On The Use of User Interface Specific Domain Gherkin in Behaviour Driven Development

Abdullah Alshammari

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

May, 2025

Abstract

There has been a noticeable rise in the popularity of behaviour driven development (BDD) in the software industry. Gherkin is a popular language for describing system behaviours, since it uses simple natural language that can be understood easily by all those involved. Gherkin describes system behaviours as examples of how the system should behave in different scenarios. However, existing research has identified several challenges with current approaches to BDD and Gherkin. First, stakeholders struggle to express and negotiate requirements in Gherkin when they are expressed in terms of business domain concepts. Second, the automated acceptance tests associated with requirements expressed in business domain concepts require additional developer effort. Third, stakeholders find selecting an appropriate abstraction level when developing Gherkin scenarios can be challenging.

To address these challenges, this thesis investigates the feasibility and efficiency of using user interface specific domain concepts for expressing requirements. This approach emphasises the expression of system behaviours in terms of user interface elements, such as buttons, text entry fields and so on. The intuition behind this approach is that the user interface represents a common 'language' for stakeholder discussion of requirements, that is independent of business specific concepts. The research followed an empirical methodology and used four different methods to evaluate the approach: multivocal literature review, design science, case studies, and laboratory experiment.

First, a multivocal review of BDD and its best practices was undertaken to establish the prevailing understanding of Gherkin usage. Second, a framework that comprises standardised low level Gherkin scenario steps and the associated acceptance test functions was developed. The objective was to investigate the applicability of user interface domain specific Gherkin use in BDD.

Third, three case studies were undertaken to improve and validate the design and implementation of the user interface domain specific Gherkin language and the automated acceptance test suites associated with it. The objective was to identify weaknesses in the design and implementation of the user interface specific domain Gherkin language. The results of the

case studies suggest that using user interface specific domain concepts for communicating system requirements is feasible.

Fourth, a controlled laboratory experiment was conducted to find out whether it is easier to comprehend and document scenarios written in user interface specific domain Gherkin than scenarios written in business domain Gherkin. The results indicate that non-software practitioners, who represent customers, found scenarios written in user interface specific domain Gherkin to be significantly easier to document than scenarios written in business domain Gherkin. Further, the results show that for the same group, scenarios written in user interface specific domain Gherkin were marginally easier to comprehend than scenarios written in business domain Gherkin. In addition, the results suggest that for software practitioners, scenarios written in user interface specific domain Gherkin were marginally easier to comprehend and document than scenarios written in business domain Gherkin.

Acknowledgements

In honour of those people without whom it would not have been possible for me to complete this research and write this thesis.

I am grateful to my supervisor, Dr Tim Storer, for his guidance, patience and support over the years. You have tremendously transformed my skills in scientific research. I have been extremely grateful that you have taken me on as a student and continued to believe in me.

My sincere gratitude goes out to my deceased father, Farhan Alshammari for his support throughout my life and especially during my PhD. Dad, you passed away in the middle of my PhD journey and it saddens me that you will not be there to attend my graduation ceremony. My sincere gratitude also goes out to my mother, Masirah Alshammari for her continuous support and prayers. Without encouragement and support from both of you, this work would not have seen the light. I also would like to thank my wife, Dalal Alshammari, for standing by me and taking great care of me and our children, Faris, Zaid, and Farhan.

Thank you to Yanbu Industrial College for sponsoring this research, and the School of Computing Science at the University of Glasgow for their support.

Declaration

I declare that this thesis has been composed by myself, that the research presented embodies the results of my own work and that it does not include work forming part of a thesis presented for a degree in this or any other University.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

A software system's success is primarily determined by its ability to fulfill its intended purpose [1]. An engineering process referred to as Requirements Engineering (RE) involves identifying the goals to be achieved by an envisioned system. A number of processes are involved in RE, including *domain analysis*, *elicitation*, *specification*, *assessment*, *negotiation*, *documentation*, and *evolution*. A growing body of research has shown that high quality requirements remain difficult to obtain and that requirements engineering is a critical area of software engineering practice and research that must be addressed carefully [2].

Complementary to the requirements process, Software Testing helps to determine the quality of a product. In particular, *Acceptance Testing* is the process of verifying that the software product meets its intended purpose [3]. Miller and Collins states that *"Acceptance tests are a contract between the developers and the customer."* [4]. In order to prove that there has not been a breach of contract, acceptance tests should be maintained, run frequently, and amended as requirements change [4]. The objectives of acceptance testing are: First, to validate user-machine interactions. Second, to validate the system's functionality. Third, to ensure that the system meets its specifications. Fourth, to check the external interfaces of the system [5].

By constantly evaluating requirements, plans, and outcomes, Agile methods have been advocated as enabling swift adaptability and facilitates rapid changes through an iterative approach [6]. *"Agile Development is a set of methods and practices where solutions evolve through collaboration between self-organising, cross-functional teams."* [7]. Test Driven Development (TDD), Acceptance Test Driven Development (ATDD), and Behaviour Driven Development (BDD) are common Agile software development approaches that integrate the development of requirements and acceptance tests. For example, in TDD, developers begin

with writing a single unit test that describes a feature of the program. We then run the test, which should fail because the program lacks that feature. After that, we write just enough code to pass the test. We then refactor the code and repeat this process over and over to accumulate unit tests [8]. The value of ATDD lies in the collaboration between customers, developers, and testers to develop acceptance tests prior to implementing the corresponding functionality [9].

North introduced Behaviour Driven Development (BDD) in 2006 [10]. Based on existing Agile practices, BDD incorporates aspects from Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD). Besides promoting Agile software methods to teams with no prior experience, North wanted to make them more accessible and useful. Prior to developing software, BDD uses acceptance tests to define its behaviour. BDD is an Agile software development process that involves documenting and designing applications around how users expect to interact with them [11]. As part of the BDD process, a domain-specific language (DSL) is used utilising natural language constructs (for example, sentences in English-like form) to describe the behaviour and outcomes expected. Most BDD frameworks use Gherkin language for writing behaviour scenarios [12]. Gherkin is a domain-specific language that uses plain English to describe business behaviour. It is easy to read and understand for both technical and non-technical team members.

The practice of BDD has now been around for over two decades, during which it has gained popularity and usage. According to Binamungu et al.'s survey of 75 BDD practitioners, the majority (60%) of organisations used BDD approach on some projects, while 20% used it on all projects [13]. Furthermore, more than a quarter of the respondents reported that BDD will be a key tool on all projects in the future, while nearly half said it would be used as an optional tool. As part of the 14th Annual State of Agile Report [14], 40,000 Agile practitioners, executives, and consultants were interviewed. According to the report, 19% of interviewees used BDD, a decrease of 3% from the previous year.

1.2 Motivation

BDD scenario steps can be written in two styles: higher level abstraction (sometimes referred to as declarative) or lower level more concrete (sometimes referred to as imperative). Higher level steps focus on the business domain behaviours that a user wants to perform, typically, enabling scenarios to be expressed with fewer steps. However, they require more extensive glue code to be written by developers per step to implement the corresponding acceptance tests. Due to this, it can be difficult to interpret a step's purpose reference to the glue code. On the other hand, lower level steps explain the finer details of the system under development. As a result, scenarios using low level steps tend to take longer, requiring more steps to

express the same meaning as scenarios with higher level steps. However, they require less glue code to implement [15] and may be less ambiguous as to their purpose.

When scenarios are written at a consistent level of abstraction, BDD feature suites will be easier to comprehend, extend and maintain [15]. On the other hand, when a feature suite contains both low level and high level scenarios, it can be difficult for a maintenance engineer to choose the level of abstraction to apply when writing a new scenario if the feature suite contains both levels [15]. In addition, it is likely that there will be redundant steps and glue code, as well as inconsistent levels of abstraction in test harness code. This will result in additional comprehension and maintenance issues [15].

The professional literature of BDD best practices generally advocates that BDD scenarios should be written in business domain language (declarative). For example, the creator of SpecFlow, a BDD framework for .Net recommends BDD practitioners to write scenarios in a declarative way [16]. Furthermore, in the user guide of Cucumber [17], which is a BDD framework for Ruby, Java, and JavaScript, the authors encourage BDD practitioners to develop scenarios in a declarative way. They believe that writing scenarios in a declarative way makes them less brittle and easier to maintain. However, to date there has not been academic research to provide support for this advice. In addition, Binamungu et al. reviewed the scientific and professional literature of what constitutes good quality in BDD suites [15]. They also surveyed 56 BDD practitioners to learn more about what constitutes good quality in BDD suites. The authors learnt that selecting the appropriate level of abstraction when developing scenarios can be challenging, suggesting in practice there is considerable uncertainty regarding the best approach. Based on these motivations, the research presented in this thesis investigated what level of abstraction was suitable for writing user scenarios. It also investigated whether it was better to write abstract scenarios using high level steps (i.e., scenarios written in the business domain language), or concrete scenarios using low level steps (i.e. scenarios written in the user interface domain language).

1.3 Thesis Statement and Research Questions

Behaviour driven development has gained significant popularity in the software industry. However, there is significant evidence that stakeholders struggle to express and negotiate requirements in the Gherkin language when described desired system behaviours in terms of business domain goals and concepts. There is also evidence that it can be challenging to select the appropriate abstraction level when developing Gherkin scenarios. Secondly, requirements expressed in business domain concepts require additional developer effort to maintain associated automated acceptance test suites.

This thesis contends that describing desired system behaviours in Gherkin using user

interface specific domain concepts is an effective way of communicating requirements for stakeholders and mitigates the cost of maintaining acceptance tests through the provision of standardised step function suites.

Behaviour Driven Development (BDD) has been around for over two decades, during which it has gained popularity and use. Gherkin is a popular language for writing BDD scenarios, as it uses simple natural language that can be easily understood by everyone involved in writing the scenarios. In the professional literature, software practitioners have written many articles about BDD best practices. There is a presumption in the professional literature that BDD scenario steps should be written in business domain language (high level steps), as this improves scenario comprehension and documentation. This thesis argues that Gherkin scenarios, written in a user interface domain specific language that explicitly describes concrete actions within a graphical user interface (such as those in a web browser), and supported by standardised step functions, are easier to comprehend and document than scenarios written in an abstracted business domain language with custom step functions. The user interface domain specific language is distinguished from conventional approaches to BDD that emphasise describing tasks at a business domain level.

Given the thesis statement above, several questions need to be addressed. The first step is to assess the feasibility of user interface specific use of behaviour driven development. Therefore, the first question that needs to be addressed is: **RQ1: Is the proposed low level framework for user interface specific use of behaviour driven development feasible?**

After investigating the feasibility of user interface specific use of behaviour driven development and having the user interface domain specific framework ready for further investigation, the next step is to assess how easy it is for users to comprehend the framework. To do so, the following question needs to be addressed: **RQ2: Are BDD scenarios written in UI Domain Specific Gherkin easier to comprehend than scenarios written in Business Domain Gherkin?**

After assessing how easy it is for users to comprehend user interface domain specific BDD, the next step is to assess how easy it is for users to document user interface domain specific BDD. For that, the third question is: **RQ3: Are BDD scenarios written in UI Domain Specific Gherkin easier to document than scenarios written in Business Domain Gherkin?**

Figure 1.1 maps the questions to their corresponding chapters. RQ1 was addressed by three chapters and the rest of the questions (RQ2 and RQ3) were addressed by a chapter.

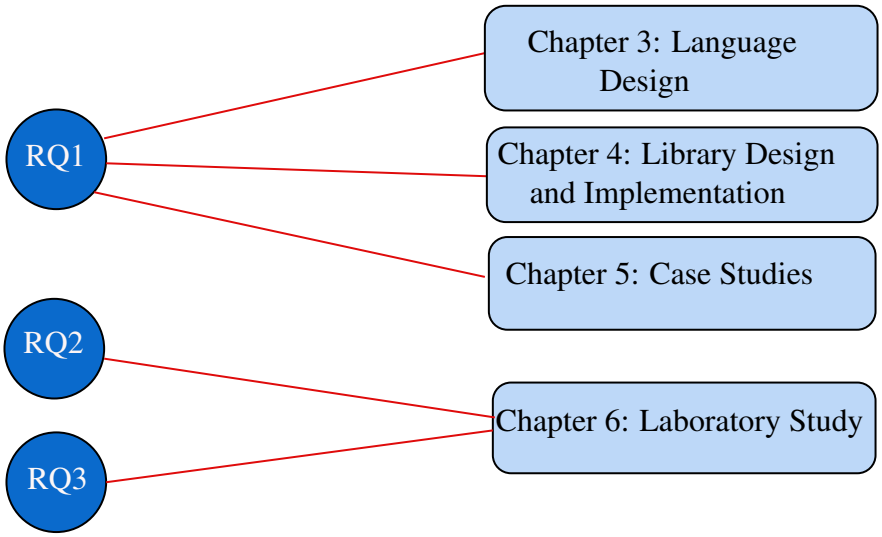

Figure 1.1: Thesis chapters and questions map.

1.4 Methods

This research follows a structured and rigorous methodological approach, integrating multiple research phases to address the overall research questions. The methodology combines empirical and experimental methods to iteratively refine and validate the proposed framework.

Research Phases and Their Relationships

The research process is divided into three main phases, each contributing to the overall goal of investigating the feasibility and effectiveness of a User Interface Domain Specific Gherkin language within Behaviour Driven Development (BDD).

Phase 1: Literature Review and Theoretical Framework Development

The study began with an extensive **multivocal literature review** to identify existing research on BDD, its benefits, challenges, and best practices. This phase aimed to establish a theoretical foundation and uncover gaps in academic and professional literature. Insights from this review guided the formulation of research questions and the design of the proposed UI Domain Specific Gherkin framework.

Phase 2: Design and Implementation of the UI Domain Specific Gherkin Framework

This phase involved the design and implementation of the **UI Domain Specific Gherkin language and its step function library**. Using an iterative development process, the research designed a set of standardised low-level Gherkin steps that explicitly describe UI interactions. These steps were mapped to corresponding step functions, implemented using Java and Selenium WebDriver. The goal was to eliminate the need for developers to write additional glue code, thus reducing the maintenance burden of acceptance tests.

Phase 3: Empirical Validation through Case Studies and Laboratory Experiment

To assess the feasibility and effectiveness of the proposed framework, the research employed two empirical evaluation methods:

1. **Case Studies:** Three real-world software systems—Lighthouse Laboratory Sample Tracker, Study Abroad Credits, and Popu e-commerce platform—were selected to refine and validate the design and implementation of the framework. These case studies provided insights into practical challenges, refinements, and the overall applicability of the framework.

2. **Laboratory Experiment:** A between-subjects laboratory experiment was conducted to compare UI Domain specific Gherkin with Business Domain Gherkin in terms of comprehension and documentation efficiency. Participants from software and non-software backgrounds were involved, and statistical analyses were applied to assess significance.

Integration of Research Phases

The literature review identified research gaps that guided the framework's design. Case studies validated its feasibility in real-world applications, while the laboratory study quantitatively measured its benefits.

Ethical Considerations

Ethical approval was obtained for the laboratory study, ensuring informed consent from participants and adherence to data protection regulations. Case study participants were engaged in a transparent manner, with confidentiality maintained throughout.

This methodological framework provides a comprehensive and empirically validated approach to investigating the research questions, ensuring that findings contribute meaningfully to both academia and industry.

1.5 Contributions

The contributions of this thesis are:

- A multivocal literature review revealing a discrepancy between industry perspectives and academic literature on BDD, highlighting potential misalignments in the theoretical foundations supporting industry practices.
- A novel standardised UI Domain Specific Gherkin language (`https://github.com/wadsg/wads-gherkin`), that enhances requirement communication among stakeholders by reducing ambiguity compared to traditional business language. The language specification is published on GitHub under the Apache Software License, enabling the specification to be adopted and reused in future research as well by practitioners.
- A novel framework of standardised step function suites directly maps to UI Domain Specific Gherkin steps, eliminating the need for ongoing developer maintenance. This framework, implemented as a Java library and published under an open-source license, is compiled as a Maven artifact (the industry-standard build tool for Java projects) for easy deployment in test suites for end-user projects. It is available for future modification and extension, with fully documented usage demonstrating that the complete set of user behaviours on a user interface can be feasibly described in a UI domain specific language.
- Empirical validation of UI domain specific behaviour driven development through three real-world software systems, demonstrating its feasibility in practice.
- A laboratory study and resulting experimental evidence that demonstrates that Gherkin specifications written in a user interface specific domain are easier to comprehend and document than specifications written in a business domain language. These results challenge widespread industry assumptions concerning best practices for BDD.

1.6 Structure of Thesis

The remainder of this thesis is structured as follows:

- **Chapter 2** presents a multivocal literature review on BDD. It begins by examining published literature on BDD, introducing its characteristics and exploring BDD tools, benefits, challenges, and related themes. Following this, it reviews professional literature on BDD best practices. The purpose of this multivocal literature review is to document the theoretical model of the BDD workflow and to identify what is known about BDD practices in academic and professional literature.

- **Chapter 3** presents the design of the standardised UI Domain Specific Gherkin scenario steps. It starts with describing the design methodology and moves to analysing each Gherkin step. The purpose of this chapter is to present the UI Gherkin language and the feasibility of it before testing its benefits in the later chapters.

- **Chapter 4** presents the step function suites that map to the UI Gherkin language steps in the previous chapter. The purpose of this chapter is to demonstrate the developed library and the feasibility of it before testing its benefits in the later chapters.

- **Chapter 5** presents the conducted case studies that drove forward the design and implementation of the standardised UI Domain Specific Gherkin steps and their corresponding step function suites. It starts by describing the case study design methodology and moves to demonstrating the results and analysis of each case study. The purpose of this chapter is to demonstrate the feasibility of the UI domain specific use of behaviour driven development before demonstrating its benefits in the next chapter.

- **Chapter 6** presents the laboratory study. It starts by describing the design methodology of the study and ends by analysing the results. The purpose of the study is to demonstrate the benefits of the UI domain specific use of BDD compared to business BDD.

- **Chapter 7** presents critical analysis of the research findings in relation to the original research questions, emphasising how the newly acquired knowledge enhances the existing body of literature. It also explores the study's limitations.

- **Chapter 8** gives a summary of the contributions and findings of this work.

Chapter 2

Multivocal Literature Review

In this chapter, the relevant multivocal literature on BDD is reviewed and the theoretical background for the thesis is provided. Multivocal literature refers to the comprehensive collection of publicly available sources related to a topic [18]. This encompasses materials such as blogs, white papers, articles, scholarly publications, and more. Incorporating this diverse range of perspectives allows for a richer, more holistic understanding of the subject, as it synthesises insights from academics, industry professionals, independent researchers, development organisations, and other experienced contributors [18].

The chapter is structured as follows. Section 2.1 presents a critical analysis of Test Driven Development (TDD), Acceptance Test Driven Development (ATDD), and the emergence of BDD. Section 2.2 demonstrates the research methodology. Section 2.3 presents the search results. Section 2.4 demonstrates literature gap identification. Section 2.5 summarises the chapter.

2.1 Critical Analysis of TDD, ATDD, and the Emergence of BDD

Test Driven Development (TDD) and Acceptance Test Driven Development (ATDD) are widely adopted methodologies in software engineering, each offering distinct advantages and challenges. This section critically examines their strengths and weaknesses, followed by a justification for Behaviour Driven Development (BDD) as an integrative approach.

TDD has been widely advocated in the software industry for its ability to improve code quality by mandating that tests are written before implementation. This practice encourages developers to design modular, cleaner, and more maintainable code [19, 20]. Additionally, TDD promotes iterative development cycles, enabling early defect detection and fostering

a robust codebase [21]. While debates persist, some studies suggest that TDD enhances developer productivity by minimising debugging and rework efforts, as developers focus on meeting predefined test criteria [22].

Despite its benefits, TDD faces criticism for being time-consuming, particularly during initial phases where writing tests precedes code implementation. This overhead can hinder progress in time-sensitive projects [23]. Furthermore, TDD's steep learning curve often discourages experienced developers accustomed to traditional workflows, leading to resistance and low adoption rates [20]. The methodology's effectiveness is also context-dependent, varying significantly across programming environments and team expertise levels [19].

ATDD emphasises aligning development with user requirements through acceptance tests derived from stakeholder input. This focus enhances customer satisfaction by ensuring the final product meets user expectations [23]. Moreover, ATDD fosters collaboration among developers, testers, and stakeholders, cultivating a shared understanding of project goals [24]. By utilising domain-specific languages, ATDD enables non-technical stakeholders to contribute to test case design, improving communication and requirement validation [25].

ATDD's complexity and resource demands pose significant challenges, especially in large or intricate projects, discouraging its adoption [25]. Similar to TDD, productivity may decline due to the effort required to maintain acceptance tests [22]. Additionally, misuse of ATDD frameworks such as repurposing them for unit testing can undermine their intended benefits [25].

BDD addresses the limitations of TDD and ATDD while building on their strengths. By prioritising collaboration, BDD employs a common language to describe system behaviour, bridging communication gaps between technical and non-technical stakeholders. This alignment with user needs enhances customer satisfaction and ensures shared requirement understanding [24, 26]. BDD also integrates automation tools to streamline testing, reducing ATDD's complexity. Its focus on system behaviour over implementation details fosters adaptable solutions, making it ideal for agile environments [27, 28].

TDD and ATDD offer valuable yet imperfect approaches to software development. TDD excels in code quality and iterative refinement but struggles with productivity trade-offs. ATDD enhances user alignment but faces scalability challenges. BDD emerges as a holistic alternative, leveraging collaboration, automation, and behaviour-centric design to balance efficiency with user-centric outcomes. For modern agile teams, BDD represents a compelling evolution of test-driven methodologies, addressing gaps while preserving core strengths.

2.2 Research Methodology

This section provides an overview of the research methodology, followed by a summary of the systematic approach used to gather relevant literature.

2.2.1 Multivocal Literature Review

After an initial search of the published literature on Behaviour Driven Development (BDD) in the software industry, I found limited academic research on the topic. Consequently, I decided to conduct a Multivocal Literature Review (MLR). Multivocal literature encompasses all accessible sources on a subject, including—but not limited to—blogs, white papers, articles, and academic publications [18]. By incorporating diverse sources, this approach provides a more comprehensive perspective, capturing insights from academics, practitioners, independent researchers, development firms, and other experienced professionals [18].

Garousi et al. [29] highlight the significance of Multivocal Literature Reviews (MLRs) in software engineering (SE), emphasising that while SE practitioners generate a vast amount of multivocal literature, much of it remains outside academic forums. They argue that excluding this literature from systematic reviews results in missed opportunities to capture valuable insights into current state-of-the-art practices in SE.

2.2.2 Study Protocol

The study protocol details the systematic approach used to identify relevant literature for this research, adhering to the guidelines established by Garousi et al. [29]. It describes the databases used, the search strategy employed, the inclusion and exclusion criteria applied to select the relevant sources, and the process used to catalog the literature. For this Multivocal Literature Review (MLR), Google's search engine was used to identify relevant sources, with Google Search helping to locate professional literature such as white papers, blogs, and articles. Additionally, academic databases, including Scopus, ACM Digital Library, IEEE Explore, and Springer, were used to retrieve published literature.

The search terms used for this study included the following string: (”behaviour driven development” OR ”bdd” OR ”gherkin”) AND (”characteristics” OR ”challenges” OR ”benefits” OR ”best practices”). After retrieving the initial search results, a selection process was conducted to filter out irrelevant papers based on the following inclusion and exclusion criteria.

- **Inclusion criteria:**
  - Literature explicitly discussing Behaviour Driven Development (BDD).

  - Literature explicitly discussing the Gherkin language.
  - Literature covering characteristics of BDD or Gherkin.
  - Literature covering best practices for BDD or Gherkin.
  - Literature covering challenges of BDD or Gherkin.
  - Literature covering benefits of BDD or Gherkin.
  - Only the first 100 results from Google Search are considered, following the research approach of Raulamo-Jurvanen et al. [30].

- **Exclusion criteria:**
  - Literature that is inaccessible.
  - Search results deemed too similar by Google.
  - Vendor tool advertisements.

Search Procedure. The process begins with a search across multiple sources, including IEEE Explore, ACM Digital Library, Scopus, Springer, and Google Search. The retrieved literature is then systematically reviewed using the predefined inclusion and exclusion criteria, ensuring that only relevant sources are selected for the primary study. This process is illustrated in Figure 2.1.

2.3 Results

In the following section, the results are demonstrated from executing the search followed by a review of the published and professional literature.

2.3.1 Search Execution

The initial search yielded 1075 results. After applying the inclusion and exclusion criteria, 202 results remained. Table 2.1 provides a summary of the search process.

2.3.2 Published Literature

In this section, the published (formal) literature on BDD (e.g., journal and conference papers) is presented. This section is divided into seven subsections. Subsection 2.3.2 introduces Behaviour Driven Development (BDD) features such as its ubiquitous language and its iterative decomposition process. Subsection 2.3.2 demonstrates the benefits of the BDD

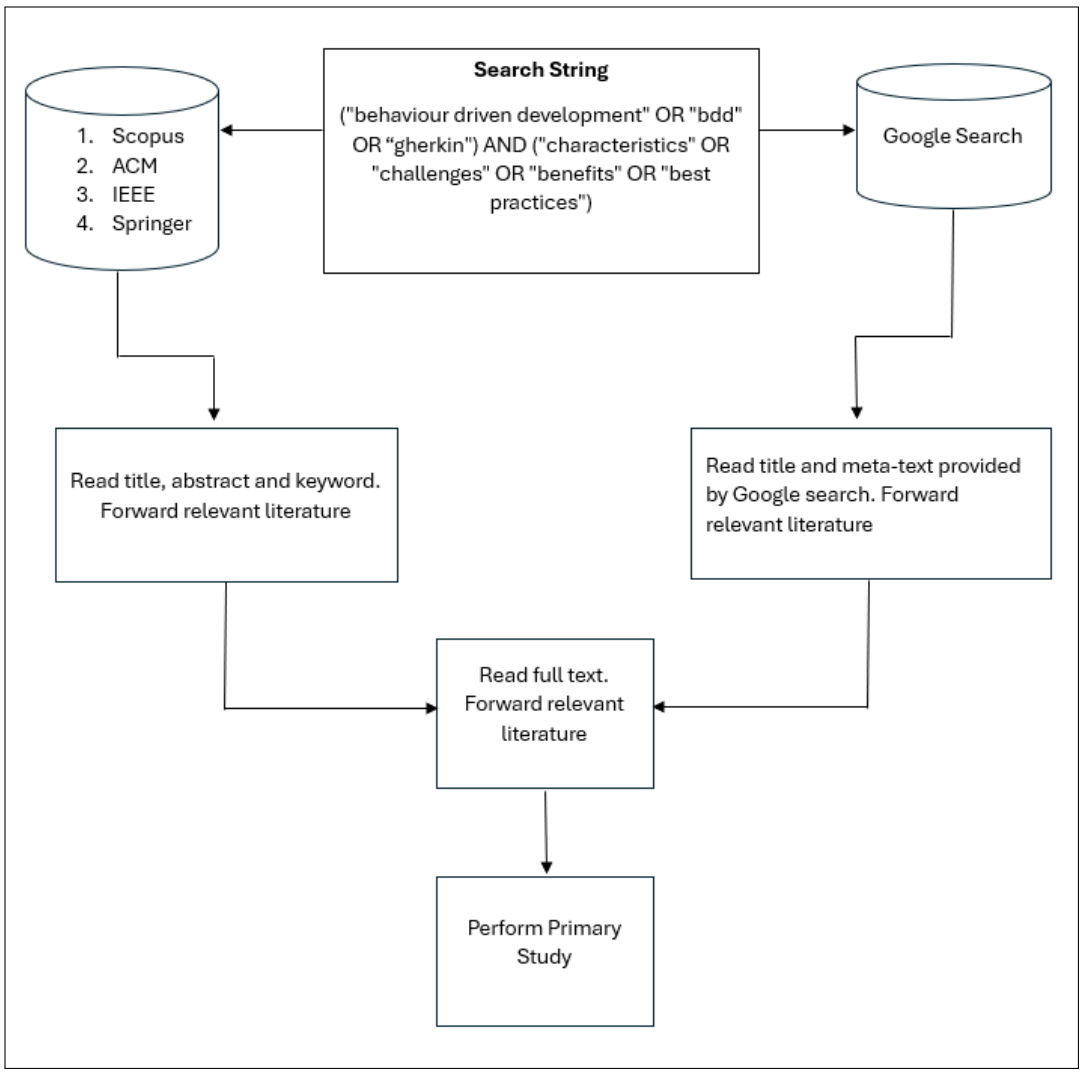

Figure 2.1: An overview of the search process to find relevant literature for this research.

Search engine	Initial results	Title, abstract, keywords and Meta text results	Full text
IEEE Explore	229	117	75
ACM Digital Library	225	98	41
Scopus	359	99	46
Springer	162	67	20
Google Search	100	41	20

Table 2.1: Summary of search results for the multivocal literature study.

approach such as improving communication and collaboration between development teams and customers. Furthermore, the section demonstrates BDD challenges such as convincing developers and customers to use BDD. Subsection 2.3.2 presents BDD tools for multiple programming languages such as Java, Ruby and Python. Subsection 2.3.2 presents work on the automated generation of tests in BDD. This includes the automated generation of step implementation functions from user scenarios, and the automated generation of stub step definition functions. Subsection 2.3.2 looks at related work on BDD maintenance such as the reusability of BDD acceptance tests, and refactoring BDD tests. Subsection 2.3.2 illustrates related work on the quality of BDD specifications. Subsection 2.3.2 looks at BDD case studies in specialist areas such as hardware design, and computer networking.

BDD Characteristics

Solís and Wang [31] identified six key features of BDD based on the existing literature on BDD. The first feature is *Ubiquitous Language*. By using the same simple language, developers and customers are able to communicate more effectively, reducing ambiguity. As development progresses, ubiquitous language should be used throughout the development life cycle. It should be derived from the analysis phase, but can be added as development progresses.

The second feature is *Iterative Decomposition Process*. In this process, the behaviour of the system is defined, which then becomes business outcomes. Through discussions between developers and customers, business outcomes can be narrowed down to a feature set, and customers will specify which outcomes are of high priority. After identifying features, user stories can then be constructed to describe how users interact with the system. Each user story contains multiple written scenarios, which represent its context and outcomes.

Plain Text Description with User Story and Scenario Templates are provided by BDD. These templates allow user stories and scenarios to be written in a ubiquitous language following North [10] guidelines. Outlined by North [10], the template for developing a user story is as follows:

[Story Title] (One line describing the story)
As a [Role]
I want a [Feature]
So that I can get [Benefit]

The following is the template for developing user scenarios:

Scenario 1: [Scenario Title]

Given [Context]
And [Some more contexts]....
When [Event]
Then [Outcome]
And [Some more outcomes]....
Scenario 2: [Scenario Title] ....

The fourth feature is *Automated Acceptance Testing with Mapping Rules*, which means each statement in the user scenario is automatically mapped to a test function in the system's programming language. Mapping rules vary in different toolkits.

Readable Behaviour Oriented Specification Code is ensured by mapping rules. Since the code is readable and includes the specification, it can be incorporated into the system documentation. In order to ensure readable code, methods should be named using sentences that display what they should do.

Finally, *BDD is Behaviour Driven at Different Phases*, where behaviour is considered throughout the entire software development process. The system's expected behaviour is mapped to business outcomes during the planning phase. Business outcomes become features that show system behaviour in the analysis phase. In the implementation phase, developers of the BDD system should pay attention to the behaviour of components and how each component interacts with other objects.

BDD Benefits and Challenges

Scandaroli et al. [32] presented the benefits of adopting BDD approach in the development of two software systems. The reported benefits are as follows: improved communication and collaboration between customers and developers, better software equality, living documentation, improved confidence among team members. This paper effectively demonstrates the practical application of BDD and its benefits, however, if would be helpful if quantitative data, broader case studies, and clearer directions for future research were incorporated.

BDD was also found to have benefits and challenges in another study. Binamungu et al. [13] surveyed 75 BDD practitioners, aiming to investigate the benefits, challenges, and opportunities associated with maintaining BDD specifications. The authors reported four main benefits of using BDD approach in industry. The first identified benefit is *The Use of Domain Specific Terms*. Second, *Improving Communication among Stakeholders*. System specifications can be easily understood by BDD teams due to the fact that it is written in a ubiquitous domain specific language. Third, *The Executable Nature of BDD Specifications*. The final benefit identified is *Facilitating Comprehension of Code Intention*; BDD produces maintainable and

scalable code, enhances collaboration between business stakeholders and developers, and improves productivity and quality of software systems.

Additionally, the authors identified some challenges of adopting BDD in industry. The first reported challenge is that according to most respondents, BDD requires a change in the way teams approach software development. Willingness to learn a new software development methodology is crucial to resolving this challenge. Second, BDD tests have similar maintenance challenges to automated tests such as fault detection and refactoring. Consequently, the authors believe it is crucial to study test maintainability in BDD context. The final identified challenge is that manual duplication detection is a complicated task for BDD practitioners. This study provides a clear direction for future research in BDD maintenance and makes a significant contribution to the understanding of BDD maintenance. Despite this, a more diverse sample and a more detailed quantitative analysis would be beneficial to the study.

Furthermore, Pereira et al. [33] investigated the benefits and challenges of exploiting BDD in software development through interviewing 24 IT professionals. The paper reported the following benefits; improved communication and collaboration between BDD team members, and generating living documentation and reports, which is very useful since software projects change very often. The identified challenges are; in order to write good quality scenarios, novel BDD adopters need training and experience. Also, it can be difficult to convince business stakeholders and developers to use BDD. This study presents useful insights into BDD's benefits and challenges. However, in order to fully realise the implications of BDD adoption in industry, further research is needed to address the limitations of the study, such as the small sample size of participants and the lack of quantitative data.

BDD Tools

A large number of development teams have been practising BDD. BDD has succeeded in drawing the attention of the software industry and as a result of this, multiple toolkits have been developed to aid the implementation of BDD for different programming languages and platforms. Tools have been developed for Java, Ruby, Python, .Net, and several others [34].

Solís and Wang [31] studied and analysed the support that the existing toolkits provide to the characteristics of BDD. The authors analysed a number of BDD software tools in terms of the support they bring to the six BDD features mentioned earlier in Section 2.3.2. The tools that the authors selected were based on their popularity among BDD practitioners. The tools are as follows; Specflow [35], JBehave [36], RSpec [37], NBehave [38], MSpec [39], StoryQ [40], and Cucumber [41]. Table 2.2 presents the summary of their findings.

By examining seven popular BDD toolkits, the paper provides insight into how well each toolkit supports BDD characteristics. Practitioners looking to select an appropriate toolkit

Support of the BDD Characteristics		xBehave Family		xSpec Family		StoryQ	Cucumber	SpecFlow
		JBehave	NBehave	RSpec	MSpec			
Ubiquitous language definition		✗	✗	✗	✗	✗	✗	✗
Iterative decomposition process		✗	✗	✗	✗	✗	✗	✗
Editing plain text based on	User story template	✓	✓	✗	✗	✗	✓	✓
	Scenario template	✓	✓	✗	✗	✗	✓	✓
Automated acceptance testing with mapping rules		✓	✓	✗	✗	✗	✓	✓
Readable behaviour oriented specification code		✓	✓	✓	✓	✓	✗	✓
Behaviour driven at different phases	Planning	✗	✗	✗	✗	✗	✗	✗
	Analysis	✓	✓	✗	✗	✗	✓	✓
	Implementation	✓	✓	✓	✓	✓	✗	✓

Table 2.2: Summary of features of Behaviour Driven Development toolkits. ✓– The toolkit supports the BDD characteristic, ✗– The toolkit does not support the BDD characteristic, taken from [31].

Support of Features	Cucumber	Concordion	Spock	JBehave	easyb
Business readable input	+	+	-	+	-
Business readable output	+	+	+	+	+
Creation of a ubiquitous language	-	+	-	-	-
Support of a predefined ubiquitous language	+	-	+	+	+/-
Automated acceptance tests	+	+	+	+	+
Plain text description of user stories and scenarios	+	+	+/-	+	-
Unit-testing facilities	-	-	+	+/–	+/-
Facilities for testing Web applications	+	+	+	+	+

Table 2.3: Comparison of BDD tools, taken from [42].

can benefit from this analysis. Furthermore, in the study, gaps in current BDD toolkits were identified, particularly in terms of their lack of support for the planning and analysis phases. Therefore, this encourages the development of more comprehensive BDD tools.

A second study conducted by Okolnychyi and Fögen [42] compared five BDD tools that were built for JVM-based programming languages, which are Java, Scala, and Groovy. The tools that they compared are Concordion [43], Spock [44], Cucumber [41], JBehave [36], and easyb [45]. The comparison is based on eight BDD features that these tools support either fully or partially. The features identified; business readable input, business readable output, creation of a ubiquitous language, support of a predefined ubiquitous language, automated acceptance tests, plain text description of user stories and scenarios, unit-testing facilities and facilities for testing Web applications. Complete support is marked by ”+”, whereas partial support is marked by ”+/-” or ”+/–” depending on the level of support (see Table 2.3).

The research provides a clear framework for reviewing and comparing BDD tools for JVM-based languages. However, it lacks empirical validation and quantitative metrics. Furthermore, this research could benefit from incorporating more BDD tools for other programming environments. Consequently, a broader audience could benefit from the study.

Another study related to BDD tools was conducted by Zampetti et al. [25]. The authors investigated the utilisation of BDD tools in open-source projects and how BDD specifications develop in the projects' code. The authors analysed 50,000 popular open-source projects developed in five programming languages to understand the extent of usage of BDD frameworks in these sampled projects. Furthermore, they studied how scenarios, fixtures and code bases co-evolved in 20 projects. Additionally, they surveyed 31 BDD practitioners to understand how they utilise BDD frameworks. The authors learnt that approximately 27% of the 50,000 projects adopt BDD frameworks, see Table 2.4. Furthermore, co-development

Language	Behave	Cucumber	Jasmine	Jbehave	Mocha	Rspec	Overall
Java	✗	78 0.78%	✗	57 0.57%	✗	✗	125 1.25%
Javascript	✗	39 0.39%	1206 12,06%	✗	2915 29.15%	✗	3,902 39.02%
Python	2522 25.22%	108 1.08%	✗	✗	✗	✗	2,578 25.78%
PHP	✗	226 2.26%	✗	✗	✗	✗	226 2.26%
Ruby	✗	706 7.06%	✗	✗	✗	6735 67.35%	6,794 67.94%
Overall	2522 5.04%	1157 2.31%	1206 2.41%	57 0.11%	2915 5.83%	6735 13.47%	13,625 27.25%

Table 2.4: Usage of BDD tools in 50,000 projects (10,000 for each programming language) in GitHub, taken from [25].

between scenarios, fixtures and the code base was identified in around 37% of cases. They also learnt that most of the 31 survey participants wrote scenarios and tests during or after writing code. This is even though they understand that tests should be written before coding.

In a similar study, Chandorkar et al. [46] investigated how Gherkin language is utilised in open-source software projects. The study aimed to understand the actual usage of Gherkin features, such as data tables, which are intended to make specifications more compact and readable. The authors used the GitHub API to identify and retrieve a list of open-source repositories that use the Gherkin language. They focused on repositories with more than 500 stars to ensure a certain level of popularity and quality. The authors ended up with 23 repositories for analysis, containing a total of 1,572 Gherkin specification files. The research highlights a gap between the theoretical benefits of using Gherkin features and their actual adoption in practice. They concluded that while data tables and other Gherkin features are recommended to improve specification quality, their limited use suggests that these benefits may not be fully realised in real-world projects.

