14
2.2.1 Acceptance Testing 14
2.3 Test-Driven Development 15
2.4 Behaviour-Driven Development 15
2.4.1 Tools 17
2.4.2 Challenges 19
2.4.3 Challenges related to DSLs 20

3 Problem Statement **22**
3.1 Addressing challenges 22

4 Methodology **25**
4.1 Design Science 25

5 Objectives **27**

6 Design & Development **29**
6.1 Design overview 29
6.1.1 Main components 29
6.1.2 Main technologies 30
6.1.3 Client-server architecture 30
6.1.4 Data model 31
6.2 Clients 32
6.2.1 Manager client 32
6.2.2 Runner client 33
6.3 Behaviour ontology implementation 34

7 Demonstration **37**
7.1 Creating specification 37

7.1.1 User authentication 37
7.1.2 Project creation 37
7.1.3 Feature creation 38
7.2 Executing specification 42

8 Evaluation 48
8.1 Adoption of the tool in a task management application 48
8.2 User testing session 49
8.2.1 Feature creation 49
8.2.2 Demonstration of specification execution 51
8.2.3 Feedback and conclusion 51
8.3 Comparison to the DSL-based approach 52
8.4 Conclusion 53

9 Discussion 54
9.1 Advantages 55
9.2 Limitations 55

10 Conclusion 57

References 58

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis 61

Appendix 2 – Code samples 62

List of Figures

1 *Proposed specification creation process* 23

2 *Client-server architecture* 30
3 *ERD* 31
4 *Test execution sequence* 34

5 *Login view* 37
6 *Projects view* 38
7 *New project created* 38
8 *New project opened* 38
9 *New feature created* 39
10 *Feature details specified* 39
11 *Scenario created* 40
12 *Adding step* 41
13 *Steps created* 41
14 *Parameters tab view* 42
15 *Parameters created* 42
16 *Specifying a step parameter* 43
17 *Scenario parameters added* 44
18 *Combined behaviours creation* 44
19 *Combined behaviour added to scenario* 45
20 *Execution output - failure* 45
21 *HTML reporter on failure 1* 46
22 *HTML reporter on failure 2* 47
23 *Execution output - success* 47

24 *Hummus feature created during user testing session* 51

List of Tables

1 Behaviour representation table 35
2 Parameter attributes 35
3 Parameters table for SUT 43

1. Introduction

Software development methodologies have evolved significantly over the past few decades, with an increasing focus on improving collaboration among stakeholders, efficiency, and effectiveness in delivering high-quality software products.

Requirements Engineering (RE) facilitate communication, design, testing, and management in software projects, while supporting activities such as elicitation, validation, verification, and documentation [1, 2].

Agile RE (ARE) methods, which integrate requirements, design, implementation, and testing processes, rely on test cases that serve as requirements themselves in adapting to evolving needs [3, 2]. However, due to the emphasis on individual skills and knowledge in ARE, the process tends to be more informal [4].

Acceptance Testing (AT), is a software testing approach used to evaluate if software meets end-user needs and requirements, serving as the final validation before customer acceptance [5]. While ATs are used to demonstrate that a system fulfills the requirements, developing these tests from requirements specifications can be a subjective process [6].

Behavior-Driven Development (BDD), emerged from the difficulties encountered in Test-Driven Development (TDD) [7], provides an integrated approach to RE and testing activities, demonstrating its suitability for agile contexts through its its widespread adoption in the industry [8]. By using natural language and domain-specific terms to express software requirements, BDD enables stakeholders to define the expected behavior of a software system from the end-user's perspective, enabling direct translation into executable tests, while producing a "living" documentation and facilitating AT [8].

Steep learning curve has been identified as a significant barrier to BDD adoption [8]. Alongside several maintenance issues reported [8], the essential aspect of collaboration in BDD can be challenging and easily overlooked, leading to potential issues in the process [8]. Most BDD tools utilize Domain-Specific Languages (DSLs), such as Gherkin [9], which facilitate a common understanding of the specification. Due to the file-based nature of DSL, this leads to a process where informally captured requirements are handed to developers, who are accustomed to working with Integrated Development Environments (IDEs) and managing text-based files. Developers often need to translate informal requirments into

suitable DSL format, which might lead to subjective testing of the required system behavior and as a result, developers also often bear the responsibility of maintaining specifications. Additionally, DSLs are found to have low usability [10].

This thesis addresses the challenges in BDD by proposing and evaluating an open-source tool designed to simplify the process of capturing BDD artifacts using a web-based User Interface (UI), offering an alternative to the DSL-based approach for BDD. By providing an ontology of browser-based UI interactions, the tool seeks to facilitate the direct elicitation of testable requirements from less-technical stakeholders, removing the need for additional test code. The proposed approach eliminates the need to learn new DSL syntax and incorporates a test execution module that preserves the executable nature of the specifications, eliminating the need for developer test-suite maintenance. Consequently, the tool aims to improve overall software development effectiveness, lower the barriers to adopting the behavior-driven approach, promote enhanced collaboration among stakeholders and enable the execution of specification to ensure the software's conformation to requirements.

To develop and evaluate the proposed tool, the Design Science Research (DSR) methodology is employed, providing a structured approach to inventing and evaluating the effectiveness of the software system [11]. This thesis follows the six DSR activities as outlined by Peffers et al., which include problem identification, objective definition, design and development, demonstration, evaluation, and communication [11]. The viability of the proposed tool is evaluated through its implementation in a company case-study context, by measuring the conversion rate of existing requirements for the company's software, referred to as the System Under Test (SUT) and the succesfulness of test execution within the new system.

The remainder of this thesis is organized as follows: Chapter 2 provides an provide an overview of the key concepts and methodologies that form the foundation for this research. Chapter 3 presents the problem statement, outlining the challenges in the current state of BDD approach. Chapter 4 introduces the research methodology, describing the Design Science Research (DSR) process used to develop and evaluate the proposed tool. Subsequent chapters detail each step of the DSR process, culminating in the evaluation and communication of the research findings.

1.1 Research motivation

The motivation for this research arose from the need to address challenges of growth in complexity of software, experience during several software projects. A need for a more

structured approach to software specification and requirements management activities arose, as well as the need to improve and automate testing processes.

Existing approaches did not effectively connect requirements to testing activities, and there was constant hesitance in implementing automated testing due to concerns over management overhead of maintaining a large test suite in a rapidly changing environment. A more structured approach to linking specification to testing activities was needed.

Motivation comes from the need to bring testing practices closer to the requirements specification.

1.2 Research goal

The goal of this research is to develop a method for conducting BDD that eliminates the need for using DSLs and instead adopts a UI-based management system for handling BDD specification artifacts. This method focuses on leveraging granular user interactions as the basis for creating BDD scenarios, allowing stakeholders to author testable requirements directly. By utilising granular user interactions, offered by the system, the method aims to streamline the authoring of testable requirements and minimize the need for developer involvement in maintaining test suites. Additionally, the approach aims to offer additional management capabilities by enabling structured way of creating and updating BDD artifacts, accessible via a user-friendly interface. To facilitate the implementation of this method, an open-source tool is developed for managing and executing specifications.

2. Background

In the background chapter of this thesis, an overview of the key concepts and methodologies that form the foundation for this research, are provided. The chapter begins with an examination of Requirements Engineering (RE) and its evolution towards Agile Requirements Engineering, highlighting the shift in focus towards collaborative and iterative processes. Following this, an overview of software testing is given in the context of agile software development, outlining the concept of Acceptance Testing as a way to ensuring that a software system meets the needs and expectations of its users. Next, Test-Driven Development (TDD) is explored and its influence on software quality and maintainability discussed. Finally, Behaviour-Driven Development (BDD), a natural extension of TDD, is discussed, highlighting challenges in the BDD landscape.

2.1 Requirements Engineering

This section provides an overview of the characteristics associated with traditional Requirements Engineering (RE) processes, setting the stage for a comparison and overview of Agile RE and its implications for software development.

Requirements engineering (RE) plays a crucial role in software development, as it lays the foundation for all software products. RE involves identifying, modeling, communicating, and documenting the requirements of a system and its context of use [12]. It encompasses the process of gathering, analyzing, documenting, and managing requirements throughout the software engineering lifecycle, with the aim of interpreting and understanding stakeholders' goals, needs, and beliefs [13].

Requirements serve multiple purposes, such as facilitating communication among stakeholders, driving design and testing, and acting as a reference for project managers and system evolution [1]. They also support various requirements activities, including eliciting and validating stakeholders' requirements, software verification, tracing and managing requirements, and documenting customer agreements for contractual purposes [2].

Requirements Engineering serves as the foundation for software development by identifying, modeling, communicating, and documenting system requirements. However, the nature of traditional RE processes can be limiting in today's fast-paced development environments, necessitating adaptations and improvements to better suit evolving project

needs.

2.1.1 Agile Requirements Engineering

This section will discuss the core principles, methodologies, and implications of Agile RE, with a focus on the use of test cases as requirements and the role of automated acceptance tests in the software development process.

Agile Requirements Engineering (Agile RE) has emerged as a response to the challenges and limitations of traditional RE processes in fast-paced software development environments. By emphasizing flexibility, responsiveness to change, and the integration of requirements, design, implementation, and testing processes, Agile RE aims to address these issues and improve overall development practices [3].

In Agile RE, the main activities, such as elicitation, documentation, validation, negotiation, and management, are not distinctly separated, but rather intertwined [14]. A user-centric perspective is adopted, with requirements often presented in the form of user stories. Test cases play a central role in Agile RE, serving as requirements and providing both benefits and challenges in various aspects of the development process [2]. While acceptance tests are traditionally used to demonstrate that a system fulfills requirements, developing these tests from requirements specifications can be subjective and not always comprehensive [6]. Moreover, outdated requirements documentation can exacerbate the problem, leading to potential inaccuracies in acceptance tests [15]. Agile development addresses these challenges by employing automated acceptance tests, which drive implementation and document requirements in an executable format [16]. However, it is important to note that Agile RE heavily relies on the skills and knowledge of individuals, resulting in a more informal process [4].

In conclusion, Agile RE aims to offer a more flexible and responsive approach to handling software requirements in rapidly changing environments. The use of test cases as requirements and the adoption of automated acceptance tests are key aspects of Agile RE, promoting better integration between requirements and testing processes. However, the informal nature of Agile RE highlights the importance of individual skills and knowledge in the overall success of the approach.

2.2 Software Testing

Software testing is a fundamental aspect of agile software development, as it not only ensures the quality of the software but also enhances visibility, communication, and feedback among developers [17]. This section will discuss the core objectives, methodologies, and benefits of software testing in the context of agile development.

Software testing in agile context aims to facilitate continuous improvement and maintain a high level of software quality throughout the development process. It serves as a valuable measure for assessing the development process itself by monitoring the number of tests that pass or fail and conducting regression tests [18]. These tests enable developers to identify and address potential defects as soon as code changes are made, fostering a proactive approach to ensuring software quality.

2.2.1 Acceptance Testing

As a critical part of software testing, acceptance testing focuses on evaluating whether the software meets the end-user's needs, requirements, and business processes [5]. It serves as the final validation of the software's functionality and usability before it is accepted by the customer or end-user.

Automated acceptance testing is a method of streamlining the acceptance testing process by enabling customers or their representatives to express requirements as input to the software, along with expected results [17]. Unlike unit testing, which focuses on low-level components such as methods, acceptance tests are integrated at a higher level between the business logic and user interface or directly with the user interface. Automation of acceptance tests can reduce the time and cost associated with manual testing while improving the overall efficiency of the process.

Acceptance tests not only help ensure that the software meets its intended requirements but also serve as valuable documentation of the system's intended behavior [17]. By writing acceptance tests, developers can reflect on the design and system behavior before programming, resulting in a more robust and well-designed software product. Furthermore, the practice of expressing requirements in the form of acceptance tests has been shown to be well-received by developers [17].

In conclusion, software testing, and particularly acceptance testing, plays a vital role in ensuring the quality and functionality of software products in agile development.

2.3 Test-Driven Development

Test-Driven Development (TDD) is a fundamental aspect of eXtreme Programming (XP) proposed by Kent Beck [19]. TDD, also known as test-first programming, mandates developers to create automated unit tests as assertions to define code requirements before writing the code itself. This approach allows developers to evolve systems through cycles of testing, development, and refactoring [7].

However, several factors limit the industrial adoption of TDD. One such factor is the increased development time due to the iterative nature of the process and the time spent on writing tests [7]. Additionally, developers may have insufficient TDD experience or knowledge, hindering its effective application. The lack of practical experience or theoretical insight can result in suboptimal test cases and inadequate code coverage.

TDD also emphasizes a minimal up-front design, which can lead to insufficient design and a need for frequent refactoring to maintain architectural quality. Moreover, developers may possess inadequate testing skills, negatively affecting the efficiency and effectiveness of the automated test cases. Insufficient adherence to the TDD protocol, such as not following the established guidelines or not creating and executing test cases before writing code, can further impede the successful implementation of TDD. Additionally, domain-specific and tool-related limitations, such as difficulties in automated testing of graphical user interfaces (GUIs), can also pose challenges for TDD adoption. Finally, the presence of legacy code in an organisation can create obstacles, as it often represents years of development efforts and investments, serving as a backbone for both existing and future products [7].