Automated Generation of Tests

Developers need to write the step implementation that corresponds to user scenarios developed in collaboration with customers. To eliminate the need for writing step implementations, a natural language processing (NLP) approach is explored by Kamalakar et al. [47] in order to automatically generate executable software tests from scenario descriptions expressed in natural language. Furthermore, the authors introduced a tool called Kirby that automates step function development. The authors achieved this by inspecting BDD user scenarios using a Natural Language Processing (NLP) approach, then linking them to im-

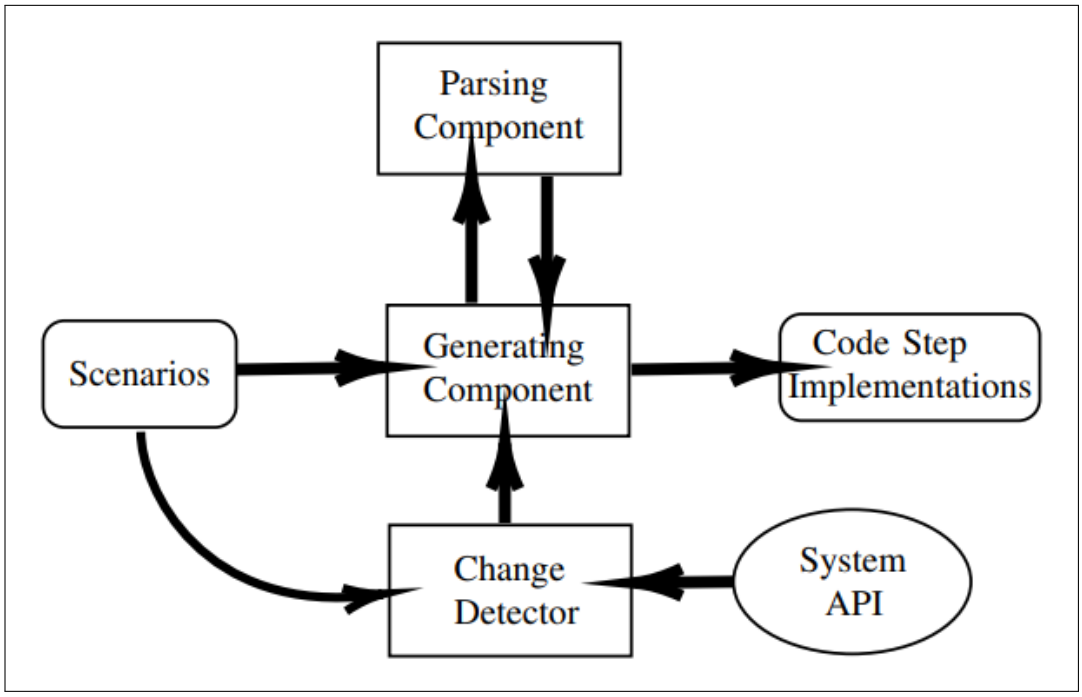

Figure 2.2: Behave-nicely's overview of the architecture, taken from [34].

plementation code already written by developers. Then test code is produced by using the project code that the user scenarios refer to. Multiple algorithms used in Kirby are evaluated in this paper in order to determine their accuracy. Kirby was evaluated on 12 BDD scenarios, testing the accuracy of various NLP algorithms in matching nouns with classes and objects, and verbs with methods. It was found that the accuracy was significantly improved when a combination of algorithms was used. While the evaluation highlighted the benefits of NLP processing, it also pointed out the limitations, including a dependency on comprehensive dictionaries that have a negative impact on the accuracy and speed of the process.

Another tool, named behave-nicely, to automatically produce executable step implementation functions is proposed by Storer and Bob [34]. The overall architecture for behave-nicely is illustrated in Figure 2.2. The Code generation is achieved by translating user scenarios into glue code using natural language processing (NLP) techniques. Usually, when software teams and business stakeholders update user scenarios, the implementation functions need to be changed accordingly to achieve consistency. This maintenance challenge was observed by the software industry and motivated the authors to develop behave-nicely. The authors evaluated behave-nicely on 50 black box projects and the tool successfully generated step definition functions for 17% of the projects.

In both previous papers, NLP was used to automate test generation, the Kamalakar et al.'s research taking a more broad conceptual approach, and the Storer and Bob's paper using the Python-based "behave_nicely" tool for BDD. This latter study investigates the practical challenges and success rates of NLP algorithms using mutation testing, while the former

compares them from a more theoretical perspective. They both aim to reduce manual testing maintenance, but the latter is more tool-specific, while the former explores the potential of natural language processing to automate software testing. It may be possible to enhance the applicability and efficiency of automated test generation by integrating the strengths of both approaches, expanding language support, and addressing performance challenges in future research.

Similarly, in another study, Soeken et al. [48] proposed the use of NLP techniques for generating step definitions and code skeletons from BDD scenarios written in natural language. The paper aims to reduce manual effort in the development of step definitions. The proposed design flow involves the user engaging in a dialogue with the computer. Each statement of the BDD scenarios is processed by the system and code constructs such as classes, attributes, and operations are suggested. By accepting, rejecting, or refining these suggestions, the user is training the system to better understand future statements. A candy machine case study was used to validate the methodology, describing six use cases in natural language. As a result, the class diagram and test cases were successfully generated with minimal user intervention, demonstrating the efficiency and effectiveness of the approach.

The previous work significantly reduces the manual effort required to develop step definitions and code skeletons from BDD scenarios, addressing a common BDD challenge. The approach leverages the widely used Cucumber tool to seamlessly integrate into existing BDD workflows. However, in evaluating the approach, a relatively simple case study was used that might not fully capture the complexities and edge cases present in larger, more complex systems.

Another study focuses on automated testing generation. In spite of the fact that adopting BDD methodology results in high code coverage with tests, Diepenbeck et al. [49] conducted an empirical study and observed that test coverage tends to decline towards the final stages of software development particularly in large-size projects. To address this issue, Diepenbeck et al. proposed automatically generating test cases from uncovered code. They developed an algorithm to generate test cases in BDD scenario style from uncovered methods. The authors evaluated their approach against 2 existing projects and concluded that it is efficient.

An automated scenario generation technique is presented by Diepenbeck et al. [49] to address coverage gaps in BDD by automatically generating BDD scenarios from uncovered code. Although BDD emphasises test-first development as a way to achieve high code coverage, practical implementations often fail to meet that goal in the long run. A BDD-style scenario generation algorithm is proposed by the authors to enhance test coverage. By exploiting a feature graph, the algorithm can identify and reuse existing steps for global coverage, and use symbolic execution techniques for local coverage to address uncovered lines of code within methods. In the paper, the researchers empirically examined two BDD projects to evalu-

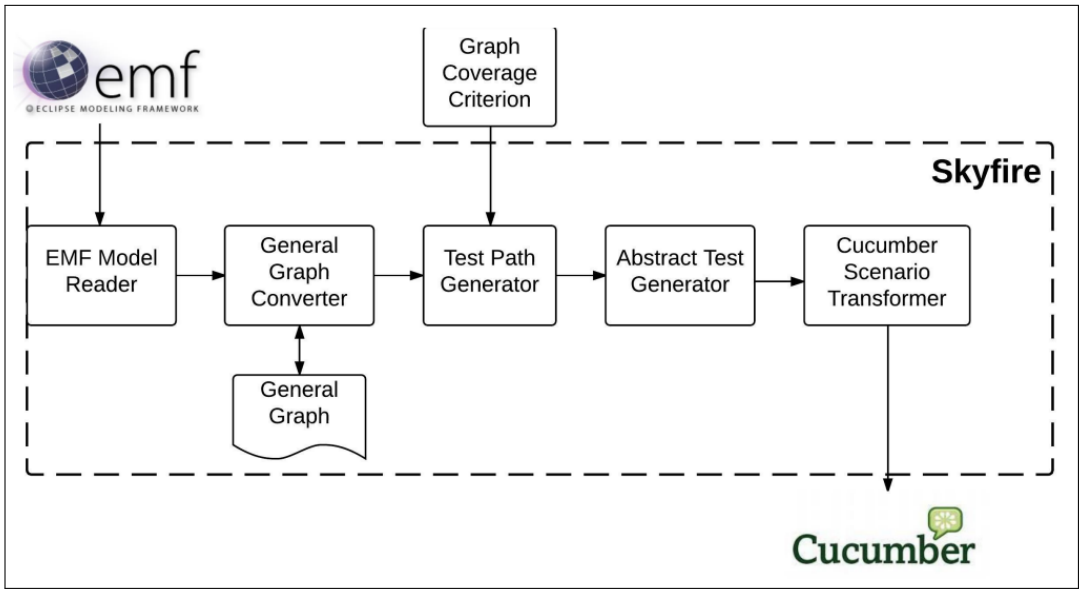

Figure 2.3: Overview of the architecture of Skyfire, taken from [50].

ate their approach. Based on the evaluation results, the algorithm proposed for automatic scenario generation was found to be effective.

The previous paper presented the innovative use of feature graphs and automated scenarios to address declining test coverage in BDD. In addition, the proposed approach proved to be practical and efficient through empirical validation in real-world projects. However, a validation based on only two projects may not fully represent the generalisability of the findings. Additionally, since the proposed method relies on reusing and integrating existing BDD scenario steps, it may be ineffective and impractical for larger projects.

In another study, Li et al. [50] presented a tool called Skyfire, see Figure 2.3, that automates the generation of BDD test scenarios for Cucumber, a popular BDD framework, through Model-Based Testing (MBT) techniques. BDD test scenarios are typically written manually. This approach addresses the manual effort required, providing a more efficient and systematic approach. According to the authors, Skyfire was evaluated in an industrial setting, showing that it generates more comprehensive and effective test scenarios than those developed manually. By integrating MBT with Cucumber, the paper highlights its innovative approach and practical application. In spite of this, the evaluation is limited because it is based mainly on a single case study, and the approach relies heavily on UML state machine diagrams, which may not be standard in all development environments.

BDD Maintenance

Maintaining traceability between Gherkin scenarios and their corresponding step definitions demands effort from BDD team developers. Yang et al. [51] attempted to improve

BDD maintenance by tracing the changes between Gherkin scenarios and the step definitions linked to them. The authors utilised Natural Language Processing (NLP) techniques to measure language similarities between feature files and their implementation code. They achieved 79% accuracy in automatically pinpointing co-changes between feature files and implementation code. Then the authors proposed a Machine Learning (ML) model that comprises of Naive Bayes, random forest and logistic regression to predict when user scenarios need to be edited. The authors evaluated their proposal on 50 BDD projects and achieved an AUC of 0.77. They claim that their work can help software teams keep software documentation up-to-date [51]. In this study, valuable insights and tools are provided for improving the maintainability and traceability of BDD projects. However, considering that this study focuses on Java projects with English commit logs, there is a possibility of sample bias. As a result of this limitation, the findings may not be generalisable to other languages or projects with documentation in languages other than English.

Duplicate user scenarios are a waste of space and effort. When BDD suites grow in size, duplicate test scenarios are introduced. Having duplicate specifications increases maintenance costs, extends execution times, and makes managing large BDD suites a challenge [52]. Binamungu et al. [52] attempted to control redundant BDD specifications by identifying duplicate Gherkin scenarios. They proposed a dynamic tracing technique to detect duplicate specifications. They evaluated their tool against 61 duplicate user scenarios across 3 projects. The tool detected more than 70% of duplicate scenarios. In this study, a practical problem faced by many software development teams using BDD is addressed. However, despite the positive results, the evaluation is limited to only three systems, which may not be representative of all BDD practices and systems.

In another BDD maintenance work, agile software practices such as BDD revise models, specifications, the system under test (SUT), and test components. Maintaining consistency among them is difficult and usually requires a lot of time and effort manually. Sathawornwichit and Hosono [53] proposed an automatic updating technique utilising metadata that maintains consistency among specifications, design models, SUT and test components. This is when changes occur. The authors reported that their approach is being evaluated. As an expansion to the foregoing study, Sathawornwichit and Hosono [54] developed a prototype for the automatic updating tool discussed earlier. The authors evaluated their tool and concluded that it is efficient and eliminates manual consistency maintenance among design models, specifications, SUT and test components.

The former paper presents a promising approach to addressing a challenge in agile software development: consistency reflection for automatic updating of test environments. Since further validation was necessary for this approach to improve the efficiency and effectiveness of automated testing, the latter paper evaluated the approach and the results of its application were reported. Although the proposed framework exploits the existing BDD tools such as

Cucumber to make it more accessible to developers, the focus on Cucumber and BDD may limit the generalisability of the framework to other agile practices and tools.

Another study examined BDD maintenance; in general, existing research has not focused on refactoring BDD specifications, but rather on refactoring code in BDD. Irshad et al. [55] propose and evaluate techniques for refactoring BDD specifications to improve their maintainability. Using Action Research, the study iteratively tests and refines similarity measures to identify refactoring candidates among BDD specifications in real-world software development environments. Evaluation results show that the proposed measures are effective for identifying refactoring candidates, ensuring a high degree of alignment between practitioners' assessments and similarity measures. Despite the innovative use of similarity measures for refactoring BDD specifications in the study, it has limitations, including a limited generalisability due to the fact that the study focuses on two specific products and the possibility of dependency on specific tools.

Quality of BDD Specifications

Binamungu et al. [15] investigated the characteristics that determine the quality of BDD suites. They surveyed BDD practitioners in order to understand what determines good BDD specifications. Four principles were proposed by the authors in order to assess how good BDD specifications are. The first principle proposed is *Conservation of Steps*; The focus should be on developing easily understandable steps. *Conservation of Domain Vocabulary*; As long as the specification remains readable by customers, new domain terms should be allowed whenever necessary. Third, *Elimination of Technical Vocabulary*; in some cases implementation words can be used.

The final principle identified was *Conservation of Proper Abstraction*; There are two styles of BDD scenario steps: declarative (higher level steps) and imperative (lower level steps). Higher level steps often contain more semantics, so scenarios can be expressed with fewer steps and relate to existing domain concepts. This means that it may take more time and effort for developers to write extensive glue code. As a result, it can be difficult to interpret scenarios without referring to glue code. A lower level step, on the other hand, describes the system under development in more detail. Due to this, scenarios that use low level steps tend to take longer, requiring more steps to express the same meaning. However, they are easier to implement because there is less glue code. It will be easier for BDD feature suites to comprehend, extend, and maintain when scenarios are written at a consistent level of abstraction. A maintenance engineer may have trouble deciding which level of abstraction to apply when writing a new scenario when a feature suite contains both low level and high level scenarios. Also, there will likely be redundant steps and glue code, as well as inconsistent abstraction levels in the test harness code, resulting in additional comprehension

and maintenance difficulties Binamungu et al. [15].

The research contributes significantly to the understanding and improvement of BDD specifications. However, this survey had 56 respondents, which might make it difficult to generalise the results to all BDD practitioners. Furthermore, the majority of respondents were from Europe and North America, which could lead to geographic bias. The generalisability of the findings could be improved by increasing the number of survey participants and ensuring a more diverse geographical representation. Additionally, the research does not provide details on the tooling and automated support required to implement the proposed principles. Therefore, it would be beneficial to develop or recommend tools that facilitate adherence to these quality principles in the future.

In another study, Oliveira and Marczak [56] investigated the quality of BDD scenarios and proposed a number of attributes that can help in evaluating the quality of BDD scenarios. The authors performed a study to evaluate BDD scenarios with students who used to work as software developers and are experienced in writing use cases but have limited experience in writing user scenarios. The authors concluded their study with a list of attributes that can be used to judge the quality of BDD scenarios. The attributes are concise, estimable, feasible, negotiable, prioritised, small, testable, understandable, unambiguous, and valuable.

Furthermore, Oliveira et al. [57] redefined the attributes of the previous study and proposed a redefined set of quality attributes that can be used to evaluate BDD scenarios. The attributes are integrous, essential, unique, focused, complete, singular, ubiquitous, and clear. Furthermore, they proposed a checklist that can help software developers evaluate the quality of BDD scenarios , see Table 2.5.

By addressing the lack of formal guidelines for evaluating scenario quality, the previous two research papers contribute significantly to the field of BDD. Although the proposed quality attributes and checklist provide practitioners with valuable resources, the study could benefit from a larger sample size and further validation in real-world settings. The long-term effects of utilising these guidelines could be explored in future research, and the checklist could be extended or refined based on wider feedback.

Similarly, in the Test Driven Development (TDD) world, Borle et al. [58] investigated its impact on software development productivity and quality. Their findings revealed that, despite the theoretical advantages of TDD, its practical application in the studied projects did not yield the anticipated benefits, raising concerns about its effectiveness in real-world settings.

Case Studies in Specialist Areas

BDD's adaptability and challenges have been explored across diverse domains, offering insights into its practical utility and limitations. In hardware design, BDD facilitated unified

ID	Question	Scope	Attribute
1	Can the feature file business value or outcome be identified by its description?	Feature	Unique
2	Does the feature file has any missing scenarios?	Feature	Complete
3	Does the scenario carry all the information needed to understand it?	Scenario	Complete
4	Does the scenario has steps that can be removed without affecting its understanding?	Scenario	Essential
5	How different each scenario is from the others?	Scenario	Unique
6	Can the scenario single action be identified on its title and match what the scenario is doing?	Scenario	Singular
7	Can the scenario outcome or verifications be identified on its title and match what the scenario is doing?	Scenario	Singular
8	Does the scenario respect Gherkin keywords meaning and its natural order?	Scenario	Integrous
9	Does the step correctly employs business terms, including a proper actor?	Step	Ubiquitous
10	Does the step has details that can be removed without affecting its meaning?	Step	Essential
11	Does the step express ”what” it is doing by being written in a declarative way?	Step	Focused
12	Does the step allow different interpretations by being vague or misleading?	Step	Clear

Table 2.5: Question-based checklist for BDD scenarios, taken from [57].

specifications for testing and formal verification [59, 60], though scalability for complex systems remains unproven. Healthcare applications demonstrated BDD's role in automating compliance testing [61] and enhancing traceability [62], yet revealed gaps in stakeholder alignment. IoT and security studies highlighted BDD's potential for improving interoperability [63] and defect detection [64], though broader validation in industrial settings is needed. Educational initiatives leveraged BDD to teach problem-solving [65] and logical reasoning [66], though long-term pedagogical impacts require further investigation.

Common themes include BDD's capacity to bridge technical and non-technical stakeholders through shared language and its effectiveness in automating workflows. However, recurring limitations such as small sample sizes, tool dependency, and context-specific constraints underline the need for more rigorous, generalisable evaluations. These studies collectively affirm BDD's versatility while emphasising domain-specific adaptations to address unique challenges.

Summary

The published literature review indicates that although there is extensive tool support for BDD, these tools are not always employed in accordance with the theoretical model. For instance, the survey conducted by Zampetti et al. [25]—as detailed in Section 2.3.2—suggests that BDD tooling is primarily used for documenting unit tests, rather than for the construction, design, or implementation of software. Moreover, the findings by Chandorkar et al. [46] highlight a discrepancy between the theoretical advantages of Gherkin features and their practical implementation; while these features are promoted for enhancing specification quality, their limited adoption implies that such benefits may not be fully realised in real-world projects. Similarly, research in the Test Driven Development (TDD) domain by Borle et al. [58] found that, despite the theoretical benefits of TDD, its practical application in the projects examined did not yield the expected improvements in productivity and quality, thereby questioning its real-world effectiveness.

Additionally, the published literature provides limited insight into the actual practices of BDD and the evidence supporting purported best practices. The comparative effectiveness of using Business Domain language versus a User Interface Domain language for Gherkin scenarios remains unexplored, and the assumption that Business Domain language offers superior comprehension and documentability has not been empirically tested. Furthermore, Binamungu et al. [15] reported that BDD practitioners often struggle to determine the appropriate level of abstraction when crafting new scenarios—a challenge that can lead to frustration and, in some cases, the abandonment of BDD.

2.3.3 Professional Literature

This section presents the results of the professional literature on BDD and its best practices.

Overview

BDD best practices analysed from twenty sources in the professional literature reveal a strong focus on maintaining clarity, independence, and proper structure. Table 2.6 demonstrates the most frequently discussed BDD best practices in the literature, ordered by occurrence frequency.

Write declarative steps

This practice is the most frequently mentioned across the sources demonstrated in Table 2.6, emphasising the importance of writing BDD scenarios in a way that focuses on what the system should do rather than how it should do it. Software specifications in the BDD domain can be written in two styles:

- **Declarative Style:** In the context of writing Gherkin steps for BDD, a declarative style focuses on what the system should do, without specifying the details of how it should be done. This approach abstracts away the implementation details.
- **Imperative Style:** Conversely, an imperative style describes how the system should accomplish a task, often including technical details that explain the sequence of actions needed to achieve a particular result. This style tends to be more detailed and specific.

For example, Figure 2.5 demonstrates a BDD scenario written in declarative style. In this style, instead of detailing the steps to log a user in, a declarative step might simply state, “Given the user is logged in”. In this approach, the scenario is focused on the behaviour, rather than the mechanics of the process. In contrast, the imperative style spells out how the system should achieve the desired outcome, often including detailed instructions. Figure 2.4 demonstrates a BDD scenario written in imperative style. In this style more guidance and details are provided.

The twelve sources that reported writing declarative steps as a BDD best practice, agree on the core benefit of declarative steps: they enhance the readability, maintainability, and flexibility of BDD scenarios. By focusing on what should happen rather than how it happens, declarative steps make it easier to understand the desired behaviour of the system, regardless of the underlying implementation. This approach also makes the tests more resilient to changes in the code base, reducing the need for frequent updates to the scenarios.

Best Practice	Occurrence	Citations
Write declarative steps	12	TestQuality [67], Hartill [68], Honkanen [69], Helm [70], Toledo and Zambra [71], Sheffer [72], Shah [73], Thorn [74], Jorente [75], Knight [12], Azevedo [76], Ghahrai [77]
Ensure scenarios are independent	11	TestQuality [67], Hartill [68], Toledo and Zambra [71], Sheffer [72], Shah [73], Hamilton [78], Thorn [74], Khunteta [79], Topcu [80], Azevedo [76], Specflow [81]
Write clear scenario steps and avoid using technical terms	10	Ghahrai [77], Azevedo [76], John [82], Knight [83], Thorn [74] , Hamilton [78], Paul [84], SmartBear [85], Sheffer [72], Toledo and Zambra [71]
Structure scenarios in the correct order (Context-Action-Outcome)	7	TestQuality [67], Hartill [68], Toledo and Zambra [71], Sheffer [72], Shah [73], Knight [12], Ghahrai [77]
Use Background structure to avoid repeating steps	6	Shah [73], Paul [84], Toledo and Zambra [71], Helm [70], Honkanen [69], TestQuality [67]
Refactor and reuse step definitions	6	John [82], Shah [73], SmartBear [85],Toledo and Zambra [71], Helm [70], Paul [84]
Avoid redundancy in steps	5	Azevedo [76], Topcu [80], Thorn [74], Shah [73], SmartBear [85]
Use Scenario Outline for multiple cases	5	Shah [73], Paul [84], SmartBear [85], Sheffer [72], TestQuality [67]
Avoid writing long scenarios	4	Knight [12], Thorn [74], Shah [73], TestQuality [67]
Use proper grammar, spelling, and punctuation	4	Knight [12], Knight [83], SmartBear [85], Sheffer [72]

Table 2.6: BDD Best Practices Ordered by Occurrence Frequency

```
Given the user navigates to the home page
And enters "username" in the "username" field
And enters "password" in the "password" field
When the user clicks on the login button
Then the user is logged in
```

Figure 2.4: BDD scenario written in imperative style, taken from Honkanen [69].

```
Given the user is logged in
And is on the home page
```

Figure 2.5: BDD scenario written in declarative style, taken from Honkanen [69].

The differences among these sources are minor and mostly related to the emphasis on certain aspects of declarative writing. For example, some sources like [12] and [83] focus on the readability and maintenance aspects, while others, like [69] and [67], emphasise the adaptability and reduced brittleness of tests written in a declarative style. Additionally, while most sources discuss the advantages of declarative steps, they do not provide specific imperative-style examples to contrast, but instead, they implicitly highlight the downsides of imperative steps by promoting the benefits of declarative steps.

Ensure scenarios are independent

The concept of ensuring scenarios are independent in BDD is a critical best practice discussed across eleven sources. At its core, scenario independence means that each test scenario should be able to run on its own, without depending on the outcomes or states established by other scenarios. This principle is essential for maintaining a reliable, maintainable, and robust test suite.

When these sources refer to scenario independence, they emphasise the importance of isolating each test case so that it does not rely on the side effects or results of another test. This practice helps to prevent a failure in one scenario from causing cascading failures in others, which can make debugging complex and time-consuming.

For example, Toledo and Zambra [71] believe that it is crucial to ensure that scenarios are as independent as possible, meaning they should not be interdependent or coupled. For example, it is not advisable for one scenario to insert records into a database and for subsequent scenarios to rely on the existence of those records. Coupled scenarios can lead to issues, such as errors when running tests in parallel or when a single scenario fails, potentially causing a ripple effect across other tests.

The eleven sources that reported ensuring independent scenarios as a BDD best practice, agree that scenario independence is essential for creating a reliable and maintainable test

suite. Furthermore, there is a consensus that independent scenarios simplify debugging because they can be run and tested individually without interference from other tests. Additionally, the sources highlight that independence helps prevent cascading failures, where one failed scenario could potentially cause others to fail if they are dependent.

Write clear scenario steps and avoid using technical terms

The practice of writing clear scenario steps and avoiding technical terms is a significant practice in BDD, as highlighted by the ten sources identified. This best practice is aimed at ensuring scenarios are clear and accessible to all stakeholders, regardless of their technical background. It ensures that the focus remains on the intended behaviour of the system rather than being trapped by technical details.

Writing Clear Scenario Steps refers to the practice of using simple, straightforward language to describe what the system should do in a way that is easily understandable by anyone involved in the project. This includes developers, testers, business analysts, and even non-technical stakeholders like product owners or customers.

Avoiding technical terms means refraining from using any language that might be confusing or inaccessible to non-technical members of the team. The goal is to ensure that the scenarios describe the behaviour of the system in plain language, focusing on what the system should achieve rather than how it achieves it. For example, using a phrase like "Given the user is authenticated" instead of a more technical "Given the user has a valid session token".

The ten sources that reported writing clear scenario steps and avoiding using technical terms as a best practice, agree that writing clear, non-technical scenarios makes them accessible to all team members, regardless of their technical background. This inclusivity is crucial for effective collaboration and communication. Furthermore, by avoiding technical jargon, scenarios remain aligned with business requirements and are easier for business stakeholders to understand and validate. Additionally, the consistent message is that scenarios should describe the intended behaviour of the system in plain language, keeping the focus on what the system should do rather than how it does it.

While all the ten sources support the idea, some, like Knight [12] and SmartBear [85], place a stronger emphasis on the inclusivity aspect, ensuring that everyone on the team can participate in the discussion. Others, like Toledo and Zambra [71], emphasise the alignment with business objectives and the clarity of communication with non-technical stakeholders.

Structure scenarios in the correct order (Context-Action-Outcome)

The best practice of structuring scenarios in the correct order, referred to as Context-Action-Outcome or Given-When-Then in BDD emphasises the importance of organising each sce-

```
Feature: Google Searching

Scenario: Search from the search bar
Given a web browser is at the Google home page
When the user enters "panda" into the search bar
Then links related to "panda" are shown on the results page
```

Figure 2.6: Correct structure of Given-When-Then, taken from Knight [12].

nario logically and clearly. This structure is essential for creating scenarios that are easy to understand, maintain, and communicate among all team members, regardless of their technical background.

Context-Action-Outcome (Given-When-Then) refers to the sequence in which steps are organised in a BDD scenario:

1. **Given (Context):** This step sets up the initial state or context. It describes the conditions that must be met or the state the system should be in before the action takes place.

2. **When (Action):** This step specifies the action that triggers the behaviour you want to test. It is the event or change that occurs in the system.

3. **Then (Outcome):** This step details the expected result or outcome after the action has been executed. It describes what should happen if the system behaves as expected.

Figure 2.6 demonstrates a Gherkin scenario example, where the correct order of Given-When-Then is followed:

The seven sources that reported structuring scenarios in the correct order as a best practice, agree that using the Given-When-Then structure is crucial for maintaining clarity in BDD scenarios. This format ensures that scenarios are easy to follow and understand, which is vital for effective communication among team members. Furthermore, this structure mirrors the natural flow of describing events: setting the stage (Given), taking an action (When), and observing the result (Then). This helps in clearly communicating what is being tested and why.

Use Background structure to avoid repeating steps

The “Background” structure in Gherkin is a feature that allows common setup (Given) steps to be defined once and reused across multiple scenarios within the same feature file. This practice is discussed and recommended by six sources among the twenty sources identified

```
Feature: Money Withdrawal

Background:
Given The credit card is enabled
And The available balance in my account is positive
And the ATM has enough money

Scenario: ...
```

Figure 2.7: Background structure example, taken from Toledo and Zambra [71].

in Table 2.6. The key idea is to avoid redundancy, enhance readability, and maintain the clarity of the scenarios by centralising shared context (Given) steps.

Using the "Background" structure means placing any common preconditions or setup steps that are required by all scenarios within a feature into a special "Background" section. This section runs before each scenario, ensuring that the scenarios are focused only on the specific actions and outcomes relevant to the behaviour being tested, rather than repeating the same setup (Given) steps over and over.

Figure 2.7 demonstrates a Background structure example, where this set of setup (Given) steps will be executed before each scenario in the feature.

The primary benefit reported in the literature of using the "Background" section is to avoid the repetition of common setup steps. This not only reduces redundancy but also makes the feature files easier to read and maintain. By using the "Background" structure, the scenarios remain focused on the specific actions and outcomes being tested, rather than being cluttered with repetitive setup steps.

Refactor and reuse step definitions

The best practice of refactoring and reusing step definitions in BDD focuses on the need to keep step definitions in Gherkin scenarios clean, efficient, and reusable. This approach is vital for making sure that BDD scenarios remain easy to manage and adapt as a project grows.

Refactoring step definitions involves revising and improving them to make them more concise and applicable across multiple scenarios. Reusing step definitions means using the same, well-written steps across different scenarios or features, rather than creating new steps each time. The consensus is that refactoring and reusing step definitions reduce maintenance cost, and ensure consistency across the test suite.

```
Scenario outline: Withdraw money with different card keys

Given The credit card is enabled
And The available balance in my account is positive
And the ATM has enough money
When I put the card in the ATM
And Enter the pin of the card

...

Examples:

| pin |

| 1234 |

| 9876 |

| 5432 |
```

Figure 2.8: Scenario outline structure example, taken from Toledo and Zambra [71].

Avoid redundancy in steps

Five sources highlight the importance of avoiding redundancy in steps as a best practice in BDD. By eliminating duplicate steps, teams can develop Gherkin scenarios that are more efficient, easier to read, and simpler to maintain. This approach ensures that every step has a specific role and directly supports the behaviour being tested. All five sources agree that this practice is essential for keeping a test suite clean and effective.

Use Scenario Outline for multiple cases

"Scenario Outline" structure in Gherkin lets you create a single scenario template that can run multiple times with different sets of data. This is especially helpful when you need to test the same actions under various conditions or edge cases. Rather than writing several scenarios that are almost identical except for the input data, you can use one "Scenario Outline" along with an "Examples" table to cover all the cases efficiently.

Figure 2.8 demonstrates a Scenario outline structure example, where the scenario will be executed multiple times with different card keys.

Five among the twenty sources demonstrated in Table 2.6 recommend using "Scenario Outline" in Gherkin to handle multiple cases. This approach enables teams to run the same test scenario multiple times with different sets of data, which helps eliminate redundancy and keeps the test suite organised and efficient.

Avoid writing long scenarios

Avoiding long scenarios means keeping each Gherkin scenario short, clear, and focused on just one function or behaviour. When scenarios try to cover too many actions or outcomes at once, they can become confusing, harder to understand, and more complicated to maintain.

Four sources recommend writing short and concise scenarios. The consensus across these sources is that long scenarios should be broken down into smaller scenarios to improve clarity, readability, and overall effectiveness of the test suite. For example, Knight [12] claims that keeping scenarios short and concise ensures they are easy to read and understand. Furthermore, he recommends that each scenario should have no more than nine steps.

Use proper grammar, spelling, and punctuation

Although using proper grammar, spelling, and punctuation in Gherkin scenarios is important, only three of the twenty sources specifically focused on this practice. These sources emphasise that clear and correct language is crucial for effective communication among team members and stakeholders. By using proper grammar, spelling, and punctuation, teams can minimise the risk of misunderstandings and ensure that scenarios are easy to read and comprehend.

Summary

This section described the professional literature review on the best practices of BDD. Multiple best practices have been identified by BDD practitioners. Among the identified best practices of BDD are: avoid writing long scenarios, refactor and reuse step definitions, structure scenarios in the correct order, and write clear scenario steps.

System behaviour scenarios can be written in two styles: declarative and imperative. In this professional literature review, twelve authors advocate for the declarative style as the best practice for writing behaviour scenarios in BDD. However, one author claims that the specific style is less important than ensuring scenarios are clear and not too abstract or overly imperative. It is noteworthy that none of the twelve authors advocating for the declarative style of writing system behaviour scenarios have provided empirical evidence to support their preference.

Discussing the writing styles for system behaviour scenarios in thirteen sources highlights just how important it is to make these scenarios readable and easy to understand for everyone. This extensive focus shows that clear and effective communication is highly valued in BDD.

2.4 Literature Gap Identification

The multivocal literature review underlines Behaviour Driven Development (BDD) as a transformative methodology in Agile software engineering, praised for its ability to bridge communication gaps through natural language specifications (Gherkin) and foster collaboration across technical and non-technical stakeholders. However, critical challenges persist:

1. **Ambiguity in Abstraction Levels:** While declarative scenarios that are written in business domain language are promoted as a best practice, stakeholders often struggle to articulate requirements in abstract business terms. This creates friction in requirement negotiation and test maintenance, as highlighted by Binamungu et al. [15].
2. **Empirical Validation Deficits:** Professional literature overwhelmingly advocates for Business Domain Specific Gherkin, asserting it improves comprehension and reduces brittleness. Yet, no rigorous empirical studies validate these claims, leaving a disconnect between industry assumptions and academic scrutiny.
3. **Maintenance Overhead:** Automated acceptance tests tied to business domain steps require extensive custom glue code increasing developer effort. Binamungu et al. [15] observed that inconsistent abstraction levels in scenarios increase maintenance costs.

The Research Gap: Despite widespread adoption, **no empirical evidence exists to support the presumption that Business Domain Gherkin outperforms alternative abstraction paradigms** (e.g., user interface specific language) in comprehension, documentation, or maintenance. Furthermore, user interface domain specific languages may ease maintenance by standardising 'glue' code, while also improving the documentation and comprehension of tests by referencing specific, concrete artifacts within user interfaces.

This thesis addresses these gaps by:

- Proposing a UI Domain Specific Gherkin framework that standardises user interface interactions, reducing ambiguity and glue code maintenance.
- Empirically testing whether UI-centric scenarios improve comprehension and documentation efficiency compared to business domain equivalents for both technical and non-technical stakeholders.

By challenging industry norms and providing evidence-based alternatives, this work aims to refine BDD practices and enhance their applicability in complex, UI-driven systems.

2.5 Summary

It can be concluded from the multivocal literature review that there is a lot of tool support for BDD and widely used, but not necessarily as per theoretical model. The results of the survey conducted by Zampetti et al. [25] and demonstrated in 2.3.2, suggest that BDD tooling is used for documentation of unit tests rather than for construction, design, and implementation of software. Additionally, the findings of Chandorkar et al. [46] reveal a gap between the theoretical advantages of Gherkin features and their implementation in practice. Although Gherkin features are advocated to enhance specification quality, their limited adoption indicates that these benefits may not be fully realised in real-world projects. Correspondingly, in Test Driven Development (TDD) domain, Borle et al. [58] investigated the impact of TDD on software development productivity and quality. They concluded that despite the theoretical advantages of TDD, its practical application in the projects studied did not demonstrate the expected benefits, raising questions about its effectiveness in real-world scenarios.

In addition, the multivocal literature reveals little about how BDD is practiced, or what evidence there is for supposed best practices. Furthermore, the efficacy of Business Domain language for Gherkin scenarios versus a User Interface Domain language has not been explored. Furthermore, the presumption that Business Domain language is better for both comprehension and documentability has not been actually tested in research. Additionally, Binamungu et al. [15] reported that BDD practitioners experienced confusion in deciding the appropriate level of abstraction to apply when writing a new scenario, which may lead to frustration and abandonment of BDD.

In this thesis, I argue that a Gherkin scenario written in a user interface domain specific language (imperative style) with corresponding step functions is easier to understand and document than one written in an abstracted Business Domain language with custom step functions.

In the next Chapter, the thesis presents the design of the User Interface Domain Specific Gherkin language.

Chapter 3

Language Design

As outlined in Chapter 2, scenario steps in BDD can be written in two styles: declarative (higher level) or imperative (lower level). In most cases, higher level steps contain more semantics, therefore scenarios can be expressed with fewer steps and relate to existing domain concepts. However, developers need to write more extensive glue code to execute the steps. As a result, interpreting the scenario meaning without referencing the glue code can be challenging. Conversely, lower level steps provide a more in-depth explanation of the system being developed. In this way, scenarios with low level steps tend to take more time, requiring more steps to convey the same meaning as scenarios with higher level steps. However, they require less glue code to implement [15].

Binamungu et al. found out that selecting the appropriate level of abstraction when developing BDD scenarios can be challenging. In addition, it will be easier to comprehend, extend, and maintain BDD feature suites when scenarios are written at a consistent level of abstraction. These observations were based on their review of both the scientific and professional literature concerning what constitutes good quality in BDD suites [15]. There is a lack of knowledge within the software industry as a whole regarding the appropriate level of abstraction for documenting user scenarios. Nevertheless, the professional literature of BDD best practices suggests that BDD scenario steps should be written in business domain language (high level steps), in order to improve scenario comprehension and documentation.

The aforementioned challenges motivated the researcher to develop a framework that comprises standardised low level Gherkin scenario steps and their corresponding glue code. The objective was to investigate whether or not scenario steps written in domain specific language are easier to comprehend and document than high level scenario steps written in business domain language. The chapter is structured as follows. Section 3.1 describes the first step of designing the User Interface Domain Specific Gherkin Language, which is surveying Gherkin projects on GitHub. Section 3.2 presents my analysis of the identified Gherkin project repositories on GitHub. Section 3.3 demonstrates the step definitions of the UI Domain Specific

Gherkin language. Section 3.4 summarises the chapter.

3.1 Survey of Gherkin Projects on GitHub

This section describes the design of the User Interface Domain Specific Gherkin Language. This was done by identifying common user interface actions in web applications through a survey of Gherkin Selenium projects.

I used the following search term to search for the relevant projects "gherkin selenium web-driver projects site: github.com". I performed the search in September 2020 and used the regular Google search engine. The search resulted in about 34,400 results and since this number of sources is huge, screening all the results would be time-consuming. Therefore, I selected and stored locally the first 100 results as my initial pool of sources.

Sources for the language design were selected in two phases as follows:

1. The researcher reviewed the initial pool of sources to identify the relevant sources in terms of whether or not the result was a repository listed on GitHub [86]. This filtering process resulted in 95 GitHub repositories.

2. Each search result was reviewed for relevance in terms of whether or not the repository contained Gherkin feature files and the glue code was implemented using Selenium WebDriver [87]. In addition, each project had to have at least one feature file to be considered. Furthermore, to identify that Selenium was being used to drive the glue code, I searched for Selenium import statements in the classes that contained the glue code. Further, I did not remove example projects and tutorial projects. These were identified by reading the project description and the README files. Similarly, the real projects were also identified by reading the project description and the README files. A real project is a software solution that addresses specific needs or problems. This process resulted in 26 relevant project repositories, see Figure 3.1.

Table 3.1 shows the list of repositories I surveyed. The table reports the repository URL, and the analysis results. I inspected each project and identified the number of features, the number of scenarios in these features, and the number of raw steps in these scenarios. I found that they contained 74 features, 150 scenarios, and 640 raw steps.

The survey filtering process for Gherkin Selenium sources for the language design was reviewed by a colleague in the Computing Science School at The University of Glasgow. The colleague replicated my Gherkin Selenium sources filtering process and eventually reached the same results as I did.

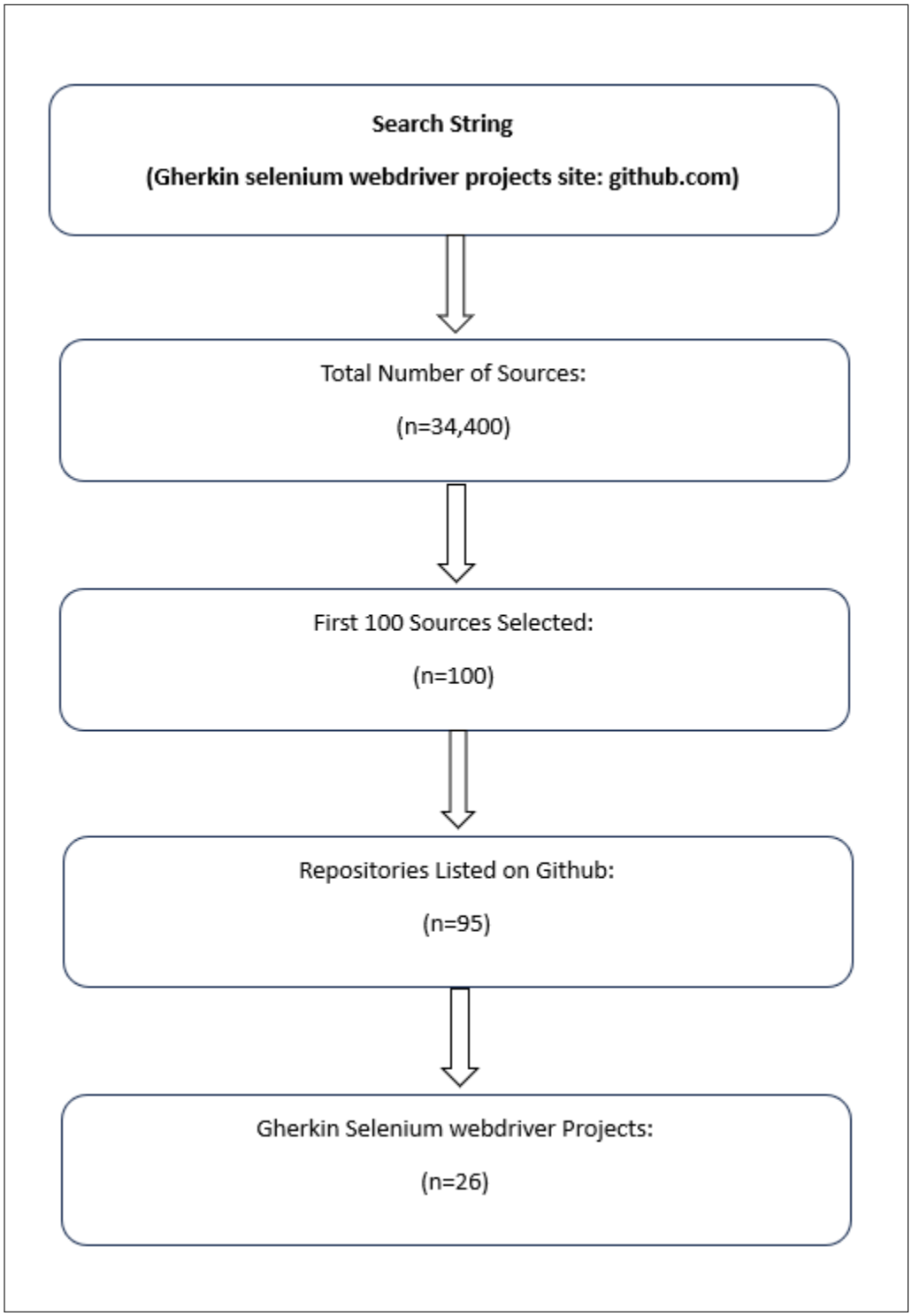

Figure 3.1: Flowchart that demonstrates the stages of the filtering process of Gherkin Selenium WebDriver projects.

Serial	Repository	Features	Scenarios	Steps
1	https://github.com/KeyurBhavsar/BDD-POM-Hybrid	4	5	23
2	https://github.com/sucinm/B2B-Web-Automation-Testing	2	2	8
3	https://github.com/amiya-pattnaik/sima	3	7	26
4	https://github.com/nirmalaC/HotelBookingSystem	2	4	13
5	https://github.com/CapgeminiNorway/learnmake-cucumber-selenium	1	3	9
6	https://github.com/narottamgla/selenium-bdd-cucumber	3	5	21
7	https://github.com/selenium-cucumber/selenium-cucumber-java-maven-example	2	4	12
8	https://github.com/workpeter/Selenium-Cucumber-framework	4	16	64
9	https://github.com/zenoyu/java-maven-cucumber-selenium	1	1	5
10	https://github.com/rahulrathore44/SeleniumCucumber	4	10	64
11	https://github.com/markwinspear/Selenium-cucumber-java-main-test-project	1	1	3
12	https://github.com/crossbrowsertesting/monta	6	13	42
13	https://github.com/jukafah/java-cucumber-selenium	1	4	16
14	https://github.com/waheedahmed55/Hybrid-Selenium-Cucumber-BDD	4	7	32
15	https://github.com/bo32/spring-cucumber-selenium-sample	2	3	12
16	https://github.com/testdouble/java-cucumber-example	2	3	6
17	https://github.com/AshisRaj/bdd-with-cucumber-selenium	8	21	97
18	https://github.com/reportportal/mizran	12	26	107
19	https://github.com/LambdaTest/monta-TestNG-Selenium	1	1	7
20	https://github.com/rohinegi548/Cucumber-BDD-Automation-Framework	1	2	25
21	https://github.com/shankybnl/selenium_BDD_framework	2	2	5
22	https://github.com/sevilayagil/Sample-Selenium-Project-With-Cucumber	1	1	11
23	https://github.com/estefafdez/selenium-cucumber	2	2	7
24	https://github.com/ahmedelshal2/Automation-Testing-of-ECommerce-WebsiteUsingSelenium	2	2	9
25	https://github.com/luciemars/cucumber-java-selenium-webdriver-example	2	3	8
26	https://github.com/executeautomation/SeleniumWithCucucumber	1	2	8

Table 3.1: Analysis of Gherkin Selenium project repositories on GitHub.

```
When I enter "tomsmith" into input field having id "username"
When I enter username as "testusername" into the text box
```

Figure 3.2: Steps with synonymous objects that can be represented in one unique step.

```
When I enter "tomsmith" into input field having id "username"
When I fill in "q" by name "<Keyword>"
When I type the todo "Do Things!"
```

Figure 3.3: Steps with synonymous actions that can be represented in one unique step.

3.2 Analysis

The aim of analysing the github repositories was to identify the unique steps in the features, and the common unique steps. To decide that two different steps were the same, a judgment was necessary so some heuristics were agreed. First, if the two steps referred to synonymous objects in the user interface, e.g. text field, input field, and text box. For example, Figure 3.2 demonstrates two action steps of entering text into synonymous objects in the user interface, i.e. input field and text box. Second, if the two steps referred to synonymous actions on the objects in the user interface, e.g. “enter text”, “write”. For example, Figure 3.3 demonstrates three action steps that synonymously represent entering text into objects in the user interface. The synonymous actions shown in the Figure: enter, fill, and type, represent one unique action of entering text into a text box object. Third, if the two steps used similar Selenium API calls, e.g. looking up a text box element. For example, radio, check box, and regular buttons have the same Selenium API call, i.e. button.click(). The aim here was to account for misnamed steps. Each of these three factors contributed to a judgment as to whether two sentences were the same. Eventually, and according to my criteria, if a unique step appears 5 times or more in the projects features, then it is considered a common unique step and it goes to my language steps.

The uniqueness of the identified steps were reviewed by a colleague in the Computing Science School at The University of Glasgow. The colleague followed my heuristics to judge whether two sentences were the same or not.

During the process of analysing the projects features, I classified each step according to the user interface action taken. First, each step that performed an action on the user interface

Serial	Unique When Step	Count	Example Step
1	Enter text in text box	194	When I enter “tomsmith” into input field having id “username”
2	Click button	134	When the user clicks on login button
3	Click link	43	When the user click on join link
4	Select value from drop down list	26	When I select option with value “2”
5	Time delay	12	When I wait for 2 seconds
6	Hover on element	2	When I hover on “Your Profile” heading on the profile page
7	Save file	1	When I save the csv file
8	Take screenshot	2	When I take screenshot

Table 3.2: The identified unique (When) steps.

were considered. These steps were treated as Gherkin “When” steps. I counted the number of occurrences for each unique step in the scenarios of the repositories. In total 8 unique types of user interface “When” actions were identified, as shown in Table 3.2. Since a unique step has to appear 5 times or more in the projects features to be considered a common unique step according to my criteria, only the first five unique steps were considered.

The first step in the unique “When” actions represents interacting with text boxes and entering text into them. This includes the basic text box, and password field in which the text is obscured in order that it cannot be read. This is usually achieved by replacing each character with a symbol such as the asterisk (”*”).

The second step represents clicking a button in the user interface. This includes the regular button, submission button, radio button, and check box button. Each of these buttons has a different behaviour within the HTML standard. For example, regular buttons are typically stateless (although they can be disabled as a result of an event) whereas check boxes are normally toggled and radio buttons are implemented as groups.

The third step is to navigate to a new page through clicking on a link. Alternatively, a user can navigate to a specific page that the user supplies. This would happen in a real browser by entering an address in the browser address bar and then pressing the return key on the keyboard. This is included as a separate precondition type in Table 3.3.

The fourth step is about interacting with drop-down lists and selecting a value from the lists. In a drop-down list, options are displayed vertically when the user hovers their cursor over it or clicks on it. In addition, this list disappears when the user clicks again or takes the cursor away from it after the user has stopped interacting with it.

The fifth step type identified is about delaying time to allow actions to happen. For example, waiting for an application to respond to a button click. Furthermore, waiting for the browser to load certain elements that naturally take more time to load. This is necessary because

Serial	Unique Given Step	Count	Example Step
1	Navigate to page	72	Given I navigate to “http://www.codigofacilito.com”
2	Use particular browser	5	Given The chrome web browser is on the search page
3	Upload file	1	Given I have an account file

Table 3.3: The identified unique (Given) steps.

various web elements on a web page may load at various intervals when the user interface opens in a browser.

The sixth step represents the action of hovering on web elements with the mouse pointer or cursor. This action is commonly used to reveal additional information in an application, such as image alt texts or tooltip guides providing additional information on data to be entered in a field.

Additional steps were identified that were associated with supporting auditing of applications. These include the seventh step for saving files on the user’s machine. Downloads of files is a common action by real users in web applications, for retrieving content that can’t be displayed natively by a browser. The eighth step is for taking a screenshot of a web page.

Second, “Given” precondition steps were also identified during the process of analysing the features of the repositories. Furthermore, 3 unique types of “Given” preconditions were identified, as shown in Table 3.3. Since a unique step has to appear 5 times or more in the projects features to be considered a common unique step according to my criteria, only the first two unique steps were considered.

The first step is to navigate to a specific web page that the user supplies in the step. This would happen in a real browser by entering an address in the browser address bar and then pressing the return key on the keyboard.

The second unique “Given” step represents setting up a particular web browser for interacting with a web page. The user needs to supply their favorite browser name that they wish to use into the step definition.

The third and last “Given” step is for selecting and uploading a file that resides on the user’s machine in order to be processed by the web application. This would happen by giving the browser access to the user’s file system and allowing the browser to upload the file.

Third, “Then” assertion steps were also identified during the process of analysing the features of the repositories. Seven unique types of “Then” assertions were identified, as shown in Table 3.4. Since a unique step has to appear 5 times or more in the projects features to be considered a common unique step according to my criteria, only the first three unique steps were considered.

Serial	Unique Then Step	Count	Example Step
1	Assert content appears in page	76	Then I should see “success” message as “Comment added to the Post successfully!”
2	Assert page redirection	49	Then I should land on the “Forgot Password” page
3	Assert page title	13	Then page title should be “Automation Exercise”
4	Assert button is disabled	4	Then I should see “Log In” button disabled
5	Close browser	3	Then I close browser
6	Assert text box should be populated	2	Then “email” field should be populated on the profile page
7	Assert link should be active	1	Then “My Profile” link should be active on the profile page

Table 3.4: The identified unique (Then) steps.

The first step represents asserting the appearance of content as an outcome of the actions described in the “When” unique steps. Such content includes but is not limited to, success text messages and search results.

The second step is for affirming page redirection as an outcome of a previous user action. For example, a user can be redirected to their personal profile page of a web application after successful login to the web system.

Additional steps were identified associated with supporting auditing of applications. These include the third step for asserting a web page title, where the user inputs the expected web page title as a result of a previous user action.

The fourth step is for asserting that a button is disabled. This would happen to prevent a user from submitting a form or benefiting from a service when they do not meet particular criteria.

The fifth step is for closing the web browser when the user wishes to terminate their interaction with the user interface.

The sixth step represents asserting a particular text box being populated by texts. This is to ensure that the text box is not empty and the user has populated the text box with the required data.

The seventh and last unique “Then” step represents asserting that a link should be active. A link is a connection from one web resource to another, and they become active when a user clicks on them.

Serial	When Steps (Actions)
1	When I enter the value “some value” into the text box nearest the label “some label”
2	When I click the button labelled “some label”
3	When I click the button nearest the label “some label”
4	When I click the check box button nearest the label “some label”
5	When I click the radio button nearest the label “some label”
6	When I click the link labelled “some label”
7	When I select the value “some value” from the drop down list nearest the label “some label”
8	When I wait for “time in seconds” seconds

Table 3.5: When steps (actions) for the UI Domain Specific Gherkin language.

3.3 Step Definitions

The guiding principles for my UI Domain Specific language are that steps should be explicit about what is being done. Therefore, types are mentioned in steps. For example, button type should be specified in the step such as check box, and radio button. Therefore, if no button type is mentioned in the step then it is assumed that it is a regular button. Furthermore, only what can be seen on the user interface can be described in the steps. For example, locating elements on the user interface should be based on their visual description and not based on their id, class name, CSS Selector, etc.

Based on the guiding principles discussed earlier, I transformed the common unique Gherkin steps resulted in the previous section into my UI Domain Specific Gherkin language. The first five common unique “When” steps shown in Table 3.2 were transformed into “When” steps as shown in Table 3.5.

The first step represents entering a text into a text box that is located nearest a specified label. Locating the text box in this step is based on their visual description on the user interface, which is in line with my guiding principles. For example, assume we have the following step: When I enter the value “Paul” into the text box nearest the label “First name:”. This step is used to locate the text box that is visually nearest to the label “First name:”. Figure 3.4 demonstrates the text box that is visually nearest to the label “First name:” in the interface.

The second step represents clicking buttons that visually have labels on them. For example, assume we have the following step: When I click the button labelled “Submit”. This step is used to locate the button that is visually labeled “Submit” in the user interface. Figure 3.5 demonstrates the button that is labeled “Submit” in the user interface.

The third, fourth, and fifth steps are for clicking regular, check box, and radio buttons that visually reside nearest a label in a user interface. For example, the following step: When I click the radio button nearest the label “international student”, is used to look up the radio

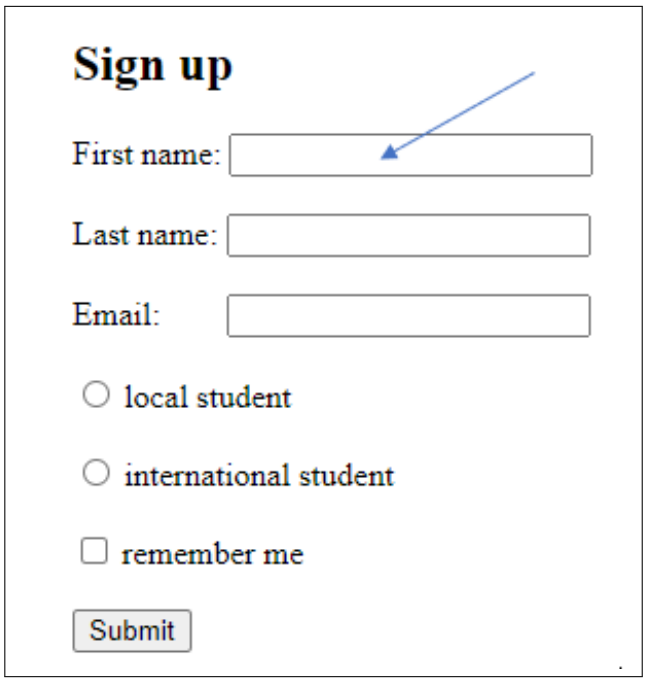

Figure 3.4: Locating a text box visually in a user interface.

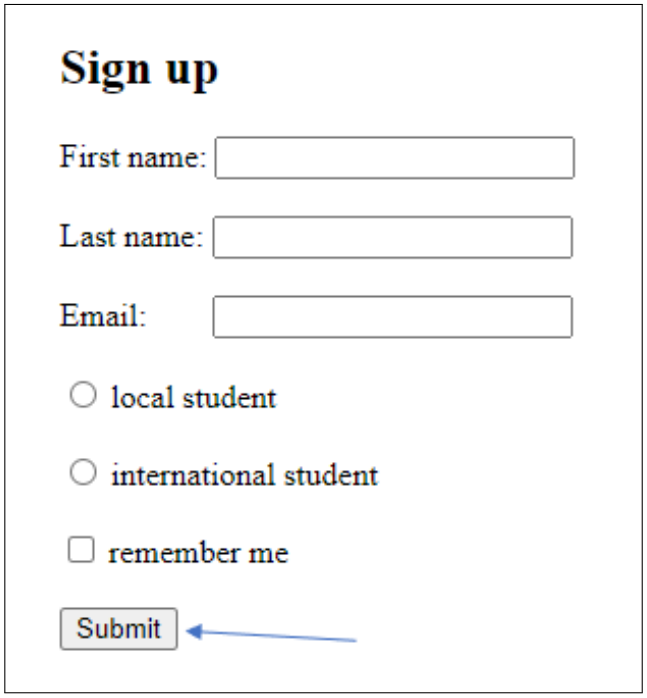

Figure 3.5: Locating a labeled button visually in a user interface.

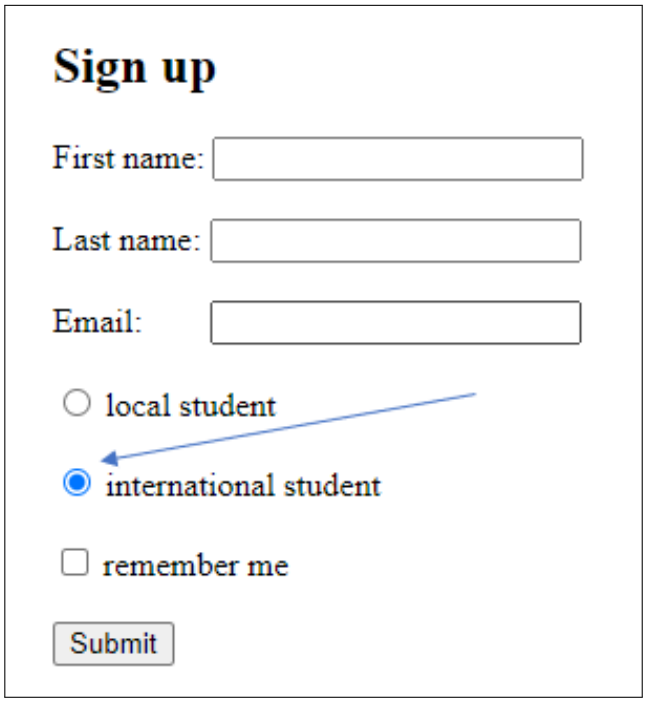

Figure 3.6: Locating a radio button that visually resides nearest a label in a user interface.

button that is visually located nearest to the label "international student" in the user interface. Figure 3.6 demonstrates the radio button that is visually located nearest to the label "international student" in the user interface.

The sixth step represents clicking on labeled links in the user interface to navigate to another page or document. For example, the following step: When I click the link labelled "Terms and conditions", is used to locate the link that is visually labeled "Terms and conditions" in the user interface, see Figure 3.7.

The seventh step is for locating a drop-down list that visually resides nearest to a specified label in the interface and selecting a value from the list. For example, assume we have the following step: When I select the value "Main Campus" from the drop down list nearest the label "Choose a campus:". This step is used to locate the drop-down list that is visually nearest the label "Choose a campus:" in the interface and select the value "Main Campus' from it, see Figure 3.8.

The eighth and last "When" step represents pausing step execution for a certain amount of time (in seconds) before moving onto the next step. For example, When I wait for "2" seconds. This is necessary when the loading of a web element with which Selenium WebDriver must interact is delayed.

Furthermore, I transformed the common unique "Given" Gherkin steps resulted in Section 3.2 into my UI Domain Specific Gherkin language. The common unique "Given" steps shown in Table 3.3 were transformed into "Given" steps as shown in Table 3.6.

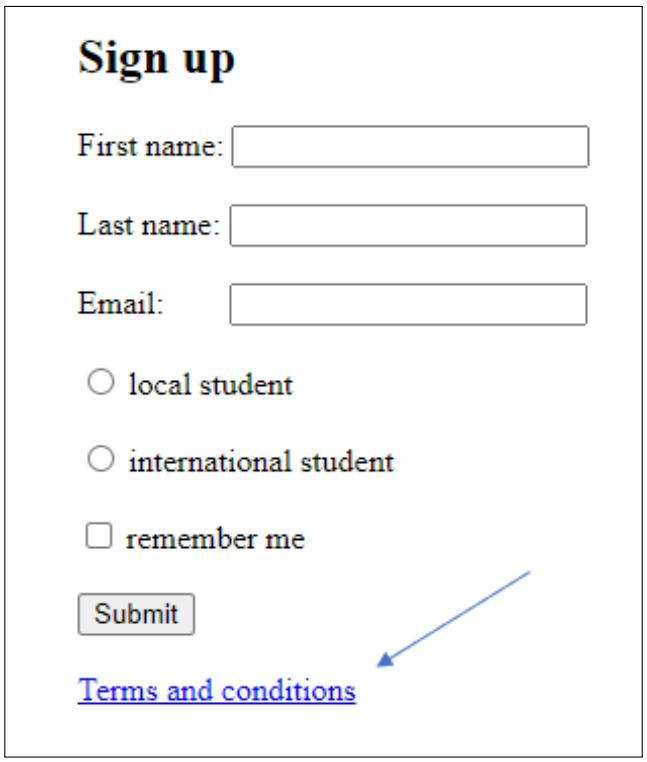

Figure 3.7: Locating a labeled link visually in a user interface.

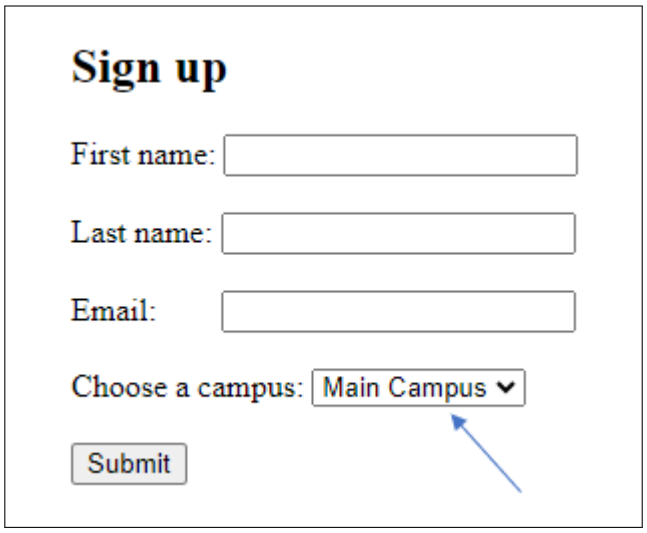

Figure 3.8: Locating a drop-down list visually in a user interface and selecting a value from it.

Serial	Given Steps (Preconditions)
1	Given I am using the (Chrome|Firefox|Edge|IExplorer|Opera) web browser
2	Given I have opened the web page “page url”

Table 3.6: Given steps (preconditions) for the UI Domain Specific Gherkin language.

Serial	Then Steps (Assertions)
1	Then I should be taken to the page “some page”
2	Then content appears on the page including the word “some phrase”
3	Then I should be on the page titled “some title”

Table 3.7: Then steps (assertions) for the UI Domain Specific Gherkin language.

The first step represents selecting a user's favorite web browser to prepare for interacting with a web user interface. The web browsers that are currently supported are as follows: Chrome, Firefox, Edge, IExplorer and Opera. These web browsers were chosen based on their popularity among web surfers. For example, if a user prefers to use Chrome web browser for interacting with a web page, then they should go for the following step: Given I am using the “Chrome” web browser.

The second and last “Given” step is for opening a web page that the user wishes to interact with. The user needs to supply the page URL in the step. For example, Given I have opened the web page “https://www.google.com/”.

In addition, I transformed the common unique “Then” Gherkin steps resulted in Section 3.2 into my UI Domain Specific Gherkin language. The common unique “Then” steps shown in Table 3.4 were transformed into “Then” steps as shown in Table 3.7.

The first step represents affirming that a user should be redirected to a new web page as a result to a previous user action step. For example, a user should be forwarded to their personal profile page once they have successfully logged into a web application.

The second step is for asserting that particular content should appear on the web page as a result of a previous user action step. For example, particular content should be displayed on the web page including the user's name and personal details when the previous action has been successful.

The third and last “Then” step represents affirming that a user should be redirected to a web page that has a particular title. For example, Then I should be on the page titled “My profile”.

3.4 Summary

This Chapter presented the design of the User Interface Domain Specific Gherkin language. As described in Section 3.1, the UI Domain Specific Gherkin language was developed by identifying common user interface actions in web applications through a survey of Gherkin projects. Searching for relevant projects on GitHub was performed using the regular Google search engine with the following search term ”gherkin selenium webdriver projects site: github.comb”. In order to determine if a search result was relevant, I examined whether it

was a repository on GitHub, and whether contained Gherkin feature files and Selenium glue code. As a result of this process, 26 Gherkin Selenium projects were identified as relevant. As part of my analysis, I identified the number of features, the number of scenarios within the features, the number of unique steps within the scenarios, and the number of common unique steps in the scenarios.

As described in Section 3.2, my analysis of the repositories began by categorising each step based on the actions that were taken on the user interface. Firstly, steps that performed an action on the user interface were considered, and they were treated as Gherkin "When" steps. Counting the number of occurrences of each unique step in the scenarios in the projects enabled us to identify 8 unique types of "When" actions. Additionally, three unique types of "Given" preconditions were identified during the process of analysing the features of the projects. Furthermore, seven unique types of "Then" assertions were identified as part of analysing the project features.

As described in Section 3.3, the identified common unique steps contributed to the design of my UI Domain Specific Gherkin language. The common unique steps were transformed into Given, When, and Then steps forming the steps of the language. In my UI Domain Specific Language, steps should explicitly explain what is being done. In order to do that, types are explicitly mentioned in steps. Furthermore, the steps can only describe what can be seen on the user interface.

In the next Chapter, the thesis presents the design and implementation of the standardised step function library.

Chapter 4

Library Design and Implementation

In Chapter 3, I described the design of the User Interface Domain Specific Gherkin Language, which comprises standardised low level Gherkin scenario steps. These scenarios describe how users will interact with the user interface of a software system, or how the software should behave. They can also be used as test cases to verify whether the software behaves as expected in addition to acting as software requirements. In order to determine whether or not the System Under Test (SUT) is working properly, the scenarios are connected to the application code using glue code, which is step functions that interpret each scenario step by step and execute the corresponding test code. These step functions can be written in the programming language of choice. Mapping the scenario steps to each step function is accomplished with the help of BDD tools such as Cucumber [41], Behave [88], Specflow [35], etc. In this chapter, I will discuss the design and implementation of the standardised step function library. The chapter is structured as follows. Section 4.1 presents an overview of the framework. In Section 4.2, the design and implementation of the step function library is presented. Section 4.3 shows an example of how the framework works and Section 4.4 demonstrates how integrate the framework into your project. Section 4.5 summarises the Chapter.

4.1 Framework Overview

The standardised low level user interface Gherkin steps that I discussed in the previous chapter need to be mapped to corresponding step functions (glue code) with the help of a BDD tool. To eliminate the need for writing glue code by developers, I developed a library of step functions that interact with the user interface of a web-based system. These functions execute the corresponding test code to determine whether or not the System Under Test (SUT) is working properly. Java programming language [89] was utilised to develop the step functions.

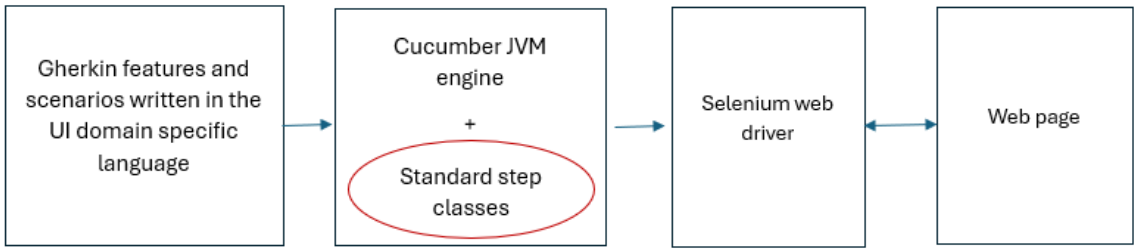

Figure 4.1: Framework design.

Figure 4.1 shows the library and framework design. In BDD, user stories and test scenarios describe system requirements and are developed using structured natural language, Gherkin. These scenarios reside in plain-language text files called feature files. Once the user runs a feature file, the Cucumber JVM [41] engine maps each scenario step to its corresponding standardised step method. These standardised methods, that I developed, reside in Java's step classes. Cucumber scans for matching step definitions using regular expressions when executing a Gherkin step. Regular expressions should start with the "^" character and end with the "$" character. After that, the step implementation methods interact with web elements in user interfaces using Selenium WebDriver [87], which is one of the most commonly used libraries for creating web application End-to-end (E2E) tests [90].

Cucumber JVM is a Java-based tool that provides support for Behaviour Driven Development by the execution of scenarios written in Gherkin language as demonstrated in Chapter 3. A variety of different programming languages is supported by the Cucumber project, including Java, which was originally written in Ruby [41]. Cucumber scans for matching step definitions using regular expressions when executing a Gherkin step. Regular expressions should start with the "^" character and end with the "$" character [41].

Selenium WebDriver is a web framework that allows you to perform cross-browser tests. Using this tool, you can automate web-based application testing to make sure it performs as expected [91]. To provide wider flexibility, Selenium WebDriver supports multiple programming languages such as Java, Python, Ruby, Perl, etc. You can create test scripts using Selenium WebDriver using a programming language of your choice. Four major components make up the Selenium WebDriver Architecture; First, Selenium client library. Second, JSON wire protocol over HTTP. Third, browser drivers. Fourth, browsers [91]. See Figure 4.2.

In Figure 4.2, test cases are written in a client library provided by Selenium. WebDriver code can be written in any major programming language using Selenium's bindings. Code specific to each browser is wrapped around these libraries. When the test cases have been constructed, the data must be communicated to the browser driver. With Selenium WebDriver, HTTP is used as the communication handler and JSON as the wire protocol. The browser driver then communicates with both Selenium and the actual browser to run the tests [91].

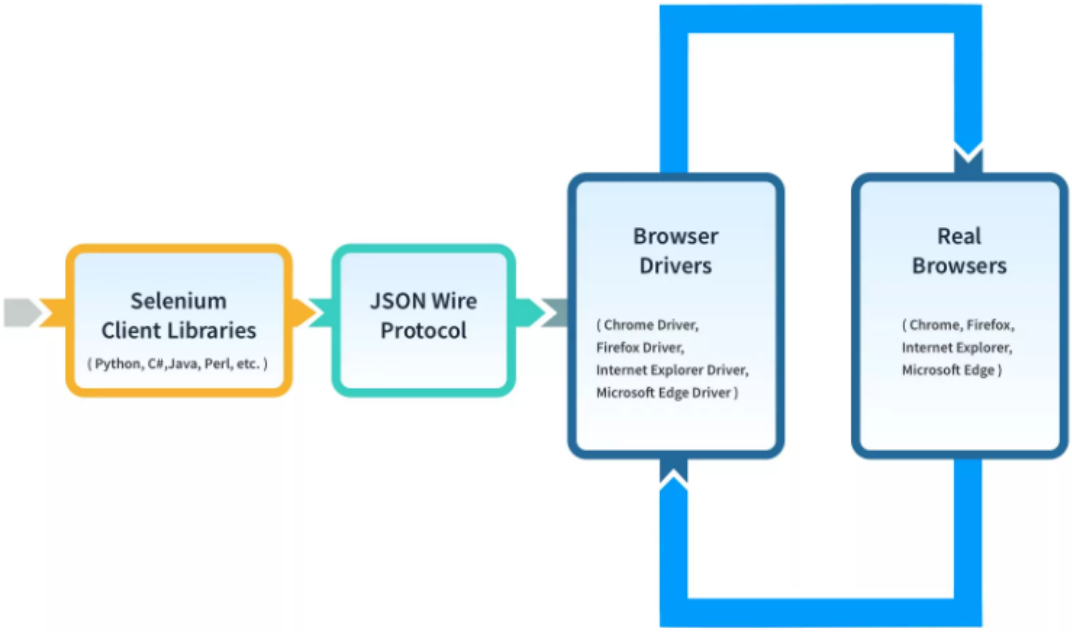

Figure 4.2: Selenium WebDriver framework architecture, taken from [91].

To interact with a web element using Selenium WebDriver, we must first locate it on the web page. Selenium provides support for 8 traditional location strategies in WebDrive, using which we can locate elements on the page. The location strategies are as follows: element's class name, CSS selector, ID, name, link text, partial link test, tag name, and xpath. See Table 4.1.

As demonstrated in Table 4.1, web elements on HTML pages can have attribute classes, and by using Selenium's class name locator, we can identify these elements. Furthermore, HTML pages are styled with CSS, and we can identify web elements on a page using the CSS

Locator	Description
class name	Locates elements whose class name contains the search value (compound class names are not permitted)
CSS selector	Locates elements matching a CSS selector
ID	Locates elements whose ID attribute matches the search value
name	Locates elements whose NAME attribute matches the search value
link text	Locates anchor elements whose visible text matches the search value
partial link text	Locates anchor elements whose visible text contains the search value. If multiple elements are matching, only the first one will be selected
tag name	Locates elements whose tag name matches the search value
xpath	Locates elements matching an XPath expression

Table 4.1: Location strategies of web elements in Selenium WebDriver, taken from [87].

```
WebDriver driver = new ChromeDriver();
driver.findElement(By.xpath("//input[@name='fname']"));
```

Figure 4.3: Locating an HTML element using the xpath locator strategy, taken from [87].

selector locator strategy. In addition, a web page element's ID attribute can be used to locate it; generally, each element's ID property should be unique. Similarly, an element's NAME can be used to locate it on a web page; it should be unique for each element. Additionally, we can identify a link on a web page using the link or partial link text locator if the element we want to locate is a link. Furthermore, to identify web elements on a page, we can use the HTML tag itself as a locator. In addition, to locate an element in a HTML document, we can use xpath, which is the path traversed to reach the element of interest. For example, the Java code snippet shown in Figure 4.3 demonstrates how we can locate the first name text box using the xpath locator strategy [87]. In the first line of the code snippet we create an instance of the Chrome driver, while in the second line we locate the first name text box.

In my framework, I utilised three of the locator strategies demonstrated earlier in Table 4.1, these are: xpath, tag name, and link text. While keeping in mind the guiding principles for my UI Domain Specific language, that steps should be explicit about what is being done, and that only what can be seen on the user interface can be described in the steps. Given that, to locate a web element such as text box, which typically resides near a label in a user interface, I developed an algorithm that first finds the reference label, and then finds all the text boxes that reside near that reference label. After that, the algorithm calculates and finds the closest text box to the reference label, which represents the target text box.

For example, to locate the text box that resides near GUID (Glasgow University ID) in Figure 4.4, the algorithm first finds the location of the GUID label with the help of xpath locator strategy provided by Selenium API. Then After that, the algorithm finds all the text boxes that reside within 160-pixel proximity zone from the reference label (GUID). In the Figure, there are two text boxes that reside within 160-pixel proximity zone from the reference label, the GUID text box and the password text box, using the formula below. The algorithm then calculates the distance from each text box to GUID and finds the closest one, which is in this case the text box below GUID.

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

d: distance between the reference and the target web element
x1, y1: coordinates of the reference web element
x2, y2: coordinates of the target web element

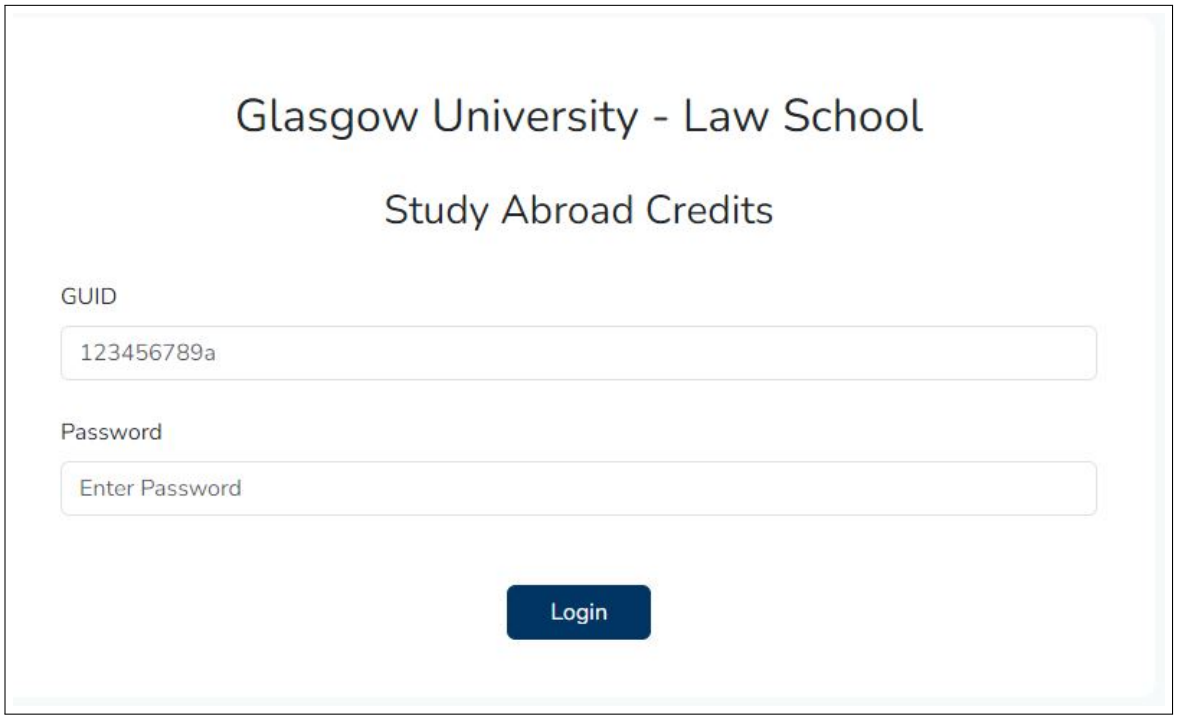

Figure 4.4: Login page for study abroad credits web application, taken from case study II in Chapter 5.

The location of a web element on a web page is determined by its relative position on the web page. The web element's position is calculated relative to the top-left corner of the web page, where the (x, y) coordinates are assumed to be (0, 0) [92]. Using JavaScript, Selenium exploits the function *getBoundingClientRect()* to locate web elements on a web page based on their position. In my algorithm, obtaining the location of a web element on a web page is achieved with the help of Selenium API through the method *getLocation()*. The web element's relative position on a web page represents the (x, y) coordinates of the top-left hand corner of the rendered element, see Figure 4.5.

4.2 Step Library Design and Implementation

In this section, I will discuss the design and implementation of the step methods that are needed to execute each of the UI Domain Specific Gherkin steps that I identified in Chapter 3. The step methods are as follows:

1. Time Delay: Time delay represents pausing step execution in a scenario for a certain amount of time (in seconds) before moving onto the next step in the scenario. For example, When I wait for "2" seconds. This is necessary when the loading of a web element with which Selenium WebDriver must interact is delayed. Figure 4.6 demonstrates how time delay step function is implemented. The user needs to enter delay time in seconds into the step.

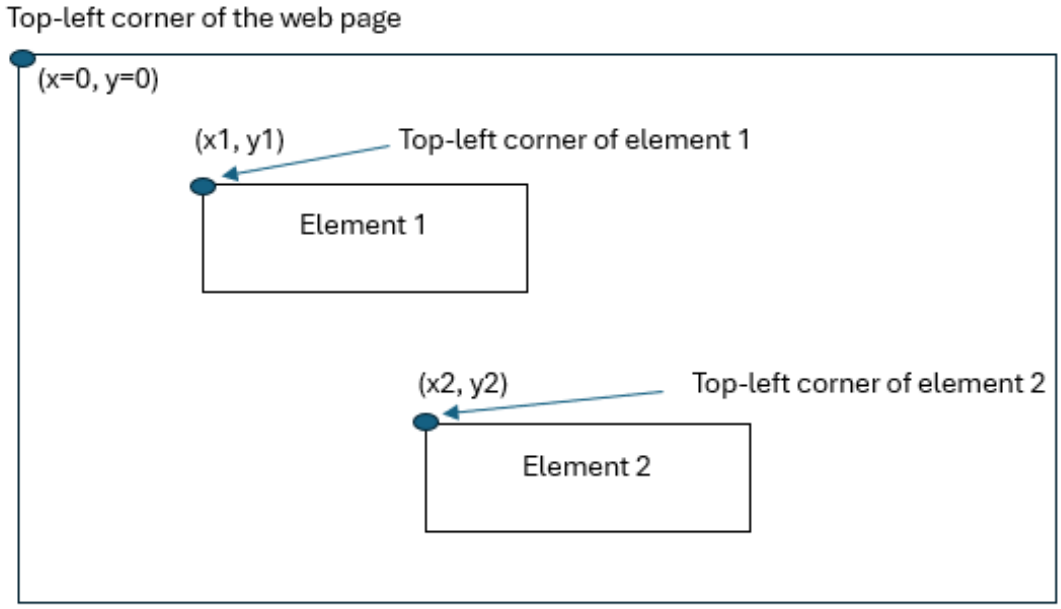

Figure 4.5: Relative position of web elements on a web page.

```
@When(``^I wait for \``(.*)\'' seconds$'')
public void i_wait_for_seconds(String seconds) throws
       InterruptedException {
String millis = seconds + "000";
Thread.sleep(Long.parseLong(millis));
}
```

Figure 4.6: Time delay step function implementation.