These limitations of TDD contributed to the development of Behaviour-Driven Development (BDD), which aims to address some of these issues by focusing on the behavior of the system and improving communication between stakeholders. BDD employs a domain-specific language that is easily understood by both technical and non-technical members, thus enabling a better understanding of the system requirements.

2.4 Behaviour-Driven Development

BDD is an increasingly prevailing agile development approach and has gained attentions of both research and practice. It was originally developed by Dan North as a response to the conceptual difficulty of specifying tests before implementation [20, 7]. BDD is focused on defining fine-grained specifications of the behaviour of the targeting system, in a way that they can be automated [20]. The use of test cases as specifications placed this methodology

into the category of acceptance test-driven development methodologies (e.g., story-driven development, specification driven development) [21] to bridge the gap between customer's business needs and technical aspects of software development.

BDD approach incorporates aspects of requirements analysis, requirements documentation and communication, and automated acceptance testing [2]. Solis and Wang [22] highlights five fundamental characteristics that capture the overall essence of BDD:

1. **Ubiquitous language**: A common language that comes from a domain model and helps customers and developers speak the same language without ambiguity. It should be used throughout the development lifecycle.
2. **Iterative decomposition process**: Starts with identifying the expected behaviors of a system, which are then broken down into feature sets and finally realized by user stories. This process should be iterative and involve barely enough up-front analysis.
3. **Plain text description with user story and scenario templates**: BDD uses pre-defined templates for specifying features, user stories, and scenarios, which are written using a simple ubiquitous language. Templates help in providing a clear structure for user stories and scenarios.
4. **Automated acceptance testing with mapping rules**: BDD includes automated acceptance testing, which verifies the interactions or behaviors of objects rather than their states. Mapping rules are used to map scenarios to test code.
5. **Readable behaviour oriented specification code**: Code should be readable, with the specification being a part of the code. The names of classes and methods should be written in sentences and in the project's ubiquitous language, describing the behaviors of objects.
6. **Behaviour driven at different phases**: BDD happens at different phases of the software development process. BDD starts with business outcomes, then features, and finally moves to the implementation phase, where testing classes are derived from scenarios and their names follow mapping rules.

BDD is in active use in the industry used by many software teams [8] to allow them to capture the requirements for software systems in a form that is both readable by their customers and detailed enough to allow the requirements to be executed to check whether the production code implements the requirements successfully or not. According to [8] the main benefits of BDD was reported to be the usage of domain specific terms, improving communication among stakeholders, the executable nature of BDD specifications and facilitating comprehension of code intention. The resulting feature descriptions, as sets of concrete scenarios describing units of required behaviour, provides a form of living documentation for the system under construction.

In modern development practices, BDD is frequently associated with Domain Specific Languages (DSLs) tools like Gherkin. DSLs are “computer programming languages of limited expressiveness focused on a particular domain” as defined by Fowler [23]. DSLs are aimed to facilitate construction of software artifacts through specialized abstractions and notations [24], and are increasingly being used in many software engineering activities, including designing and checking architectural rules [25].

Gherkin is used to describe the desired software behavior through a collection of example interactions with the system, expressed using natural language sentences arranged around a "Given-When-Then" structure as demonstrated in 1. These interactions are then executed using automation frameworks, such as Cucumber, SpecFlow for C#, and Behave for Python, which all support Gherkin syntax. In addition, a browser automation library such as Selenium WebDriver can also be used in combination with these frameworks to interact with web applications through various browsers, as demonstrated in code sample listing 3 using Python with Behave and Selenium packages.

```
Feature: User authentication
  As a registered user
  I want to log in to the website
  So that I can access my account

  Scenario: Successful login
    Given I am on the login page
    When I enter my valid username and password
    And I click the log in button
    Then I should be redirected to my account dashboard
    And I should see a welcome message
```

Listing 1. Gherkin example

2.4.1 Tools

In the realm of open-source and commercial tools for executing BDD-based specifications, there are numerous options available for various programming languages. However, there are not many alternatives to Gherkin for specifying behavior. One such alternative mentioned in existing literature is the Robot language, which is incorporated in the Robot Framework [26].

Most open-source BDD tools are based on Gherkin and do not come with a user-interface offering additional capabilities, compared to text-based DSL. Nevertheless, some tools can be used in conjunction with other platforms to enhance collaboration and comprehension

of specifications. Examples of such tools include:

1. **Serenity BDD**: Serenity BDD is an open-source library for writing and executing BDD tests. While it does not include a UI for creating and managing scenarios, it generates comprehensive, living documentation in the form of an interactive web-based report. This report can be shared with team members, fostering collaboration and keeping everyone on the same page. [27]
2. **Pickles**: Pickles is an open-source living documentation generator that works with various BDD tools, such as SpecFlow, Cucumber, and Behat. Although not a BDD tool itself, Pickles takes the output of your BDD scenarios and generates a web-based documentation site that can be used for easy visualization, sharing, and collaboration. [28]
3. **BDDFire**: BDDfire is a Ruby-Cucumber BDD framework aimed at automating mobile and web applications. It generates a default toolkit around BDD and integrates with various popular open-source libraries to provide a range of testing capabilities. BDDfire includes pre-defined steps for browser, accessibility, and API testing, which contribute to its utility in handling different aspects of mobile and web application testing within a BDD context. [29]

While these open-source options may not provide a fully-featured UI for creating and managing BDD scenarios, they can be used in conjunction with other web-based tools for collaboration and mroe efficient BDD process.

In addition to the open-source tools, there are several commercial tools available:

1. **Behave Pro**: Behave Pro focuses on integration with Jira and is based on Gherkin for writing BDD specifications. Jira integration enables collaborative environment for working with BDD scenarios. [30]
2. **CucumberStudio**: Initially called HipTest, CucumberStudio was acquired by SmartBear and integrated into their product offering. It provides a collaborative BDD platform that helps teams define, execute, and maintain their BDD specifications using Gherkin language. [31]
3. **SpecFlow+ LivingDoc**: SpecFlow+ LivingDoc is an extension for SpecFlow that generates living documentation from your BDD scenarios. Although the tool itself does not include a web-based interface, it is based on Gherkin language and creates interactive, living documentation within the IDE. [32]

These commercial tools provide some additional capabilities and ways of engaging with BDD specifications, most of them leveraging Gherkin or other DSLs to streamline the

behavior specification process. While CucumberStudio stands out as the most comprehensive among the aforementioned tools, providing a collaborative BDD platform that assists teams in defining, executing, and maintaining their BDD specifications, its commercial nature may present certain limitations. These may include cost constraints, limited customisability, and potential vendor lock-in. In such cases, the need for an open-source alternative becomes apparent, as it would offer greater flexibility, customisation options, and adaptability to specific project requirements without the burden of licensing fees or restrictions. Moreover, an open-source tool fosters a community-driven approach, encouraging continuous improvement and innovation.

2.4.2 Challenges

The management of BDD specifications is reported to be challenging, particularly when they grow beyond a handful of features and involve multiple development team members writing and updating them over time. This can lead to redundancy, resulting in bloated BDD specifications that are more costly to maintain and use [8]. The main drawbacks include challenges in adapting to changes in software development methods, similar maintenance challenges to those encountered in any automated test suite, and a scarcity of tools supporting the maintenance and evolution of BDD specifications [8].

In their study of BDD tools in open-source projects, Zampetti et al. [33] found that developers often use BDD frameworks for unit testing activities rather than strictly applying BDD. They tend to write tests during or after coding, perceiving BDD as effort-prone and requiring more than just framework adoption. Moreover, the majority of BDD-supporting tools necessitate composing tests using low-level events and components, which only become available once the system has been implemented. Consequently, BDD tests face challenges in terms of reusability across various artifacts and different versions of the system [34].

Furthermore, Silva, Hak, and Winckler [34] observed recurring patterns of low-level behaviors and semantic inconsistencies in BDD specifications during the development of e-Government applications. They explored the use of a formal ontology for defining pre-established behaviors that could be employed to specify scenarios. While Lenka, Kumar, and Mamgain [9] presented several BDD tools, they also highlighted that BDD's effectiveness is limited when developers or testers alone are responsible for testing and maintaining specification.

In conclusion, the main challenges in managing BDD specifications are the redundancy and maintenance costs [8], difficulties in strictly applying BDD [33], reusability challenges

across artifacts and system versions [34], and limited effectiveness when collaboration aspects are ignored [9].

2.4.3 Challenges related to DSLs

Several studies have concluded that he difficulties of using DSLs have become more apparent when exposed to software maintenance circumstance. One important factor that contributes to increased maintenance effort is the low usability of such DSLs [10]. The usability of a DSL artifact (e.g., a specification built using the DSL) is the quality that makes it easy for users to understand, learn, and interact with it [24, 10].

From the observations of Micallef and Colombo [35] the use of Gherkin as DSL for test automation initially led to improved communication among team members. However, issues arose as the number of test scenarios increased. The loose grammar allowed for organic language growth, which resulted in substantial duplication and inconsistency. For example, "Given I log in to the system," "Given I log on to the system," and "Given I log in correctly" all express the same notion and user actions. Similarly, some team members condensed sequences of actions into one, while others used a longer atomic format, for example:

> Given I log in and purchase a product

Was also be expressed as:

> Given I log in
> And I search for a product
> And I select the first item in the list
> And I add the item to my shopping cart

The problem was further compounded when product specifications changed. In one case, the perceived cost of maintaining the language and automated tests became so high that the project was nearly abandoned as technical testers were reassigned to manual testing jobs to meet deadlines.

Key lessons learned the study of Micallef and Colombo [35] include the need for a dedicated language owner to ensure consistency, a development process that accommodates DSL development, tool support for language lookup as the language grows, and essential management buy-in to prevent competition between DSL development and software

delivery.

3. Problem Statement

The BDD approach has significantly impacted the software development industry by facilitating a common understanding of specifying software requirements among stakeholders [8]. BDD emphasizes the expression of software specifications in natural language and domain-specific terms, focusing on the software's intended behavior from the end-user's perspective. The executable nature of these requirements specifications help confirm correctness and identify problematic software behavior.

The current state of BDD tools relies on DSL, such as Gherkin or Robot, for writing specifications [9]. Difficulties of using DSLs become more evident during software maintenance, with low usability being a significant factor, contributing to the persistance of maintenance challenges related to BDD specifications [10, 8].

DSLs files typically reside within the codebase and most language-support tools are designed for use in IDEs, which might cause struggles for non-technical stakeholders to participate in the process, hindering collaboration and leading to limited effectiveness. The need to learn DSL syntax also contributes to an increased learning curve, potentially discouraging BDD adoption.

In many cases, especially in agile methodologies, software requirements are initially captured in an informal notion [4]. For BDD, this often involves translating requirements into a suitable DSL format, such as Gherkin, leaving the developer responsible to efficiently manage specifications. This process introduces a degree of subjectivity in testing, which may not guarantee that the requirement is fulfilled even when the corresponding test passes. As a result, inconsistencies between requirements and their implementation may arise, leading to potential issues in the final software product.

3.1 Addressing challenges

This thesis proposes and evaluates a method for conducting BDD by introducing a UI for streamlined capture and management of BDD artifacts, such as features, scenarios, steps and step parameters.

The process includes selecting scenario step behaviour from available low-level behaviours, offered by the system. Figure 1 illustrates the proposed process of creating a BDD feature

through UI. With this approach, a scenario for the user story of "As a user, I want to log in to the system" could be represented with the following steps:

> I am on "login page"
> I fill the "username field" with "my username"
> I fill the "password field" with "my password"
> I click the "login button"
> I see "my name" in "navigation bar"

In this example, the terms wrapped in parentheses, such as "username field," can be referred to as step parameters. By utilising these parameters, it becomes possible to reuse atomic behaviors across different scenarios. In case the login status is required in further scenarios, these steps could be combined into a single step, such as "I am logged in as a customer" and later selected into scenarios when needed.

This approach aims to promote reusability and consistency by allowing stakeholders to select available steps and parameters for scenarios, potentially eliminating issues related to loose grammar. Furthermore, it seeks to enable the creation of concise scenarios by allowing the combination of low-level steps into a single step when necessary.

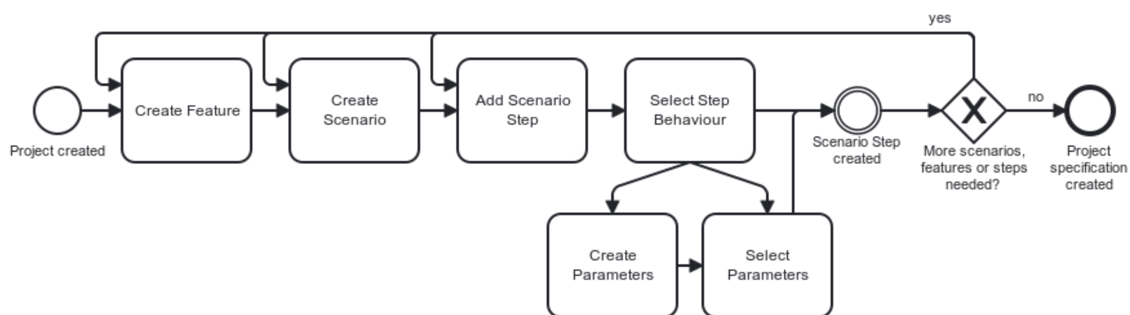

Figure 1. *Proposed specification creation process*

The proposed approach includes pre-written test-code for low-level behaviors used in scenarios. By providing the test-code, this approach aims to eliminate the need for developers to write additional tests themselves and allow for direct elicitation of testable requirements. Additionally, this approach eliminates the risk of subjective interpretation of requirements, which enables developers to focus on executing the specification and writing application code to make it pass, promoting the strict application of BDD principles, such as having test cases before implementation.