Cucumber will then scan and match the step against the step definition's regular expression. Next, Cucumber gathers the delay time variable and passes it to the step definition's method and executes it. In the step definition method, the time variable in seconds will be converted to milliseconds. With the help of *Thread* class in Java, the milliseconds variable is passed to the *sleep* method as an argument. This will invoke the method *sleep* which causes the thread to temporarily cease execution for the stated number of milliseconds.

2. Click Link: Clicking a link in a user interface to navigate to another page or document in a web application was a common action step I identified in Chapter 3. For example, the following step: When I click the link labelled "Read our policy", is used to locate and click the link that is visually labeled "Read our policy" in the user interface, see Figure 4.7. Figure 4.8 demonstrates how the step function of clicking a link is implemented. The user needs to enter the link label that is visually described in the user interface into the step. Cucumber will then scan and match the step against the step definition's regular expression. After

Figure 4.7: Locating and clicking a labeled link visually in a user interface.

```
@When(``I click the link labelled \``(.*)\''$'')
public void i_click_the_link_labeled(String label){
WebElement element = driver.findElement(By.linkText(label));
element.click();
}
```

Figure 4.8: Step function implementation for clicking a link.

that, Cucumber gathers the link label variable and passes it to the step definition's method and executes it. Subsequently, in the step definition method, Selenium API is exploited by calling *driver.findElement(By.linkText(label));* to look up the specified link element and then click it through *element.click();*.

3. Click Button: Clicking a button in a user interface was also a common action step I identified in the previous chapter. For example, the following step: When I click the button labelled "Sign up", is used to locate and click the button that is visually labeled "Sign up" in the user interface, see Figure 4.9. Figure 4.10 demonstrates how the step function of clicking a button is implemented. The user needs to enter the button label that is visually described in the user interface into the step. Cucumber will then scan and match the step against the step definition's regular expression. Subsequently, Cucumber gathers the button label variable and passes it to the step definition's method and executes it. Consequently, in the step definition method, Selenium API is exploited by first calling *webElement = driver.findElement(By.xpath("//button[contains(.," + label + ")]"));* to try look up the specified button element. If this fails, the same Selenium method will be called to try look up the specified button as a submission button *driver.findElement(By.xpath ("//input[@type='submit' and @value=" + label + "]"));*. If the button has been located, then it

Figure 4.9: Locating and clicking a labeled button visually in a user interface.

will be clicked through *webElement.click();*.

An alternative step, and step function for clicking a button is shown in Figure 4.11. This can come in useful when there exist two buttons with the same label in a user interface. In this figure, *getButtonNearestLabel(label)* is a utility method I developed to locate the button element nearest the specified label with the help of Selenium WebDriver API. When the button has been located, it will be clicked with the help of Selenium API through *getButtonNearestLabel(label).click();*.

4. Click Check box Button: Clicking a check box button in a user interface was also a common action step I identified in the previous chapter. For example, the following step: When I click the check box button nearest the label "Remember me", is used to look up and click the check box button that is visually labeled "Remember me" in the user interface, see Figure 4.12. Figure 4.13 demonstrates how the step function of clicking a check box button nearest a label is implemented. In this figure, *getInputWebElementNearest(label, "checkbox")* is a utility method I developed to locate the check box button element nearest the specified label with the help of Selenium WebDriver API. When the check box button has been located, it will be clicked with the help of Selenium API through *getInputWebElementNearest(label, "checkbox").click();*.

5. Click Radio Button: Clicking a radio button in a user interface was also a common action step I identified in the previous chapter. For example, the following step: When I click the radio button nearest the label "international student", is used to look up and click the radio button that is visually labeled "international student" in the user interface, see Figure 4.14. Figure 4.15 demonstrates how the step function of clicking a radio button nearest a label is implemented. In this figure, *getInputWebElementNearest(label, "radio")* is a utility method I developed to locate the check box button element nearest the specified label with the help of Selenium WebDriver API. When the radio button has been located, it will be clicked with

```
@When(``^I click the button labelled \``(.*)\''$'')
public void i_click_the_button_labelled(String label){
WebElement webElement = null;
try{
webElement = driver.findElement(By.xpath(``//button
[contains(.,`` + label + '')]''));
} catch(NoSuchElementException noSuchElementException){

}

if (webElement == null){
try{
  webElement = driver.findElement(By.xpath
  (``//input[@type='submit' and @value='' +
  label + '']''));
    }catch(NoSuchElementException noSuchElementException){

    }
}

if (webElement != null)
     webElement.click();
else throw new NoSuchElementException(``Element cannot be
                                        found!'');
}
```

Figure 4.10: Step function implementation for clicking a button.

```
@When(``^I click the button nearest the label \``(.*)\''$'')
public void i_click_the_button_nearest_the_label(String label) {
getButtonNearestLabel(label).click();
}
```

Figure 4.11: Alternative step function implementation for clicking a button.

Figure 4.12: Locating and clicking a check box button that is visually located nearest a label in a user interface.

```
@When(``^I click the check box button nearest the label
     \``(.*)\''$'')
public void i_click_the_check_box_button_nearest_the_label(String
                                                          label){
getInputWebElementNearest(label, ``checkbox'').click();
}
```

Figure 4.13: Step function implementation for clicking a check box button.

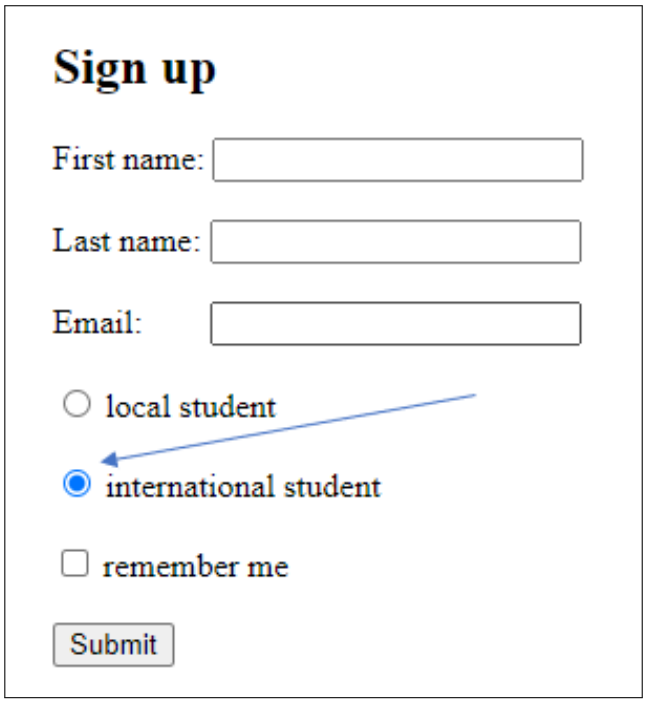

Figure 4.14: Locating and clicking a radio button that is visually located nearest a label in a user interface.

the help of Selenium API through *getInputWebElementNearest(label, "radio").click();*.

6. Select Value from Drop-down List: Selecting a value from a drop-down list was also a common action step I identified in the previous chapter. For example, the following step: When I select the value "Main Campus" from the drop down list nearest the label "Choose a campus:", is used to look up a drop-down list that visually resides nearest the label "Choose a campus:" in the user interface. When the drop-down list has been identified, the value "Main Campus" is selected, see Figure 4.16.

Figure 4.17 demonstrates how the step function of selecting a value from a drop-down list nearest a label is implemented. In this figure, *getSelectNearest(label)* is a utility method I developed to locate the drop-down element that resides nearest the specified label with the help of Selenium WebDriver API. When the drop-down list has been located, the value passed by the user will be selected with the help of Selenium API through *selectByVisibleText(value);*.

7. Enter Text into Text Box: Entering text into a text box was also a common action step I

```
@When(``^I click the radio button nearest the label \``(.*)\''$'')
public void i_click_the_radio_button_nearest_the_label(String
                                                        label){
   getInputWebElementNearest(label, ``radio'').click();
}
```

Figure 4.15: Step function implementation for clicking a radio button.

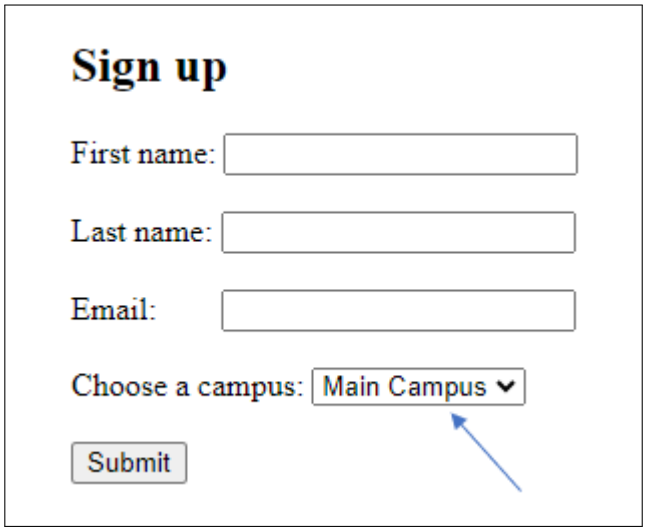

Figure 4.16: Locating a drop-down list visually in a user interface and selecting a value from it.

```
@When(``^I select the value \``(.*)\'' from the drop down list
      nearest the label \``(.*)\''$'')
public void i_select_the_value_from_the_drop_down_list_nearest_the
                                _label(String value, String label) {
getSelectNearest(label).selectByVisibleText(value);
}
```

Figure 4.17: Step function implementation for selecting a value from a drop-down list.

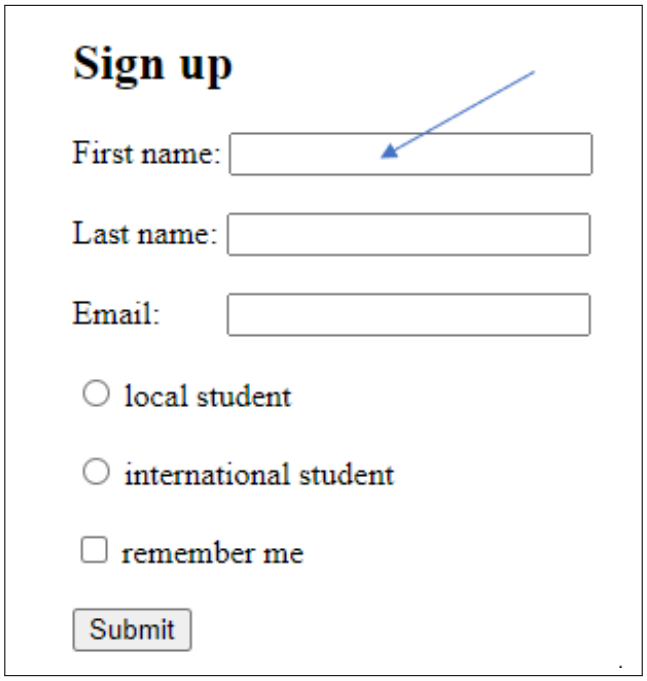

Figure 4.18: Locating a text box visually in a user interface.

```
@When(``^I enter the value \``(.*)\'' into the text box nearest the
      label \``(.*)\''$'')
public void i_enter_the_value_into_the_text_box_nearest_the_label
                                    (String value, String label) {
getInputWebElementNearest(label, ``text'').sendKeys(value);
}
```

Figure 4.19: Step function implementation for entering text into a text box.

identified in the previous chapter. For example, the following step: When I enter the value "Paul" into the text box nearest the label "First name:", is used to look up the text box that visually resides nearest the label "First name:" in the user interface. When the text box has been identified, the value "Paul" will be entered into it, see Figure 4.18.

Figure 4.19 demonstrates how the step function of entering text into a text box nearest a label is implemented. In this figure, *getInputWebElementNearest(label, "text")* is a utility method I developed to locate the text box element that resides nearest the specified label with the help of Selenium WebDriver API. When the text box has been located, the value passed by the user , "Paul", will be entered into the text box in the user interface with the help of Selenium API through *sendKeys(value);*.

8. Utilise Specific Web Browser: Exploiting a specific web browser to open a web page was a common precondition step I identified in the previous chapter. For example, the following step: Given I am using the "Chrome" web browser', is used to utilise the Chrome web browser to interact with a web interface. The web browsers that the framework currently

```
@Given(``^I am using the \``(.*)\'' web browser$'')
public void i_am_using_the_web_browser(String browser) {
if (browser.equalsIgnoreCase(``chrome'')) {
WebDriverManager manager = WebDriverManager
                    .getInstance(CHROME).setup();
ChromeOptions options = new ChromeOptions();
driver = new ChromeDriver(options);
} else if (browser.equalsIgnoreCase(``firefox'')) {
WebDriverManager.getInstance(FIREFOX).setup();
FirefoxOptions options = new FirefoxOptions();
driver = new FirefoxDriver(options);
} else if (browser.equalsIgnoreCase(``edge'')) {
WebDriverManager.getInstance(EDGE).setup();
EdgeOptions options = new EdgeOptions();
driver = new EdgeDriver(options);
} else if (browser.equalsIgnoreCase(``iexplorer'')) {
WebDriverManager.getInstance(IEXPLORER).setup();
driver = new InternetExplorerDriver();
} else if (browser.equalsIgnoreCase(``opera'')) {
WebDriverManager.getInstance(OPERA).setup();
driver = new OperaDriver();
} else {
Assert.fail(``Unknown browser selection.'');
}
}
```

Figure 4.20: Step function implementation for utilising a particular web browser to interact with a web interface.

supports are as follows: Chrome, Firefox, Edge, IExplorer and Opera.

Figure 4.20 demonstrates how the step function of utilising a particular web browser to interact with a web interface is implemented. *WebDriverManager* is a Java library that manages the drivers required by Selenium WebDriver in an automated manner [87]. Selenium WebDriver launches Google Chrome using *ChromeDriver*, which is a separate executable or a standalone server. With *ChromeOptions*,*FirefoxOptions*, etc, you can customise the ChromeDriver session and by default, Selenium opens Chrome, Firefox, or any other supported browser without any extensions or history [87].

9. Open web page: Opening a web page to interact with was also a common precondition step I identified in the previous chapter. For example, the following step: Given I have opened the web page "https://www.google.com/", is used to open Google search web page to interact with. Figure 4.21 demonstrates how the step function of opening a web page to interact with is implemented. The user needs to enter the URL of the web page that they wish to interact with. Cucumber will then scan and match the step against the step definition's regular expression. Subsequently, Cucumber gathers the url variable and passes

```
@Given(``^I have opened the web page \``(.*)\''$'')
public void i_have_opened_the_web_page(String WebPageUrl) {
try {
URL url = new URL(WebPageUrl);
driver.get(url.toString());
} catch (MalformedURLException malformedURLException) {
Assert.fail(``Invalid URL specification'');
}
}
```

Figure 4.21: Step function implementation for opening a web page to interact with.

```
@Then(``^I should be taken to the page \``(.*)\''$'')
public void i_should_be_taken_to_the_page(String url) {
Assert.assertEquals(url, driver.getCurrentUrl());
}
```

Figure 4.22: Step function implementation for affirming page redirection.

it to the step definition's method and executes it. Consequently, in the step definition method, a URL object is created from the url string variable that the user passed and then Selenium WebDriver API is exploited by calling *driver.get(url.toString());* to open the specified web page. The step will fail if the user enters an invalid URL specification.

10. Affirming Page Redirection: Asserting a page redirection as a result of a previous user interface action was a common outcome step I identified in the previous chapter. For example, the following step: Then I should be taken to the page "https://www.sac.guss.gla.ac.uk/profile", is used to affirm that the user is redirected his personal profile page as a result of a previous user interface action. Figure 4.22 demonstrates how the step function of affirming page redirection is implemented. The user needs to enter the URL of the web page that they should be redirected to as an outcome of a previous user interface action. Cucumber will then scan and match the step against the step definition's regular expression. Subsequently, Cucumber gathers the url variable and passes it to the step definition's method and executes it. Consequently, in the step definition method, JUnit, which is a unit testing framework for Java, is exploited through *Assert.assertEquals(url, driver.getCurrentUrl())*. This JUnit method tests if the two objects, the expected url and the actual resulted url, are equal. The step will pass if they are equal and will fail if they are not.

11. Affirming Content Appears on Page: Asserting the appearance of content as an outcome of a previous user interface action was also a common outcome step I identified in the previous chapter. For example, the following step: Then content appears on the page including the word "Form submitted successfully!!", is used to affirm that the specified phrase should be displayed to the user as an outcome of a previous action step. Figure 4.23 demon-

```
@Then(``^content appears on the page including the word \``(.*)\''
                                                              $'')
public void content_appears_on_the_page_including_the_word(String
                              word) throws InterruptedException {

    String actualWord = null;
    try{
        actualWord = driver.findElement(By.xpath(``//
        span[contains(.,`` + word + '')]'')).getText();
    } catch(NoSuchElementException noSuchElementException){
        noSuchElementException.printStackTrace();
    }

    if(actualWord != null) {
        if (actualWord.equalsIgnoreCase(word))
            Assert.assertTrue(true);
        else
            Assert.assertTrue(false);
    } else throw new NoSuchElementException(``No such element
                                                    exists!'');
}
```

Figure 4.23: Step function implementation for affirming content appearance on user interface.

strates how the step function of affirming content appearance is implemented. The user needs to enter the phrase or word that should be displayed to the user as an outcome of a previous user interface action. Cucumber will then scan and match the step against the step definition's regular expression. Subsequently, Cucumber gathers the word variable and passes it to the step definition's method and executes it. Consequently, in the step definition method, Selenium API is exploited to look up the specified word or phrase in the user interface through *driver.findElement(By.xpath("//span[contains(.," + word + ")]")).getText()*. Furthermore, JUnit is utilised to test whether or not the expected word and the actual resulted word are equal. The step will pass if they are equal and will fail if they are not.

12. Affirming Page Title: Asserting page title as an outcome of a previous user interface action was also a common outcome step I identified in the previous chapter. For example, the following step: Then I should be on the page titled "My profile", is used to affirm that the user should be on the specified page title as an outcome of a previous action step. Figure 4.24 demonstrates how the step function of affirming page title is implemented. The user needs to enter the page title that they should be on as an outcome of a previous user interface action. Cucumber will then scan and match the step against the step definition's regular expression. Subsequently, Cucumber gathers the page title variable and passes it to the step definition's method and executes it. Consequently, in the step definition method, Selenium API is ex-

```
@Then(``^I should be on the page titled \``(.*)\''$'')
public void i_should_be_on_the_page_titled(String title) {
    String actualTitle = driver.getTitle();
Assert.assertEquals(actualTitle, title);
}
```

Figure 4.24: Step function implementation for affirming page title on user interface.

```
Feature: Login

Scenario: Login scenario
 Given I am using the ``Chrome'' web browser
 And   I have opened the web page ``https://www.sac.guss.gla.ac.uk/
       login''
 When  I enter the value ``123456789'' into the text box
       nearest the label ``GUID''
 And   I enter the value ``gherkin'' into the password box nearest
       the label ``Password''
 And   I click the button labelled ``Login''
 Then  I should be taken to the page ``https://www.sac.guss.gla.ac.
       uk/account''
```

Figure 4.25: Login feature.

ploited to look up the specified page title in the user interface through *driver.getTitle()*. When the page title has been identified, JUnit is exploited to test whether or not the expected page title and the actual resulted page title are equal through *Assert.assertEquals(actualTitle, title)*. The step will pass if they are equal, otherwise, it will fail.

4.3 Framework Demonstration

As an example of the relationship shown in Figure 4.1, I will provide a user login scenario. The first two statements demonstrated in Figure 4.25 illustrate a user login scenario with five UI Domain Specific Gherkin steps. The first two steps describe the preconditions for this login scenario. In the first step, the user specifies that they wish to utilise Chrome web browser to open the web page. Second, the user enters the web page that they wish to open and interact with. The following three Gherkin steps represent the actions that the user performs in the login scenario. The first and second action steps represent the user's actions of entering their student id and password into the text boxes near the labels "GUID" and "Password" respectively. Then the user clicks on the button with the label "Login". The final step illustrates user actions' outcomes. Further, the user should be logged in and taken to their designated account page.

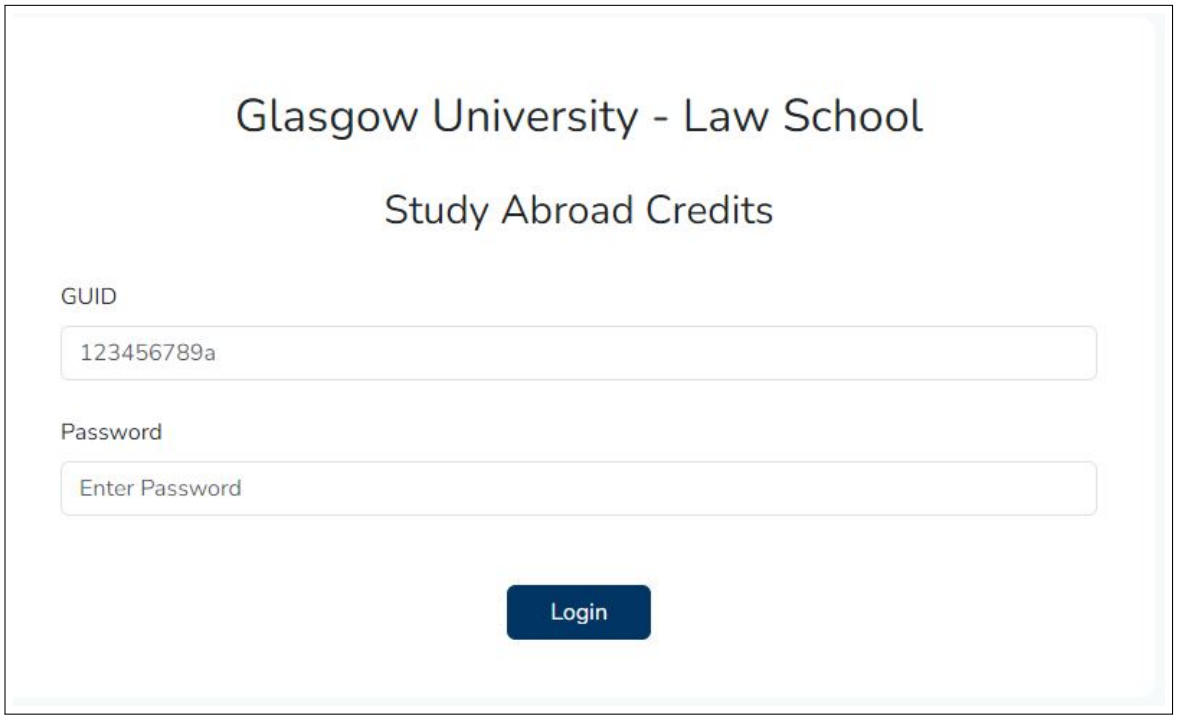

Figure 4.26: Login page for study abroad credits web application, taken from case study II in Chapter 5.

With the help of Cucumber JVM engine, each step in the scenario shown in Figure 4.25 is then mapped to its corresponding step function in the standardised step function suite using regular expressions. Figure 4.27 shows the step function that corresponds to step 3. In this figure, the label "GUID" is passed to the utility method *getInputWebElementNearest()* as a first argument. The second argument, which is "text", represents the type of input field we aim to locate. Selenium WebDriver API is then exploited to locate the text box object that resides nearest the label element "GUID". The utility methods that were developed to locate the text box and label objects are shown in Figure 4.28. In this Figure, *driver.findElement(By.xpath("//label[contains(.,'" + label + "')]"));* and *driver.findElements (withTagName("input").near(webElement))* represent Selenium API calls to locate both label and text box objects in the user interface. When the target text box element has been received through *getInputWebElementNearest()* as shown in Figure 4.27, Selenium API will be exploited again through *.sendKeys(value);* to send the student id that was entered by the user to the located text box.

4.4 Framework Integration

To integrate the framework into a project, there two options depending on the build automation tool of choice. The options are as follows:

```
@When(``^I enter the value \``(.*)\'' into the text box nearest the
     Label \``(.*)\''$'')
public void i_enter_the_value_into_the_text_box_nearest_the_label(
       String value, String label) {
getInputWebElementNearest(label, ``text'').sendKeys(value);
}
```

Figure 4.27: Step function that maps to the third scenario step.

```
public static WebElement getInputWebElementNearest(String label,
              String type) {

WebElement labelElement = driver.findElement(By.xpath
           ("//label[contains(.,'" + label + "')]"));
return getInputWebElementNearest(labelElement, type);
}

public static WebElement getInputWebElementNearest(WebElement
              webElement, String type) {
ListWebElement webElements =
driver.findElements(withTagName(``input'').near(webElement))
.stream()
.filter(e-e.getAttribute(``type'').equalsIgnoreCase(type))
.collect(Collectors.toList());

return findNearestWebElementTo(webElement, webElements);
}

public static WebElement findNearestWebElementTo(WebElement origin,
              ListWebElement candidates){
WebElement closest = candidates.get(0);
for (WebElement webElement: candidates)
    if (distanceBetween(origin, webElement)  distanceBetween(
       origin, closest))
       closest = webElement;

return closest;
}

private static Double distanceBetween(WebElement webElement1,
               WebElement webElement2){
Point point1 = webElement1.getLocation();
Point point2 = webElement2.getLocation();
return Math.sqrt(Math.pow(point1.getX() - point2.getX(), 2)
            + Math.pow(point1.getY() - point2.getY(), 2));
}
```

Figure 4.28: Utility methods to locate the text box element nearest the label “GUID” with the help of Selenium WebDriver API.

```
<!-- https://mvnrepository.com/artifact/io.github.wadsg/wads-gherkin -->
<dependency>
    <groupId>io.github.wadsg</groupId>
    <artifactId>wads-gherkin</artifactId>
    <version>1.3.1</version>
</dependency>
```

Figure 4.29: Web application Domain Specific Gherkin (wads-gherkin) Maven dependency.

```
// https://mvnrepository.com/artifact/io.github.wadsg/wads-gherkin
implementation group: 'io.github.wadsg', name: 'wads-gherkin', version: '1.3.1'
```

Figure 4.30: Web application Domain Specific Gherkin (wads-gherkin) Gradle dependency.

1. First, if you use Maven as your build automation tool, then you must include my web application domain specific gherkin (wads-gherkin) framework as a Maven dependency into your Maven project object model (pom) file. The group id, artifact id, and version of the dependency are demonstrated in Figure 4.29.

2. Second, if you use Gradle as your build automation tool, include my wads-gherkin framework in your Gradle build file as shown in Figure 4.30.

The web application UI Domain Specific Gherkin framework currently supports two web application flavors. First, the framework supports web applications developed with HTML5, which is the latest standard of Hypertext Markup Language. Second, the framework supports web applications developed with ReactJS, which is a JavaScript library for building user interfaces. The support for ReactJS is needed because the unique dynamic and component-based architecture of ReactJS requires some considerations when automating it with Selenium.

Figure 4.31 demonstrates how to configure the glue code path for a ReactJS web application in the TestRunner class. The common glue code for all web application frameworks reside in io/github/wadsg/selenium_bdd/steps/common. Whereas the specific glue code for ReactJS reside in io/github/wadsg/selenium_bdd/steps/react. To configure the glue code path for an HTML5 web application, only replace react with html5 as follows: io/github/wadsg/selenium_bdd/steps/html5.

```
import io.cucumber.junit.Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner.RunWith;

@RunWith(Cucumber.class)
@CucumberOptions(
        glue = {"io/github/wadsg/selenium_bdd/steps/common", "io/github/wadsg/selenium_bdd/steps/react"},
        plugin = {"json:target/cucumber.json","html:target/site/cucumber-pretty"}
)
public class TestRunner {
}
```

Figure 4.31: Configure glue code path for a ReactJS web application.

4.5 Summary

This Chapter presented the design and implementation of the standardised step function library. In order to maintain automated acceptance test suites associated with requirements expressed in business domain concepts, additional developer effort is required. The objective of the framework demonstrated in this Chapter was to mitigate the cost of maintaining acceptance tests through the provision of standardised step function suites. As described in Section 4.1, in order to eliminate the need for developers to write glue code (step functions), I developed a library of step functions that interact with web-based user interfaces. To test whether or not the System Under Test (SUT) is working properly, these functions execute the corresponding test code with the help of Selenium WebDriver.

As shown in figure 4.1, BDD uses the structured natural language, Gherkin, to describe system requirements. Cucumber's JVM engine maps each scenario step to a standardised method. These methods, which I developed, reside in Java step classes. Using Selenium WebDriver, the step methods interact with web elements in user interfaces.

As described in Section 4.2, I discussed the implementation of the framework's step methods needed to execute each Given, When, and Then step identified in Chapter 3. Cucumber JVM framework and Selenium WebDriver were exploited in the steps implementation. Cucumber was utilised to match each step with its designated step method, whereas Selenium was exploited to look up objects in the user interface and interact with them.

As described in Section 4.3 and Section 4.4, an example of how the framework works is presented. It showed a user login scenario and demonstrated how the framework maps each step to its corresponding step function with the help of Cucumber JVM. Furthermore, the example showed how Selenium WebDriver were exploited in executing cross-browser tests. Additionally, integrating the framework into a project was demonstrated.

In the next Chapter, the thesis presents the case studies that I undertook to improve and validate the design and implementation of the User Interface Domain Specific Gherkin language

and its corresponding step function suites.

Chapter 5

Case Studies

As described in Chapter 3, I designed the User Interface Domain Specific Gherkin Language, which consists of standardised low level Gherkin scenario steps. The scenario steps describe how a user will interact with the software system, or how the software should behave. In addition to acting as software requirements, they are also useful as test cases for verifying whether the software behaves as expected. Furthermore, I presented in Chapter 4 the design and implementation of the step function library that connects the scenario steps to the application code in order to determine whether or not the System Under Test (SUT) is working properly. The step functions interpret each scenario step by step and execute the corresponding test code. Three case studies have been undertaken to help me improve and validate the design and implementation of the UI Domain Specific Gherkin Language and the step function library. In this Chapter, the three case studies are presented. The chapter is structured as follows. Section 5.1 demonstrates research design considerations. In Section 5.2, the research method is presented. Section 5.3 shows the first case study, which is the Lighthouse Laboratory Sample Tracker. In Section 5.4, the second case study is presented, which is Study Abroad Credits. Section 5.5 presents the third case study, which is Popu e-commerce platform. Section 5.6 presents a summary of findings. Section 5.7 summarises the chapter.

5.1 Research Design Considerations

Recall (Section 2.3.2) that researchers [15] learnt that selecting the appropriate level of abstraction when expressing and negotiating requirements can be challenging. Furthermore, when expressing requirements in business domain concepts, developers need to write more glue code. Without reference to the glue code, it can be difficult to interpret the Gherkin scenario meaning. In addition, when scenarios are written at a consistent level of abstraction, BDD feature suites will be easier to comprehend, extend, and maintain. In contrast, if

a feature suite contains both high level and low level scenarios, a maintenance engineer may find it difficult to decide which level of abstraction to use when writing a new scenario if the feature suite consists of both levels. Moreover, the test harness code is also likely to contain redundant steps, glue code, and inconsistent abstraction levels, resulting in additional comprehension and maintenance issues.

In addition to the scientific BDD literature, according to the professional literature of BDD best practices presented in Chapter 2, it is assumed that business domain language should be used when writing BDD scenarios. Furthermore, it is also presumed in the professional literature that writing BDD scenarios in business domain concepts makes them less brittle and easier to maintain. While these practitioners specifically recommend writing BDD scenarios in the business domain language, there is currently no evidence to suggest that writing BDD scenarios in this way makes them less brittle and easier to maintain than writing them in in the application domain. Therefore, based on call from academia to write BDD scenarios in a consistent level of abstraction, coupled with the confusion in the professional literature on the abstraction level to write BDD scenarios, a user interface specific domain Gherkin language was developed coupled with the provision of standardised step function suites to mitigate the cost of maintaining acceptance tests. The objective of this chapter is to investigate whether the proposed low level framework for user interface specific use of behaviour driven development is feasible or not. One research question guided this experiment:

1. Is the proposed low level framework for user interface specific use of behaviour driven development feasible?

In this research, feasibility is defined by the framework's ability to describe automatically executable acceptance test scenarios, similar to conventional BDD. For the framework to be considered feasible, it should achieve equivalent functionality coverage, describe similar interactions, and support common user stories. Additionally, feasibility is demonstrated by minimising or eliminating the need for custom step definitions, ensuring that reusable, generic step definition functions can be applied across multiple projects. Finally, the framework must show that step definitions are truly generic by proving their applicability in different projects.

In order to answer this research question, case study methodology was considered suitable for this research. This is due to the fact that I needed to identify weaknesses in the design and implementation of user interface specific domain Gherkin language coupled with standardised step function suites. Other research methods, such as surveys and controlled experiments, were considered but the case study method was selected. The survey approach can provide a researcher with a broad but shallow overview of many instances of the case under investigation, but is unlikely to aid the researcher who wants a more detailed understanding of a particular issue [93, 94].

5.2 Research Method

This chapter employs a series of sequential, self-contained case studies to improve and validate the design and implementation of the UI Domain Specific Gherkin language and its associated acceptance test suites. Each case study is conducted independently, allowing iterative improvements based on quantitative performance metrics.

5.2.1 Research Design

Each case study is focused on a distinct real-world application. The sequential approach ensures that the findings from one case study directly inform improvements made before the next study is undertaken. The objectives for each case study are to evaluate the technical integration, the efficiency of scenario implementation, and the overall performance of the acceptance test suites based on measurable criteria.

5.2.2 Data Collection

For each case study, quantitative data is gathered. The key performance metrics include:

- The number of scenarios implemented.
- The number of steps contained within those scenarios.
- The maximum number of scenarios for a feature.
- The minimum number of scenarios for a feature.
- The number of custom steps required.
- The number of scenarios that could not be implemented.

These metrics provide a detailed, objective basis for evaluating the performance and efficiency of the UI Domain Specific framework.

5.2.3 Data Analysis

Quantitative data from each case study are analysed using statistical methods. Descriptive statistics summarise the performance metrics. This analysis determines whether the changes lead to enhanced efficiency, minimised the number of custom steps, and overall improved framework performance.

5.2.4 Evaluation Procedure

1. Baseline Assessment: Record existing metrics related to scenario implementation and test suite performance.
2. Implementation: Deploy the improved UI Domain Specific Gherkin language and associated test suites in the case study's context.
3. Data Collection: Gather quantitative data on the number of scenarios, steps, custom steps, and feature scenario range.
4. Data Analysis: Compare the new metrics against the baseline to assess improvements and identify areas for further improvement.
5. Iterative Improvement: Use the insights gained to adjust and improve the framework before initiating the next case study.

By focusing on these quantitative measures, this methodology provides a clear, objective framework for evaluating the efficiency and effectiveness of the UI Domain Specific Gherkin language and its test suites. Each case study offers measurable insights that guide continuous improvement, ensuring a robust and performance-driven final design.

5.2.5 Case Study Selection

I explored the available case studies with my collaborators. The criteria I used to select suitable projects was as follows: First, the application has to be web-based. Second, it has to be developed using ReactJS framework. Third, the application should not have more than 65 user stories and should not have less than 15 user stories.

During the process of exploring suitable case studies, four potential case studies were identified to partner with in order to improve and validate the design and implementation of the UI domain specific Gherkin language and its associated step function suites. These identified potential case studies were as follows: First, Safari Wallet, which is an online marketplace for personalised travel. Travelers in Africa and beyond can use Safari Wallet to search, find, design, compare, and purchase holiday experiences from local African tour operators, and independent guides, with the flexibility of their income flow, time preferences, and payment options. Second, the Lighthouse Laboratory Sample Tracker, which is a sample tracker for Covid-19 lab samples. Third, Study Abroad Credit System, which is a credit recording and grade mapping web application for the University of Glasgow's School of Law. Fourth, Popu, which is an e-commerce platform that allows small businesses to sell products online.

I started the process of filtering the preceding case studies in order to select the appropriate case studies according to the criteria that I discussed earlier. First, all the case study were web based, therefore, the first criterion applies to all the case studies, and therefore no case study was excluded. Second, all the case studies were developed using ReactJS, therefore, no case study was excluded as they all met the second criterion. Third, all the remaining case studies had between 15 to 65 user stories except for Safari Wallet which had 78 user stories and thus was excluded as it did not meet the third criterion. As a result, three case studies resulted from the filtering process according to my criteria, which were the Lighthouse Laboratory Sample Tracker, Study Abroad Credit System, and Popu.

I collaborated with Glasgow University Software Service (GUSS) to work on the first and second case studies, which were the Lighthouse Laboratory Sample Tracker, and Study Abroad Credit System. The aim was to improve and validate the design of the UI Domain Specific Gherkin language and the step function suites. GUSS is a project created and run by the School of Computing Science at the University of Glasgow. With their managed software development services, they employ professional software developers and designers for a variety of internal and external clients. Furthermore, I collaborated with The Unicorn Hub [95], which is a startup based in Nigeria that provides software and business services to customers, to work on the third case study i.e Popu.

5.2.6 Study Protocol

The first step was for me to participate in a recorded online walk-through of the application with the software development team. This was for me to understand the functionality of the system and to have a chance to ask questions and take notes. The second step was to participate in an recorded online user story workshop with the software team to gather user stories and to take notes and ask questions. The third step was for me to turn these user stories into Gherkin scenarios starting from the user stories with highest priority. Furthermore, I communicated with the team to validate the Gherkin scenarios and the specific parameters. The fifth step was for me to start executing the Gherkin scenarios against the web application to see whether they pass or not. If a Gherkin scenario passes, it goes to the main application suite. If it does not pass, I investigate the reasons behind the failing scenario, fix it, or report a bug to the development team. Finally, I had a debrief period where he did an evaluation of the number of scenarios implemented, the number of steps in the scenarios, the number of steps that required custom code, and the number of scenarios that I could not manage to implement.

5.3 Case Study I: The Lighthouse Laboratory Sample Tracker

In this section, I demonstrate the first case study.

5.3.1 Background

As part of the global response to the COVID-19 pandemic, the University of Glasgow has had to make significant adjustments. The University of Glasgow has designated a space at its campus at Queen Elizabeth University Hospital as a Lighthouse Laboratory in response to the outbreak of COVID-19 in March 2020. The Lighthouse Laboratory is part of the National Lighthouse Laboratory Network in the UK, which is devoted to mass-testing of samples for COVID-19. For the purpose of automating the tracking of Covid-19 lab samples, the Lighthouse Laboratory collaborated with Glasgow University Software Service (GUSS) to develop a web-based sample tracking system. GUSS was created by, and operates within the School of Computing Science at the University of Glasgow. It employs professional software developers, as well as designers to provide managed software development services to its internal and external customers.

Furthermore, the software development team used multiple technologies in developing the system. They adopted ReactJS [96] for developing the user interface components. ReactJS is an open-source JavaScript library used for building user interfaces. Furthermore, the team adopted Flask [97] in developing the back-end side of the project. Flask is a web application development framework written in Python. The team has also adopted the software framework Docker [98]. They used it to build, run, and manage containers on servers and in the cloud.

5.3.2 Software Team

The software team of the lab sample tracker consists of a group of five professional software developers. Two developers were responsible for developing and maintaining the front-end side of the system. One developer was also involved in the front-end development as well as the back-end. One developer was assigned to build and maintain the back-end side of the project as well as taking care of the IT operations. Lastly, one developer was responsible for working on the front, back-end side of the system and also IT related operations.

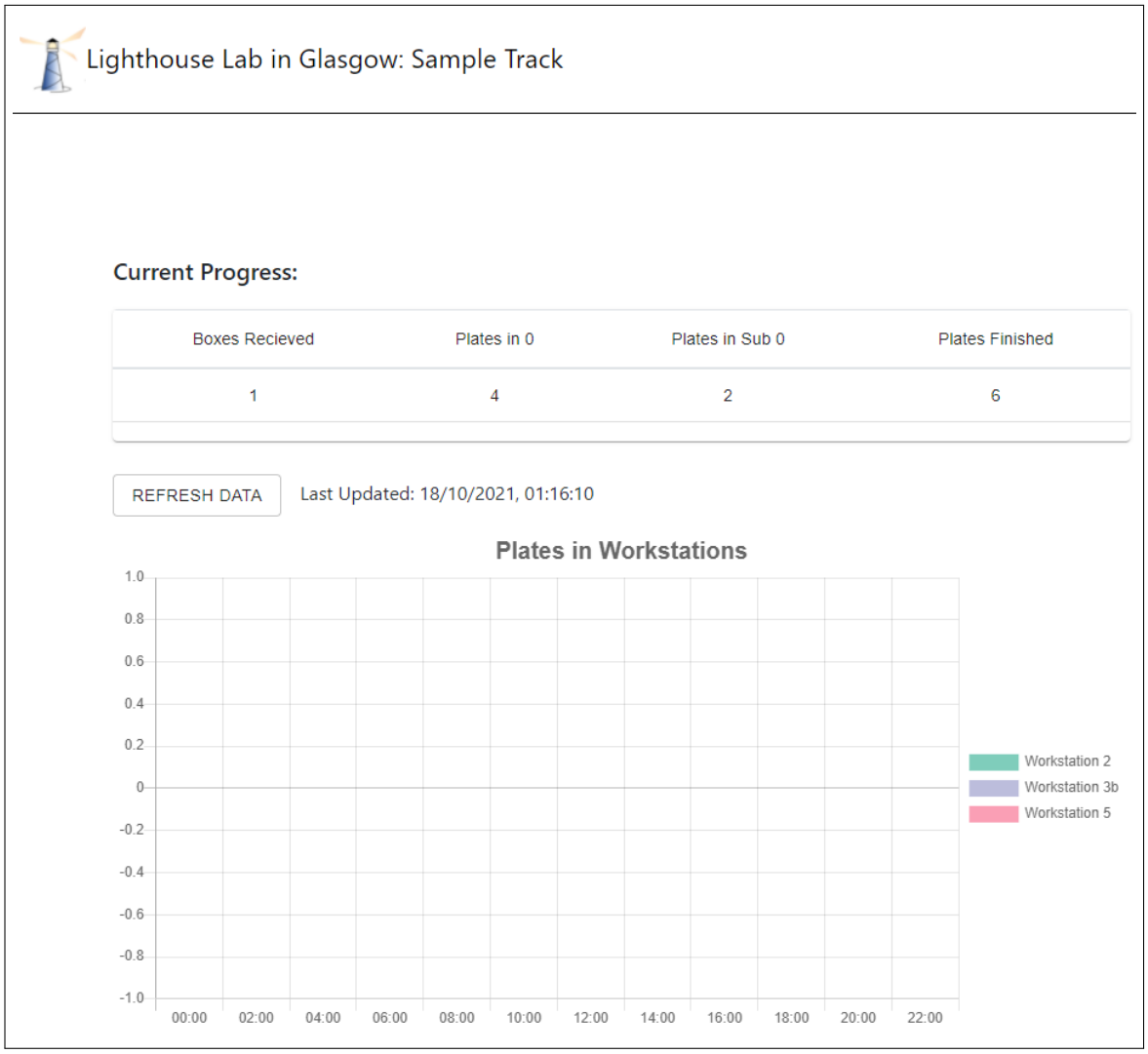

Figure 5.1: Lighthouse home Page I.

5.3.3 Software Functionality

Processing samples in the lab happens in 12 workstation. These are Receipt, Sub 0, 0, 1A, 1B, 2, 3A, 3B, 4A, 4B, 5, and lastly 5QC. The home page of the sample tracker web application contains information about the work progress in the lab. It shows the number of sample boxes that have been received in the lab as well as the number of plates that have been processed. Moreover, it shows live data of the plates that are being processed in each workstation. The home page also illustrates an overview of the messages that have been posted by lab operators, see Figures 5.1 and 5.2.

Furthermore, sample crates are received in workstation Receipt. Operators then need to process them by entering/scanning their operator id and then entering/scanning the unique crate id. After that, the same operator needs to scan/enter the source location of the incoming samples. This can be done by either manually typing the source or by selecting the source

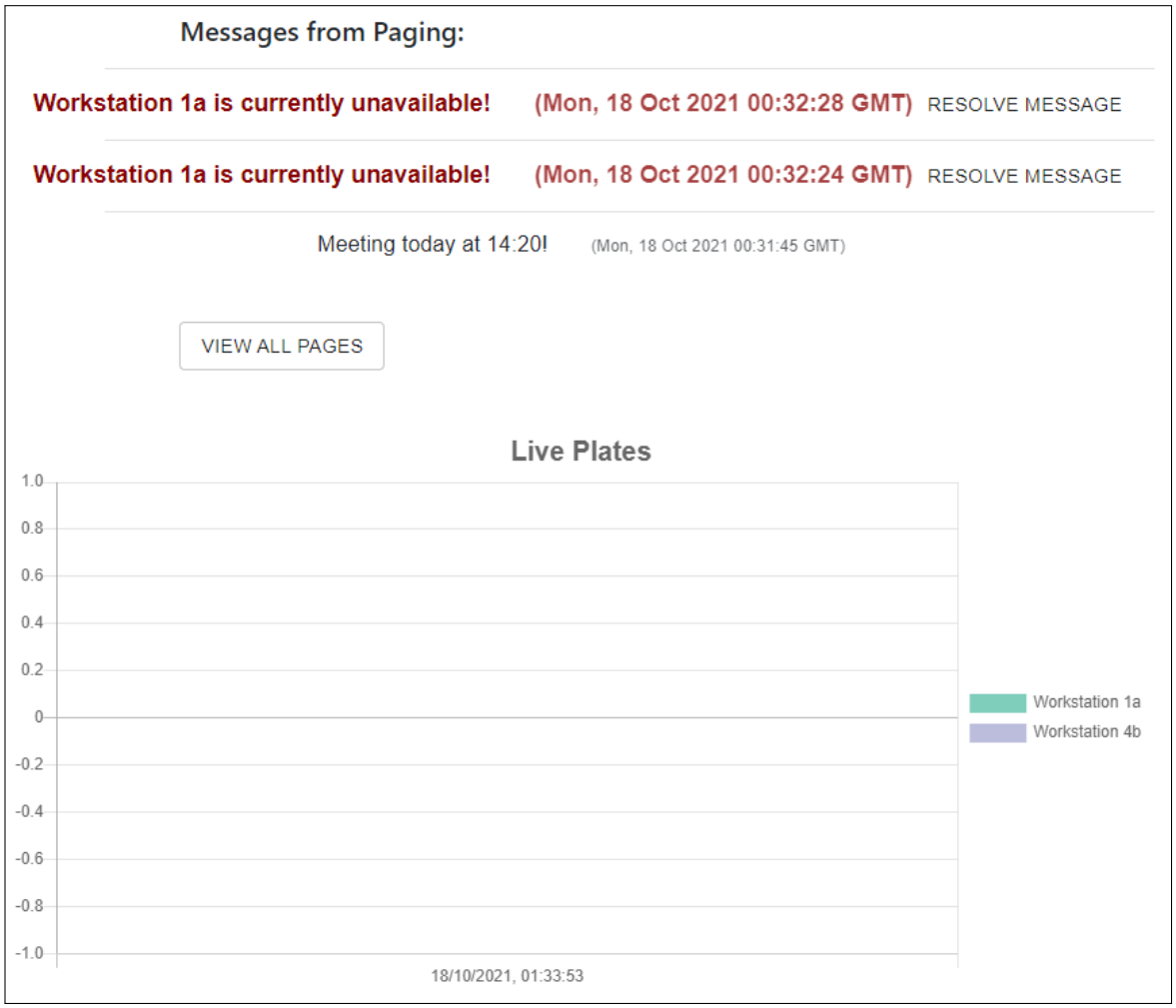

Figure 5.2: Lighthouse home page II.

from the drop-down list. Then the operator enters the receipt priority. This can be either low, medium or high. This is achieved by selecting one of these priority values from the drop down list. Once the operator submits the form that contains the foregoing values, then the samples become available for the other workstations to process them.

In addition, reagents are substances used to carryout laboratory tests. Reagents such as ethanol and magnetic beads can be utilised in detecting the presence of other substances. However, they need lab's managers approval before being used in the lab. The web application supports the process of uploading reagents data to be viewed by managers before authorising them. It starts with the operator uploading the reagents data. Then they are available for managers to view and validate. Once they are validated by managers, then lab operators can start utilising them.

Further, the web application also supports many other functionalities. For instance, users can log into the system to show their attendance. Users can also send messages that can be viewed by all other users. It also can be used to create tickets and report work issues such as process error, equipment malfunctioning, etc. Moreover, search functionality is implemented in the system and the users can search for operators, instruments, workstations, samples, etc. Users can also void samples and plates for various reasons such as user error, plate dropped, contamination, etc.

5.3.4 Software Analysis

Table 5.1 illustrates the various languages and technologies that the software team utilised in developing the system. It also shows the estimate lines of code for each individual language and technology, and also for the overall system. Furthermore, the table demonstrates the number of files, comments and blank lines for each language and for the system as a whole. For example, the system has 65 Python files, 3001 lines of code, 54 comment lines and 738 blank lines. Similarly, the software system has 7383 lines of JavaScript code, 51 lines of comments, 512 blank line and 50 files. Overall, the system has a sum of 144 files, 35458 lines of code, 139 lines of comments and 1539 blank line. The data in the table was automatically generated using an open source tool called cloc [99].

5.3.5 Results

The quantitative results for the Lighthouse Laboratory Sample Tracker are as follows:

- Scenarios implemented: 80
- Total steps (across all scenarios): 585

Language	Files	Blank	Comment	Code
JSON	3	0	0	14605
SQL	2	24	0	9452
JavaScript	50	512	51	7383
Python	65	738	54	3001
CSS	9	66	2	436
Others	15	199	32	581
SUM:	144	1539	139	35458

Table 5.1: Estimate lines of code for the Lighthouse Laboratory Sample Tracker.

Serial	Given Steps (Preconditions)
1	Given I have opened the webpage “page url” in tab “tab number”

Table 5.2: The custom “Given” steps (preconditions) resulted from the Lighthouse Laboratory Sample Tracker.

- Average steps per scenario: 585 / 80 = 7.31
- Custom steps for the case study: 17
- Average custom steps per scenario: 17 / 80 = 0.21
- Maximum scenarios for a feature: 6
- Minimum scenarios for a feature: 1
- Number of scenarios that could not be implemented: 0

These metrics indicate that the framework was efficiently integrated within this system, with an average of approximately 7.3 steps per scenario and a minimal reliance on custom steps (about 0.21 per scenario).

The custom steps that resulted from the case study are shown in Tables 5.2, 5.3, and 5.4.

5.4 Case Study II: Study Abroad Credits

In this section, I demonstrate the second case study.

5.4.1 Background

Study abroad credit system is a credit recording and grade mapping web application for the University of Glasgow’s School of Law. The students can utilise the application to record the host institution they attend, record courses, grades earned in courses, etc. Furthermore,

Serial	When Steps (Actions)
1	When I switch to tab “tab number”
2	When I switch to the tab titled “page title”
3	When I enter the value “some value” into the text box above the label “some label”
4	When I enter the value “some value” into the text box below the label “some label”
5	When I enter the value “some value” into the text box that is both nearest the label “some label” and below the label “some other label”
6	When I enter the value “some value” into the password box nearest the label “some label”
7	When I enter the date “mmddyyyy” into the date picker nearest the label “some label”
8	When I click the button above the label “some label”
9	When I click the button below the label “some label”
10	When I upload the file “some path” by clicking the “some label” button
11	When I select the value “some value” from the drop down list below the label “some label”

Table 5.3: The custom “When” steps (actions) resulted from the Lighthouse Laboratory Sample Tracker.

Serial	Then Steps (Assertions)
1	Then the check box button nearest the label “some label” is selected
2	Then the radio button nearest the label “some label” is selected
3	Then the value “some value” in the drop down list nearest the label “some label” is selected
4	Then I should see the message “some message”
5	Then the file “filename.extension” is downloaded

Table 5.4: The custom “Then” steps (assertions) resulted from the Lighthouse Laboratory Sample Tracker.

the application can be utilised by the School of Law administrators to add host institutions, specify study abroad students, set email reminder notifications for students, etc.

In addition, the software development team used various technologies in developing the system. They adopted ReactJS in developing the user interface components. Furthermore, the team adopted Django [100] in developing the back-end side of the project. Django is a web application development framework that is written in Python. The team has also adopted Docker in building, running, and managing containers on servers and the cloud.

5.4.2 Software Team

The software team of the School of Law study abroad credit recording and grade mapping web application consists of a group of six professional software developers. Two developers were responsible for developing and maintaining the front-end side of the system. One developer was also involved in the front-end development as well as the back-end. Two developers were assigned to build and maintain the back-end side of the project as well as taking care of the IT operations. Lastly, one developer was responsible for working on the front, back-end side of the system and also IT related operations.

5.4.3 Software Functionality

The end users of the study abroad credits web application are the students and administrators of the Law School at the University of Glasgow. The landing page of the web application as shown in Figure 5.3 contains the login page for both end user groups. The students can login to the application using their student id and password, whereas the school administrators have unique credentials provided by the development team. Each end user group has a different set of features.

Furthermore, once the students log in to the web application successfully, they will be able to record their host institution, law plan, and study duration in the host institution. Further, they will have a chance to express their feelings about the fairness of grade conversion into Glasgow University grades. In addition, the students can also record any special circumstances that affected their performance while studying abroad. Further, they can record transcript grades, upload transcript, and submit it for review. When the transcripts are submitted by the students, they will become available for review by the Law School administrators.

5.4.4 Software Analysis

Table 5.5 illustrates the various languages and technologies that the software team utilised in developing the web application. It also shows the estimate lines of code for each individual

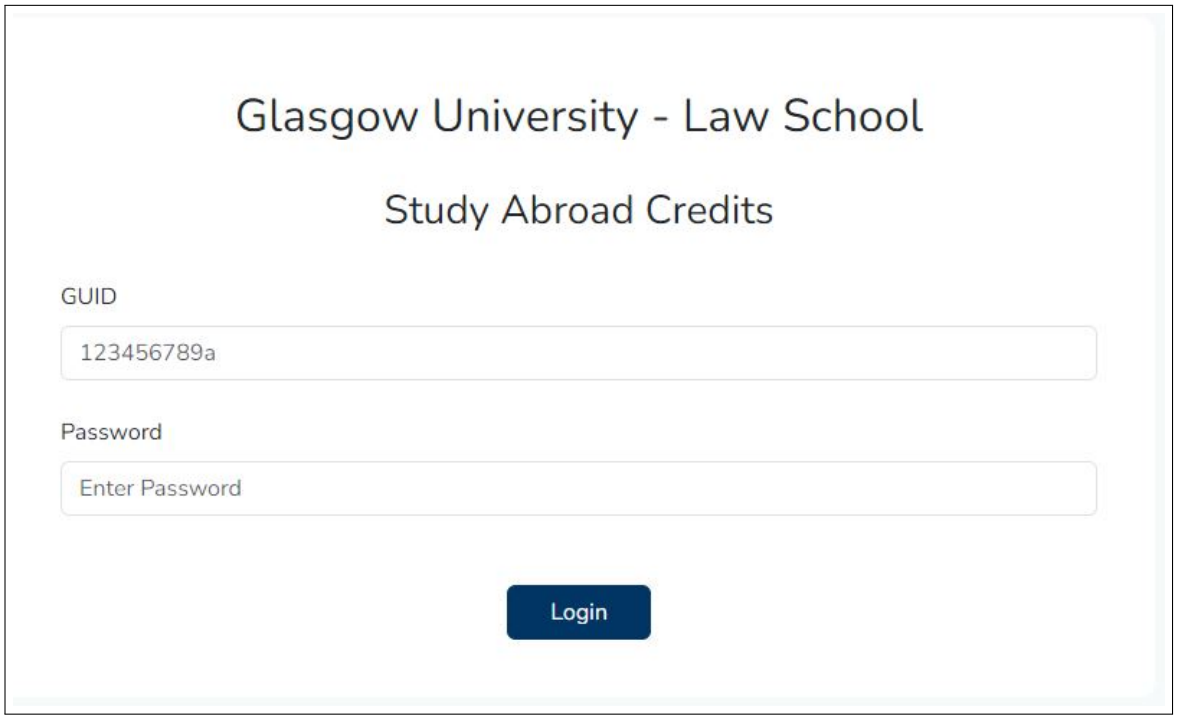

Figure 5.3: Login page for study abroad credits web application.

Language	Files	Blank	Comment	Code
JSON	3	0	0	35667
Python	14	247	64	2099
JavaScript	29	467	50	2095
CSS	10	126	58	521
Others	11	120	33	385
SUM:	67	960	205	40777

Table 5.5: Estimate lines of code for Study Abroad Credits system.

language and technology, and also for the overall system. Furthermore, the table demonstrates the number of files, comments and blank lines for each language and for the system as a whole. For example, the system has 14 Python files, 2099 lines of code, 64 comment lines and 247 blank lines. Similarly, the software system has 2095 lines of JavaScript code, 50 lines of comments, 467 blank lines and 29 files. Overall, the system has a sum of 67 files, 40777 lines of code, 205 lines of comments and 960 blank lines. The data in the table was automatically generated using cloc.

5.4.5 Results

The quantitative results for the Study Abroad Credits system are as follows:

- Scenarios implemented: 28

Serial	When Steps (Actions)
1	When I select the value “some value” from the multiple select list nearest the label “some label”
2	When I select the values “some value” and “some other value” from the multiple select list nearest the label “some label”

Table 5.6: The custom When steps (actions) resulted from Study Abroad Credits.

Serial	Then Steps (Assertions)
1	Then I should not see the message “some message”

Table 5.7: The custom Then step (assertion) resulted from Study Abroad Credits.

- Total steps (across all scenarios): 286
- Average steps per scenario: 286 / 28 = 10.21
- Custom steps for the case study: 3
- Average custom steps per scenario: 3 / 28 = 0.11
- Maximum scenarios for a feature: 4
- Minimum scenarios for a feature: 1
- Number of scenarios that could not be implemented: 0

This case study shows that even with a higher average number of steps per scenario (10.2), the framework maintained a very low requirement for custom steps (0.11 per scenario). The fact that no scenarios failed to be implemented confirms the robustness of the approach.

The custom steps that resulted from the case study are shown in Tables 5.6 and 5.7.

5.5 Case Study III: POPU

In this section, I demonstrate the third case study.

5.5.1 Background

Popu is an e-commerce platform that allows small businesses to create online stores and sell online. The software development team used various technologies in developing the application. They adopted ReactJS in developing the front-end side of the application. In addition, the team adopted Django in developing the back-end side of the application. Furthermore, the team has adopted Docker in building, running, and managing containers on servers and the cloud.

5.5.2 Software Team

The software team of Popu e-commerce platform consists of a group of five professional software developers. Two developers were responsible for developing and maintaining the front-end side of the application. Two developers were assigned to build and maintain the back-end side of the project as well as taking care of the IT operations. Lastly, one developer was also involved in both the front-end development as well as the back-end side of the application.

5.5.3 Software Functionality

Popu is an e-commerce platform that enables small business to create customisable online stores and start selling their products online. The landing page of the web application as shown in Figure 5.4 contains the login page for existing customers. It also contains a link that takes potential customers to a new page where they can create an account. Furthermore, existing customers can reset their passwords by clicking on the "Reset it here" link and following the given instructions. In addition, both existing and potential customers can chat with Popu team by clicking on the "Chat with us" link and following the given instructions.

As soon as the user logs in, they will be taken to the overview page where they can view total sales, top selling products, etc. The user can also view existing customers, available products, product categories, etc. Furthermore, the platform provides many other functionalities. For example, users can add categories and products to their online store.

5.5.4 Software Analysis

Table 5.8 illustrates the languages and technologies that the software team exploited in developing the Popu e-commerce platform. It also shows the estimate lines of code for each individual language and technology, and also for the overall system. In addition, the table demonstrates the number of files, comments and blank lines for each language and for the system as a whole. For example, the system has 27 Python files, 2375 lines of code, 92 comment lines and 378 blank lines. Similarly, the software system has 2435 lines of JavaScript code, 62 lines of comments, 536 blank lines and 47 files. Overall, the system has a sum of 102 files, 45361 lines of code, 446 lines of comments and 1189 blank lines. The data in the table was automatically generated using cloc.

5.5.5 Results

The quantitative results for the POPU e-commerce platform are as follows:

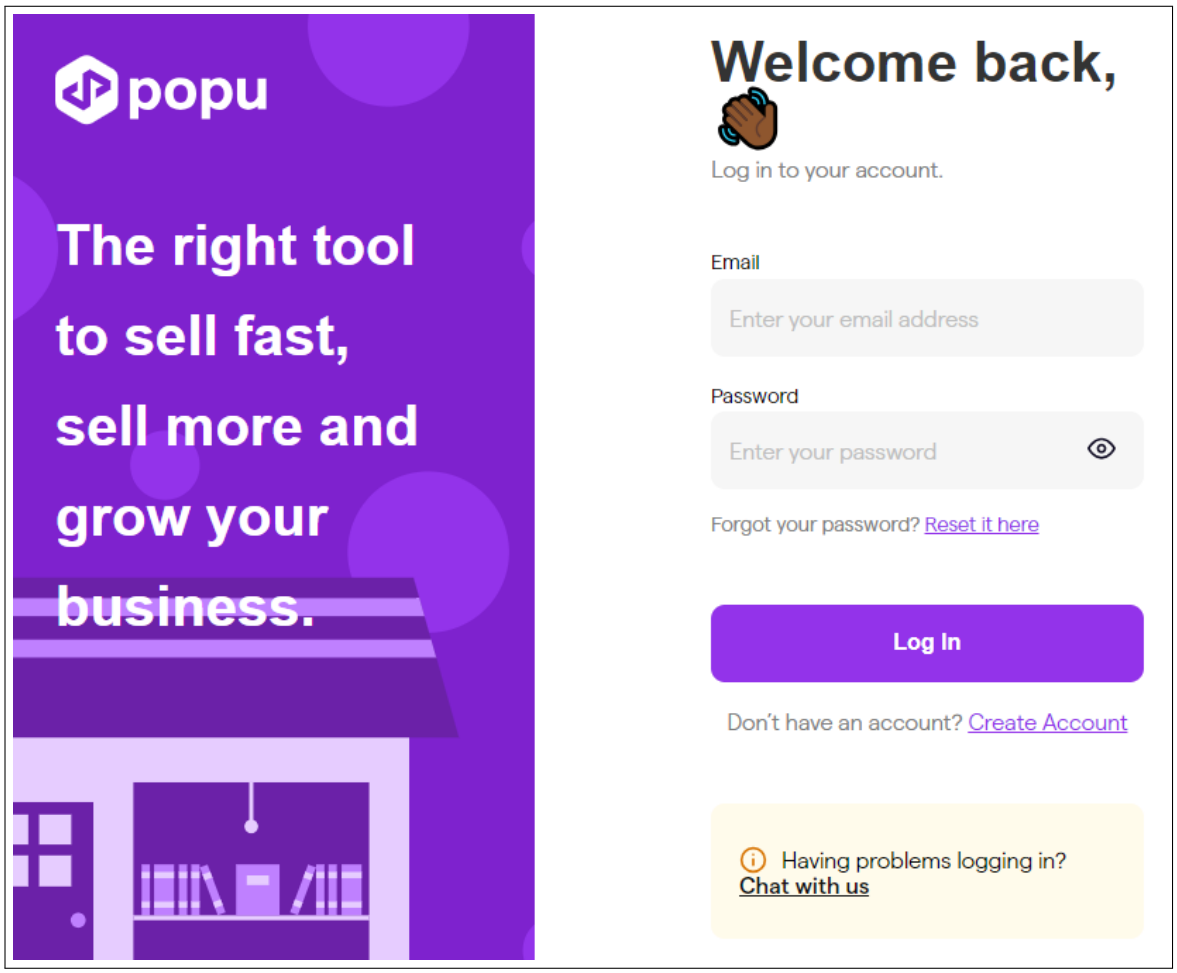

Figure 5.4: Popu customer login page.

Language	Files	Blank	Comment	Code
JSON	4	0	0	39234
Python	27	378	92	2375
JavaScript	47	536	62	2435
CSS	12	133	64	832
Others	12	142	23	485
SUM:	102	1189	446	45361

Table 5.8: Estimate lines of code for Popu e-commerce platform.

- Scenarios implemented: 42
- Total steps (across all scenarios): 510
- Average steps per scenario: 510 / 42 = 12.14
- Custom steps for the case study: 0
- Average custom steps per scenario: 0 / 42 = 0
- Maximum scenarios for a feature: 10
- Minimum scenarios for a feature: 1
- Number of scenarios that could not be implemented: 0

In this case study, the higher average steps per scenario (12.1) reflect the increased complexity of the e-commerce platform. Notably, no custom steps were required, and all planned scenarios were successfully implemented, reinforcing the framework's scalability.

5.6 Summary of Findings

Across all three case studies, the following conclusions can be drawn from the quantitative metrics:

- Every planned scenario was successfully implemented (0 scenarios could not be implemented), demonstrating comprehensive coverage and the framework's reliability.
- The average number of steps per scenario varied with system complexity—from approximately 7.3 in Case Study I to 12.1 in Case Study III—indicating that the language adapts well to different contexts.
- The consistently low average number of custom steps per scenario (ranging from 0 to 0.21) underscores that the standardised UI domain specific approach minimises additional glue code development requirements.
- The variation in maximum scenarios per feature (from 4 to 10) further illustrates the framework's flexibility in accommodating diverse feature complexities.

Overall, these results confirm that the UI Domain Specific Gherkin language and its acceptance test suites are both effective and scalable across diverse systems, achieving full implementation with minimal custom steps and glue code. In spite of that, it is likely that some custom steps will be needed, e.g. more likely for "Then" steps. For example, a custom "Then" step will be needed when JavaScript animation libraries and functions are utilised in moving a Document Object Model (DOM) element across a web page or changing its style.

5.7 Summary

This chapter presented an evaluation of the UI Domain Specific Gherkin language and its associated acceptance test suites through a series of independent, sequential case studies. Each case study was designed to quantitatively assess key performance metrics such as the number of implemented scenarios, total steps, custom steps, and the variability of scenario counts per feature.

Across all three case studies, every planned scenario was successfully implemented, with minimal reliance on custom steps. The framework demonstrated adaptability, as indicated by the variation in average steps per scenario—from approximately 7.3 in the Lighthouse Laboratory Sample Tracker to 12.1 in the POPU e-commerce platform—while consistently maintaining low custom steps and glue code requirements. These findings underscore the feasibility, scalability, and robustness of the UI domain specific approach.

In the next chapter, the thesis details a laboratory study designed to evaluate whether scenarios written in UI domain specific Gherkin are more comprehensible and easier to document than those written in Business Gherkin.

Chapter 6

Laboratory Study

In Chapters 3 and 4, I presented the design of UI Domain Specific Gherkin language. Gherkin scenario steps describe how a software system should behave, or how a user will interact with it. Additionally, I presented the design and implementation of the step function library that connects the scenario steps to the application code for testing the System Under Test (SUT). In Chapter 5, I presented the case studies that helped improve and validate the design and implementation of the UI Domain Specific Gherkin framework. In this Chapter, I present the laboratory study that aimed to test whether or not scenarios written in UI Domain Specific Gherkin are easier to comprehend and document than scenarios written in Business Gherkin. This chapter is structured as follows. Section 6.1 presents the methodology used to design the study. Sections 6.2 and 6.3 present the study's results and findings for software and non-software practitioner participants, respectively. Section 6.4 gives a summary of the findings of this laboratory study.

6.1 Research Method

This study employed a **between-subjects laboratory experiment** to evaluate whether UI Domain Specific Gherkin scenarios are easier to comprehend and document than Business domain Gherkin scenarios. A between-subjects laboratory experiment is a research method where different groups of participants experience different conditions, making it easier to compare their responses. This design is especially useful when researchers want to see how an independent variable influences a dependent variable without worrying about repeated testing affecting the results. In this study, the independent variable is the type of Gherkin documentation style used, and the dependent variable is how well participants perform in comprehending and documenting the scenarios. I am investigating how changing the style of Gherkin impacts the participants' performance.

Egele et al. [101] found that this approach led to more consistent responses across different instructions, showing that individual differences can play a big role when the same participants go through multiple conditions. Charness et al. [102] also pointed out that using a between-subjects design helps prevent carryover effects, which can happen in within-subjects experiments when participants experience all conditions and their previous experiences influence later responses. On top of that, Bathke et al. [103] highlighted how this design strengthens statistical analysis, especially in multivariate studies, where keeping data independent is crucial for drawing accurate conclusions. Another big advantage of this method is that it reduces issues like participant fatigue or learning effects, since each person only takes part in one condition, keeping their responses fresh and unbiased.

I chose a between-subjects laboratory experiment to examine how the independent variable (the type of Gherkin documentation style used) affects the dependent variable (how well participants perform in comprehending and documenting the scenarios) without the risk of repeated testing influencing the results. This approach also helps minimise issues like participant fatigue and learning effects, as each person experiences only one condition, ensuring their responses remain clear and unbiased. The research experiment was designed to test the following hypotheses:

- **H1:** UI Domain Specific Gherkin scenarios are easier to comprehend than Business domain Gherkin scenarios.
- **H2:** UI Domain Specific Gherkin scenarios are easier to document than Business domain Gherkin scenarios.

6.1.1 Experimental Setup

The experiment involved two groups: one working with **UI Domain Specific Gherkin** and the other with **Business domain Gherkin**. Participants were randomly assigned to either condition to mitigate bias. The study followed these steps:

1. **Task Assignment:** Each participant received a set of BDD scenarios written in either UI domain specific Gherkin or Business domain Gherkin.
2. **Comprehension Task:** Participants read and answered questions assessing their understanding of the given scenarios.
3. **Documentation Task:** Participants rewrote missing steps in scenarios to assess their ability to document behaviour correctly.
4. **Time Measurement:** The time taken to complete comprehension and documentation tasks was recorded.

6.1.2 Data Collection and Analysis

The collected data included:

- **Comprehension Accuracy:** The percentage of correctly answered comprehension questions.
- **Documentation Accuracy:** The percentage of correctly answered documentation questions.
- **Time Metrics:** Time taken to complete comprehension and documentation tasks.

For analysis, Student t-test was used to compare the two groups, ensuring results were statistically significant. The Student's t-test is a widely used statistical method for determining whether there is a significant difference between the means of two groups, particularly effective with small sample sizes and normally distributed data [104]. It is valued for its simplicity, requiring only basic calculations, and its versatility in handling both independent and paired samples [105]. The test is especially useful in small-sample scenarios, maintaining reliable results even with very few observations, provided the data meets assumptions [106]. With a long history of application in fields ranging from healthcare to industry, the Student's t-test remains a fundamental and practical tool in statistical analysis [107].

6.1.3 Survey of GitHub Projects

In order to find BDD suites that have suitable Gherkin features for my study, I surveyed 594 GitHub repositories provided by Islam and Storer [108]. The criteria for selecting suitable Gherkin features were as follows: First, the BDD suite should have a significant number of features and scenarios. I calculated the 90th percentile of the 594 repositories in terms of the numbers of features and scenarios, which was 47 features and 176 scenarios. Second, the application with Gherkin features should be accessible and familiar to participants, e.g., an e-commerce application, and should not contain technical terms that might be unfamiliar, such as testing and documenting REST APIs. Third, Gherkin scenarios should be written in the business domain language and be abstracted. Filtering by size resulted in 45 repositories. Further filtering by accessibility and familiarity resulted in six repositories. Finally, filtering by abstraction level left us with only one result, PrestaShop [109]. PrestaShop is an open-source e-commerce platform that serves both merchants and customers.

Furthermore, I selected three features from the PrestaShop BDD suite [109]: category management, add standard product, and update details feature. From these features, I selected ten scenarios for my study. Figure 6.1 shows an example of a scenario written in Business

```
Scenario: delete category cover image
 Given I edit category ''category1'' with following details:
   | Name                | dummy category name    |
   | Displayed           | false                  |
   | Parent category     | Home Accessories       |
   | Description         | dummy description      |
   | Meta title          | dummy meta title       |
   | Meta description    | dummy meta description |
   | Friendly URL        | dummy                  |
   | Group access        | Visitor,Guest,Customer |
   | Category cover image | logo.jpg              |
 And category ''category1'' has cover image
 When I delete category ''category1'' cover image
 Then category ''category1'' does not have cover image
```

Figure 6.1: Scenario written in Business Domain Gherkin, taken from [109].

Gherkin. Additionally, I translated the same ten scenarios into my UI specific Gherkin. Figure 6.2 shows the scenario that resulted from translating the foregoing scenario into my UI Domain Specific Gherkin.

6.1.4 Study Protocol

To address my two research questions, an online questionnaire was developed in two versions, referred to as Business domain Gherkin and User Interface Domain Specific Gherkin. Both versions start with a participant information sheet containing general information about the study, such as the purpose of the study, possible benefits, and who can participate. The second page contains a consent form that participants had to agree to in order to proceed. After that, participants were optionally requested to supply their current job title and years of experience in the software industry. Then, the researcher provided a BDD overview and training on documenting software features using either Business Domain Gherkin or UI Domain Specific Gherkin, depending on which version was assigned to the participant. This training was planned to take around twelve minutes and took place virtually in a video call with the researcher, where participants had a chance to interact and ask questions.

Furthermore, the next page in the online questionnaire contained information about Presta Shop's user interface, which was used to write both Business and UI Domain Specific Gherkin scenarios. The participants were provided with the URL to the site so that they could familiarise themselves with the interface. Then, I provided them with five untitled scenarios, and their task was to select a title from the multiple-choice list that best described each scenario's overall purpose. The intention here was to gauge how easy it was to comprehend scenarios written in both styles, Business and UI Domain Specific Gherkin. The

```
Scenario: delete category cover image
 Given I click the link labelled ''PC parts''
 And I click the link labelled ''Edit''
 And I enter the value ''dummy category name'' into the text box
     nearest the label ''Name''
 And I click the radio button nearest the label ''Displayed''
 And I click the radio button nearest the label ''Home''
 And I click the radio button nearest the label ''Accessories''
 And I click the radio button nearest the label ''Home Accessories''
 And I enter the value ''dummy description'' into the text box
     nearest the label ''Description''
 And I enter the value ''dummy meta title'' into the text box
     nearest the label ''Meta title''
 And I enter the value ''dummy meta description'' into the text box
     nearest the label ''Meta description''
 And I enter the value ''dummy'' into the text box nearest the
     label ''Friendly URL''
 And I upload the file ''src\test\resources\logo.jpg'' by clicking
     the ''Browse'' button
 And I click the button labelled ''Save''
 And I click the link labelled ''dummy category name''
 And I click the link labelled ''Edit''
 When I click the button labelled ''Delete Cover Image''
 Then I should see the message ''The cover image was successfully
      deleted''
```

Figure 6.2: Equivalent scenario to Figure 6.1, written in UI Domain Specific Gherkin, prepared by the researcher.

number of correctly selected scenario names was taken as measure of how easy the scenario was to comprehend.

After that, the participants were provided with five incomplete scenarios that were missing the last "Then" step(s), and they were requested to complete each scenario. The intention here was to gauge how easy it was to document scenarios written in both styles, Business and UI Domain Specific Gherkin. For the Business Domain Gherkin version, they were requested to select from the multiple-choice list a step function that matched the desired behaviour of the "Then" step(s). However, for the UI Domain Specific Gherkin version, the task was to select from the multiple choice list a scenario step that best completed the scenario. The number of correctly selected step functions and scenario steps was taken as measure of how easy the scenario was to document. Finally, participants had a chance to optionally input their comments about the study. Additionally, they were optionally asked to enter their email address in order to receive updates about the study.

6.1.5 Study Design Validation

To validate the study design and length before release, I asked two PhD students in the School of Computing Science at The University of Glasgow to voluntarily participate in the two studies and provide feedback. The reviewers agreed, and we met each virtually via video call, assigning one study to each reviewer. The Business Gherkin study took 34 minutes to complete, while the UI Domain Specific Gherkin study took 32 minutes. I anticipated each study should take participants 30 to 40 minutes, and the reviewers' times aligned with this expectation. I received positive feedback from the reviewers.

6.1.6 Participant Recruitment

Participants were recruited through academic and industry networks, ensuring a mix of software and non-software practitioners. Several platforms were utilised to reach as many participants as possible. These platforms included X, LinkedIn, Slack, and Google Groups. On X, I used the UofG School of Computing Science account, which had 2,200 followers at the time of writing this chapter. Similarly, I leveraged LinkedIn through the UofG School of Computing Science account, which had 1,351 followers. I also engaged with the Cucumber Community on Slack, which had 18,486 members. Additionally, I promoted my studies via the Behaviour Driven Development Google Group, which had 803 members at the time of writing.

6.1.7 Ethics Statement

Whenever human participants are involved in research, ethical approval is required. I obtained this approval from the College of Science and Engineering at The University of Glasgow before approaching participants and beginning data collection, under approval number 300210184. As a result of the ethical review, research participants can feel confident that potential risks have been considered.

This methodological framework ensures a rigorous evaluation of the proposed approach, allowing for empirical validation of UI Domain Specific Gherkin in BDD contexts.

6.2 Software Practitioner Results

This section describes the results and findings of my study regarding software practitioners. This includes participant demographics, such as job titles and the number of participants for each title, and the years of experience for each job title. Additionally, I demonstrate the statistical significance of comprehending and documenting scenarios written in Business, and UI Domain Specific Gherkin. The statistical significance is based on the correct answers provided by participants in the comprehension and documentation sections, as well as the time they spent answering the questions in both sections.

6.2.1 Demographics

In this subsection, I discuss the demographics for the participants of both Business Domain Gherkin and UI Domain Specific Gherkin study. Figure 6.3 illustrates the demographics of the participants in terms of job title and the number of participants for each job title. Axis x represents the job title groups for the participants, whereas axis y represents the number of participants for each job title. The blue bar represents Business Domain Gherkin participants, and the red bar represents UI Domain Specific Gherkin. I divided the job title groups based on participants response into four groups. The groups are Student, Adademic/Educator, Software Engineer, and Executive Level. The first group represents students who have already studied software development courses in their degrees. The second group represents software engineering/computer science faculty members and computing teachers. The third group illustrates software engineers who work in various organisations. The last group represents the participants who work in executive level careers in various organisations. The figure shows that five of the Business Domain Gherkin participants are students, while three of the UI Domain Specific Gherkin participants are students. It also displays that only one Business Domain Gherkin participant is among the group Academic/Educator, whist there

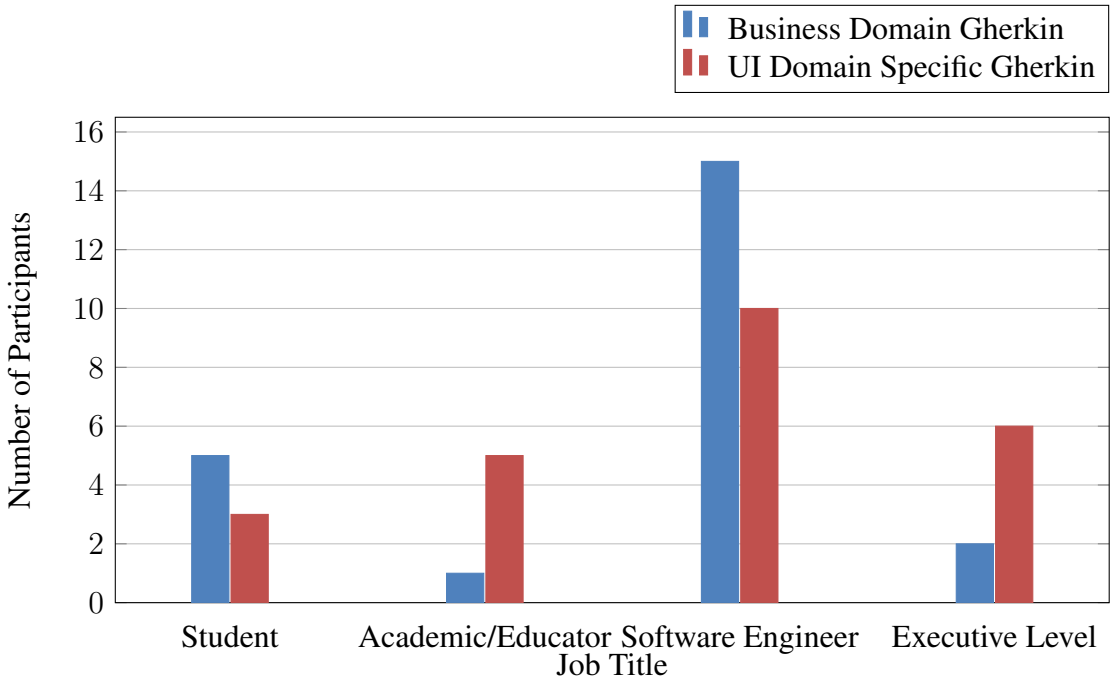

Figure 6.3: Demographics. Job title and number of participants.

are five UI Domain Specific Gherkin participants from the same group. The figure also shows that fifteen software engineers participated in the Business Domain Gherkin study, whereas ten software engineers participated in the UI Domain Specific Gherkin study. Lastly, the figure illustrates that two participants who occupy executive level jobs participated in the Business Domain Gherkin study, whereas six participants from the same level participated in the UI Domain Specific Gherkin study.