Furthermore, the proposed approach aims to eliminate the need for stakeholders to learn a DSL syntax, making the adoption of the behavior-driven approach more accessible

and convenient. This, in turn, seeks to foster better collaboration among stakeholders, ultimately aiming to improve the effectiveness of software development processes.

In conclusion, the UI-based approach for conducting BDD proposed in this thesis seeks to address the key challenges associated with BDD by providing a more accessible, reusable, and collaborative environment for managing specifications. By streamlining the process of capturing and maintaining testable requirements, this approach aims to promote the adoption of BDD principles and reduce effort in specification and test-suite management, potentially leading to more efficient development process, higher-quality software products and improved collaboration among stakeholders.

4. Methodology

This chapter describes the method used to develop a software system, supporting the proposed method of conducting BDD described in 3.1, further on referred to as System Under Development (SUD).

The principles of Design Science Research (DSR) process are applied to deliver the SUD. The usefulness of SUD is evaluated by its adoption in requirements management and testing activities of the System Under Test (SUT), presented in chapter 7.

4.1 Design Science

Design science research (DSR) is a scientific methodology used to create knowledge in the form of information systems that are useful, usable and desirable. In short, DSR is a research that invents and evaluates technological artifacts [11].

This research is structured following the 6 activities of DSR process defined by Peffers et al. [11]. The six activities are explained below:

1. **Problem Identification.** The first step of the DSR process involves the identification of a problem that needs to be addressed. Consequently, after the problem is identified, there remains the step of determining the performance objectives for a solution.
   In this thesis, the problem is captured in chapter 3.
2. **Objective Definition.** The objective definition stage is the second step of the DSR process. The purpose of this step is to define the goals that must be achieved to solve the problem identified in the previous step. The objective definition should be specific and measurable, so that the success of the research and the developed artifact can be evaluated, to ensure whether the research objective is achieved, and the proposed solution is effective.
   In this research, the detailed objectives and requirements for the SUD are defined in chapter 5.
3. **Design & Development.** The third step of the DSR process involves designing and developing the proposed software artefact to solve the defined problem. This activity includes determining the artifact's desired functionality, its architecture and the development of that artifact. This step requires careful consideration of the goals that have been set in step two, as well as the existing literature that has been reviewed

previously.

In this research, the software design and development is captured in chapter 6.

4. **Demonstration.** The fourth step of the DSR process involves demonstrating the proposed solution to solve one or more instances of the problem.

   In this step, the capture of SUT specification is demonstrated using the SUD, and the corresponding testing of these requirements is conducted and results portrayed in chapter 7.

5. **Evaluation.** The fifth step observes and measures how well the designed artefact supports a solution to the problem.

   In this research, the viability of the solution is evaluated by examining its adoption in chapter 7 and conducting a single user testing session. The evaluation results are presented in chapter 8.

6. **Communication.** The sixth and last activity consists of communicating the previous activities and their outputs. In this thesis, chapter 9 discusses and summarises the research by discussing the strengths, weaknesses and limitations of the proposed approach.

5. Objectives

The objective section of the DSR process outlines the primary goals and specific objectives that the research aims to achieve. These objectives guide the development of the proposed solution and provide a roadmap for evaluating its success.

The primary goal of the proposed tool is to streamline the process of capturing and managing testable specifications through UI, to reduce the effort involved in test-suite management and minimize the subjectivity associated with testing against requirements.

To achieve this goal, we have divided it into two primary objectives and their corresponding requirements (REQ). The primary objectives are focused on the two main components of the system, while the requirements describe the necessary features and functionality for each objective.

Objective 1: Enable capturing and management of requirements specifications through UI

The first primary objective is to develop a web application that provides an intuitive user interface for stakeholders to capture, manage and store behavior-driven requirements specifications.

REQ 1 **Specification management**

The system should enable user view, create and modify specification artifacts via a user-friendly interface.

REQ 2 **Access control**

The system should have authentication mechanisms in place to restrict unauthorized access and ensure secure collaboration.

REQ 3 **Predefined ontology of browser-based user interactions**

The system should provide a predefined ontology of browser-based user interactions enabling non-technical stakeholders to author testable requirements without necessitating supplementary test code.

Objective 2: Enable execution of specifications

The second primary objective is to develop a specification execution module that automates the execution of the scenarios in the specification, streamlining the test execution process and minimizing the effort for test-suite maintenance.

REQ 4 **Execution predefined behaviours**

The system should be able to read and execute the scenarios based on predefined behaviours.

Additionally, another requirement is added to emphasize the extensibility aspect of the tool:

REQ 5 **Extensibility of predefined behaviors**

The system should allow users to extend the predefined ontology of behaviors to address specific requirements, thereby enabling a higher level of customization and adaptability in different project contexts.

6. Design & Development

In this chapter the design process for the System Under Development (SUD) is outlined, focusing on the architectural and structural components that will enable the system to meet the objectives and requirements stated in the previous chapters.

The SUD is designed to be used by the all members of a software development team, facilitating collaboration and effective communication of requirements throughout the development process. As an example potential users, we consider 3 types of team members, each with distinct responsibilities and involvement in a software development process:

Product Owner (PO) or Product Manager (PM): Responsible for describing the requirements of software to be developed and creating specifications. They work closely with stakeholders to gather and prioritise requirements, ensuring that the vision for the product aligns with the needs of the business and the end-users.

Developer: Responsible for developing the system and ensuring that the development is in conformance with the requirements. Developers translate specifications into functional code and continuously refine the product through iterative development cycles.

Designer: Designers focus on creating user interfaces that facilitate interactions between users and the system. They work closely with both the PO or PM and developers to ensure that the design aligns with the requirements and is feasible for implementation. Designers are mainly interested in viewing the specification in order to understand the use-cases for designing the UI of the specified system.

6.1 Design overview

The proposed SUD will be given a name - Hummus. This section provides an overview of the main components and architecture of the Hummus system.

6.1.1 Main components

Hummus consists of several main components:

- **Manager** - A web-based client used for managing specifications.

- **Runner** - A Command Line Interface (CLI) client for executing the specifications.
- **API** - A server-side component facilitating the exchange of data between the clients and the database.
- **Database** A database for storing data entities related to the specification.

6.1.2 Main technologies

The primary programming language used for both the client and server components is TypeScript, running on the Node.js runtime environment. TypeScript is a strongly-typed superset of JavaScript, providing enhanced safety and maintainability.

For the web-based user interface of the Manager component, React framework is utilised. React is a popular and widely-used library for building user interfaces.

The database technology chosen for storing BDD-related data entities is relational database management system MySQL.

For executing the specification and running the tests, Playwright is used. Playwright is a an open-source library for automating testing of web applications across multiple browsers.

6.1.3 Client-server architecture

The client-server architecture is used to enable a clear separation of responsibilities between the clients and the data processing components. In this architecture, the client side is responsible for presenting the data and handling user interactions, while the server side manages the data storage and processing. This separation ensures that the BDD artifacts are maintained centrally, allowing multiple clients to access and interact with the same set of specifications.

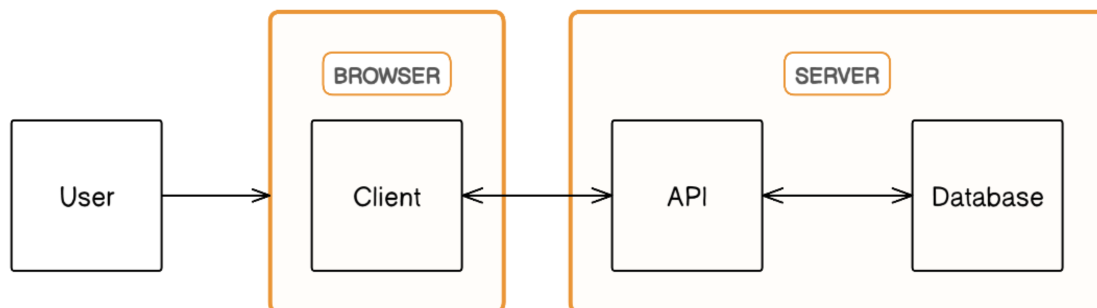

Figure 2. *Client-server architecture*

In the context of Hummus development, two distinct clients are employed: the browser-

based Manager and the CLI-based Runner. Both clients interact with the API to exchange data, utilising HTTP connections for communication.

6.1.4 Data model

To store a specification, a database is required to hold the specification entities. Thus, a data model is introduced for managing the entities. Figure 3 illustrates the data model using an Entity-Relationship Diagram (ERD).

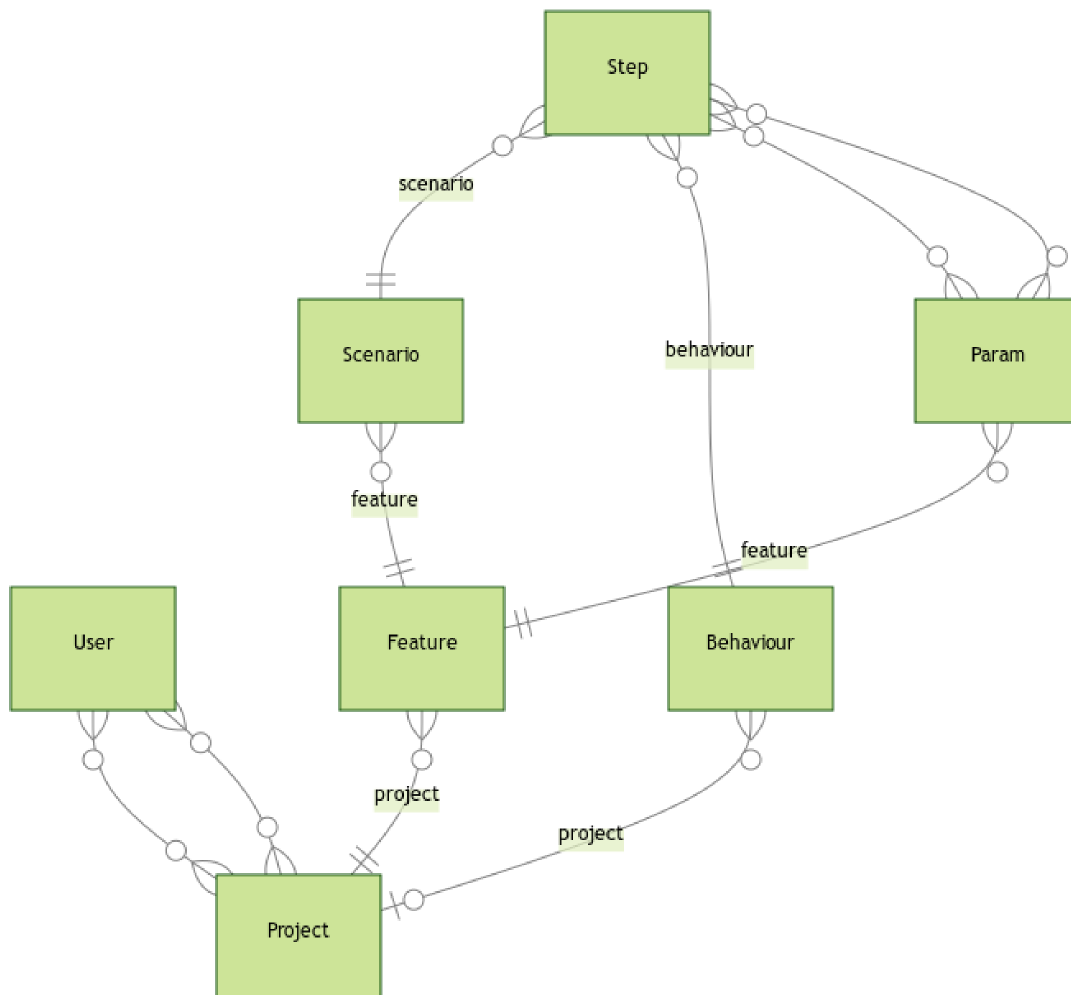

Figure 3. *ERD*

The following descriptions provide an overview of each entity and their relations represented in the diagram:

1. **User**: Represents an individual user of the system. Each user has a unique username, password, and access token. A user can be associated with multiple projects.
2. **Project**: Represents a project that contains features and behaviors. A project can be

associated with multiple users, features, and behaviors.

3. **Feature**: Represents a feature within a project. Each feature has an optional title and description, and can be associated with multiple scenarios. A feature belongs to a specific project.
4. **Scenario**: Represents a scenario within a feature. Each scenario has an optional name and description and can be associated with multiple steps. A scenario belongs to a specific feature.
5. **Step**: Represents a step within a scenario. Each step can be associated with multiple params and a behaviour. A step belongs to a specific scenario.
6. **Behaviour**: Represents a behavior within a project. Each behavior has text value and can be associated with multiple steps.
7. **Param**: Represents a parameter within a feature. Each parameter has a unique name and value, and a type. A parameter can be associated with multiple steps and belongs to a specific feature.

6.2 Clients

In this section, we will describe the two clients developed for the Hummus system - Manager and Runner.

6.2.1 Manager client

The purpose of the Manager is to enable the process of capturing and managing specifications. In the example development team provided in chapter, all team members would be interested in using the Manager client for viewing or creating the specification. The resulting Manager client exposes the following views to its users:

Authentication view

In authentication view, the user of the Manager is able to log into the system or register an account.

Projects listing view

After the authentication has been successful, the user is directed to the projects listing view that lists the projects user has created. In this view, the user can select an existing project or create a new one.

Project view

Upon selecting a specific project, users are directed to the project view, where they can create or view specifications related to the project. This view lists the features within the project, along with a button for creating new features. When a feature is created and selected, users can access the following tabs: Feature tab, Parameters tab, Behaviours tab and Options tab.

- Feature tab - displays and enables the creation or modification of the feature title, description, scenarios and scenario steps.
- Parameters tab - displays and enables the creation or modification of parameters for usage in scenario steps.
- Behaviours tab - displays existing behaviors that can be used in scenarios, and allows the creation of new behaviors combining multiple existing behaviors.
- Setting tab - displays the deletion button of a feature, as well as potential configuration options for features in the future.

6.2.2 Runner client

The purpose of Runner client is to execute a specification specified via the Manager. In the example development team provided, a developer would be the primary user interested in using the Runner client to execute the specification.

Runner aims to be installable via a Node.js package manager and can be invoked via the command line. Runner retrieves the specification via HTTP request and generates tests based on the response. It uses Playwright for the execution of the tests. When the run command is invoked, the following sequence of steps occur:

1. Load and read the configuration file - in order to retrieve the specification, authentication credentials and project identifier must be provided in the configuration file. An example of the configuration file is show in in listing 6.
2. Retrieve project specification - in case configuration file is specified correctly, the project specification is fetched.
3. Generate test files - given the specification is retrieved, the tests will be generated.
4. Running the tests - after successful generation of tests, Playwright will be executed to run the tests.

Figure 4 represents the specification retrieval and execution sequence. Listing 5 represents the Runner invocation process.

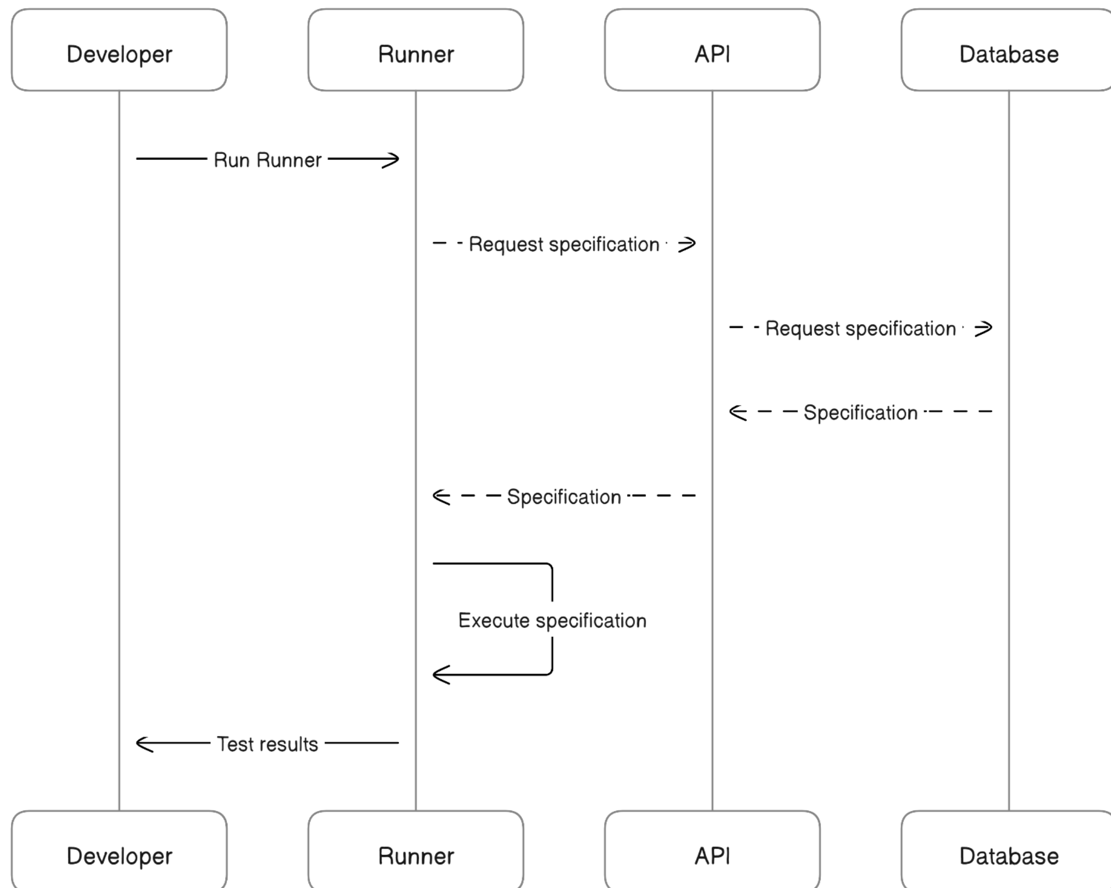

Figure 4. *Test execution sequence*

6.3 Behaviour ontology implementation

Silva, Hak, and Winckler [34] present a behavior-based ontology designed for test automation to help validate functional requirements when building interactive systems. The ontology acts as a common vocabulary to map user behaviors to interaction elements in the UI, enabling automated testing. It also improves the way teams write requirements for testing purposes, allowing for the reuse of described behaviors in natural language and providing test automation with minimal effort.

Manager client enables creation of scenarios into a feature. Table 1 represents the user behaviours implemented into the Hummus system, which, are made available in the Manager for usage in scenario steps. The Parameters column represents the types of parameters needed for a behaviour to be executable. The Test Function column represents corresponding function invoked, when the step is executed. A parameter consists of a name, value and type attribute. Table 2 describes the attributes of a parameter.

The test function takes behaviour parameters as function arguments. Additionally, the

Behavior	Parameters	Test Function
I click the "selector"	selector	clickElement
I fill the "selector" with "text"	selector, text	inputElementValue
I go to "location"	location	navigateToLocation
I am directed to "location"	location	verifyDirectedToLocation
The "selector" is visible	selector	verifyElementVisibility
The "selector" is not visible	selector	verifyElementVisibility
The text "text" in "selector" is visible	selector, text	verifyElementContainsText
The text "text" in "selector" is not visible	selector, text	verifyElementContainsText

Table 1. Behaviour representation table

Attribute	Description
Name	The name displayed in the scenario step.
Value	The value used by the test function.
Type	Determines the parameter's compatibility with scenario steps.

Table 2. Parameter attributes

current page object from Playwright is passed into the function, to make it possible to assert and interact with elements on the current page. Listing 4 represents the functions invoked for clicking and inputting behaviours. Within these functions, an element is retrieved by the selector and corresponding interactions are invoked using the Playwright test library.

Using these behaviours, an example scenario for user authentication feature could be constructed with the following steps:

1. I am on "login page"
2. I fill the "username field" with "my username"
3. I fill the "password field" with "my password"
4. I click the "login button"
5. I am directed to "dashboard page"

As a result, when this scenario is executed by Playwright, the following actions occur:

1. Playwright opens the browser with the value of "login page" parameter, for example "/login"
2. Playwright selects the "username field" by its parameter value and fills it with the "my username" value
3. Playwright selects the "password field" by its parameter value and fills it with the "my password" value
4. Playwright selects the "password field" and clicks on it

5. Playwright waits for the page to become stable and expects the URL of the page to be the "dashboard page" parameter value, for example "/dashboard"

In case all the interactions complete successfully, the scenario test will pass.

7. Demonstration

In this section, the process of capturing and executing a software specification is demonstrated using the developed system. For the demonstration, a task management application will be used as a SUT. The specification for SUT will captured, executed, and the results will be portrayed, showcasing the overall functionality and utility of the proposed approach.

7.1 Creating specification

In this section, we will demonstrate the process of creating a specification using the Hummus Manager client. In the example development team provided in chapter 6, a PO or PM would be the main actor interested in creating the specification. However, this is something all team members can be involved in.

7.1.1 User authentication

To start using Hummus, users must first authenticate themselves by logging into the system. This ensures that the user has the necessary permissions to create or manage specifications within the system. User authentication view is displayed on 5

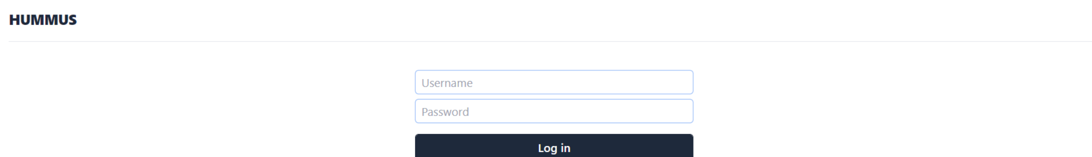

Figure 5. *Login view*

7.1.2 Project creation

After successful authentication, users can create a new project or select an existing one. Projects serve as containers for features and other related specification entities.

Project can be created by inserting a project name into the corresponding field and clicking the "Create" button. Screenshot 6 displays the projects view before, and screenshot 7, after a successful creation of a project "Task Management App".

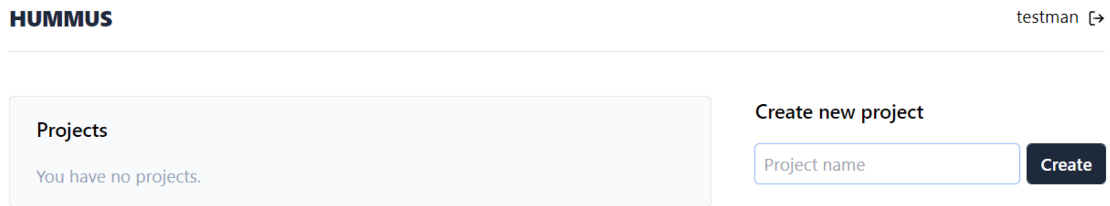

Figure 6. *Projects view*

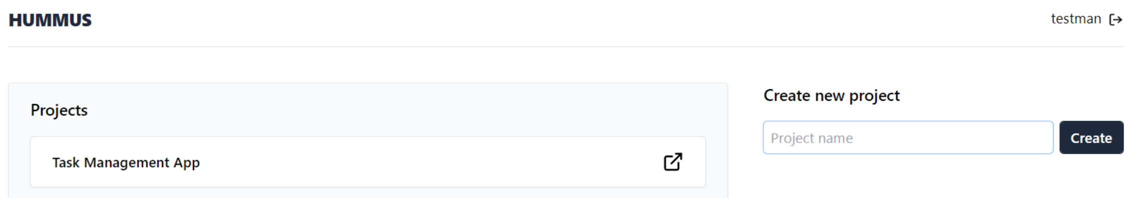

Figure 7. *New project created*

When a project exists in the system, clicking on the project directs the user to a specific project view as displayed on screenshot 8

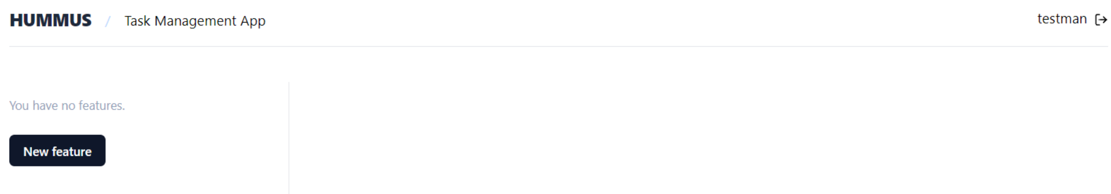

Figure 8. *New project opened*

7.1.3 Feature creation

Once a project has been created and selected, users can create new features into the project. Features represent high-level requirements or functionalities of the system under development. Each feature consists of one or more scenarios.

A feature can be created by clicking the "New Feature" button. Once created the feature appears with corresponding fields for specifying additional feature details, such as feature name and description, as displayed on screenshot 9. Once the details are specified, the new details will be saved as displayed on screenshot 10.

Scenario creation

Scenarios are created within a feature to describe specific situations or use cases. They contain a series of steps that define the expected behavior of the system under test.