Figure 6.4 illustrates the demographics of the participants in terms of job title group and the number of years of experience for each group. Axis x represents the job title groups for the participants, whereas axis y represents the years of experience for each job title. The blue bar represents Business Domain Gherkin participants, and the red bar represents UI Domain Specific Gherkin. I grouped job titles the same way I did in the preceding paragraph. The figure shows that the Business Domain Gherkin student participants have three years of experience in software practicing, while the UI Domain Specific Gherkin student participants have one year of software practicing experience. It also displays that Business Domain Gherkin academic/educator participants have fourteen years of experience, whereas UI Domain Specific Gherkin participants from the same group have fifty six years of experience. The figure also shows that the software engineers who participated in the Business Domain Gherkin study have forty years of experience, whereas the software engineers who participated in the UI Domain Specific Gherkin study have thirty seven years of experience. Lastly, the figure illustrates that the participants who occupy executive level jobs and participated in the Business Domain Gherkin study have forty years of experience, whereas those

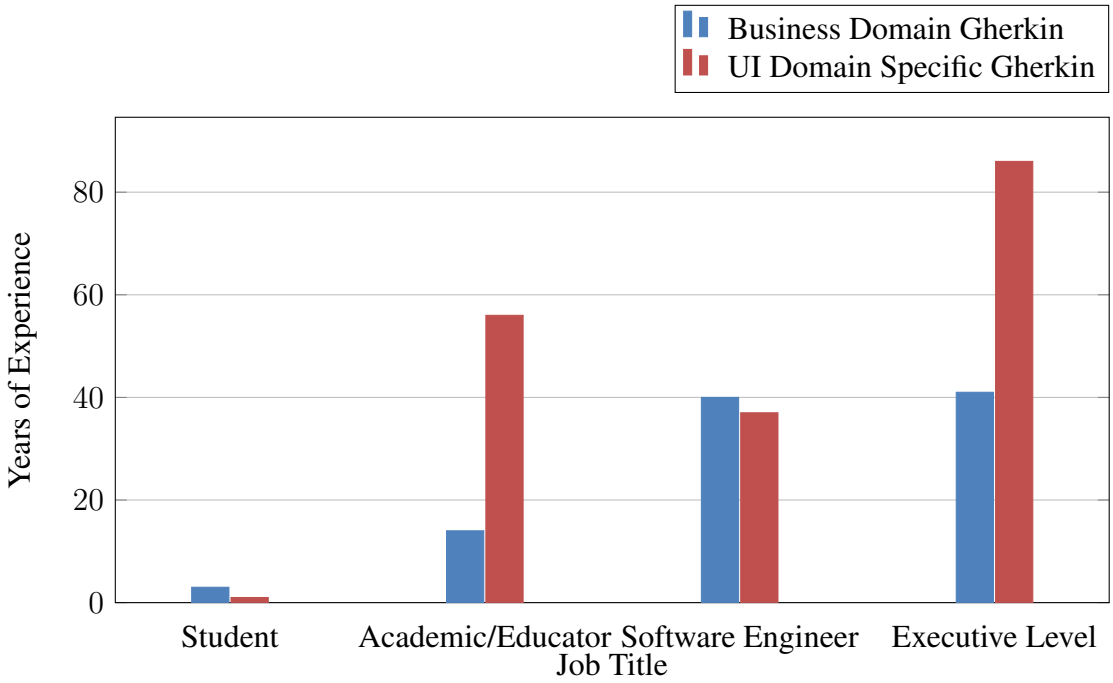

Figure 6.4: Demographics. Job title and years of experience.

from the same group who participated in the UI Domain Specific Gherkin study have eighty six years of experience.

6.2.2 Statistical Significance for Comprehension and Documentation

In this subsection, I discuss the statistical significance for the results that are related to comprehending and documenting Gherkin scenarios that were written in Business and UI Domain Specific Gherkin. Student's t-test was used to indicate whether or not there is a significant difference between the means of two groups [110]. By doing so, I could determine whether those differences were due to chance. Data sets following a normal distribution are usually tested using the t-test because the variance of the population cannot be determined.

I employed t-test in order to find out whether there is statistical significance in the comprehension test or not. To do this, I first calculated the t-value using the following formula:

$$t - value = \frac{|\overline{m}_1 - \overline{m}_2|}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$$

m1: the mean for the number of correct answers for Business Domain Gherkin.
m2: The mean for the number of correct answers for UI Domain Specific Gherkin.
s1: Standard deviation for the number of correct answers for Business Domain Gherkin.

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of correct answers	102	115
Number of participants (n)	23	24
Mean	4.43	4.79
Standard Deviation	0.99	0.41
T-value	1.59	
Critical value	2.01	

Table 6.1: T-test for comprehending Business and UI Domain Specific Gherkin scenarios.

s2: Standard deviation for the number of correct answers for UI Domain Specific Gherkin.
n1: The number of participants in Business Domain Gherkin.
n2: The number of participants in UI Domain Specific Gherkin.

Then, in order to determine whether there is a statistically significant difference between the two samples, I used a two-sample t-test with a 95% confidence level and a 0.05 significance level. Additionally, in t-test, we begin with the hypothesis that there is no statistically significant difference between the results of comprehending Business and UI Domain Specific Gherkin. We also identify a critical value, which is a number if our t-value is lower than that then we do not reject the hypothesis. However, if our t-value is higher than the critical value then we reject our no statistical significance hypothesis and assume that there is statistical significance between the results of the two samples. In order to calculate this critical value, I first calculated the value of degrees of freedom using the following formula:

$$df = n1 + n2 - 2.$$

Then, I used the t-distribution table of critical values to map the resulted degrees of freedom with its corresponding critical value. Table 6.1 shows the calculations of t-value for the comprehension test for Business Domain Gherkin and UI Domain Specific Gherkin. The number of participants for Business Domain Gherkin is twenty three, whereas the number of UI Domain Specific Gherkin participants is twenty four. The total number of correct answers for Business Domain Gherkin is 102, while it is 115 for UI Domain Specific Gherkin. The mean for Business Domain Gherkin is 4.43, whereas it is 4.79 for UI Domain Specific Gherkin. The standard deviation for Business Domain Gherkin is 0.99, whereas it is 0.41 for UI Domain Specific Gherkin. The t-value is 1.59 and the critical value is 2.01. Since the t-value is less than the critical value, we do not reject the no statistically significant hypothesis. Thus, there is no statistical significance difference between the two groups in terms of comprehending Business, and UI Domain Specific Gherkin scenarios.

Also, I employed t-test in order to find out whether there is statistical significance difference

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of correct answers	71	83
Number of participants (n)	23	24
Mean	3.08	3.45
Standard Deviation	1.23	1.47
T-value	0.93	
Critical value	2.01	

Table 6.2: T-test for documenting Business and UI Domain Specific Gherkin scenarios.

in the documentation test or not. To do this, I first calculated the t-value using the same formula that I used in the comprehension test. Table 6.2 shows the calculations of t-value for the documentation test for Business Domain Gherkin and UI Domain Specific Gherkin. The number of participants for Business Domain Gherkin is twenty three, whereas the number of UI Domain Specific Gherkin participants is twenty four. The total number of correct answers for Business Domain Gherkin is 71, while it is 83 for UI Domain Specific Gherkin. The mean for Business Domain Gherkin is 3.08, whereas it is 3.45 for UI Domain Specific Gherkin. The standard deviation for Business Domain Gherkin is 1.23, whereas it is 1.47 for UI Domain Specific Gherkin. The t-value is 0.93 and the critical value is 2.01. Since the t-value is less than the critical value, we do not reject the no statistically significant hypothesis. Thus, there is no statistical significance difference between the two groups in terms of documenting Business, and UI Domain Specific Gherkin scenarios.

6.2.3 Comprehension and Documentation Time Statistical Significance

In this subsection, I discuss the statistical significance difference for the results that are related to the time that the participants spent on comprehending and documenting Gherkin scenarios that were written in Business and UI Domain Specific Gherkin. T-test was used to indicate whether or not there was a significant statistical difference between the means of the two samples. By doing so, I could determine whether those differences were due to chance. Data sets following a normal distribution are usually tested using the t-test because the variance of the population cannot be determined.

Table 6.3 shows the calculations of t-value for the time in minutes participants spent on comprehending Business, and UI Domain Specific Gherkin scenarios. The number of participants for Business Domain Gherkin is twenty three, whereas the number of UI Domain Specific Gherkin participants is twenty four. The total number of minutes the participants spent on comprehending Business Domain Gherkin scenarios is 114, while it is 137 minutes for UI Domain Specific Gherkin scenarios. The mean for time spent on Business Domain

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of minutes	114	137
Number of participants (n)	23	24
Mean	4.96	5.70
Standard Deviation	1.85	2.16
T-value	1.29	
Critical value	2.01	

Table 6.3: Time in minutes participants spent on comprehending Business and UI Domain Specific Gherkin scenarios.

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of minutes	172	180
Number of participants (n)	23	24
Mean	7.30	7.50
Standard Deviation	2.74	3.34
T-value	0.22	
Critical value	2.01	

Table 6.4: Time in minutes participants spent on documenting Business and UI Domain Specific Gherkin scenarios.

Gherkin is 4.96 minutes, whereas it is 5.70 minutes for UI Domain Specific Gherkin. The standard deviation for Business Domain Gherkin is 1.85, whereas it is 2.16 for UI Domain Specific Gherkin. The t-value is 1.29 and the critical value is 2.01. Since the t-value is less than the critical value, we do not reject the no statistically significant hypothesis. Thus, there is no statistical significance difference between the two groups in terms of the time participants spent on comprehending Business, and UI Domain Specific Gherkin scenarios.

Also, I employed t-test in order to find out whether there is statistical significance difference in the time participants spent on documenting Business and UI Domain Specific Gherkin. Table 6.2 shows the calculations of t-value for the time participants spent on documenting Business, and UI Domain Specific Gherkin. The number of participants for Business Domain Gherkin is 23, whereas the number of UI Domain Specific Gherkin participants is 24. The total number of minutes participants spent on Business Domain Gherkin scenarios is 172, while it is 180 for UI Domain Specific Gherkin scenarios. The mean of time in minutes for Business Domain Gherkin is 7.30, whereas it is 7.50 for UI Domain Specific Gherkin. The standard deviation for Business Domain Gherkin is 2.74, whereas it is 3.34 for UI Domain Specific Gherkin. The t-value is 0.22 and the critical value is 2.01. Since the t-value is less than the critical value, we do not reject the no statistically significant hypothesis. Thus, there is no statistical significance difference between the two groups in terms of the time participants spent on documenting Business, and UI Domain Specific Gherkin scenarios.

6.3 Non-software Practitioner Results

This section describes the results and findings of my study regarding non-software practitioners. This includes participant demographics, whether they are professionals or students and the number of participants that represent each category. Additionally, I demonstrate in this section the statistical significance for comprehending and documenting scenarios written in Business, and UI Domain Specific Gherkin. The statistical significance is in terms of the correct answers of participants in comprehension and documentation sections and also the time they spent on answering the questions in both sections.

6.3.1 Demographics

In this subsection, I discuss the demographics for the participants of both Business Domain Gherkin and UI Domain Specific Gherkin study. Figure 6.5 illustrates the demographics of the participants in terms of whether they are professionals or students, and the number of participants for each category. Axis x represents the job title groups for the participants, whereas axis y represents the number of participants for each job title. The blue bar represents Business Domain Gherkin participants, and the red bar represents UI Domain Specific Gherkin. I divided the job title groups based on participants response into two groups which are Student, and Professional. The first group represents non-software practicing students, while the second group represents non-software practitioner professionals such as pharmacists, lawyers, etc. The figure shows that sixteen of the Business Domain Gherkin participants are non-software practitioner students, while nineteen of the UI Domain Specific Gherkin participants are non-software practitioner students. It also displays that seven Business Domain Gherkin participant is among the group non-software practitioner professionals, whist there are five UI Domain Specific Gherkin non-software practitioner participants.

6.3.2 Statistical Significance for Comprehension and documentation for Non-software Practitioners

In this subsection, I discuss the statistical significance for the results of comprehending and documenting Business and UI Domain Specific Domain Gherkin scenarios by non-software practitioner participants. As in Section 6.2 t-test was used to indicate whether or not there is a significant difference between the means of the two groups.

Table 6.5 shows the calculations of t-value for comprehending Business, and UI Domain Specific Gherkin. The number of participants for Business Domain Gherkin is twenty three, whereas the number of UI Domain Specific Gherkin participants is twenty four. The total

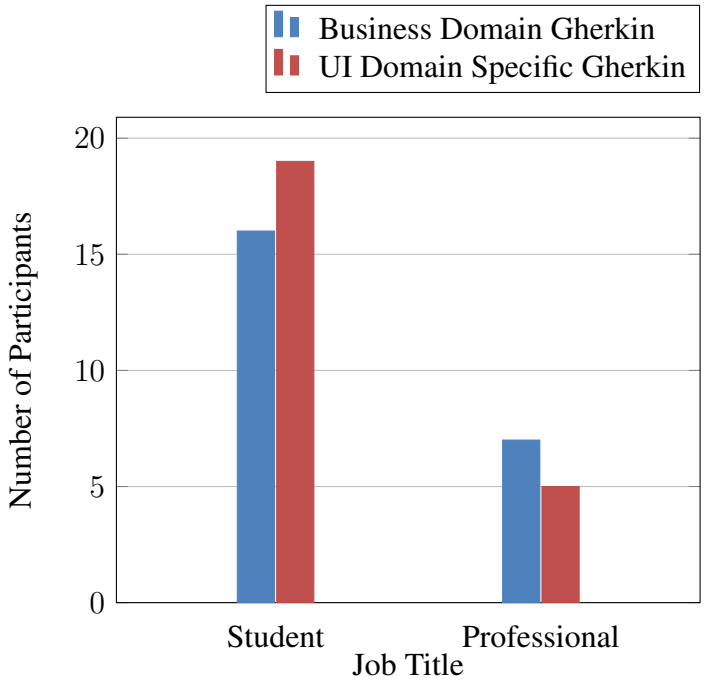

Figure 6.5: Non-software practitioner demographics (student/professional) and number of participants.

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of correct answers	82	91
Number of participants (n)	23	24
Mean	3.57	3.79
Standard Deviation	1.04	1.10
T-value	0.73	
Critical value	2.01	

Table 6.5: T-test for comprehending Business, and UI Domain Specific Gherkin scenarios for non-software practicing participants.

number of correct answers for Business Domain Gherkin is 82, while it is 91 for UI Domain Specific Gherkin. The mean for Business Domain Gherkin is 3.57, whereas it is 3.79 for UI Domain Specific Gherkin. The standard deviation for Business Domain Gherkin is 1.04, whereas it is 1.10 for UI Domain Specific Gherkin. The t-value is 0.73 and the critical value is 2.01. Since the t-value is less than the critical value, we do not reject the no statistically significant hypothesis. Thus, there is no statistical significance difference between the two groups in terms of comprehending Business, and UI Domain Specific Gherkin scenarios.

Also, I employed t-test in order to find out whether there is statistical significance in documenting Business, and UI Domain Specific Gherkin scenarios or not for non-software practitioners. To do this, I first calculated the t-value using the same formula that I used previously. Table 6.6 shows the calculations of t-value for documenting Business, and UI Domain Spe-

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of correct answers	52	83
Number of participants (n)	23	24
Mean	2.26	3.46
Standard Deviation	1.32	1.10
T-value	3.37	
Critical value	2.01	

Table 6.6: T-test for documenting Business, and UI Domain Specific Gherkin scenarios for non-software practicing participants.

cific Gherkin scenarios. The number of participants for Business Domain Gherkin is twenty three, whereas the number of UI Domain Specific Gherkin participants is twenty four. The total number of correct answers for Business Domain Gherkin participants is 52, while it is 83 for UI Domain Specific Gherkin participants. The mean for Business Domain Gherkin is 2.26, whereas it is 3.46 for UI Domain Specific Gherkin participants. The standard deviation for Business Domain Gherkin is 1.32, whereas it is 1.10 for UI Domain Specific Gherkin. The t-value is 3.37 and the critical value is 2.01. Since the t-value is greater than the critical value, we reject the no statistically significant hypothesis. Thus, based on the given evidence, there is statistical significance difference between the two groups of participants in terms of documenting Business, and UI Domain Specific Gherkin scenarios.

6.3.3 Time Statistical Significance for Non-software Practitioner Participants

In this subsection, I discuss the statistical significance difference for the results that are related to the time that the non-software practitioner participants spent on comprehending and documenting Gherkin scenarios that were written in Business and UI Domain Specific Gherkin. T-test was used to indicate whether or not there was a significant statistical difference between the means of the two samples. By doing so, I could determine whether those differences were due to chance. Data sets following a normal distribution are usually tested using the t-test because the variance of the population cannot be determined.

Table 6.7 shows the calculations of t-value for the time in minutes participants spent on comprehending Business, and UI Domain Specific Gherkin scenarios. The number of participants for Business Domain Gherkin is twenty three, whereas the number of UI Domain Specific Gherkin participants is twenty four. The total number of minutes the participants spent on comprehending Business Domain Gherkin scenarios is 110, while it is 135 minutes for UI Domain Specific Gherkin scenarios. The mean for time spent on Business Domain Gherkin is 4.78 minutes, whereas it is 5.63 minutes for UI Domain Specific Gherkin. The

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of minutes	110	135
Number of participants (n)	23	24
Mean	4.78	5.63
Standard Deviation	2.07	1.88
T-value	1.46	
Critical value	2.01	

Table 6.7: Time in minutes non-software practicing participants spent on comprehending Business, and UI Domain Specific Gherkin scenarios.

	Business Domain Gherkin	UI Domain Specific Gherkin
Total number of minutes	139	134
Number of participants (n)	23	24
Mean	6.04	5.58
Standard Deviation	3.13	2.02
T-value	0.60	
Critical value	2.01	

Table 6.8: Time in minutes non-software practicing participants spent on documenting Business, and UI Domain Specific Gherkin scenarios.

standard deviation for Business Domain Gherkin is 2.07, whereas it is 1.88 for UI Domain Specific Gherkin. The t-value is 1.46 and the critical value is 2.01. Since the t-value is less than the critical value, we do not reject the no statistically significant hypothesis. Thus, there is no statistical significance difference between the time that non-software practicing participants spent on comprehending Business, and UI Domain Specific Gherkin scenarios.

Also, I employed t-test in order to find out whether there is statistical significance difference in the time non-software practicing participants spent on documenting Business and UI Domain Specific Gherkin scenarios. Table 6.8 shows the calculations of t-value for the time participants spent on documenting Business, and UI Domain Specific Gherkin scenarios. The number of participants for Business Domain Gherkin is twenty three, whereas the number of UI Domain Specific Gherkin participants is twenty four. The total number of minutes participants spent on Business Domain Gherkin scenarios is 139, while it is 134 for UI Domain Specific Gherkin scenarios. The mean of time in minutes for Business Domain Gherkin is 6.04, whereas it is 5.58 for UI Domain Specific Gherkin. The standard deviation for Business Domain Gherkin is 3.13, whereas it is 2.02 for UI Domain Specific Gherkin. The t-value is 0.60 and the critical value is 2.01. Since the t-value is less than the critical value, we do not reject the no statistically significant hypothesis. Thus, there is no statistical significance difference between the time non-software practicing participants spent on documenting Business, and UI Domain Specific Gherkin scenarios.

6.4 Summary

This chapter described a laboratory study designed to evaluate the efficiency of UI Domain Specific Gherkin compared to Business Domain Gherkin, specifically in terms of scenario comprehension and documentation. The results show that scenarios written in UI Domain Specific Gherkin are marginally easier to comprehend than Business Domain Gherkin for both software practitioners and non-software practitioners, see Tables 6.1 and 6.5. Although these results lack statistical significance, they align with my assumption. The study also indicates that comprehending UI Domain Specific Gherkin scenarios takes slightly more time than Business Domain Gherkin scenarios for both groups, see Tables 6.3 6.7, likely due to the verbosity of UI Domain Specific Gherkin.

Additionally, the results reveal that software practitioners find UI Domain Specific Gherkin scenarios marginally easier to document than Business Domain Gherkin scenarios, see Table 6.2. For non-software practitioners, UI Domain Specific Gherkin scenarios are significantly easier to document than Business Domain Gherkin scenarios, see Table 6.6. Whether marginally or significantly easier, this finding aligns with my assumption. However, documenting UI Domain Specific Gherkin scenarios takes slightly more time for software practitioners compared to Business Domain Gherkin scenarios, see Table 6.4. Conversely, non-software practitioners find it marginally quicker to document UI Domain Specific Gherkin scenarios, see Table 6.8.

Furthermore, I acknowledge that five of the six relevant repositories discussed in Section 6.1 contained concrete Gherkin scenarios rather than Business Domain Gherkin scenarios. This indicates I am on the right track, as maintainers of open-source software appear to prefer concrete Gherkin scenarios over business-oriented Gherkin scenarios. Therefore, we should support the practice of writing concrete Gherkin scenarios rather than mandating business-oriented Gherkin scenarios.

Chapter 7

Discussion

This thesis proposed that describing desired system behaviours in Gherkin using user interface specific domain concepts are an effective way of communicating requirements for stakeholders and mitigates the cost of maintaining acceptance tests through the provision of standardised step function suites. Three case studies presented in Chapter 5 were undertaken to find out if the use of user interface specific domain Gherkin in behaviour driven development is feasible. A controlled laboratory experiment presented in Chapter 6 was used to evaluate if Gherkin scenarios written in user interface specific domain are easier to comprehend and document than Gherkin scenarios written in business concepts. This chapter critically examines the findings of this research in relation to the original research questions, highlighting how the new knowledge contributes to the existing body of literature. It also discusses the limitations of the study. Section 7.1 of this chapter addresses the research questions presented in Chapter 1. Section 7.2 presents research validity. Section 7.3 concludes the chapter.

7.1 Addressing Research Questions

This section critically examines the findings of this research in relation to the original research questions, highlighting how the new knowledge contributes to the existing body of literature.

RQ1: Is the proposed low-level framework for user interface specific use of Behaviour Driven Development (BDD) feasible?

The feasibility of using a low-level, UI specific Gherkin framework was investigated through the design and implementation of a standardised step function library. Case studies conducted on real-world software systems—Lighthouse Laboratory Sample Tracker, Study Abroad

Credits, and Popu e-commerce platform—provided empirical validation of the framework's practicality.

The findings confirm that a UI specific Gherkin framework is feasible, addressing a key gap in BDD literature. Existing research (e.g., Binamungu et al. [15]) emphasises the importance of abstraction consistency in BDD scenarios, but does not specifically explore UI-based abstraction. This study extends previous work by demonstrating that maintaining UI-level step definitions enhances scenario maintainability without requiring additional glue code. The case study results align with prior work on BDD maintainability challenges (Irshad et al. [55]), reinforcing the framework's validity.

Across three diverse case studies—the Lighthouse Laboratory Sample Tracker, the Study Abroad Credits system, and the Popu e-commerce platform—every planned scenario was fully implemented using the UI Domain Specific Gherkin framework, with zero scenarios left unaddressed. Quantitatively:

- Average steps per scenario rose from 7.3 (Lighthouse) to 12.1 (Popu), reflecting adaptability to increasing system complexity.
- Custom steps per scenario remained extremely low (0–0.21), demonstrating that the standardised step library covered nearly all UI interactions without additional glue code.

These metrics confirm the framework's scalability (handling both simple and complex systems) and robustness (minimal developer effort to extend step definitions).

Prior work by Binamungu et al. [15] has emphasised the importance of consistent abstraction in BDD scenarios—mixing high and low-level steps increases maintenance overhead and ambiguity. However, existing research stops short of exploring a UI-centric abstraction. By demonstrating that a UI domain specific language can fully express system behaviour without additional glue code, this thesis fills that gap. Moreover, it aligns with Irshad et al. [55] findings that refactoring BDD specifications requires clear, reusable step patterns to reduce technical debt.

RQ2: Are BDD scenarios written in UI Domain Specific Gherkin easier to comprehend than scenarios written in Business Domain Gherkin?

The laboratory study evaluated comprehension efficiency by comparing UI specific and Business domain Gherkin scenarios among software and non-software practitioners. The results showed that participants found UI Domain Specific Gherkin marginally easier to comprehend. This result contributes to ongoing debates in the BDD community regarding the appropriate level of abstraction for test scenarios. While existing best practices recommend

business domain abstraction for readability, demonstrated in Chapter 2, my study suggests that UI-level Gherkin can be similarly effective, especially for stakeholders involved in software development.

The controlled laboratory experiment compared participants' ability to comprehend scenarios written in UI Domain Specific Gherkin against equivalent Business Domain Gherkin. Key findings:

- Software practitioners found UI-specific scenarios marginally easier to comprehend, though the difference did not reach statistical significance.
- Non-software practitioners also trended toward better comprehension with UI-specific steps, suggesting that concrete UI language reduces ambiguity.
- However, UI-specific scenarios were slightly more verbose, leading to marginally longer reading times across both groups.

The professional BDD community often advocates writing in declarative, business-level language to maximise readability and reduce brittleness. For example, the creator of SpecFlow, a BDD framework for .Net, recommends BDD practitioners to write scenarios in a declarative way [16]. Furthermore, in the user guide of Cucumber [17], which is a BDD framework for Ruby, Java, and JavaScript, the authors encourage BDD practitioners to develop scenarios in a declarative way. They believe that writing scenarios in a declarative way makes them less brittle and easier to maintain. Yet, without empirical validation, these guidelines rely on hearsay. By empirically showing that UI-specific steps are at least as comprehensible—particularly for practitioners already familiar with UI testing—this study challenges the prevailing assumption and suggests that concreteness can aid understanding.

RQ3: Are BDD scenarios written in UI Domain Specific Gherkin easier to document than scenarios written in Business Domain Gherkin?

The same laboratory study assessed participants' ability to document new scenarios. Key findings:

- Non-software practitioners were significantly faster and more accurate when writing UI Domain Specific Gherkin compared to Business Domain Gherkin.
- Software practitioners also showed a marginal improvement in documentation accuracy with UI-specific steps, though at a slight time cost due to verbosity.

The results indicated that non-software practitioners documented UI Domain Specific Gherkin scenarios more efficiently than Business Domain scenarios. This finding highlights the

importance of stakeholder background in selecting appropriate Gherkin abstraction levels. Previous research on scenario quality (Binamungu et al. [15]) emphasised step clarity but did not differentiate between technical and non-technical stakeholders. My study extends this perspective by demonstrating that UI-specific Gherkin reduces ambiguity for non-software practitioners, suggesting its potential for bridging communication gaps in cross-functional teams. It also calls into question Cucumber's best-practice guidance to favour business domain language exclusively, suggesting instead a hybrid approach where UI steps serve as a common language for cross-functional collaboration.

Traditional BDD guidance often urges practitioners to eliminate technical vocabulary and write scenarios in purely declarative, business-domain language to maximise reusability and maintainability. The laboratory study results indicate that, for cross-functional teams, a UI-centric approach can significantly lower the barrier for non-technical stakeholders, improving both the accuracy and speed of scenario authoring. This calls for a more nuanced best practice that balances business-domain clarity with UI-domain concreteness.

Together, these analyses show that a UI Domain Specific Gherkin framework is not only feasible in real-world systems but also enhances comprehension and documentation, especially for non-technical stakeholders, without imposing undue maintenance burdens on developers. By grounding these findings in empirical data and contrasting them with existing BDD literature, this work offers a practical, evidence-based alternative to purely business-level Gherkin, with the potential to streamline collaboration and test automation in agile teams.

7.2 Research Validity

Potential threats to internal, external, and construct validity are important to consider in order to assess the rigour of my findings.

7.2.1 Chapter 2

To evaluate the internal and external validity threats in my multivocal literature review in Chapter 2, I followed the relevant guidelines and observations from Sjoeberg et al. [111].

1. **Internal validity:** It refers to whether the methodology I used for my multivocal literature review allows us to confidently attribute my findings to the process I followed, rather than to uncontrolled factors.

   (a) **Selection Bias:** Since my initial pool of sources in the professional literature review in the multivocal literature review was limited to the top 100 Google search

results, there is a risk of selection bias. The ranking of Google search results can be influenced by search engine algorithms, commercial interests, or search history. This was mitigated by following the guidance provided by Raulamo-Jurvanen et al. [30] demonstrated in Section 2.2. Furthermore, Google's search engine uses a sophisticated algorithm that ranks pages based on various factors, such as relevance to the query, and user engagement. The first results are those that the algorithm deems most relevant and authoritative on the topic of interest. Additionally, Google is a viable tool for identifying professional literature, provided its limitations (e.g., personalisation and potential bias) are managed [112]. Further, the researcher attempted to mitigate the personalisation bias by logging out of their Google account to reduce search result tailoring. In addition, the search process was reviewed by a Software Engineering researcher, which adds an extra layer of validation. Furthermore, conducting exhaustive searches through all search results is impractical, especially for professional literature reviews [113]. They describe how searching vast numbers of results would be unfeasible and suggest limiting the number of pages or results to create a manageable and focused screening process.

2. **External validity:** It concerns whether the findings from my literature review can be generalised to other contexts or populations.

   (a) **Generalisability:** Since the sources were drawn from general Google search results, they may not be representative of all software engineering professionals. For example, best practices for BDD in large enterprise environments might differ from those in startups or open-source projects. To mitigate this and to improve generalisability, I consider including sources from a variety of industry sectors in future research to ensure that my review covers BDD best practices across different domains and project sizes.

7.2.2 Chapters 3, 4, and 5

The language design method in Section 3.1 led to a complete UI Domain Specific Gherkin language. To evaluate the internal and external validity threats in my UI domain specific language demonstrated in Chapters 3, 4, and 5, I followed the relevant guidelines and observations from Sjoeberg et al. [111].

1. **Internal validity:** It refers to whether the methodology I used allows us to confidently attribute my findings to the process I followed, rather than to uncontrolled factors.

(a) **Selection Bias:** Since my initial pool of Gherkin Selenium projects demonstrated in Section 3.1 was limited to the top 100 Google search results, there is a risk of selection bias. The ranking of Google search results can be influenced by search engine algorithms, search history, or commercial interests. The researcher attempted to mitigate the personalisation bias by logging out of their Google account to reduce search result tailoring. Furthermore, Google's search engine uses a sophisticated algorithm that ranks pages based on various factors, such as relevance to the query, and user engagement. The first results are those that the algorithm deems most relevant to the topic of interest. In addition, the search process was reviewed by a Software Engineering researcher, which adds an extra layer of validation. Also, the selection of case studies used to validate the UI steps in Section5.2 was necessarily limited to those available through the software partners, which were relatively small in size.

(b) **Instrumentation:** The way I measured project relevance demonstrated in Section 3.1 (e.g., presence of Gherkin files and Selenium glue code) may affect internal validity. This was mitigated by having a Software Engineering researcher review the process.

2. **External validity:**

(a) **Generalisability:** The selection of case studies used to validate the UI steps in Section5.2 was necessarily limited to those available through the software partners, which were relatively small in size. This raises the question of whether the proposed solution would be scalable in larger, more complex environments. In a study on the real world practice of behaviour driven development, Islam [114] surveyed 493 open-source BDD projects on GitHub and found that most BDD suites in open-source projects are small in size. According to the author, 90% of the surveyed BDD projects on GitHub contain less than 100 features, and 80% of the projects contain less than 100 scenarios. Furthermore, the case studies that I conducted, see Sections 5.3, 5.4, and 5.5, were as big as most BDD suites found in the previous study by Islam.

Furthermore, the number of case studies I undertook to improve and validate the UI steps and the language design was a limitation. In spite of the fact that the last case study did not result in additional steps being added to the language, I cannot be certain that if I did not undertake another case study I would not add more steps to the language.

(b) **Temporal Validity:** The technology stack and best practices in web development evolve rapidly. My findings might not hold in the future if frameworks like ReactJS or tools like Selenium are no longer widely used. To mitigate this, I will

periodically review and update the language and step function suites to ensure they align with current practices and tools in the industry, which will help ensure ongoing relevance.

7.2.3 Chapter 6

To evaluate the internal and external validity threats in my controlled laboratory study in Chapter 6, I followed the relevant guidelines and observations from Sjoeberg et al. [111]. Furthermore, to evaluate the construct validity threats, I followed the relevant guidelines and observations from Sjøberg and Bergersen [115].

1. **Internal validity:** It refers to the extent to which a study can accurately demonstrate a cause-and-effect relationship between the treatment (independent variable) and the observed outcome (dependent variable). In other words, it ensures that the changes in the dependent variable are genuinely caused by the manipulation of the independent variable, and not influenced by other uncontrolled factors. This means that internal validity confirms that the observed effects are attributable to the experimental treatment rather than external or confounding variables.

   (a) **Selection Bias:** If participants self-select into the study, those who choose to participate may not be representative of the general population. This could affect how well your results reflect the treatment effect. For example, more experienced participants may find one style of Gherkin easier to work with simply because they are more familiar with documenting software features, regardless of the Gherkin version. This was mitigated by randomising participant assignment to the two Gherkin versions to reduce selection bias.

   (b) **Training Confound:** If some participants engaged more during the training (e.g., ask more questions or interact more with the researcher), their understanding of the task might be better than others, thus skewing the results. It would be important to ensure that the training is consistent across participants and that variations in participant engagement are controlled for. This concern was mitigated by standardising the training and ensuring all participants received the same level of interaction and instruction during the session, see appendices C and D. Furthermore, an interactive exercise was offered to all participants at the end of training, see appendices C and D, enabling them to engage directly with the researcher. This approach ensured that they fully comprehended the training material before proceeding to the assigned tasks.

(c) **Maturation Threat:** It refers to the natural changes that occur in participants over time, which could affect the outcome of a study independently of the experimental treatment or intervention. These changes can be psychological, emotional, or physical, and they include factors like fatigue, boredom, and hunger. This was mitigated by setting reasonable time limits for participants to complete each task. This prevents participants from spending too much time on a single task, which could lead to exhaustion. Furthermore, I asked participants about their engagement and fatigue levels at different points in the study. For example, after the comprehension tasks and again after the documentation tasks, I asked them how focused or tired they felt. This could help us assess whether maturation effects like fatigue were influencing their performance.

(d) **History Effects:** External factors unrelated to the study that occur during the experiment (e.g., distractions in a virtual environment) could influence participants' performance in one group differently than the other. This was mitigated by ensuring that participants complete the study in similar environments, encouraging participants to minimise distractions (e.g., suggesting they find a quiet place during the video call for the training session).

(e) **Mortality Threat (Attrition):** If participants dropped out before completing the study, especially if the dropout was not random, the results might be biased. To mitigate this, I ensured that participants were motivated to complete the entire study. As a result, there were no cases of dropout, ensuring that the study's results were not influenced by participant attrition.

(f) **Regression to the Mean:** If participants were selected based on extreme scores (e.g., very high or low performance), their performance might naturally shift closer to the average in later measurements, making it appear as though there is an effect when there is not. This was mitigated by randomising participant assignment to the two Gherkin versions.

2. **External validity:** It refers to the extent to which the results of a study can be generalised to other settings, populations, times, or situations beyond the specific conditions of the experiment. In other words, it is about how well the findings of a study apply to real-world situations or different groups of people, outside of the controlled environment where the study took place.

   (a) **Population Validity:** If the participants in the study were not representative of the broader population of software professionals (e.g., using mostly students or only junior developers), the results might not generalise to more experienced professionals working in different roles or industries. Furthermore, if non-software practicing participants were not recruited in the study, the results might not be

representative of potential stakeholders. This was mitigated by recruiting a diverse set of participants with varying levels of experience and from different software engineering roles to improve generalisability, see Section 6.2.1. Further, non-software practicing participants were recruited to improve generalisability and inclusion of potential stakeholders, see Section 6.3.1.

(b) **Task Representativeness:**If the tasks in the study were too simplified compared to real-world tasks, this could be an external validity threat. This was mitigated by ensuring that the tasks were as realistic as possible. The scenarios were based on a real-world example, an e-commerce platform, see phase II in appendices C and D.

(c) **Ecological Validity (Realism of the Environment):** The study was conducted virtually with an online questionnaire and a video call, which might not reflect the real-world environments where software engineers typically document features. The lack of interaction with real team dynamics, tools, and deadlines could limit how well my findings generalise to actual work settings. To mitigate this, in future work, I consider conducting a follow-up study in a more realistic work environment or involving participants in team-based tasks to examine how Gherkin documentation works in collaborative, real-world conditions.

(d) **Treatment Generalisation:** The concrete scenario steps in the laboratory study were drawn from a single project. This raises the question of whether incorporating multiple projects would have ensured that the observed effects were not merely artifacts of the particular target application. To mitigate this, I consider conducting follow-up studies with different types of software projects or industries (e.g., finance, healthcare). This helps test whether the findings hold across a range of real-world projects and enhances the generalisability of the results.

3. **Construct Validity:** It refers to the degree to which a test, experiment, or measurement tool accurately represents the theoretical concept or construct it is intended to measure.

(a) **Inadequate Definition of the Concept:** If the concepts in my study (e.g., the ease of scenario documentation and comprehension) were not well-defined, this could lead to issues in how well the measurements represent those concepts. To mitigate this, I provided clear, unambiguous definitions for scenario documentation and comprehension in my study, see Section 6.1.

(b) **Construct Underrepresentation:** If the tasks or indicators in the study only captured a narrow aspect of the concept, the study might not represent the full construct. To mitigate this, I allowed the participants to include qualitative feedback, see appendices C and D.

(c) **Construct-Representation Bias:** If the indicators or tasks used in the study systematically misrepresent the concept (e.g., if multiple-choice questions were overly simplistic for assessing documentation ease), study results might not accurately reflect the real-world application of Gherkin. To mitigate this, I ensured that the tasks represent the concept accurately. For example, I designed tasks that closely mimic real-world documentation challenges to reduce bias, see appendices C and D.

(d) **Mono-Operation Bias:** If I relied on only one indicator to measure a concept, such as only using multiple-choice questions to gauge scenario comprehension, the measurement might be inadequate or biased. I consider to incorporate multiple indicators for each concept in future studies. For example, in addition to scenario comprehension tasks, I could include open-ended questions or ask participants to perform tasks that require deeper interaction with Gherkin scenarios, thus providing a fuller picture of their comprehension.

(e) **Subjectivity in Measurement:** If the tasks required subjective judgments (e.g., rating their own comprehension or ease of use), it could lead to biased results. To mitigate this, I used objective measures, such as task accuracy as measure of how easy the scenario was to comprehend and document, see Section 6.1.

(f) **Consult Experts:** If the operationalisation of study constructs lacks external validation, it could undermine construct validity. To mitigate this, the study was reviewed by two Software Engineering researchers to validate project choice, and to confirm that steps were objectively concrete rather than abstract.

7.3 Summary

This chapter discusses the research findings in relation to the original research questions, highlighting the feasibility, comprehension, and documentation efficiency of UI Domain Specific Gherkin. The case studies confirmed that a standardised UI domain specific BDD framework is practical, improving maintainability and reducing the need for additional glue code. A controlled experiment showed that UI Domain Specific Gherkin provided a slight comprehension advantage, particularly for software practitioners, while non-software practitioners found it significantly easier to document. These findings challenge the industry preference for business domain Gherkin by demonstrating that a UI specific approach can be equally effective. The study contributes to BDD literature by addressing standardisation and abstraction challenges while acknowledging limitations such as statistical significance concerns and the generalisability of case studies. Overall, UI Domain Specific Gherkin is

presented as a viable alternative that enhances test automation, comprehension, and maintainability.

Chapter 8

Conclusion

This thesis proposed that describing desired system behaviours in Gherkin using user interface specific domain concepts are an effective way of communicating requirements for stakeholders and mitigates the cost of maintaining acceptance tests through the provision of standardised step function suites. Three case studies presented in Chapter 5 were undertaken to find out if the use of user interface specific domain Gherkin in behaviour driven development is feasible. A controlled laboratory experiment presented in Chapter 6 was used to evaluate if Gherkin scenarios written in user interface specific domain are easier to comprehend and document than Gherkin scenarios written in business concepts. This chapter concludes this research. Section 8.1 provides an overview of the research. Section 8.2 demonstrates an overview of the research questions. Section 8.3 presents a reflection on the research process. Section 8.4 presents a summary of the research contributions. Section 8.5 presents areas for future work. Section 8.6 concludes the chapter.

8.1 Overview

This thesis investigated the feasibility and efficiency of using UI Domain Specific Gherkin for behaviour driven development (BDD). The research addressed challenges in BDD, particularly the difficulty stakeholders face in expressing and negotiating requirements using business domain concepts, the maintenance overhead of automated acceptance tests, and the selection of appropriate abstraction levels in Gherkin scenarios. The study proposed a UI Domain Specific Gherkin framework, providing standardised step functions that reduce the need for additional glue code and improve maintainability.

8.2 Research Questions Overview

This research answered three key questions:

1. **Feasibility of UI Domain Specific Gherkin (RQ1):** Through multivocal literature review, framework development, and empirical validation via case studies, the study confirmed that UI Specific Domain Gherkin is a viable approach for writing BDD scenarios. Case studies demonstrated its practical applicability and reduced maintenance overhead.

2. **Comprehension of UI vs. Business Domain Gherkin (RQ2):** A controlled laboratory study showed that UI Domain Specific Gherkin is at least as comprehensible as Business Domain Gherkin. While software practitioners showed only marginal differences, non-software practitioners found UI Domain Specific scenarios slightly easier to understand.

3. **Documentation Efficiency (RQ3):** Results indicated that non-software practitioners documented UI Domain specific Gherkin scenarios more efficiently than Business Domain scenarios, suggesting that this approach lowers the barrier to participation for non-technical stakeholders.

8.3 Reflection on the Research Process

The research followed an empirical methodology, integrating multivocal literature review, framework design, case studies, and a laboratory experiment to validate the findings. While the study demonstrated the feasibility and benefits of UI Domain Specific Gherkin, limitations include the controlled nature of the laboratory study, the small number of case studies, and the need for broader industry validation.

8.4 Contribution Summary

This thesis has addressed the stakeholders struggle to express and negotiate requirements in the Gherkin language when described desired system behaviours in terms of business domain goals and concepts. This thesis has also evaluated whether user interface specific use of behaviour driven development can be a more efficient alternative than the use of Business Domain Gherkin. The work starts by providing a multivocal literature review on BDD and its best practices. It then moves to the design and implementation of the User Interface Domain Specific Gherkin and its associated automated acceptance test suites.

The thesis then moves to evaluating the feasibility of user interface specific use of behaviour driven development, and finishes with evaluating the efficiency of user interface specific use of behaviour driven development in comparison with business domain concepts, in terms of scenario comprehension and documentation. The remainder of this section discusses the contributions and conclusions of this thesis. The main contributions of this thesis are summarised in the following points:

- **A multivocal literature review of behaviour driven development (BDD)**
  This thesis began by reviewing the literature of BDD and its best practices. It presented the characteristics of BDD and its current tools and frameworks, BDD benefits and challenges, automated generation of tests, BDD maintenance, and case studies in specialist areas. Furthermore, it presented a professional literature review on BDD best practices. It demonstrated general BDD best practices, Gherkin best practices, and BDD's Cucumber framework best practices.

- **In Chapter 3 and 4**, the design and implementation of the framework comprising of the user interface specific Gherkin language and its associated automated acceptance test suites are presented. The objective was to design and implement the language and the associated library before refining and validating the proposed framework in the following chapter.

- **Chapter 5** presented three case studies of the Lighthouse Laboratory Sample Tracker, the Study Abroad Credits, and Popu e-commerce platform. The objective of undertaking the case studies was to refine and validate the design and implementation of the user interface specific Gherkin language and its associated automated acceptance test suites presented in chapter 3 and 4. In addition, the objective was to evaluate the feasibility of user interface specific use of behaviour driven development. The results from the case studies suggest that user interface specific Gherkin use of behaviour driven development is feasible.

- **In Chapter 6**, the laboratory experiment to evaluate the efficiency of User Interface Specific Gherkin and Business Domain Gherkin. The objective was to determine whether UI Specific Gherkin use of behaviour driven development is more efficient than Business Domain Gherkin in terms of comprehending and documenting Gherkin scenarios. The results of the laboratory study show that scenarios written in UI Domain Specific Gherkin are marginally easier to comprehend than scenarios written in Business Domain Gherkin. This includes both software practicing participants, and non-software practitioners. Although the results are marginal and have no statistical significance, this finding is still aligned with my assumption.

Additionally, the results show that comprehending UI Domain Specific Gherkin scenarios marginally takes more time than comprehending Business Domain Gherkin scenarios for both software and non-software practitioners. This is anticipated due to the fact that my standardised UI Domain Specific Gherkin language is verbose and thus takes more time to comprehend.

Furthermore, the results of the laboratory study show that for software practicing participants, scenarios written in UI Domain Specific Gherkin are marginally easier to document than scenarios written in Business Domain Gherkin. Additionally, the results demonstrate that for non-software practicing participants, scenarios written in UI domain specific Gherkin are significantly easier to document than scenarios written in Business Domain Gherkin. Whether it is marginally or significantly easier to document scenarios written in UI Domain Specific Gherkin than business Domain Gherkin, this finding is aligned with my assumption.

The results also demonstrate that for software practicing participants, documenting UI Domain Specific Gherkin scenarios marginally takes more time than documenting Business Domain Gherkin scenarios. However, it marginally takes non-software practicing participants less time to document scenarios written in UI Domain Specific Gherkin than Business Domain Gherkin.

8.5 Future Work

This thesis has highlighted the potential of adopting User Interface Domain Specific Gherkin in the behaviour driven development methodology. The following paragraphs outline some possible scenarios that could be researched further.

8.5.1 Adopt the UI Domain Specific Gherkin Framework in an Industrial Case Study

This thesis provides answers to some fundamental questions regarding the feasibility of UI Domain Specific Gherkin. It also examines the easiness of comprehending and documenting the language in comparison to Business Domain Gherkin. However, there is still a gap around using UI Domain Specific Gherkin in the industrial context. In the short term, one demanding dimension that needs to be investigated in the industrial context is the reliability and efficiency of the step functions that are mapped to each step. Another demanding dimension that needs to be investigated in the industrial case study is how efficient the software development team finds the language to be in terms of easiness to comprehend and document.

```
Scenario: Fill out and submit interest form
  When I enter the value ''Sam James'' into the text box nearest
       the label ''Name''
  And  I enter the value ''samjames@example.com'' into the text box
       nearest the label ''Email''
  And  I click the button labelled ''Submit''
  Then I should see the message ''Form submitted successfully''
```

Figure 8.1: Example scenario written in UI domain specific Gherkin.

8.5.2 Describe Workflows Alongside UX Design Tools

UX design tools like Figma, Sketch, and Adobe XD are typically used to create interactive prototypes, wireframes, and high-fidelity designs that define the user interface layout and interaction points. The UI Domain Specific Gherkin language can be used to describe the exact interactions defined in these prototypes. For example, consider a prototype created in Figma that includes a form with labels, text boxes, and buttons. The UI Gherkin steps can map directly to interactions with these elements. The design in Figma will have a text box next to the ”Name” and ”Email” labels, and the ”Submit” button. The UI Gherkin language describes the exact interactions with those elements, see Figure 8.1, making it clear how the design will function once implemented.

The UI Domain Specific Gherkin can provide a common ground for collaboration between UX designers and developers. Designers create layouts and prototypes, while developers translate those designs into code. The Gherkin steps provide the link between these two worlds by making the interactions explicit and testable. When designers define a layout with clear interaction points, e.g., buttons, forms, and labels, the UI Gherkin can be used to describe how those elements should behave in the final implementation. Developers can use these detailed steps to ensure that the behaviour of each UI element matches the design.

The key research challenge here is how to effectively integrate the UI Domain Specific Gherkin language with UX design tools (such as Figma, Sketch, and Adobe XD) to create a seamless workflow that aligns UI prototypes with detailed, testable interactions, bridging the gap between design and development. This involves ensuring that the Gherkin language can accurately represent the intent behind UX design elements and that it remains flexible enough to accommodate iterative design changes, while facilitating clear communication and collaboration between UX designers and developers. One possible approach is to develop a structured framework that maps UX design elements (e.g., buttons, forms) from tools like Figma or Sketch to corresponding Gherkin steps. This can be tested by applying the mapping to real-world design projects.

8.5.3 The Integration of Large Language Models (LLMs) with the UI Domain Specific Gherkin

Integrating LLMs with the UI Domain Specific Gherkin could bring several meaningful improvements, making the language more flexible, user-friendly, and efficient. LLMs, like GPT models, can help automate tasks, make the language easier to use, and improve collaboration between design and development teams. Below are some key ways LLMs could impact the UI Gherkin language:

1. **Greater Flexibility with Natural Language:** LLMs excel at understanding natural language, meaning that users could write Gherkin steps in more conversational or flexible ways. Instead of sticking to exact syntax, users could describe actions more naturally, and the LLM would convert them into the appropriate Gherkin steps. For example: a user might write, "Fill out the email text box and click submit", and the LLM could automatically generate:

   When I enter the value "user@example.com" into the text box nearest the label "Email"
   And I click the button labelled "Submit"

   This flexibility would make writing test cases much easier and more intuitive, especially for users who are not developers.

2. **Automatic Scenario Generation:** LLMs could take high level descriptions of how a user should interact with a UI and generate Gherkin scenarios automatically. This would save time by reducing the need for manually writing each step and make it quicker to create new test cases. For example: a user might write, "The user fills out a form with their name, email, and phone number, then submits it", and the LLM could automatically generate:

   Scenario: Submit user information
   When I enter the value "user" into the text box nearest the label "Name"
   And I enter the value "user@example.com" into the text box nearest the label "Email"
   And I enter the value "447760897447" into the text box nearest the label "Phone"
   And I click the button labelled "Submit"
   Then I should see the message "Form submitted successfully"

3. **Mapping Natural Language to Pre-Developed Step Functions:** Since my framework already has pre-developed step functions, LLMs could make it even easier by letting users describe what they want in simple language, and the LLM would map it to the correct step functions. This eliminates the need for memorising exact syntax and reduces errors.

4. **Context-Aware Suggestions:** LLMs can understand the flow of a user interaction and make intelligent suggestions for the next steps based on context. This means that as a user writes a scenario, the LLM can predict what comes next and help complete more complex workflows automatically. For example, after defining the steps for logging in, the LLM might suggest additional steps for navigating to the dashboard or updating profile information. This predictive ability can save time and make writing longer scenarios easier.

The research challenge here is how to effectively integrate LLMs with the UI Domain Specific Gherkin language to automate the generation of flexible, accurate, and context-aware test cases while ensuring that these generated scenarios align with pre-developed step functions and real-world application behaviour. The key challenge lies in maintaining accuracy, reducing ambiguity in natural language inputs, and ensuring that the LLM-generated Gherkin steps correspond to the expected UI interactions. One possible approach is to implement an LLM-integrated version of the UI Domain Specific Gherkin and evaluate its ability to generate accurate and relevant Gherkin scenarios based on natural language inputs. Then conduct experiments with different user groups (technical and non-technical) to assess ease of use, flexibility, and accuracy.

8.5.4 Extend Support to other Web Frameworks

The tool currently offers support for web applications developed using the ReactJS framework and HTML5. Future research should explore the potential necessity of extending support to additional web frameworks. If such a need is identified, the initial step would involve determining the most widely adopted web frameworks within the industry. Subsequently, an analysis of how various user interface elements, such as buttons, text boxes, and other interactive elements, are implemented within these frameworks should be conducted. Based on this analysis, modifications to the cross-browser interactions, specifically within the step functions, should be made to ensure seamless interaction with UI components across the identified frameworks.

To make it even more interesting, research needs to investigate how we can make the approach more robust and less dependent on the framework, integrating AI-driven computer vision tools would be a smart approach. By using AI, we can detect and interact with UI elements based on what appears visually on the screen rather than relying on the underlying framework's structure. This would allow my system to work with different frameworks without requiring significant changes. The following is how this can be achieved:

1. **Using Computer Vision to Detect UI Elements:** Rather than relying on a web application's internal code structure, e.g., DOM elements, we can use computer vision

techniques to visually identify and interact with UI components such as buttons, text boxes, and check boxes. This approach makes the testing independent of the framework used, as the system focuses on what is displayed on the screen. OpenCV and Tesseract OCR are examples of computer vision tools to consider. OpenCV is an open-source computer vision library that enables real-time image processing. It can be used to recognise patterns, such as buttons or form fields, by analysing the visual layout of the screen. Tesseract is a powerful optical character recognition (OCR) tool that can detect and extract text from images. By using this tool, the system can identify UI elements based on visible text labels, making interactions independent of how these elements are coded in frameworks like React, Angular, and Vue.

This can be implemented using the following steps:

(a) Utilise Selenium WebDriver or another web automation tool to capture a screenshot of the current web page.

(b) Apply Tesseract OCR to detect and recognise text elements visible on the UI.

(c) Once text is recognised, use OpenCV to visually locate UI elements near the identified text, such as buttons or input fields.

(d) After identifying the element visually, the system can interact with it, e.g., sending text to a text box or clicking a button, through Selenium or a similar automation tool.

By applying this approach, the framework will focus on the visual layout of the UI, allowing it to operate independently of the underlying framework.

2. **Leveraging AI-Based Object Detection for UI Components:** AI-driven object detection can further enhance the framework-agnostic capabilities of the UI Domain Specific Gherkin by enabling it to recognise UI elements based purely on their visual features. Deep learning models can be trained to detect common UI components (buttons, check boxes, text boxes) by analysing patterns in their appearance. TensorFlow and YOLO (You Only Look Once) are example tools to consider. TensorFlow is a machine learning library that provides pre-trained object detection models, which can be fine-tuned to detect UI elements based on visual data. YOLO is a fast and efficient object detection system capable of identifying multiple objects in real time. It can be trained to recognise various UI components on a screen, enabling interaction based on their visual representation.

   This can be implemented using the following steps:

   (a) Gather a data set of labeled screenshots that contain UI elements, e.g., buttons, and forms, from different web applications developed using various frameworks.

(b) Train an object detection model to recognise specific UI elements, such as buttons, text boxes, and check boxes, based on their visual characteristics.

(c) When testing, the system can capture the screen and pass it through the trained model to detect and locate UI elements.

(d) Once the elements are detected, their coordinates on the web interface can be passed to Selenium to interact with them, e.g., clicking a button or entering text.

This AI-driven method allows the detection of UI elements based purely on their visual appearance, making my framework adaptable to a variety of web development frameworks.

The research challenge here is how to extend the UI Domain Specific Gherkin language to reliably detect and interact with user interface elements across diverse web frameworks (e.g., Angular, Vue) using AI-driven computer vision tools, making the approach framework-independent. How to ensure accurate, consistent, and efficient interaction with UI components, regardless of the underlying web framework, while reducing the dependency on code structure like the DOM.

8.5.5 Expand the UI Domain Specific Gherkin to Accommodate Audio and Gesture-Based Interfaces

A research challenge here is how to extend the UI Domain Specific Gherkin language to effectively support multimodal interactions, such as voice and gesture-based interfaces. This involves integrating speech recognition, speech synthesis, and gesture detection into the Gherkin language in a way that ensures accurate interpretation of user input and seamless interaction across diverse platforms, such as voice-activated systems, Virtual Reality (VR), and Augmented Reality (AR). Additionally, the challenge includes ensuring that the language remains intuitive for both technical and non-technical users, while also validating its effectiveness in real-world environments with different input modalities. Below is a possible scenario for how I might adapt and improve my UI Gherkin to handle audio interfaces. To accommodate this, my UI Gherkin language will need to handle speech recognition (interpreting what users say) and speech synthesis (responding via voice). My UI Gherkin language can be extended to include voice commands and system responses. For example,

When I say “What is the weather today?”
Then the system should respond with “The weather is sunny”

This interaction allows my Gherkin language to work with voice interfaces, opening it up to systems like virtual assistants.

Furthermore, I could expand my language to work with gesture-based interfaces (where actions are controlled by physical movements). For example,

When I swipe left with my hand
Then the system should go to the next page

In this way, my language could handle interactions that do not require traditional mouse or keyboard input, making it suitable for more immersive environments like VR and AR.

8.5.6 Linking Business Domain Requirements to User Interface Domain

A key research challenge here is effectively bridging the gap between abstract business requirements and concrete user interface (UI) design, ensuring that UI elements directly support and reflect business goals while maintaining a seamless user experience. How can a UI Domain Specific Gherkin language be extended to effectively integrate and align with business domain requirements, ensuring both technical accuracy and fulfillment of business objectives? One possible approach is to develop a structured mapping process where each business requirement is translated into user stories or business rules and then into corresponding UI Gherkin steps. Tools like Domain Driven Design (DDD) can assist in modeling the business logic. After that, I implement this approach in a real-world project to evaluate the effectiveness of translating business goals into UI interactions using my UI Gherkin language.

8.6 A Final Thought...

BDD was introduced as an approach to integrate requirements specification with acceptance testing to achieve traceability. However, the literature suggests that this integration has not been fully successful due to the language gap between the high-level business domain and the lower-level user interface domain. Describing tests is generally easier for systems with visible interfaces, while starting with abstract requirements and refining them is more effective because the connection to business value is maintained. This thesis has demonstrated that stakeholders can work within the lower-level domain, but further efforts are required to fully bridge the gap between these domains.

Appendix A

UI Domain Specific Gherkin Steps

The User Interface Domain Specific BDD uses the same underlying syntax as general BDD (https://cucumber.io/docs/gherkin/reference). However, the possible sentences that can be used are restricted to actions that can be applied to a web browser's user interface.

Given Steps (Preconditions)
Given statements are used to configure the setup of the web browser. For example, to define browser version, or render a particular URL.

- **Given I am using the "browser" web browser**
  This step represents configuring the setup of a specified web browser.

  ```
  Given I am using the "Chrome" web browser
  ```

- **Given I have opened the web page "page_url"**
  This step represents rendering a specified URL in the web browser.

  ```
  Given I have opened the web page "https://example.com"
  ```

- **Given I have opened the web page "page_url" in tab "tab number"**
  This step represents rendering a specified URL in a new browser tab.

  ```
  Given I have opened the web page "https://example.com" in tab
  "2"
  ```

When Steps (Actions)
When steps define the action or event that triggers a specific behaviour in the system. These steps describe what happens when a user or system interaction occurs, typically representing an action performed by the user or an event in the system. Elements on the web browser's user interface can be identified based on the visual layout of the screen. For example, you can specify a button to click by identifying a nearby label.

- **When I enter the value “value” into the text box nearest the label “label”**
  This step represents entering a text into a text box that is visually located nearest a specified label.

  ```
  When I enter the value "John Doe" into the text box nearest the label "Full Name"
  ```

- **When I click the button labelled “label”**
  This step represents clicking a button that visually has a label on it.

  ```
  When I click the button labelled "Submit"
  ```

- **When I click the button nearest the label “label”**
  This step represents clicking a button that is visually located nearest a specified label.

  ```
  When I click the button nearest the label "Submit"
  ```

- **When I click the check box button nearest the label “label”**

  ```
  When I click the check box button nearest the label "Remember Me"
  ```

- **When I click the radio button nearest the label “label”**

  ```
  When I click the radio button nearest the label "Male"
  ```

- **When I click the link labelled “label”**

  ```
  When I click the link labelled "Terms and Conditions"
  ```

- **When I select the value “value” from the drop down list nearest the label “label”**
  This step represents selecting a value from a drop down list that is visually located nearest a specified label.

  ```
  When I select the value "United Kingdom" from the drop down list nearest the label "Country"
  ```

- **When I wait for “time_in_seconds” seconds**
  This step represents pausing step execution for a certain amount of time (in seconds) before moving onto the next step.

  ```
  When I wait for "5" seconds
  ```

- **When I switch to tab “tab_number”**
  This step represents switching control to a different tab in the browser by entering the tab number.

```
When I switch to tab "3"
```

- **When I switch to the tab titled “tab_title”**
  This step represents switching control to a different tab in the browser by entering the tab title.

  ```
  When I switch to the tab titled "Home Page"
  ```

- **When I enter the value “value” into the text box above the label “label”**
  This step represents entering a text into a text box that is visually located above a specified label.

  ```
  When I enter the value "John Doe" into the text box above the
  label "Full Name"
  ```

- **When I enter the value “value” into the text box below the label “label”**

  ```
  When I enter the value "John Doe" into the text box below the
  label "Full Name"
  ```

- **When I enter the value “value” into the text box that is both nearest the label “label” and below the label “label”**

  ```
  When I enter the value "John Doe" into the text box that is
  both nearest the label "Full Name" and below the label
  "username"
  ```

- **When I enter the value “value” into the password box nearest the label “label”**
  This step represents entering a text into a password box that is visually located nearest a specified label.

  ```
  When I enter the value "mypassword" into the password box
  nearest the label "password"
  ```

- **When I enter the date “date” into the date picker nearest the label “label”**
  This step represents entering a date into a date picker object that is visually located nearest a specified label.

  ```
  When I enter the date "02132024" into the date picker nearest
  the label "Hire date"
  ```

- **When I click the button above the label “label”**

  ```
  When I click the button above the label "Submit"
  ```

- **When I click the button below the label “label”**

```
When I click the button below the label "Submit"
```

- **When I upload the file “file_path” by clicking the “label” button**
  This step is for uploading a file from a user’s machine by clicking a button with a specified label.

```
When I upload the file "C:\Users\sample.pdf" by clicking the
"Upload Resume" button
```

- **When I select the value “value” from the drop down list below the label “label”**

```
When I select the value "United Kingdom" from the drop down
list below the label "Country"
```

- **When I select the value “value” from the multiple select list nearest the label “label”**

```
When I select the value "United Kingdom" from the multiple
select list nearest the label "Country"
```

- **When I select the values “value” and “value” from the multiple select list nearest the label “label”**

```
When I select the values "Japan" and "United Kingdom" from
the multiple select list nearest the label "Visited
Countries"
```

Then Steps (Assertions)

Then steps define the expected outcome or result after an action has been performed in a Gherkin scenario. These steps are used to verify that the system behaves as expected after executing the When step(s). For example, you can verify that a page redirection occurs as a result of a previous user action.

- **Then I should be taken to the page “page_name”**
  This step represents affirming that a user should be redirected to a specified web page as a result of a previous user action.

```
Then I should be taken to the page "Dashboard"
```

- **Then content appears on the page including the word “phrase”**
  This step is for asserting that a specified content should appear on the web page as a result of a previous user action.

```
Then content appears on the page including the word
"Welcome"
```

- **Then I should be on the page titled “title”**
  This step represents affirming that a user is on a web page that has a specified title as a result of a previous user action.

  ```
  Then I should be on the page titled "User Profile"
  ```

- **Then the check box button nearest the label “label” is selected**
  This step represents affirming that the check box button that resides nearest a specified label is selected.

  ```
  Then the check box button nearest the label "Subscribe
  to Newsletter" is selected
  ```

- **Then the radio button nearest the label “label” is selected**

  ```
  Then the radio button nearest the label "Credit Card"
  is selected
  ```

- **Then the value “value” in the drop down list nearest the label “label” is selected**

  ```
  Then the value "USD" in the drop down list nearest the
  label "Currency" is selected
  ```

- **Then I should see the message “message”**
  This step is for asserting that a specified message should be displayed to the user as a result of a previous user action.

  ```
  Then I should see the message "Registration Successful"
  ```

- **Then the file “filename.extension” is downloaded**
  This step represents affirming that a specified file has been downloaded to the user’s machine an a result to a previous user action in the user interface.

  ```
  Then the file "report.pdf" is downloaded
  ```

- **Then I should not see the message “message”**
  This step is for affirming that a specified message should not be displayed to the user as a result of a previous user action in the user interface.

  ```
  Then I should not see the message "Error"
  ```

Appendix B

Initial Analysis of Professional Literature

B.0.1 General BDD Best Practices

In this article, the author discusses a number of tips that can help you write behaviour driven test cases effectively [81]:

1. Integrate Multiple Viewpoints: Collaborate across teams to gain a far broader understanding of the problem.
2. Early testing of ideas will help you catch issues earlier in the development process.
3. Make tests living documents by keeping them up-to-date.
4. Tests should be designed to run independently so that inter-dependencies are avoided.

In addition, this article provides useful tips for effective Behaviour Driven Development (BDD) [77]:

1. Encourage Communication: Ensure everyone understands the requirements.
2. Structure Scenarios: Follow a clear Context-Action-Outcome format.
3. Write Effective Feature Files: Keep them concise and focused on “what” rather than “how”.
4. Collaborate on Feature Files: Involve developers, quality assurance (QA), and business analysts (BAs) in the process.
5. Store Feature Files Accessibly: Use source control for easy updates.

6. Prefer Declarative Style: Make feature files simple and maintainable.

Similarly, the article by Azevedo [76] on BDD best practices highlights key tips for effective Behaviour Driven Development:

1. Clear Gherkin Syntax: Write steps in present tense for consistency.
2. Organise Feature Files: Keep files properly named and located.
3. Imperative vs. Declarative: Use declarative style for better readability.
4. Scenario Independence: Ensure each scenario can stand alone.
5. Avoid Redundancy: Refactor to remove repetitive steps and keep scenarios concise.

Furthermore, this article on BeyondxScratch provides a comprehensive introduction to behaviour driven development [80]. It emphasises the use of real user scenarios to guide development, the maintenance of clear and consistent language, and the involvement of stakeholders throughout the development process. Also, the article highlights key practices for effective behaviour driven development:

1. Given-When-Then steps should be used to structure scenarios.
2. Negative test cases should be considered.
3. Each scenario should be able to run independently.
4. Redundant steps should be eliminated to keep scenarios concise and effective.

In this article, the author highlights two key software development approaches: BDD and TDD [82]. She discusses key tips for effective Behaviour Driven Development:

1. Make sure scenarios are written in a simple language. Doing so helps make them more readable by everyone involved in software development process, such as quality engineers, business analysts, product owners, and developers.
2. Run each scenario one at a time and only if it is successful move to the next scenario.
3. Refactor your code to improve readability and re-usability.

B.0.2 Gherkin Best Practices

The author discusses in this article the best practices for writing Gherkin features [12]:

1. Avoid writing imperative Given-When-Then steps, instead, write declarative steps.
2. Limit each scenario to a single When-Then pair. This is because each scenario should examine only one individual behaviour.
3. Given-When-Then steps must appear in the correct order, starting with the “Given” step, followed by “When”, and concluded by the “Then” step.
4. Steps should be written in third person perspective rather than first person.
5. Scenario steps should be written in present tense instead of future or past tense.
6. Choose a suitable and meaningful title to describe your scenarios.
7. Avoid writing long scenarios.

Additionally, the author concludes the article by providing 20 tips for better feature style and structure:

1. Customer needs should be the focus of a feature.
2. To make finding features easier, limit them to one per feature file.
3. There should be a maximum of 12 scenarios per feature.
4. Each scenario should contain no more than ten steps.
5. Each step should not exceed 80-120 characters in length.
6. Spelling should be accurate.
7. Grammar should be accurate.
8. Make sure Gherkin keywords are capitalised.
9. Make sure that the first word in scenario titles is capitalised.
10. Capitalise only proper nouns in step phrases.
11. Ending step phrases with punctuation is not necessary.
12. Words should be separated by single spaces.

13. Make sure the content is indented beneath section headers.
14. Two blank lines should be used to separate features from scenarios.
15. A blank line should be placed between each example table.
16. In a scenario, do not use blank lines to separate steps.
17. Ensure that table delimiter pipes are spaced evenly.
18. Avoid duplicate tag names and make sure all tag names are consistent.
19. The names of all tags should be written in lowercase and separated by hyphens.
20. Names for tags should be limited in length.

Furthermore, in this webinar, the presenter demonstrates the best practices for using Gherkin keywords in a feature file through a practical example [79].:

1. Encourage the use of Gherkin keywords “And”, and “But”, when there are successive “Given”, “When”, or “Then” statements in a scenario. This improves scenario readability.
2. Scenarios should run independently without dependencies on other scenarios.
3. Encourage making use of data tables to pass multiple parameters to step definitions in a single Gherkin statement. This improves the clarity of scenarios an helps manage multiple inputs.

Similarly, the author of this article presents the best practices that can help BDD practitioners develop effective Gherkin scenarios [75]:

1. Limit each scenario to one precondition, one action, and one outcome.
2. Use a maximum of two “And” clauses in each scenario.
3. A single scenario should not exceed five steps.
4. Write declarative steps instead of imperative steps.
5. Write steps in the perspective of end user instead of using a first or third-user perspective.

In addition, Knight [83] discusses in this article the steps BDD teams can take to improve their writing skills to make their Gherkin work seamlessly.

1. Write feature files in a way that readers can understand easily, this will improve team collaboration.
2. A single scenario should only examine one behaviour.
3. BDD teams should only test the unique aspects of the software.
4. Gherkin features should be written in proper grammar as this will make behaviour scenarios readable and expressive.

Furthermore, in this workshop, Thorn [74] presents the best practices for writing BDD features:

1. Behaviour scenarios should be written in a declarative format rather than imperative.
2. BDD practitioners should shorten scenarios and remove unnecessary steps.
3. Each scenario should only test a single behaviour and be limited to one pre-condition step and one outcome step.
4. Each scenario should run independently without dependencies on other scenarios in the feature file.
5. Similar and related scenarios should be organised and grouped into a feature.
6. It is critical to include both happy and unhappy scenarios in your tests.
7. Configuration data should be placed in a configuration file rather than hard-coding them.
8. Write your scenarios in a clear and understandable language, and avoid technical terms.

Additionally, in this article, Hamilton [78] discusses Gherkin language in general and highlights its best practices:

1. Each feature should run independently without being coupled with other features and each scenario should execute independently without dependencies on other scenarios.
2. Behaviour scenarios should be linked to the software requirements.
3. Everybody involved in writing scenarios should write clear and understandable steps.
4. Combine the common scenarios in your feature into one scenario.
5. Create a requirement document that keeps a record of all the scenarios included in the acceptance criteria and monitor this document.

B.0.3 Cucumber BDD Best Practices

In this article, Shah [73] discusses BDD best practices. The author divides this guidance into 4 categories: feature files, background, scenarios and steps, and tags. He starts by discussing two recommendations regarding feature files. First, avoid lengthy feature descriptions. Second, follow a unified format for all features, as this will keep them interesting to read. The author then lists his recommendations for writing "background" in a feature file:

1. It should be short.
2. Avoid technical terms in the background, as this will make it easier for users to remember its details when reading feature scenarios.
3. It is critical to remember that tags can be used with features and scenarios but not with backgrounds.
4. It is also vital to refrain from using backgrounds for features that only have one scenario. Instead, include the background steps in the "Given" clause of the scenario.
5. Background and before hooks should not be included in the same feature file. This is because context duplication might happen when they are both present in the same feature file.

The author then moves to discuss his recommendations regarding scenarios and steps.

1. If a scenario has multiple cases to address, use the "Scenario Outline" keyword. Otherwise, use the regular "Scenario" keyword if there is only one case. Having a well-organised and consistent scenario outline will keep your feature more manageable.
2. Avoid long scenarios and keep them short, as doing this will make them more resilient, and easier to read.
3. Ensure that the Given-When-Then steps appear in the correct order. Following the right order makes scenarios easier to read for users.
4. Avoid writing imperative Given-When-Then steps, instead, write declarative steps. This is because imperative steps show unnecessary details about the software.
5. Although it is tempting to cover many cases at once when developing a scenario outline, we should only cover the most useful cases.
6. Keep your scenarios independent, as doing so will speed up debugging.
7. Use the keyword "And" when combining two Gherkin statements.

8. Scenarios should address both positive and negative results.
9. Lastly, consider reusing and refactoring your step definitions.

After that, the author moves to discuss the best practices for using tags in feature files. Tags are used in BDD to organise features and scenarios.

1. Choose smart names for your scenarios. Doing so will help you manage and filter your scenarios easily.
2. You can tag an entire feature if you find that helpful.
3. A scenario can inherit the feature tag, therefore, there is no need to tag a scenario with the same tag as the feature.
4. Remove the work in progress tag from completed scenarios.

Furthermore, in this article on using Cucumber in Selenium automation, Paul [84] discusses best practices for using Cucumber in Selenium automation:

1. Writing Feature Files: Keep them concise and focused.
2. Organising Files: Separate features into specific directories.
3. Maintaining Perspective: Use a consistent point of view, e.g. first or third person perspective.
4. Using Keywords: Effectively use Background, Scenario Outline, Doc Strings, Data Table, Languages, and Tags.
5. Creating Step Definitions: Make them reusable and maintainable.
6. Leveraging Cloud Testing: Utilise platforms like LambdaTest for scalable testing.

Similarly, this article describes the best practices for developing BDD scenarios in Cucumber Studio [85].

1. Before you start developing test code, make sure you develop BDD scenarios that help you understand the software behaviour. These scenarios will help you identify potential issues and improve code quality. Once the scenarios are ready, you can implement them.
2. Instead of writing test scenarios in isolation, work as a team. Doing so will ensure that team members can easily input their ideas and suggestions.

3. When you write test scenarios, you should consider making them clear and focused on one feature at a time.

4. One factor to consider when creating feature files is keeping a sufficient number of scenarios in them. Doing so will make it easier to understand how a feature works. You can also use scenario outlines to check the various cases related to the feature.

5. Before you create the software product, remove unnecessary test scenarios. Doing so will help reduce the number of tests written.

6. A crucial factor to consider when developing test scenarios is focusing on the software core. Doing so will reduce the number of scenarios that contain unnecessary details.

7. Another critical factor to consider when developing scenarios is to avoid developing too abstract scenarios. This will make them challenging to understand.

8. Avoid writing very imperative scenarios because this will distract you from the core idea and lead misleading tests.

9. You should not have too many “And” steps in a single scenario. Write no more than three “And” steps for each “Given”,“When” or “Then” statement. Doing so will minimise scenario complexity.

10. Steps should be written in a third-person perspective. This will help provide more structured information.

11. Consider writing descriptive titles that accurately describe the scenarios.

12. Consider reusing your steps because it will keep the tests easier to maintain when the application changes.

13. Use scenario outlines in moderation. This will prevent tests from running longer, and minimise the amount of similar tests written.

14. Although test automation tools do not distinguish between “Given”,“When” or “Then” statements, they exist to help software teams develop logically structured test scenarios. Therefore, use them wisely.

15. Having clear test scenarios is a critical factor to consider when writing test scenarios.

The authors then moves to recommend BDD practitioners to adopt writing rules if multiple team members participate in writing test scenarios. The authors recommend the following writing rules:

1. Spelling and grammar should be correct.
2. You should use correct punctuation, but do not use periods, commas, or other punctuation marks to end step definitions.
3. Avoid using technical terms.
4. Separate scenarios from each other using empty lines, and separate examples from scenarios using empty lines. Test steps should not be separated by empty lines.
5. Make sure Gherkin keywords are capitalised.
6. Capitalise only proper nouns in step phrases.
7. Make sure tag names are in lower case and separated by hyphens.

In this webinar, Sheffer [72] discusses Cucumber BDD best practices:

1. Write Given-When-Then steps in the right order, as doing so promotes readability.
2. Scenario steps should run without dependencies on other scenarios
3. Features should execute independently without dependencies on other features.
4. Scenarios should be enumerated, as this can speed up debugging.
5. Make sure your scenarios are connected to your system requirements. This will help you ensure requirements are covered.
6. Utilise "Scenario Outline" keyword if most of the scenario steps are the same. This improves scenario readability and maintainability.
7. Write declarative scenario steps instead of low-level imperative steps. This is because declarative steps are more readable than imperative steps, and they hide unnecessary details.
8. Add comments to your scenarios if they contain logic that might be difficult to comprehend by other team members.

The presenter then moves to discuss tips for formatting feature files and scenarios:

1. Keep your tables organised and consistent.
2. Add a colon to the end of each step if you are using a step table.
3. Keep your language consistent.

4. Parameters should be used and explained clearly.
5. Write scenario steps in lowercase.
6. Place general comments at the top of your scenario or feature, and place other comments in the appropriate places.
7. Ensure that your steps are concise and to the point.
8. Scenario steps should not exceed 100 characters, and ideally should be between 50 and 80 characters long.

Furthermore, the authors of this article share the best practices of implementing Cucumber framework and Gherkin in BDD projects [71]:

1. Write abstract scenarios rather than concrete ones.
2. Use first or third-person point of view when developing scenarios.
3. Scenarios should be independent.
4. Use “And” clause if there are successive “Given”, “When”, or “Then” steps in a scenario.
5. Given-When-Then steps should be written in the right order.
6. Use “Background” keyword to avoid repeating scenarios’ prerequisite steps.
7. Write short and clear scenarios.
8. If the “Background” is very long, it will be difficult to understand the scenarios that follow. Therefore, make it as short as possible.
9. Develop step definitions that can be reused in many scenarios and only need to be implemented once.
10. There should be consistency between scenario steps and scenario title.

Similarly, the author of this article demonstrates the top five best practices of the Cucumber framework [70]:

1. Write high-level declarative steps instead of imperative ones. Doing so helps scenarios become clear and maintainable.

2. Write a narrative for each feature because narratives help readers understand the main idea behind features.
3. Use “And” clause in Gherkin instead of creating a conjunctive action step. Doing so promotes step re-usability.
4. Consider reusing and refactoring your step definitions as often as possible, as this improves software maintainability.
5. Avoid repeating the prerequisite steps required to run feature scenarios by using the “Background” keyword.

Additionally, Cucumber’s best practices are discussed in this article [69]:

1. Write declarative steps instead of imperative steps. Using the declarative style in writing tests is more beneficial for scenario writers, and testers in the long run.
2. Tests that have already been performed to test a particular functionality should not be repeated.
3. Each scenario should not test more than one action at a time. In other words, each scenario should have only one “When” step.
4. Use “Background” keyword in Gherkin to avoid repeating prerequisite steps.
5. Implement “Given” steps to prepare your system for performing the action you desire to test.
6. The readability of a scenario can be greatly improved by combining multiple “Given” steps into one single step.

Furthermore, Hartill [68] presents the best practices of Cucumber framework from his BDD experience:

1. Write test descriptions, as having clear comments and descriptive test names will make it easier to troubleshoot a failing test.
2. limit each scenario to one precondition step, one action step, and one assertion step. This is because having more than 3 steps might indicate that you are testing more than you should in one scenario.
3. Use tags to organise your scenarios.
4. write declarative scenarios and avoid imperative ones. This is because readers of your scenarios need to know what you are doing, but not how you are doing it.

5. You should not have many scenarios in a feature file. Each scenario should test the feature itself and not others. Therefore, a feature file should not contain multiple features.

6. Keep only useful scenarios when maintaining and reviewing your test suite.

7. Your scenarios should not depend on each other. This is because it is much harder to find the actual problem when one test fails.

Additionally, in this article, the authors discuss Gherkin language best practices in Cucumber framework [67]:

1. Avoid writing scenarios in imperative style. Writing behaviour scenarios in this style tends to lengthen them, add unnecessary complexity, make them harder to read and understand, and make them harder to maintain.

2. Given-When-Then steps must appear in the correct order starting with the setup step (Given), followed by the action step (When), and concluded by the assertion step (Then).

3. Limit each scenario to a single “Given”, “When”, and “Then” step. If you need to add more steps, this indicates that you are testing multiple behaviours, which is not advisable.

4. It is critical that the sentences are brief and consistent with each other, and consistent with the scenario description.

5. Behaviour scenarios must be independent of one another, as coupled scenarios may fail when one fails.

6. Similar and related scenarios should be organised and grouped under the same feature.

7. Use “Background” keyword if setup steps must be performed before each scenario in a feature. It is much more practical to use “Background” instead of repeating the same steps over and over again.

8. keep the background brief.

9. utilise “Scenario Outline” when you have a data set, instead of developing individual scenarios for each input to the data set.

Appendix C

Laboratory Study: Business Domain Gherkin Version

Participant information sheet

1. Study title

Comprehending and documenting software features in business & user interface Behaviour Driven Development (BDD).

2. Invitation paragraph

You are being invited to take part in a research study concerning the documentation of software features using the Gherkin specification language, which is commonly used as part of the Behaviour Driven Development (BDD) practice. Gherkin is a structured natural language for describing interactions with a software system.

The experiment will be conducted through a questionnaire. The experiment begins with a short tutorial on Gherkin. You will then be presented with descriptions of software features written in Gherkin. After you complete reading the descriptions, you will be asked questions to test how effective the descriptions are in communicating the features of the software system. Note that the questions asked are intended to test the Gherkin language specifications, not the performance of the participant.

Before you decide it is important for you to understand why the research is being done and what it will involve. Please take time to read the following information carefully and discuss it with others if you wish. Ask us if there is anything that is not clear or if you would like more information. Take time to decide whether or not you wish to take part.

3. What is the purpose of the study?

Gherkin specifications can be written from either the perspective of the high level business domain concepts, or by describing actions taken on a specific user interface. The focus of this

study is to examine which approach is more effective for documenting and comprehending software specifications.

4. Why have I been chosen?

Any adult can participate in this study. You do not need to have a software development background to participate in this study.