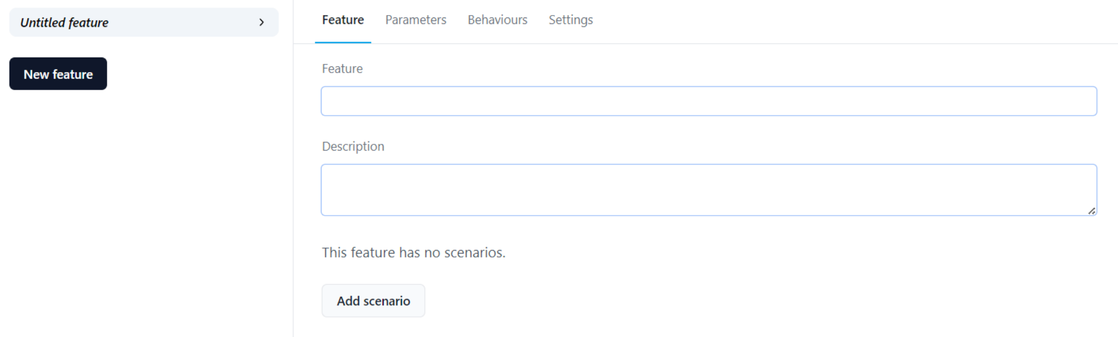

Figure 9. *New feature created*

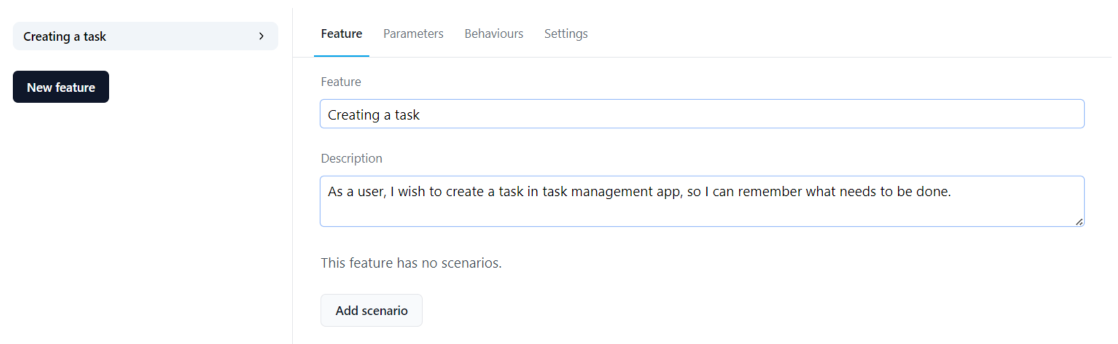

Figure 10. *Feature details specified*

Scenarios can be created by clicking the "Add Scenario" button. As a result, a field for specifying the scenario name and a button for adding a scenario step become available. Screenshot 11 displays the result after creating a scenario and specifying its name.

Scenario step creation

For each scenario, users define the necessary steps to describe the expected behavior.

Steps are selected from a list of available low-level behaviors provided by the system. Additionally, steps that are a sequence of multiple behaviours can be created and selected, as described in subsection 7.1.3.

Screenshot 12 displays the selection of behaviours available when adding a step. Screenshot 13 displays the resulting scenario, once multiple steps are added, after clicking the "Add step" and selecting the behaviour. As a result, the steps are displayed under the scenario. Additionally, the steps can be deleted or the order changed when hovering over a specific step with a cursor.

Figure 11. *Scenario created*

Parameters creation

Parameters can be created and used within scenario steps to provide additional context. This allows for greater flexibility and reusability of steps across different scenarios.

After adding one or more steps to a feature, the steps may require parameters that have not yet been created. In this case, the missing parameters are highlighted in red within the steps, indicating that the necessary parameters have not been added, as can be seen on screenshot 13.

In order to create a parameter, the user must select "Parameters" from the top of the feature view. As a result, a view for managing parameters is opened, as shown in screenshot 14. This view includes a list of created parameters and a form for creating a new parameter. To create a parameter, its attributes, as described in table 2, must be specified. After a successful creation of parameters, they are listed and sorted by their type attribute as displayed on screenshot 15. Table 3 describes the parameters created for the SUT.

When a parameter is created, it can be used in a scenario step by clicking on the parameter type in the step, as displayed on screenshot 16. In screenshot 17, a finalised scenario with the added parameters is shown.

Combined behaviors creation

Users can also combine multiple low-level behaviors into a single step. This contributes to making scenarios more concise, easier to understand and shorter.

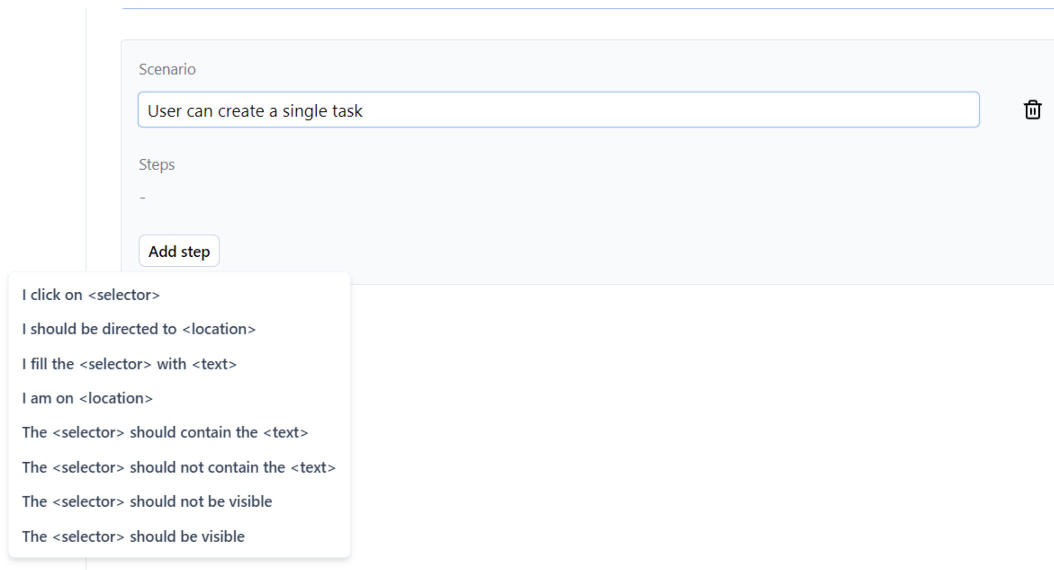

Figure 12. *Adding step*

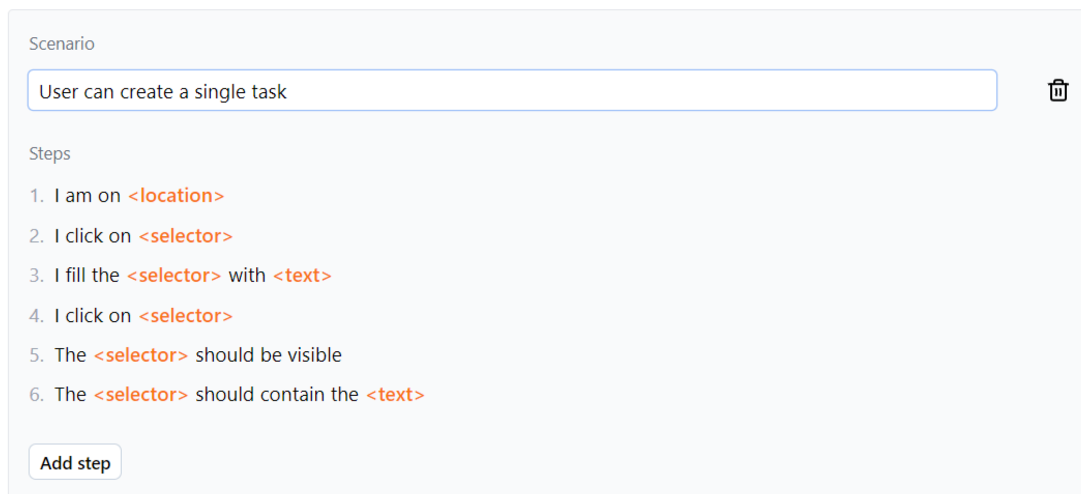

Figure 13. *Steps created*

Users can access the dedicated view for creating combined behaviors by selecting "Behaviours" from the top of the feature view. Similar to the parameters creation view, a form for creating new behaviors is displayed, and the existing ones are listed. The process of creating combined behaviors is similar to creating a scenario: insert a value for the behaviour, add a step, select a behavior, and then select a parameter. On screenshot 18, the process of creating a combined behaviour is seen, as well as the list of existing ones already in the system. As a result, a more concise scenario can be created, as displayed in screenshot 19.

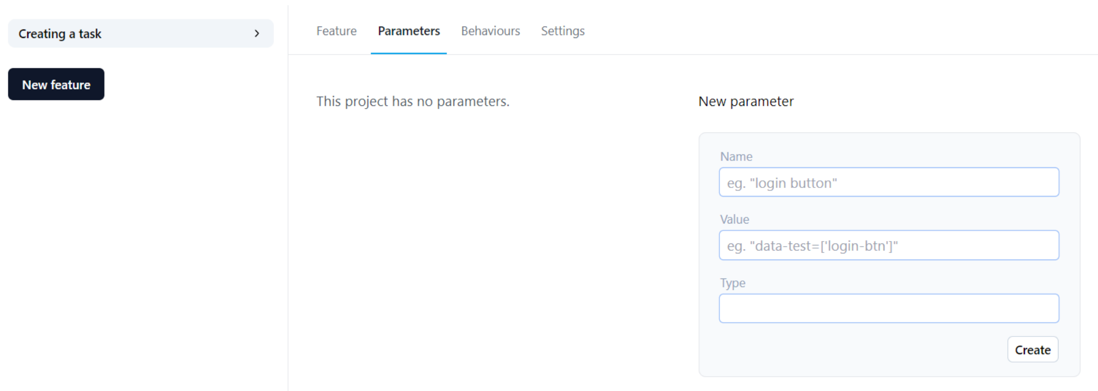

Figure 14. *Parameters tab view*

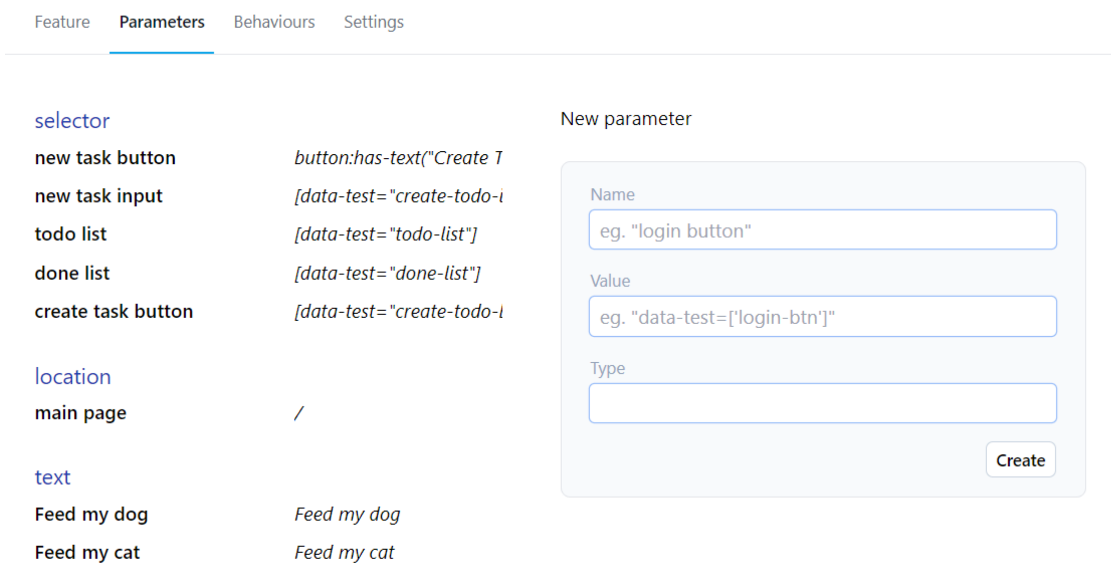

Figure 15. *Parameters created*

7.2 Executing specification

In this section, we will discuss the process of executing the created specification using Hummus.

To execute the specification, users need to provide a configuration file containing authentication credentials and project identifier in. This allows Hummus to retrieve the project specification and generate the appropriate test files.

Once the Runner package has been installed via package manager and configuration file has been set up, the specification can be retrieved and executed by running the command

Name	Value	Type	Description
new task button	button:has-text("Create Todo")	selector	Button for creating a new task
new task input	[data-test="create-todo-input"]	selector	Input field for entering a new task
create task button	[data-test="create-todo-button"]	selector	Button to confirm task creation
todo list	[data-test="todo-list"]	selector	List of tasks to be done
done list	[data-test="done-list"]	selector	List of completed tasks
main page	/	location	Main page of the application
text	Feed my dog	text	Example text for a task
text	Feed my cat	text	Another example text for a task

Table 3. Parameters table for SUT

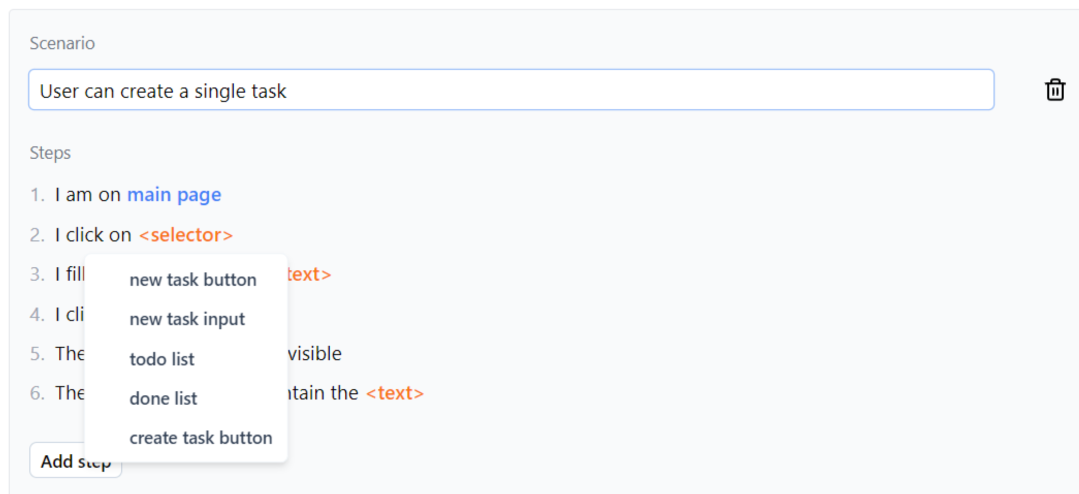

Figure 16. *Specifying a step parameter*

`hummus run`. The execution process and results are then displayed, once completed, providing insight into the system's behavior and compliance with the specified requirements.

Playwright allows for additional configuration, enabling users to customise the test execution process according to their needs. This includes options such as specifying browser or devices where the tests will be automated or setting timeouts, and configuring network conditions. Additionally Playwright has a built-in mechanism for configuring the reporter that displays test results. By specifying a reporter configuration option, users can choose how they want the test execution results to be presented.

The final specification for the SUT consists of 4 features - (1) Creating a task, (2) Completing a task, (3) Uncompleting a task, (4) Deleting a task.

Considering the feature "Deleting a task" has not yet been implemented, we expect the corresponding feature to fail on test execution. When invoking the run command, the test finishes with failure, pointing the failure to the feature "Deleting a task" as seen from

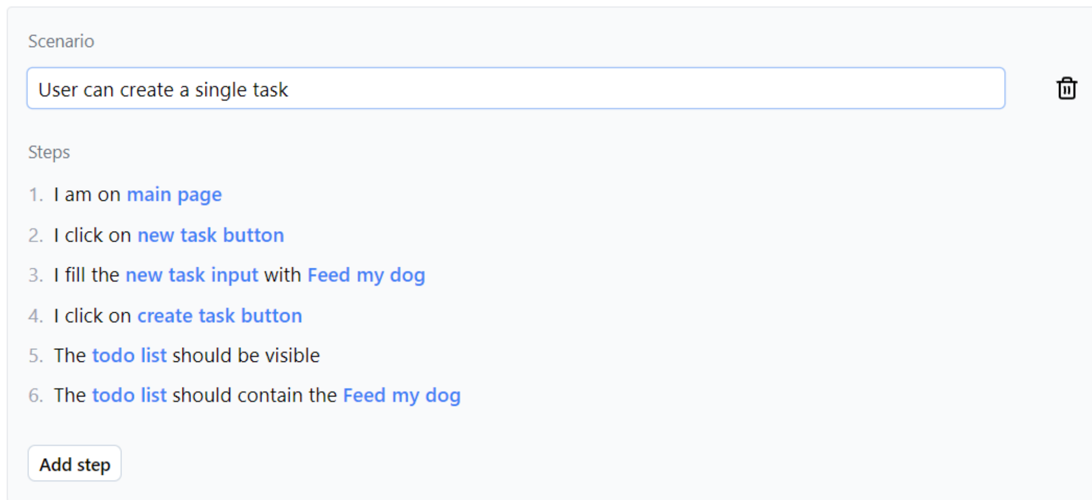

Figure 17. *Scenario parameters added*

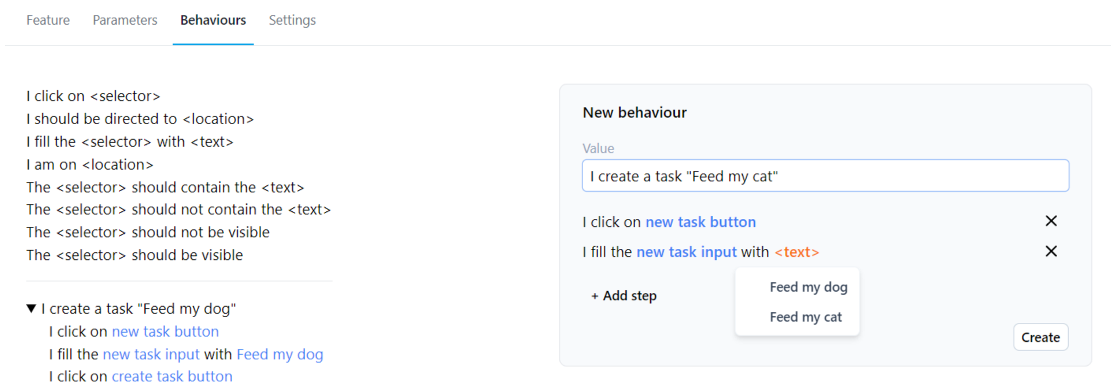

Figure 18. *Combined behaviours creation*

screenshot 20. When setting the Playwright `reporter` configuration option to `html`, a web page opens in the browser after test execution, offering additional context on the results as seen on screenshot 21. When opening the corresponding failed test in the reporter page, the specific failed step can be easily identified. Furthermore, a screenshot of the SUT is included in the report, illustrating the state of the SUT at the point of failure, as shown in screenshot 22.

Given that the developer has implemented the deletion feature and added the appropriate identifier for the delete button, the execution of 6 scenarios within 4 features will succeed within 7.84 seconds. At this point, we can conclude that the SUT conforms to its specification.

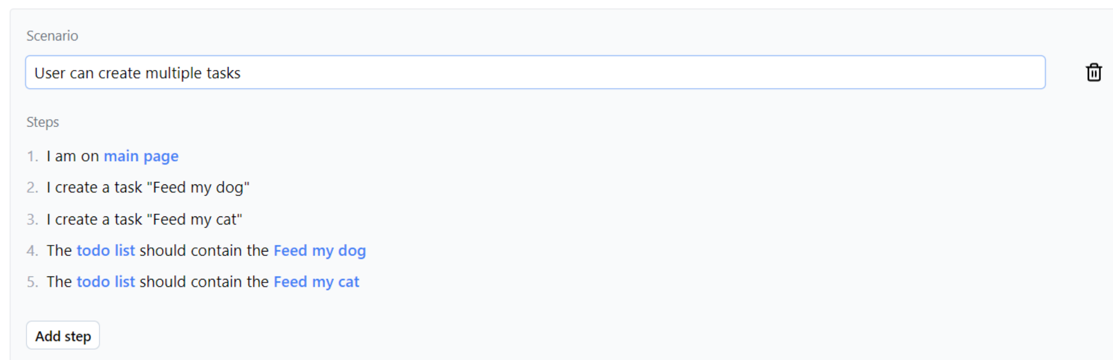

Figure 19. *Combined behaviour added to scenario*

```
PS C:\Users\hans\Projects\hummus\packages\demo> yarn test
yarn run v1.22.19
$ yarn with-env hummus run
$ dotenv -e ../../.env -- hummus run
@hummus: Loading config ...
@hummus: Config loaded!
@hummus: Retrieving project ...
@hummus: Project "Task Management App" retrieved!
@hummus: Generating test files ...
@hummus: Test files generated!
@hummus: Executing spec using Playwright ...

Running 6 tests using 6 workers

  ✓  1 [chromium] › .hummus\creating-a-task.spec.ts:4:5 › Creating a task > User can create a single task (2.7s)
  ✓  2 [chromium] › .hummus\completing-a-task.spec.ts:4:5 › Completing a task > Complete a single task (3.1s)
  ✓  3 [chromium] › .hummus\completing-a-task.spec.ts:49:5 › Completing a task > Complete multiple tasks (3.2s)
  ✘  4 [chromium] › .hummus\deleting-a-task.spec.ts:4:5 › Deleting a task > Delete a completed task (20.0s)
  ✓  5 …m] › .hummus\uncompleting-a-task.spec.ts:4:5 › Uncompleting a task > User can uncomplete a completed task (3.3s)
  ✓  6 [chromium] › .hummus\creating-a-task.spec.ts:50:5 › Creating a task > User can create multiple tasks (3.1s)


  1) [chromium] › .hummus\deleting-a-task.spec.ts:4:5 › Deleting a task > Delete a completed task ──

    Test timeout of 20000ms exceeded.

    Error: locator.waitFor: Target closed
    =========================== logs ===========================
    waiting for locator('[data-test="done-item"]:has-text("Feed my dog") > [data-test="clear-todo"]') to be visible
    ============================================================

       at ..\runner\src\lib\get-element.ts:6
```

Figure 20. *Execution output - failure*

All 6 | Passed 5 | × Failed 1 | Flaky 0 | Skipped 0

Project: chromium — Total time: 22.2s

.hummus/deleting-a-task.spec.ts

× Deleting a task > Delete a completed task — 20.0s
.hummus/deleting-a-task.spec.ts:4

.hummus/completing-a-task.spec.ts

✓ Completing a task > Complete a single task — 2.9s
.hummus/completing-a-task.spec.ts:4

✓ Completing a task > Complete multiple tasks — 3.2s
.hummus/completing-a-task.spec.ts:49

.hummus/creating-a-task.spec.ts

✓ Creating a task > User can create a single task — 2.5s
.hummus/creating-a-task.spec.ts:4

✓ Creating a task > User can create multiple tasks — 3.1s
.hummus/creating-a-task.spec.ts:50

.hummus/uncompleting-a-task.spec.ts

✓ Uncompleting a task > User can uncomplete a completed task — 3.2s
.hummus/uncompleting-a-task.spec.ts:4

Figure 21. *HTML reporter on failure 1*

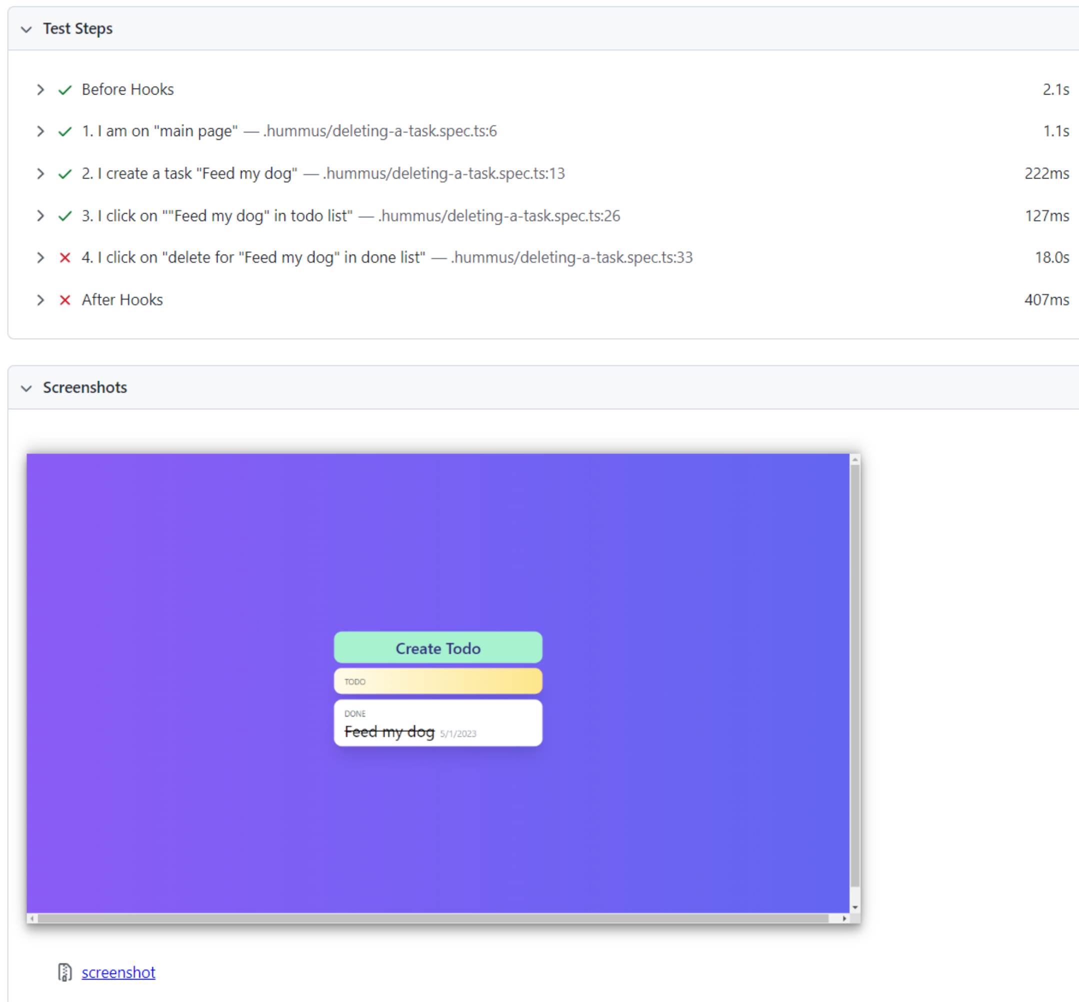

Figure 22. *HTML reporter on failure 2*

```
PS C:\Users\hans\Projects\hummus\packages\demo> yarn test
yarn run v1.22.19
$ yarn with-env hummus run
$ dotenv -e ../../.env -- hummus run
@hummus: Loading config ...
@hummus: Config loaded!
@hummus: Retrieving project ...
@hummus: Project "Task Management App" retrieved!
@hummus: Generating test files ...
@hummus: Test files generated!
@hummus: Executing spec using Playwright ...

Running 6 tests using 6 workers

  ✓  1 [chromium] › .hummus\creating-a-task.spec.ts:4:5 › Creating a task > User can create a single task (2.8s)
  ✓  2 [chromium] › .hummus\completing-a-task.spec.ts:4:5 › Completing a task > Complete a single task (3.0s)
  ✓  3 …m] › .hummus\uncompleting-a-task.spec.ts:4:5 › Uncompleting a task > User can uncomplete a completed task (3.0s)
  ✓  4 [chromium] › .hummus\deleting-a-task.spec.ts:4:5 › Deleting a task > Delete a completed task (2.6s)
  ✓  5 [chromium] › .hummus\creating-a-task.spec.ts:50:5 › Creating a task > User can create multiple tasks (2.7s)
  ✓  6 [chromium] › .hummus\completing-a-task.spec.ts:49:5 › Completing a task > Complete multiple tasks (3.3s)

  6 passed (5.2s)
Done in 7.84s.
```

Figure 23. *Execution output - success*

8. Evaluation

This section assesses the proposed approach, focusing on how effectively it addresses the presented problems. The practical adoption of the proposed approach is discussed, a comparison to the traditional DSL-based method is provided and the results obtained from a user testing session are showcased.

The primary objective of this thesis was to design an approach for conducting BDD without relying on commonly used DSLs, such as Gherkin, which are associated with several maintenance and usability issues. To enable the process of conducting BDD without DSL and achieve the primary goal, a specification management and execution tool called Hummus was developed, which consisted of the Manager web-based client and the Runner CLI client.

8.1 Adoption of the tool in a task management application

The tool was adopted in an example development process of a task management application (SUT). A specification for the SUT, was built using the Manager UI. Seven of the eight atomic behaviors offered by the Hummus tool were employed to construct a specification of 6 scenarios and 4 features. In total of 11 parameters were described, to fulfill the necessary context for a comprehensive scenario steps. In addition, two new behaviours that were commonly used in other scenarios, were created using the functionality of combining several behaviours into one, which enabled to make scenarios shorter and more concise. The specification was then executed using the Runner CLI client. Playwright was launched during the execution process, successfully automating the scenarios in a browser environment.

During the adoption of Hummus for specifying and testing the task management SUT, no major issues were encountered. However, a scenario to verify task list persistence upon browser page reload could not be specified, due to the fact that there was no corresponding behavior implemented that would enable reloading of the browser page, such as "I reload the page." This use-case could have been possible if the extensibility requirement (REQ 5) had been implemented during the development phase. Although the combined behaviors functionality provides some degree of extensibility, the original intention of the requirement was to enable the definition of custom behaviors with custom test code for more complex use-cases such as drag-and-drop or swiping, especially when the library of behaviors,

offered by Hummus, is limited. However, the intended extensibility functionality was not implemented due to time constraints.

In conclusion, the Hummus system enabled to create the specification of a task management application with all the required features, except for one scenario, due to the early stage of development of the Hummus system. Additionally, the acceptance testing of SUT was conducted successfully with all the features specified successfully passing.

8.2 User testing session

To evaluate the usability of Hummus, a user testing session was conducted with a single participant. The participant has experience in various roles within software development, as well as in managerial positions. However, as she had no practical experience with BDD, she was aware of the core principles of the method.

8.2.1 Feature creation

The session began with the explanation and drafting of an example username/password authentication feature.

Feature creation with Gherkin

Firstly, the feature was drafted in Gherkin format. The discussion focused on potential ways to structure scenarios for both successful and unsuccessful login attempts. It was discussed whether to specify as "I insert my username and password" or "I insert my username" and "I insert my password" separately. It was noted that there could be several ways to construct the steps. Additionally, the need for the test code to be written for each interaction, was identified.

A point of interest was how reusable interactions could be made in Gherkin, which led to a broader discussion on the management of specifications as they grow in size and how the structure of the scenarios affects the maintainability of the test suite. This raised the question of collaboration and how maintaining such specifications relies on technical knowledge. However, it was agreed that the responsibility of maintaining such specifications should not solely lie with the developer but also involve managerial roles.

The resulting user authentication feature created in Gherkin can be seen in listing 2.

```
Feature: User authentication
    User can log in so they can do logged in user things

Scenario: Successful login
    When I have inserted my correct username and password
    And I click log in button
    Then I am directed to dashboard page

Scenario: Unsuccessful login, user has not registered
    When I have inserted incorrect username and password
    And I click log in button
    Then I see error message - "Incorrect credentials"
```

Listing 2. Gherkin feature created during user testing session

Feature creation with Manager

Following the Gherkin discussion, the same authentication feature was created using the Manager web-based client. The participant was able to intuitively log in, create a project, and define a new feature along with its associated user story and scenarios. During the addition of scenario steps, the functionality of step parameters was acknowledged and the absence of parameters in current project was noted. The participant intuitively proceeded to add all the scenario steps by selecting from the available granular behaviours, easily identifying the necessary steps for the authentication scenario.

When specifying parameters for a step, it was quickly understood that they needed to be created separately. Subsequently, we moved on to creating parameters. The participant quickly grasped the purpose of the parameter fields that needed to be filled out. While the nuances of defining selector values required some discussion, the fundamental concept was easily comprehended. Ultimately, the participant successfully created all the parameters required for our login scenario, which were then successfully added to the steps.

There was a brief misunderstanding concerning the usage of "I am on <location>" and "I should be directed to <location>". It became clear after explaining that "I am on <location>" is used when the scenario requires opening a specific page where certain elements, like form fields, exist. Conversely, "I should be directed to <location>" is used to assert that the browser has completed its operations and checks the current URL, ensuring the user is directed to the expected location.

The concept of combining multiple steps into one for more concise scenarios was also introduced and well-received, showing the usefulness of this functionality.

The final specification created during the user testing session can be seen from screenshot 24

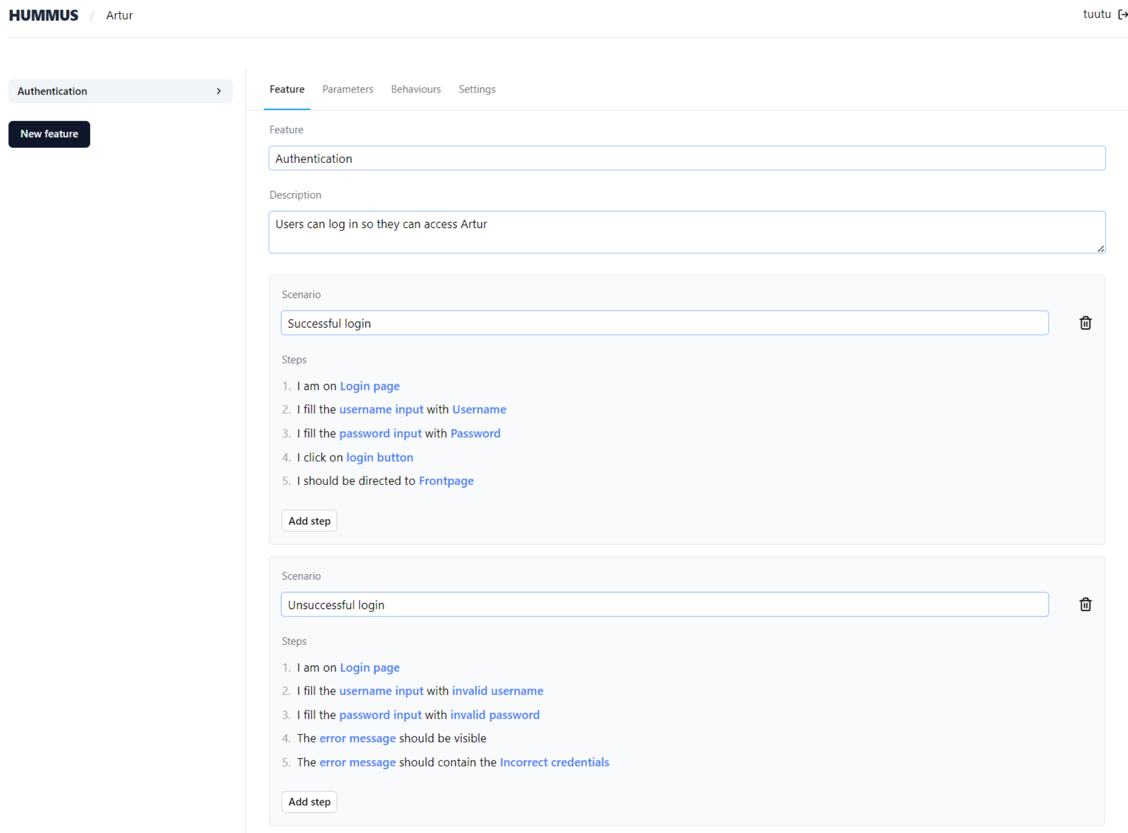

Figure 24. *Hummus feature created during user testing session*

8.2.2 Demonstration of specification execution

The testing session continued with a demonstration of the Runner package using an existing specification for the task management application described in chapter 7. After showing the installation and configuration process of the Runner package, the task management app's specification was executed, and results examined using the Playwright's reporting feature. The usage and configuration process of Runner was easily understood.

8.2.3 Feedback and conclusion

The session concluded with positive feedback, where the participant showed appreciation for the workflow enabled by Hummus for software development teams. Despite suggesting some improvements in managing larger quantities of specification artifacts, such as the ability to search parameters, she agreed that Hummus in its current state can be considered viable and could offer significant value to its users. Particularly, the ability to create

testable specifications directly without worrying about Gherkin syntax and potential test suite management overhead was noted as an important advantage.

8.3 Comparison to the DSL-based approach

To provide a more concrete understanding of the improvements the Hummus approach offers over the traditional DSL-based method, the process of defining, implementing and maintaining a feature is examined.

Defining the feature

DSL-based approach: Typically, a specification is written as files in Gherkin language within an IDE, with each file representing a feature, as presented in listing 1. Usually developers write the final Gherkin file, as they work with IDEs and have access to language supporting tools. The exact composition of a scenario, if not decided collaboratively by the team, should carefully consider already existing step definitions, to promote reusability and avoid maintenance overhead, as well as issues related to ambiguity and loose grammar. Therefore access and knowledge over existing step definitions is needed. Step definitions should be carefully constructed using an appropriate abstraction level to achieve clear intentions and reusability.

Hummus approach: Features can be created using the Manager client, which is accessible to all team members via web-based UI. Hummus approach encourages the shift of specification management responsibility towards managers, as the area of defining requirements is more appropriate to managerial roles and IDE usage is no longer needed to access existing knowledge of specifications. Scenario steps can be defined by choosing from a selection of granular user interactions, reducing issues related to loose grammar and ambiguity. Additional context can be defined and selected in the form of parameters, if required. As a result, an accurate behaviour of end-user interacting with the system is defined.

Implementing the feature

DSL-based approach: With the feature defined, developers need to write test code for each scenario step, similarly to listing 3. Potentially subjective test implementation of a step may be introduced. In case the new scenario is defined using existing step definitions, no additional test code should be needed. In case steps are reused and take parameters, these parameters need to defined in the test suite. When ensuring the test code is implemented as expected, only then the implementation can be tested against the specification.

Hummus approach: When the feature is defined in Manager client, developer can directly start working on the implementation as no additional test code is needed. Developers can execute specification using Runner and verify whether the feature implementation passes the specification.

Maintaining the feature

DSL-based approach: Over time, the application and specification evolves. As developers are responsible for modifying Gherkin files and updating the test code, inconsistencies and errors might be introduced over time, if required changes are not communicated accurately. This can be time-consuming.

Hummus approach: Any changes to the features or scenarios can be made directly in the Manager UI. No changes to test code need to be considered. The resulting modified specification can be executed to verify whether the developed system fulfills the modifications. If not, developer can deliver the required changes and not worry about the test suite maintenance.

8.4 Conclusion

In conclusion, the objectives raised in chapter 5 were largely met, except for the extensibility requirement (REQ 5). The Hummus system facilitated the development of a task management SUT with most desired features, though one scenario was not achieved due to the system's early stage. The acceptance testing of the SUT was successfully conducted, by executing the specification using the Runner CLI.

The user testing session demonstrated the usability and practicality of the Hummus system. The participant was able to successfully navigate, use the system's features and the proposed benefits were well recognised.

Despite not being tested with large specifications, the Hummus system's core concepts aim to improve maintainability and clarity of specifications throughout the development process. It can be concluded that despite the early stage, the current state of the Hummus system can be beneficial and provide value to its user.

9. Discussion

The resulting Hummus Manager enables the creation of specifications directly from the UI, eliminating the need for a DSL. Specifications can be created and accessed by logging into the web-based Manager client. This client offers additional maintenance capabilities for composing requirement specifications, such as encouraging reusability by providing a selection of granular interactions when creating scenarios. This approach reduces mental effort when determining the exact wording or sequence for scenario steps and prevents errors due to loose grammar, such as "Given I log in to the system" versus "Given I log on to the system." Additionally, the ability to combine multiple steps into one for repetitive interactions makes scenarios shorter and more concise.

Furthermore, parameter management offers a way to store additional context about the system being specified, such as specific elements, locations, or textual values that users of the new system would see or interact with. Parameters are centrally stored within projects, shared across features, and used in scenario steps to specify step behavior. This approach encourages consideration of the specific properties of the designed system before development. These kinds of properties, similar to Hummus parameters, are often abstracted in traditional test suites by developers.

Executing the specification using the Runner package provides valuable feedback for developers during development, ensuring that the proposed changes to the system are beneficial. Commonly, test automation might be overlooked due to concerns about managing test suites, the benefits of feedback and security provided by automated acceptance testing are essential in fast-paced agile software development. The Runner package allows developers to test their work directly during development or in continuous integration process without requiring additional test code or management of DSL. This feedback ensures valuable knowledge on whether the developed system conforms to its requirements, excluding the subjectivity aspect associated with acceptance testing.

In case the specification was constructed using traditional DSL based approach, by utilising Gherkin, the task management SUT specification would have required the creators to determine an appropriate abstraction level to achieve clear intentions and reusable steps, as the granular behaviours would not be available to select. Potentially issues related to ambiguity, loose grammar and different interpretations of the required behaviours could arise from definitions like "I complete the task", "I complete the task" or "I mark a task as

done". As developers, who usually work with IDEs, where the DSL-based specification lives, would be responsible for interpreting and translating steps into actionable test code, introducing subjectivity and potentially altering the system's intent while organising test suites. Consequently, developers would remain responsible for managing the specification, reducing valuable time spent on implementing functionality. However, when the Hummus system is used, the process eliminates the need for test code and language maintenance of DSL, allowing more time for developers to focus on the implementation of functionality. The responsibility of specification management could easily be transferred to a stakeholder more closer to the business needs. Someone who is likely to be less-technical than a software developer, as the usage of IDEs is no longer needed to effectively manage a specification artifacts.

9.1 Advantages

The proposed approach offers several significant advantages:

1. **Elimination of DSL:** Specifications can be created directly from the UI, circumventing the usability issues associated with DSLs and making the process more accessible.
2. **Reduced mental effort:** Selection of granular interactions and parameters simplify the construction of scenarios.
3. **Error prevention:** Loose grammar-related errors are minimised by utilising a selection of available entities in scenario creation.
4. **Concise scenarios:** The ability to combine multiple steps into one for repetitive interactions makes scenarios shorter and more concise.
5. **Parameter management:** Centrally stored parameters within projects encourage consideration of specific properties before development.
6. **Automated acceptance testing:** The Runner package enables automated acceptance testing without requiring additional test code, providing valuable feedback on adherence to the specification.

Despite the advantages of the proposed approach, there are several limitations that should be considered:

9.2 Limitations

1. **Limited to web-based software:** As it currently stands, only the software utilising web technologies and a browser environment can be specified and tested using

Hummus. To overcome this limitation, an additional Runner package should be developed, utilising a testing library that has the ability to automate the desired environment.

2. **Limited to implemented behaviors:** The implemented behaviors might not be sufficient for more complex software products. Additional behaviors could be needed for more complicated interactions, such as dragging and dropping, swiping in mobile environments or additional assertments for test oracles. Utilising the selector parameter is very dynamic but might not be enough for more complex use-cases.
3. **Limited access to testing lifecycle:** The nature of acceptance testing, which targets the system's ends, might be more difficult due to the fact that test code is generated. Access to the lifecycle of testing is limited, such as access to `before`, `after`, `beforeAll` and `afterAll` functions. This limitation can be reduced when utilising Playwright's `globalSetup` configuration option, which allows for a functionality to be passed that acts before every scenario, potentially needed for tasks like resetting mutable shared states (e.g., a database) or intercepting network traffic when communicating with third-party services beyond the developer's control.
4. **Limited features in the prototype:** In the prototype developed in this thesis, the Manager client does not enable connecting users to existing projects created by other users. However, this is a feature that can easily be added in the future. In addition, version control functionality might be necessary to implement in the future, as currently any modifications are instantly reflected when executing the specification. This might discourage making modification to the specification before the implementation is due to be developed.

10. Conclusion

The proposed approach, enabled by the Hummus tool successfully meets the research goal and objectives set by enabling the creation and execution of specifications while eliminating the need for developers to manage DSL files and test suites. The new process was made possible through the development of an open-source tool called Hummus. Although Hummus introduces new limitations that may hinder its future adoption, the approach streamlines specification management, making it more efficient and user-friendly. Consequently, Hummus has the potential to be a valuable tool for various stakeholders involved in specifying and developing software. By addressing some of the challenges associated with traditional BDD processes, Hummus offers a promising alternative for more effective and accessible software specification and testing, provided that its limitations are addressed in future iterations.

References

[1] Alan Davis. *Just enough requirements management: where software development meets marketing*. Addison-Wesley, 2013.

[2] Elizabeth Bjarnason et al. “A multi-case study of agile requirements engineering and the use of test cases as requirements”. In: *Information and Software Technology* 77 (2016), pp. 61–79.

[3] Lucas Layman, Laurie Williams, and Lynn Cunningham. “Motivations and measurements in an agile case study”. In: *Proceedings of the 2004 workshop on Quantitative techniques for software agile process*. 2004, pp. 14–24.

[4] Karina Curcio et al. “Requirements engineering: A systematic mapping study in agile software development”. In: *Journal of Systems and Software* 139 (2018), pp. 32–50.

[5] Certified Model-Based Tester. “ISTQB® Foundation Level Certified Model-Based Tester”. In: ().

[6] Pei Hsia, David Kung, and Chris Sell. “Software requirements and acceptance testing”. In: *Annals of software Engineering* 3.1 (1997), pp. 291–317.

[7] Adnan Causevic, Daniel Sundmark, and Sasikumar Punnekkat. “Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review”. In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 2011, pp. 337–346. DOI: 10.1109/ICST.2011.19.

[8] Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. “Maintaining behaviour driven development specifications: Challenges and opportunities”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 175–184.

[9] Rakesh Kumar Lenka, Srikant Kumar, and Sunakshi Mamgain. “Behavior driven development: Tools and challenges”. In: *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*. IEEE. 2018, pp. 1032–1037.

[10] Ankica Barisic et al. “Quality in use of dsls: Current evaluation methods”. In: *3rd Inforum-Simpósio de informática*. 2011.

[11] Ken Peffers et al. “A design science research methodology for information systems research”. In: *Journal of management information systems* 24.3 (2007), pp. 45–77.

[12] Frauke Paetsch, Armin Eberlein, and Frank Maurer. "Requirements engineering and agile software development". In: *WET ICE 2003. Proceedings. Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2003.* IEEE. 2003, pp. 308–313.

[13] Aybüke Aurum and Claes Wohlin. *Engineering and managing software requirements.* Vol. 1. Springer, 2005.

[14] Eva-Maria Schön, Jörg Thomaschewski, and Marıa José Escalona. "Agile Requirements Engineering: A systematic literature review". In: *Computer standards & interfaces* 49 (2017), pp. 79–91.

[15] Timothy C Lethbridge, Janice Singer, and Andrew Forward. "How software engineers use documentation: The state of the practice". In: *IEEE software* 20.6 (2003), pp. 35–39.

[16] Shelly Park and Frank Maurer. "A literature review on story test driven development". In: *International Conference on Agile Software Development.* Springer. 2010, pp. 208–213.

[17] Børge Haugset and Geir Kjetil Hanssen. "Automated acceptance testing: A literature review and an industrial case study". In: *Agile 2008 Conference.* IEEE. 2008, pp. 27–38.

[18] Ana CR Paiva, Daniel Maciel, and Alberto Rodrigues da Silva. "From requirements to automated acceptance tests with the RSL language". In: *Evaluation of Novel Approaches to Software Engineering: 14th International Conference, ENASE 2019, Heraklion, Crete, Greece, May 4–5, 2019, Revised Selected Papers 14.* Springer. 2020, pp. 39–57.

[19] Kent Beck. *Extreme programming explained: embrace change.* addison-wesley professional, 2000.

[20] Dan North. *Introducing BDD.* Accessed March, 2022. 2006. URL: http://dannorth.net/introducing-bdd.

[21] Ioan Lazăr, Simona Motogna, and Bazil Pârv. "Behaviour-driven development of foundational UML components". In: *Electronic Notes in Theoretical Computer Science* 264.1 (2010), pp. 91–105.

[22] Carlos Solis and Xiaofeng Wang. "A study of the characteristics of behaviour driven development". In: *2011 37th EUROMICRO conference on software engineering and advanced applications.* IEEE. 2011, pp. 383–387.

[23] Martin Fowler. *Domain-specific languages.* Pearson Education, 2010.

[24] Benoıt Langlois, Consuela-Elena Jitia, and Eric Jouenne. "DSL classification". In: *OOPSLA 7th workshop on domain specific modeling.* 2007.

[25] Alessandro Cavalcante Gurgel. "for Architectural Degradation Prevention". PhD thesis. Programa de Pós-Graduação em Informática of the Departamento de Informática ..., 2012.

[26] Mohsin Irshad, Ricardo Britto, and Kai Petersen. "Adapting Behavior Driven Development (BDD) for large-scale software systems". In: *Journal of Systems and Software* 177 (2021), p. 110944.

[27] *Serenity BDD website*. Accessed: 2023-04-20. 2023. URL: https://serenity-bdd.info/.

[28] *Pickles website*. Accessed: 2023-04-20. 2023. URL: https://www.picklesdoc.com/.

[29] Shashikant Jagtap. *BDDfire repository*. Accessed: 2023-04-20. 2023. URL: https://github.com/Shashikant86/bddfire.

[30] *Behave Pro website*. Accessed: 2023-04-20. 2023. URL: https://behavepro.app/.

[31] *CucumberStudio website*. Accessed: 2023-04-20. 2023. URL: https://cucumber.io/tools/cucumberstudio/.

[32] *SpecFlow LivingDoc website*. Accessed: 2023-04-20. 2023. URL: https://docs.specflow.org/projects/specflow-livingdoc/en/latest/.

[33] Fiorella Zampetti et al. "Demystifying the adoption of behavior-driven development in open source projects". In: *Information and Software Technology* 123 (2020), p. 106311.

[34] Thiago Rocha Silva, Jean-Luc Hak, and Marco Winckler. "A behavior-based ontology for supporting automated assessment of interactive systems". In: *2017 IEEE 11th International Conference on Semantic Computing (ICSC)*. IEEE. 2017, pp. 250–257.

[35] Mark Micallef and Christian Colombo. "Lessons learnt from using DSLs for automated software testing". In: *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2015, pp. 1–6.

Appendix 1 – Non-Exclusive License for Reproduction and Publication of a Graduation Thesis[1]

I Hans Hendrik Starkopf

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Behaviour-driven Specification Management and Execution System”, supervised by Gert Kanter and Dietmar Pfahl
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

18.05.2023

[1] The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Code samples

```
from behave import given, when, then
from selenium import webdriver

LOGIN_PAGE_URL = "https://example.com/login"
TEST_USERNAME = "testuser"
TEST_PASSWORD = "testpassword"

@given("I am on the login page")
def step_given_on_login_page(context):
    context.browser = webdriver.Firefox()
    context.browser.get(LOGIN_PAGE_URL)

@when("I enter my valid username and password")
def step_when_enter_credentials(context):
    username_field = context.browser.find_element_by_name("username")
    password_field = context.browser.find_element_by_name("password")

    username_field.send_keys(TEST_USERNAME)
    password_field.send_keys(TEST_PASSWORD)

@when('I click the login button')
def step_when_click_login(context):
    login_button = context.browser.find_element_by_name("login")
    login_button.click()

@then('I should be redirected to my account dashboard')
def step_then_redirected_to_dashboard(context):
    assert "dashboard" in context.browser.current_url

@then('I should see a welcome message')
def step_then_see_welcome_message(context):
    message = context.browser.find_element_by_id("welcome_message")
    assert message.is_displayed()
```

Listing 3. Behave test code for feature in listing 1

```
import { Page } from '@playwright/test';
import { getElement } from './get-element';

export async function clickElement(page: Page, selector: string) {
  const element = await getElement(page, selector);

  element.click();
}

export async function inputElementValue(
    page: Page,
    selector: string,
    input: string
) {
  const element = await getElement(page, selector);

  element.focus();
  element.fill(input);
}
```

Listing 4. runner/lib/interactions.ts

```
#!/usr/bin/env tsx

import path from 'path';
import { spawn } from 'child_process';

import { getProject } from './manager';
import { generate } from './generator';
import { resolveConfig } from './config';

function loadConfig() {
  const configPath = path.join(process.cwd(), 'hummus.config.ts');
  const config = require(configPath).default;

  return resolveConfig(config);
}

async function run() {
  /** 1. Load configuration */
  const config = loadConfig();

  /** 2. Retrieve project */
  const project = await getProject(config);

  /** 3. Generate test files */
  await generate(project, config);

  /** 4. Execute Playright */
  spawn(
    'npx',
    ['playwright', 'test', config.dir, '--headed'],
    { stdio: 'inherit', shell: true }
  );
}

run();
```

Listing 5. runner/cli.ts

```
export default {
  projectId: 'example-project-id',
  managerURL: 'http://localhost:3000',
  auth: {
    username: process.env.MANAGER_USER,
    password: process.env.MANAGER_PASSWORD,
  },
};
```

Listing 6. hummus.config.ts