5. Do I have to take part?

It is up to you to decide whether or not to take part. If you decide to take part, you are still free to withdraw at any time and without giving a reason. Information like industrial experience and qualification are optional and you can choose to not provide any information that you do not want to.

6. What will happen to me if I take part?

This study will increase your knowledge about software development methodologies and the Gherkin language that are used to document software features. This study will take place in multiple sessions (one per participant) and each session will be of approximately 30-40 minutes.

7. What do I have to do?

You have to answer questions about software features written in the Gherkin language.

8. What are the possible disadvantages and risks of taking part?

There is no physical or mental risk involved.

9. What are the possible benefits of taking part?

All participants will have the opportunity to participate in a prize draw for a £200 voucher for participating in this study. The information that is collected during this study will give us a better understanding of which method participants find easier for comprehending and documenting software features.

10. Will my taking part in this study be kept confidential?

You can choose to provide or not provide information like qualification and years of software development experience. The results of the study may be published later but general information (if any) will be anonymized. Please note that assurances on confidentiality will be strictly adhered to unless evidence of serious harm, or risk of serious harm, is uncovered. In such cases the University may be obliged to contact relevant statutory bodies/agencies.

11. What will happen to the results of the research study?

Results will likely be published after the analysis of results of the study. You will not be identified in any report/publication. You can request a copy of the publication on the condition that you will use it for academic purposes.

12. Who is organising and funding the research?

The research is organised by the University of Glasgow. The Saudi Arabian Cultural Attache in London is funding this research.

13. Who has reviewed the study?

The project has been reviewed by the College of Science and Engineering Ethics Committee in the University of Glasgow.

14. Contact for further information

My email address is a.alshammari.2@research.gla.ac.uk , Room F151 SAWB (Computing Science building) 18 Lilybank Gardens, Glasgow G12 8RZ.

15. Thank you very much.

Consent

- I agree to take part as a participant in this questionnaire.
- I understand to my satisfaction the information regarding participation in the research project.
- I understand that I am free to not answer specific items or questions in the questionnaire.
- I understand that I am free to withdraw from the study at any time without penalty.
- I understand that my continued participation should be as informed as my initial consent, so I should feel free to ask for clarification or new information throughout my participation.
- I understand that the material may be used in future publications, both print and online.

Demographics

1. Enter your years of experience in the software industry.
2. Enter your current job title, for example ”software engineer”, ”business analyst” or ”Chief Technology Officer”.

This is an experiment to test the readability and maintainability of behaviour driven development requirements specifications.

Phase I: BDD Training

What is Behaviour Driven Development?

Behaviour Driven Development is a requirements specification technique that allows you to describe a user's interactions with a software system in a structured natural language, called Gherkin. The Gherkin language organises requirements into modular features, with each feature comprising one or more scenarios describing an interaction with the software application under test. The descriptions of interactions can then be linked to the actual system itself through executable tests. This linking is supported by BDD frameworks, such as Cucumber, that match individual steps in a test scenario to a program method or function for execution. Therefore, a test is expressed as a sequence of Gherkin step/program method combinations.

The BDD workflow is therefore as follows:

Here is an example of a BDD Feature describing the management of a shopping cart for online purchase:

Feature: Shopping Cart Management
As a customer
I want to manage my shopping cart
So that I can checkout

```
Scenario: Add a new product to my cart
Given I am on ''myfavonlineshop.com'' login page
And I login with the username ''shop@gmail.com'' and password
    ''letmein''
When I add the product ''smart tv'' to the cart
Then the product ''smart tv'' should be added to the cart

Scenario: Add new products to my cart
Given I am on ''myfavonlineshop.com'' login page
And I login with the username ''shop@gmail.com'' and password
    ''letmein''
When I add the following products to the cart
    | smart tv    |
    | smart phone |
    | smart watch |
```

```
Then the following products should be added to the cart
    | smart tv    |
    | smart phone |
    | smart watch |

Scenario:
Given I am on ''myfavonlineshop.com'' login page
And I login with the username ''shop@gmail.com'' and password
    ''letmein''
When I delete all the products in my cart
Then my cart should be empty
```

In the example above, we have a customer wanting to manage their shopping cart on their favorite online shop. The feature file begins with an outline description of the feature.

As a identifies the actor which is a customer in our case.

I want to represents the general action that they want to perform with the feature, and in this example it is managing the customer's cart.

So that describes the desired outcomes of the feature.

The feature's outline description is useful in documenting the purpose of the tests that are described in the scenarios below it. Each feature has a series of scenarios associated with it. Each scenario describes an alternative sequence through this feature. Each scenario has a title that describes its purpose. **Given-When-Then** is a style of representing tests.

Given represents the pre-conditions of the tests.

When describes the action that the actor in the system performs.

Then describes the expected outcome of that action.

Now let's discuss each scenario in the previously discussed feature file:

The first scenario is about adding new products to the customer's shopping cart. In the first two 'Given' statements of the scenario, we are laying the ground for our when statement. So, we are loading the login page of myfavonlineshop.com , and then using the customer's credentials to login. The following 'When' statement represents the action that we want to

test, which is adding the smart tv product to the customer's cart. Then, the final statement in the scenario, which is asserting the action of adding the product to the cart.

What if we want to add multiple products to the customer's cart? We have done this in the second scenario. We listed the desired products in a table below the steps 'When' and 'Then'. Note that each row in the table will be executed as a separate step.

We are now going to show you an example of a question we will ask in the next section.
Exercise Consider the third scenario. Can you propose a title for the scenario, based on the described behaviour and title of the other scenarios?
Answer: Empty my cart.

This brings us to the end of our BDD introduction and training on how to document systems through features and scenarios.

Phase II:
Now that you know the workflow of BDD, let us explore another example. **Prestashop** is an e-commerce solution that helps their customers in creating online stores. They provide a demo shop for potential customers to explore their platform. Click on the following url to view Prestashop's demo shop:

`https://demo.prestashop.com/#/en/back`

Log in to the shop using the provided credentials. Now that you have logged in, take a few minutes to familiarise yourself with the user interface. Here is a list of suggestions (use the menu in the left side of the screen):

- Create a new category (Catalog - Categories - Add new category).
- Create a new product (Catalog - Products - New product).
- Edit an existing product (Catalog - Products - Edit the product you wish by clicking on the pencil icon).
- Create a new customer (Customers - Customers - Add new customer).

Note that you do not have to do them all. Doing two activities would suffice. The point here is to familiarise yourself with the user interface.

Phase III:

In phase I we learned how to document a system in Gherkin features and Scenarios. Furthermore, in phase II we explored Prestashop's user interface. In this phase, you will read 5 untitled scenarios and then select a title from the multiple choice list that best describes each scenario's overall purpose.

```
1. Scenario:
Given I have an empty default cart
And there is a product in the catalog named ''product1'' with a
    price of 19.812 and 1000 items in stock
Then the remaining available stock for product ''product1'' should
     be 1000
When I add 11 items of product ''product1'' in my cart
Then my cart should contain 11 units of product ''product1'',
     excluding items in pack
And the remaining available stock for product ''product1'' should
    be 989
```

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Add product(s) in cart

(b) Update product

(c) Delete product

(d) Disable product

```
2. Scenario:
Given I add product ''product1'' with following information:
      | name[en-US] | Presta camera |
      | type        | standard      |
And product ''product1'' should have following details:
      | product detail | value |
      | isbn           |       |
      | upc            |       |
      | ean13          |       |
      | mpn            |       |
      | reference      |       |
When I update product ''product1'' details with following values:
      | isbn      | 978-3-16-148410-0 |
      | upc       | 72527273070       |
      | ean13     | 978020137962      |
```

```
      | mpn       | mpn1              |
      | reference | ref1              |
Then product ''product1'' should have following details:
      | product detail | value             |
      | isbn           | 978-3-16-148410-0 |
      | upc            | 72527273070       |
      | ean13          | 978020137962      |
      | mpn            | mpn1              |
      | reference      | ref1              |
```

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Add new category

(b) Update product

(c) Delete product

(d) Disable product

```
3. Scenario:
Given I have an empty default cart
And there is a product in the catalog named ''product1'' with a
    price of 19.812 and 1000 items in stock
Then I am not allowed to add 1100 items of product ''product1'' in
     my cart
And my cart should contain 0 units of product ''product1'',
    excluding items in pack
And the remaining available stock for product ''product1'' should
    be 1000
```

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Update shipping details

(b) Can not add product in cart with quantity exceeding availability

(c) Delete product from category

(d) Move product to different category

```
4. Scenario:
Given there is customer ''testCustomer'' with email
      ''pub@prestashop.com''
```

And I create an empty cart ‘‘dummy_cart’’ for customer ‘‘testCustomer’’
And there is a product in the catalog named ‘‘product1’’ with a price of 19.812 and 1000 items in stock
And the product ‘‘product1’’ minimal quantity is 10
When I add 5 products ‘‘product1’’ to the cart ‘‘dummy_cart’’
Then I should get error that minimum quantity of 10 must be added to cart
And my cart should contain 0 units of product ‘‘product1’’, excluding items in pack
And the remaining available stock for product ‘‘product1’’ should be 1000

Select a title from the multiple choice list that best describes the scenario’s overall purpose.

(a) Can not add product in cart with minimal quantity enabled

(b) Delete product from category

(c) Update customer’s address

(d) Move product to different category

5. **Scenario:**
Given order out of stock products is allowed
And I have an empty default cart
And there is a product in the catalog named ‘‘product1’’ with a price of 19.812 and 1000 items in stock
When I add 1100 items of product ‘‘product1’’ in my cart
Then my cart should contain 1100 units of product ‘‘product1’’, excluding items in pack
And the remaining available stock for product ‘‘product1’’ should be -100

Select a title from the multiple choice list that best describes the scenario’s overall purpose.

(a) Can not add product in cart with minimal quantity enabled

(b) Be able to add out of stock product if configuration allows it

(c) Update customer’s credit card

(d) Delete product from category

Phase IV: Final phase

In phase I we learned how to document a system in Gherkin features and scenarios. Furthermore, in phase II we explored Prestashop's user interface. Also, in phase III, we had a chance to read untitled scenarios and then give a title for each scenario that indicates its overall purpose. In this final phase, you will read 5 scenarios, and then you will be asked to choose from the multiple choice list a step function that matches the desired behaviour of the "Then" step(s). If you think no step function matches the desired behaviour of the "Then" step(s), then choose "A new function is needed".

Assume that you have the following steps that run before each of the 5 scenarios. These steps are grouped in a background.

```
Background: Adding new Category

Given I add new category ''category1'' with following details:
        | Name                | PC parts          |
        | Displayed           | false             |
        | Parent category     | Home Accessories  |
        | Friendly URL        | pc-parts          |

1- Scenario: Delete category
When I delete category ''category1'' choosing mode ''associate_and_
     disable''
Then
```

Choose from the multiple choice list a step function that matches the desired behaviour of the "Then" step.

(a)
```
@Then category :catReference does not have menu thumbnail image
public function catDoesNotHaveMenuThumbImage(string $catReference){
$editableCategory=$this-getEditableCategory($catReference);
$menuThumbImages=$editableCategory-getMenuThumbImages();
Assert::assertCount(0, $menuThumbImages);
}
```

(b)
```
@Then category :catReference does not exist
public function categoryDoesNotExist(string $catReference){
$categoryId=SharedStorage::getStorage()-get($catReference);
try {
$this-getQueryBus()-handle(new
GetCategoryForEditing($categoryId));
} catch (CategoryNotFoundException $e) {
```

```
return;
}
throw new RuntimeException(sprintf(''Category %s exists'',
$catReference));
}
```

(c)
```
@Then category :catReference is enabled
public function categoryIsEnabled(string $catReference){
   $categoryIsEnabled=$this-getCategoryIsEnabled($catReference);
   Assert::assertTrue($categoryIsEnabled);
}
```

(d)
```
@Then category :catReference does not have cover image
public function categoryDoesNotHaveCoverImage(string $catReference){
   $editableCategory=$this-getEditableCategory($catReference);
   $coverImage=$editableCategory-getCoverImage();
   ASSERT::assertNull($coverImage);
}
```

(e) A new function is needed

2. **Scenario**: enable category
Given category ''category1'' is disabled
When I enable category ''category1''
Then

Choose from the multiple choice list a step function that matches the desired behaviour of the "Then" step.

(a)
```
@Then category :catReference is enabled
public function categoryIsEnabled(string $catReference){
   $categoryIsEnabled=$this-getCategoryIsEnabled($catReference);
   Assert::assertTrue($categoryIsEnabled);
}
```

(b)
```
@Then category :catReference does not have menu thumbnail image
public function catDoesNotHaveMenuThumbImage(string $catReference){
   $editableCategory=$this-getEditableCategory($catReference);
   $menuThumbnailImages=$editableCategory-getMenuThumbnailImages();
   Assert::assertCount(0, $menuThumbnailImages);
}
```

(c)
```
@Then category :catReference is disabled
public function categoryIsDisabled(string $catReference): void {
    $categoryIsEnabled=$this->getCategoryIsEnabled($catReference);
    Assert::assertFalse($categoryIsEnabled);
}
```

(d)
```
@Then category :catReference does not have cover image
public function categoryDoesNotHaveCoverImage(string $catReference){
    $editableCategory=$this->getEditableCategory($catReference);
    $coverImage=$editableCategory->getCoverImage();
    ASSERT::assertNull($coverImage);
}
```

(e) A new function is needed

```
3. Scenario: delete category menu thumbnail imag
Given I edit category ``category1'' with following details:
      | Name                | dummy category name    |
      | Displayed           | false                  |
      | Parent category     | Home Accessories       |
      | Description         | dummy description      |
      | Meta title          | dummy meta title       |
      | Meta description    | dummy meta description |
      | Friendly URL        | dummy                  |
      | Group access        | Visitor,Guest,Customer |
      | Category cover image | logo.jpg              |

And category ``category1'' has menu thumbnail image
When I delete category ``category1'' menu thumbnail image
Then
```

Choose from the multiple choice list a step function that matches the desired behaviour of the “Then” step.

(a)
```
@Then I should see following root categories in ``:langIso'' language
public function assertRootCategoriesTree(TableNode $tableNode, string
        $langIso): void{
    $langId=Language::getIdByIso($langIso);
    $categoriesTree = $this->getQueryBus()->handle(new
    GetCategoriesTree($langId));
    Assert::assertNotEmpty($categoriesTree,
```

```
        ''Categories tree is empty'');
        $this-assertCategoriesInTree($categoriesTree, $tableNode-
        getColumnsHash(), $langId);
    }
```

(b)
```
@Then category :catReference does not have cover image
    public function categoryDoesNotHaveCoverImage(string $catReference){
        $editableCategory=$this-getEditableCategory($categoryReference);
        $coverImage=$editableCategory-getCoverImage();
        ASSERT::assertNull($coverImage);
    }
```

(c)
```
@Then I should get error that customer was not found
    public function assertCustomerNotFound(): void{
        $this-assertLastErrorIs(CustomerNotFoundException::class);
    }
```

(d)
```
@Then I should get error that payment method is invalid
    public function assertLastErrorIsInvalidPaymentMethod(): void{
        $this-assertLastErrorIs(OrderConstraintException::class,
        OrderConstraintException::INVALID_PAYMENT_METHOD);
    }
```

(e) A new function is needed

```
4- Scenario: bulk disable selected categories
Given I add new category ''category2'' with following details:
      | Name                | PC parts 2       |
      | Displayed           | true             |
      | Parent category     | Home Accessories |
      | Friendly URL        | pc-parts2        |

When I bulk disable categories ''category1,category2''
Then
And
```

Choose from the multiple choice list a step function that matches the desired behaviour of the "Then" step.

(a)
```
@Then category :catReference does not have menu thumbnail image
    public function catDoesNotHaveMenuThumbImage(string $catReference){
```

```
        $editableCategory=$this-getEditableCategory($catReference);
        $menuThumbnailImages=$editableCategory-getMenuThumbnailImages();
        Assert::assertCount(0, $menuThumbnailImages);
    }
```

(b)
```
@Then category :catReference is enabled
    public function categoryIsEnabled(string $catReference){
        $categoryIsEnabled=$this-getCategoryIsEnabled($catReference);
        Assert::assertTrue($categoryIsEnabled);
    }
```

(c)
```
@Then category :catReference is disabled
    public function categoryIsDisabled(string $catReference): void{
        $categoryIsEnabled=$this-getCategoryIsEnabled($catReference);
        Assert::assertFalse($categoryIsEnabled);
    }
```

(d)
```
@Then the logo size configuration should be :width x :height
    public function logoSizeConfigurationShouldbe(int $width,
            int $height): void{
        $confWidth=(int)Configuration::get(''SHOP_LOGO_WIDTH'');
        $confHeight=(int)Configuration::get(''SHOP_LOGO_HEIGHT'');
        if ($confWidth!==$width) {
     throw new RuntimeException(''Width does not match'');
        }
        if ($confHeight!==$height) {
          throw new RuntimeException(''Height does not match'');
        }
    }
```

(e) A new function is needed

5. **Scenario:** delete category cover image

Given I edit category ''category1'' with following details:

Name	dummy category name
Displayed	false
Parent category	Home Accessories
Description	dummy description
Meta title	dummy meta title
Meta description	dummy meta description
Friendly URL	dummy

```
    | Group access          | Visitor,Guest,Customer |
    | Category cover image | logo.jpg                |
```

And category ``category1'' has cover image
When I delete category ``category1'' cover image
Then

Choose from the multiple choice list a step function that matches the desired behaviour of the "Then" step.

(a)
```
@Then category :catReference is disabled
public function categoryIsDisabled(string $catReference): void{
    $categoryIsEnabled=$this-getCategoryIsEnabled($catReference);
    Assert::assertFalse($categoryIsEnabled);
}
```

(b)
```
@Then category :catReference does not have menu thumbnail image
public function catDoesNotHaveMenuThumbImage(string $catReference){
    $editableCategory=$this-getEditableCategory($categoryReference);
    $menuThumbnailImages=$editableCategory-getMenuThumbnailImages();
    Assert::assertCount(0, $menuThumbnailImages);
}
```

(c)
```
@Then category :catReference is enabled
public function categoryIsEnabled(string $catReference){
    $categoryIsEnabled=$this-getCategoryIsEnabled($catReference);
    Assert::assertTrue($categoryIsEnabled);
}
```

(d)
```
@Then I should get error that payment method is invalid
public function assertLastErrorIsInvalidPaymentMethod(): void{
    $thisassertLastErrorIs( OrderConstraintException::class,
    OrderConstraintException::INVALID_PAYMENT_METHOD);
}
```

(e) A new function is needed

Final words

- Please enter your email below if you would like to enter the prize draw for a chance to win £200.
- Please enter your email below if you would like to receive updates about the study.
- Please enter your comments about the study if you have any.
- End of study. Thank you for taking part in this study!

Appendix D

Laboratory Study: User Interface Domain Specific Gherkin Version

Participant information sheet

1. Study title

Comprehending and documenting software features in business & user interface Behaviour Driven Development (BDD).

2. Invitation paragraph

You are being invited to take part in a research study concerning the documentation of software features using the Gherkin specification language, which is commonly used as part of the Behaviour Driven Development (BDD) practice. Gherkin is a structured natural language for describing interactions with a software system.

The experiment will be conducted through a questionnaire. The experiment begins with a short tutorial on Gherkin. You will then be presented with descriptions of software features written in Gherkin. After you complete reading the descriptions, you will be asked questions to test how effective the descriptions are in communicating the features of the software system. Note that the questions asked are intended to test the Gherkin language specifications, not the performance of the participant.

Before you decide it is important for you to understand why the research is being done and what it will involve. Please take time to read the following information carefully and discuss it with others if you wish. Ask us if there is anything that is not clear or if you would like more information. Take time to decide whether or not you wish to take part.

3. What is the purpose of the study?

Gherkin specifications can be written from either the perspective of the high level business domain concepts, or by describing actions taken on a specific user interface. The focus of this

study is to examine which approach is more effective for documenting and comprehending software specifications.

4. Why have I been chosen?

Any adult can participate in this study. You do not need to have a software development background to participate in this study.

5. Do I have to take part?

It is up to you to decide whether or not to take part. If you decide to take part, you are still free to withdraw at any time and without giving a reason. Information like industrial experience and qualification are optional and you can choose to not provide any information that you do not want to.

6. What will happen to me if I take part?

This study will increase your knowledge about software development methodologies and the Gherkin language that are used to document software features. This study will take place in multiple sessions (one per participant) and each session will be of approximately 30-40 minutes.

7. What do I have to do?

You have to answer questions about software features written in the Gherkin language.

8. What are the possible disadvantages and risks of taking part?

There is no physical or mental risk involved.

9. What are the possible benefits of taking part?

All participants will have the opportunity to participate in a prize draw for a £200 voucher for participating in this study. The information that is collected during this study will give us a better understanding of which method participants find easier for comprehending and documenting software features.

10. Will my taking part in this study be kept confidential?

You can choose to provide or not provide information like qualification and years of software development experience. The results of the study may be published later but general information (if any) will be anonymized. Please note that assurances on confidentiality will be strictly adhered to unless evidence of serious harm, or risk of serious harm, is uncovered. In such cases the University may be obliged to contact relevant statutory bodies/agencies.

11. What will happen to the results of the research study?

Results will likely be published after the analysis of results of the study. You will not be identified in any report/publication. You can request a copy of the publication on the condition that you will use it for academic purposes.

12. Who is organising and funding the research?

The research is organised by the University of Glasgow. The Saudi Arabian Cultural Attache in London is funding this research.

13. Who has reviewed the study?

The project has been reviewed by the College of Science and Engineering Ethics Committee in the University of Glasgow.

14. Contact for further information

My email address is a.alshammari.2@research.gla.ac.uk , Room F151 SAWB (Computing Science building) 18 Lilybank Gardens, Glasgow G12 8RZ.

15. Thank you very much.

Consent

- I agree to take part as a participant in this questionnaire.
- I understand to my satisfaction the information regarding participation in the research project.
- I understand that I am free to not answer specific items or questions in the questionnaire.
- I understand that I am free to withdraw from the study at any time without penalty.
- I understand that my continued participation should be as informed as my initial consent, so I should feel free to ask for clarification or new information throughout my participation.
- I understand that the material may be used in future publications, both print and online.

Demographics

1. Enter your years of experience in the software industry.
2. Enter your current job title, for example ”software engineer”, ”business analyst” or ”Chief Technology Officer”.

This is an experiment to test the readability and maintainability of behaviour driven development requirements specifications.

Phase I: BDD Training

What is Behaviour Driven Development?

Behaviour Driven Development is a requirements specification technique that allows you to describe a user's interactions with a software system in a structured natural language, called Gherkin. The Gherkin language organises requirements into modular features, with each feature comprising one or more scenarios describing an interaction with the software application under test. The descriptions of interactions can then be linked to the actual system itself through executable tests. This linking is supported by BDD frameworks, such as Cucumber, that match individual steps in a test scenario to a program method or function for execution. Therefore, a test is expressed as a sequence of Gherkin step/program method combinations.

The BDD workflow is therefore as follows:

Here is an example of a BDD Feature describing the management of a shopping cart for online purchase:

Feature: Shopping Cart Management
As a customer
I want to manage my shopping cart
So that I can checkout

```
Scenario: Add a new product to my cart
Given I am using the ''chrome'' web browser
And I have opened the web page ''https://myfavonlineshop.com/login''
And I enter the value ''shop@gmail.com'' into the text box nearest
    the label ''username''
And I enter the value ''letmein'' into the password box nearest
    the label ''password''
And I click the button labelled ''Login''
When I click the link labelled ''All products''
And I click the link labelled ''smart TV''
And I click the button labelled ''Add to cart''
Then I should see the message ''Product has been added to cart!''

Scenario: Edit product quantity in my cart
Given I am using the ''chrome'' web browser
```

```
And I have opened the web page ``https://myfavonlineshop.com/login''
And I enter the value ``shop@gmail.com'' into the text box nearest
    the label ``username''
And I enter the value ``letmein'' into the password box nearest
    the label ``password''
And I click the button labelled ``Login''
When I click the link labelled ``Cart''
And I click the link labelled ``Edit cart''
And I enter the value ``2'' into the number box nearest the
    label ``smart TV''
And I click the button labelled ``save''
Then I should see the message ``Cart has been updated!''
```

```
Scenario:
Given I am using the ``chrome'' web browser
And I have opened the web page ``https://myfavonlineshop.com/login''
And I enter the value ``shop@gmail.com'' into the text box nearest
    the label ``username''
And I enter the value ``letmein'' into the password box nearest
    the label ``password''
And I click the button labelled ``Login''
When I click the link labelled ``Cart''
And I click the check box button nearest the label ``Select all''
And I click the button labelled ``Delete''
Then I should see the message ``Cart has been updated!''
```

In the example above, we have a customer wanting to manage their shopping cart on their favorite online shop. The feature file begins with an outline description of the feature.

As a identifies the actor which is a customer in our case.

I want to represents the general action that they want to perform with the feature, and in this example it is managing the customer's cart.

So that describes the desired outcomes of the feature.

The feature's outline description is useful in documenting the purpose of the tests that are described in the scenarios below it. Each feature has a series of scenarios associated with it. Each scenario describes an alternative sequence through this feature. Each scenario has a title that describes its purpose. **Given-When-Then** is a style of representing tests.

Given represents the pre-conditions of the tests.

When describes the action that the actor in the system performs.

Then describes the expected outcome of that action.

Now let's discuss each scenario in the previously discussed feature file:

The first scenario is about adding a new product to the customer's shopping cart. In the first two 'Given' statements of the scenario, we are selecting our favorite browser and then we are loading the login page of myfavonlineshop.com. In the following three 'Given' statements we are using the customer's credentials to login. The following three 'When' statement represent the action that we want to test, which is adding the smart tv product to the customer's cart. Then, the final statement in the scenario, which is asserting the action of adding the product to the cart.

The second scenario is about editing the quantity of the product that we added to the customer's cart in the first scenario. We have added one more smart tv to the cart. Therefore, we have a total of 2 smart televisions in the cart.

We are now going to show you an example of a question we will ask in the next section.
Exercise Consider the third scenario. Can you propose a title for the scenario, based on the described behaviour and title of the other scenarios?
Answer: Empty my cart.

This brings us to the end of our BDD introduction and training on how to document systems through features and scenarios.

Phase II:
Now that you know the workflow of BDD, let us explore another example. **Prestashop** is an e-commerce solution that helps their customers in creating online stores. They provide a demo shop for potential customers to explore their platform. Click on the following url to view Prestashop's demo shop:

`https://demo.prestashop.com/#/en/back`

Log in to the shop using the provided credentials. Now that you have logged in, take a few minutes to familiarise yourself with the user interface. Here is a list of suggestions (use the menu in the left side of the screen):

- Create a new category (Catalog - Categories - Add new category).
- Create a new product (Catalog - Products - New product).
- Edit an existing product (Catalog - Products - Edit the product you wish by clicking on the pencil icon).
- Create a new customer (Customers - Customers - Add new customer).

Note that you do not have to do them all. Doing two activities would suffice. The point here is to familiarise yourself with the user interface.

Phase III:

In phase I we learned how to document a system in Gherkin features and Scenarios. Furthermore, in phase II we explored Prestashop's user interface. In this phase, you will read 5 untitled scenarios and then select a title from the multiple choice list that best describes each scenario's overall purpose.

1. **Scenario:**
Given I am using the ''chrome'' web browser
And I have opened the web page ''https://demo.prestashop.com/#/en/front''
And I click the link labelled ''All products''
And I click the link labelled ''product1''
Then content appears on the page including the word ''In stock 1000 Items''
When I enter the value ''11'' into the number box nearest the label ''Quantity''
And I click the button labelled ''ADD TO CART''
Then content appears on the page including the word ''There are 11 items in your cart.''
And content appears on the page including the word ''In stock 989 Items''

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Add product(s) in cart

(b) Update product

(c) Delete product

(d) Disable product

2. **Scenario:**
Given I am using the ''chrome'' web browser
And I have opened the web page ''https://demo.prestashop.com/#/en/back''
And I click the button labelled ''LOG IN''
And I click the link labelled ''Catalog''
And I click the link labelled ''Products''
And I click the button labelled ''New product''
And I enter the value ''Presta camera'' into the text box nearest the label ''Enter your product name''
And I select the value ''Standard product'' from the drop down list

nearest the label ''en''
And I click the button labelled ''Save''
When I click the link labelled ''Presta camera''
And I click the link labelled ''Options''
And I enter the value ''978-3-16-148410-0'' into the text box nearest the label ''ISBN''
And I enter the value ''72527273070'' into the text box nearest the label ''UPC barcode''
And I enter the value ''978020137962'' into the text box nearest the label ''EAN-13 or JAN barcode''
And I enter the value ''mpn1'' into the text box nearest the label ''MPN''
And I enter the value ''ref1'' into the text box nearest the label ''Tags''
And I click the check box button nearest the label ''offline''
And I click the button labelled ''Save''
Then I should see the message ''Settings updated.''

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Add new category

(b) Update product

(c) Delete product

(d) Disable product

3. **Scenario:**
Given I am using the ''chrome'' web browser
And I have opened the web page ''https://demo.prestashop.com/#/en/front''
And I click the link labelled ''All products''
And I click the link labelled ''Product1''
Then content appears on the page including the word ''In stock 1000 Items''
When I enter the value ''1100'' into the number box nearest the label ''Quantity''
Then content appears on the page including the word ''There are not enough products in stock''
And content appears on the page including the word ''Cart (0)''
And content appears on the page including the word ''In stock 1000 Items''

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Update shipping details

(b) Can not add product in cart with quantity exceeding availability

(c) Delete product from category

(d) Move product to different category

4. **Scenario:**
Given I am using the ``chrome'' web browser
And I have opened the web page ``https://demo.prestashop.com/#/en/front''
And I click the link labelled ``No account? Create one here''
And I enter the value ``test'' into the text box nearest the label ``First name''
And I enter the value ``Customer'' into the text box nearest the label ``Last name''
And I enter the value ``pud@prestashop.com'' into the text box nearest the label ``Email''
And I enter the value ``testcustomer'' into the password box nearest the label ``Password''
And I click the check box button nearest the label ``Customer data privacy''
And I click the check box button nearest the label ``I agree to the terms and conditions and the privacy policy''
And I click the button labelled ``Save''
And I have opened the webpage ``https://demo.prestashop.com/#/en/back'' in tab ``2''
And I switch to tab ``2''
And I click the button labelled ``LOG IN''
And I click the link labelled ``Catalog''
And I click the link labelled ``Products''
And I click the button labelled ``New product''
And I enter the value ``product1'' into the text box nearest the label ``Enter your product name''
And I enter the value ``19.812'' into the text box below the label ``Price''
And I enter the value ``1000'' into the text box below the label ``Quantity''
And I click the link labelled ``Quantities''

And I enter the value ``10'' into the text box nearest the label ``Minimum quantity for sale''
And I click the button labelled ``Save''
And I switch to tab ``1''
And I click the link labelled ``All products''
And I click the link labelled ``Product1''
When I enter the value ``5'' into the number box nearest the label ``Quantity''
Then content appears on the page including the word ``The minimum purchase order quantity for the product is 10.''
And content appears on the page including the word ``Cart (0)''
And content appears on the page including the word ``In stock 1000 Items''

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Can not add product in cart with minimal quantity enabled

(b) Delete product from category

(c) Update customer's address

(d) Move product to different category

5. **Scenario:**
Given I am using the ``chrome'' web browser
And I have opened the web page ``https://demo.prestashop.com/#/en/back''
And I click the button labelled ``LOG IN''
And I click the link labelled ``Shop Parameters''
And I click the link labelled ``Product Settings''
And I click the radio button nearest the label ``Allow ordering of out-of-stock products''
And I click the button labelled ``Save''
And I click the link labelled ``Catalog''
And I click the link labelled ``Products''
And I click the button labelled ``New product''
And I enter the value ``product1'' into the text box nearest the label ``Enter your product name''
And I enter the value ``19.812'' into the text box below the label ``Price''
And I enter the value ``1000'' into the text box below the label ``Quantity''

And I click the button labelled ``Save''
And I have opened the webpage ``https://demo.prestashop.com/#/en/front'' in tab ``2''
And I switch to tab ``2''
When I click the link labelled ``All products''
And I click the link labelled ``Product1''
And I enter the value ``1100'' into the number box nearest the label ``Quantity''
And I click the button labelled ``ADD TO CART''
Then content appears on the page including the word ``There are 1100 items in your cart.''
And content appears on the page including the word ``In stock -100 Items''

Select a title from the multiple choice list that best describes the scenario's overall purpose.

(a) Can not add product in cart with minimal quantity enabled

(b) Be able to add out of stock product if configuration allows it

(c) Update customer's credit card

(d) Delete product from category

Phase IV: Final phase

In phase I we learned how to document a system in Gherkin features and scenarios. Furthermore, in phase II we explored Prestashop's user interface. Also, in phase III, we had a chance to read untitled scenarios and then give a title for each scenario that indicates its overall purpose. In this final phase, you will have 5 incomplete scenarios that are missing the outcome step(s) . Your task is to select the step from the multiple choice list that completes each scenario. If you think there is no step that completes the scenario, then choose the option "It can not be done".

Assume that you have the following steps that run before each of the 5 scenarios. These steps are grouped in a background. The 'Background' keyword means that these steps run before each scenario in the feature file.

```
Background: Adding new Category
Given I am using the ``chrome'' web browser
And I have opened the web page ``https://demo.prestashop.com/#/en/
    back''
And I click the button labelled ``LOG IN''
When I click the link labelled ``Catalog''
And I click the link labelled ``Categories''
And I click the button labelled ``Add new category''
And I enter the value ``PC parts'' into the text box nearest the
    label ``Name''
And I click the radio button nearest the ``Displayed''
And I click the radio button nearest the label ``Home''
And I click the radio button nearest the label ``Accessories''
And I click the radio button nearest the label ``Home Accessories''
And I enter the value ``pc-parts'' into the text box nearest the
    label ``Friendly URL''
And I click the button labelled ``Save''

1- Scenario: Delete category
When I click the button nearest the label ``Actions''
And I click the button labelled ``Delete''
Then
```

Choose a step from the multiple choice list that best completes the scenario.

(a) I should see the message "Customer's address updated"

(b) I should see the message "Customer's credit card updated"

(c) I should see the message "Category deleted"

(d) I should see the message “Category created”

(e) It can not be done

```
2. Scenario: enable category
When I click the link labelled ``PC parts''
And  I click the link labelled ``Edit''
And I click the radio button nearest the label ``Displayed''
Then
```

Choose a step from the multiple choice list that best completes the scenario.

(a) the radio button nearest the label “Displayed” is selected

(b) I should see the message “Customer’s credit card updated”

(c) I should see the message “Category deleted”

(d) the check box button nearest the label “gift wrapping” is selected

(e) It can not be done

```
3. Scenario: delete category menu thumbnail imag
Given I click the link labelled ``PC parts''
And I click the link labelled ``Edit''
And I enter the value ``dummy category name'' into the text box
    nearest the label ``Name''
And I click the radio button nearest the label ``Displayed''
And I click the radio button nearest the label ``Home''
And I click the radio button nearest the label ``Accessories''
And I click the radio button nearest the label ``Home Accessories''
And I enter the value ``dummy description'' into the text box
    nearest the label ``Description''
And I enter the value ``dummy meta title'' into the text box
    nearest the label ``Meta title''
And I enter the value ``dummy meta description'' into the text box
    nearest the label ``Meta description''
And I enter the value ``dummy'' into the text box nearest the label
    ``Friendly URL''
And I upload the file ``src\test\resources\logo.jpg'' by clicking
    the ``Browse'' button
And I click the button labelled ``Save''
And I click the link labelled ``dummy category name''
```

And I click the link labelled ''Edit''
When I click the button labelled ''Delete category thumbnail image''
Then

Choose a step from the multiple choice list that best completes the scenario.

(a) I should see the message "Shipping details updated"

(b) I should see the message "Category deleted"

(c) I should see the message "Customer's credit card updated"

(d) I should see the message "The thumbnail image was successfully deleted"

(e) It can not be done

4- **Scenario:** bulk disable selected categories
Given I click the button labelled ''Add new category''
And I enter the value ''PC parts 2'' into the text box nearest the label ''Name''
And I click the radio button nearest the label ''Home''
And I click the radio button nearest the label ''Accessories''
And I click the radio button nearest the label ''Home Accessories''
And I enter the value ''pc-parts2'' into the text box nearest the label ''Friendly URL''
And I click the button labelled ''Save''
When I click the check box button nearest the label ''Categories''
And I select the value ''Disable selection'' from the drop down list below the label ''Categories''
Then

Choose a step from the multiple choice list that best completes the scenario.

(a) I should see the message "Shipping details updated"

(b) I should see the message "The status has been successfully updated"

(c) I should see the message "Category deleted"

(d) I should see the message "Customer's address has been updated"

(e) It can not be done

```
5. Scenario: delete category cover image
Given I click the link labelled ''PC parts''
And I click the link labelled ''Edit''
And I enter the value ''dummy category name'' into the text box
    nearest the label ''Name''
And I click the radio button nearest the label ''Displayed''
And I click the radio button nearest the label ''Home''
And I click the radio button nearest the label ''Accessories''
And I click the radio button nearest the label ''Home Accessories''
And I enter the value ''dummy description'' into the text box
    nearest the label ''Description''
And I enter the value ''dummy meta title'' into the text box
    nearest the label ''Meta title''
And I enter the value ''dummy meta description'' into the text box
    nearest the label ''Meta description''
And I enter the value ''dummy'' into the text box nearest the label
    ''Friendly URL''
And I upload the file ''src\test\resources\logo.jpg'' by clicking
    the ''Browse'' button
And I click the button labelled ''Save''
And I click the link labelled ''dummy category name''
And I click the link labelled ''Edit''
When I click the button labelled ''Delete Cover Image''
Then
```

Choose a step from the multiple choice list that best completes the scenario.

(a) I should see the message “Shipping details updated”

(b) I should see the message “Customer’s email has been updated”

(c) I should see the message “The category was successfully deleted”

(d) I should see the message “Customer’s address has been updated”

(e) It can not be done

Final words

- Please enter your email below if you would like to enter the prize draw for a chance to win £200.
- Please enter your email below if you would like to receive updates about the study.
- Please enter your comments about the study if you have any.
- End of study. Thank you for taking part in this study!

Bibliography

[1] B. Nuseibeh and S. M. Easterbrook, "Requirements engineering: a roadmap," in *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*, A. Finkelstein, Ed. ACM, 2000, pp. 35–46. [Online]. Available: https://doi.org/10.1145/336512.336523

[2] A. van Lamsweerde, "Requirements engineering in the year 00: a research perspective," in *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, C. Ghezzi, M. Jazayeri, and A. L. Wolf, Eds. ACM, 2000, pp. 5–19. [Online]. Available: https://doi.org/10.1145/337180.337184

[3] P. Pandit and S. Tahiliani, "Agileuat: A framework for user acceptance testing based on user stories and acceptance criteria," *International Journal of Computer Applications*, vol. 120, no. 10, 2015.

[4] R. Miller and C. T. Collins, "Acceptance testing," *Proc. XPUniverse*, vol. 238, 2001.

[5] P. Hsia, D. C. Kung, and C. Sell, "Software requirements and acceptance testing," *Ann. Softw. Eng.*, vol. 3, pp. 291–317, 1997. [Online]. Available: https://doi.org/10.1023/A:1018938021528

[6] H. K. Flora and S. V. Chande, "A systematic study on agile software development methodologies and practices," *International Journal of Computer Science and Information Technologies*, vol. 5, no. 3, pp. 3626–3637, 2014.

[7] A. Alliance, "Agile essentials," https://www.agilealliance.org/agile-essentials/, n.d, online; accessed 11 July 2023.

[8] ——, "What is test driven development (tdd)?" https://www.agilealliance.org/glossary/tdd/, n.d, online; accessed 11 July 2023.

[9] ——, "What is acceptance test driven development (atdd)?" https://www.agilealliance.org/glossary/atdd/, n.d, online; accessed 11 July 2023.

[10] D. North, "Introducing bdd," https://dannorth.net/introducing-bdd/, 2006, online; accessed 05 January 2020.

[11] A. Knight, "Bdd," https://automationpanda.com/bdd/, 2017, online; accessed 06 July 2020.

[12] ——, "Bdd 101: Writing good gherkin," https://automationpanda.com/2017/01/30/bdd-101-writing-good-gherkin/, 2017, online; accessed 06 July 2022.

[13] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Maintaining behaviour driven development specifications: Challenges and opportunities," in *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, R. Oliveto, M. D. Penta, and D. C. Shepherd, Eds. IEEE Computer Society, 2018, pp. 175–184. [Online]. Available: https://doi.org/10.1109/SANER.2018.8330207

[14] digital ai, "The 14th annual state of agile report," https://digital.ai/catalyst-blog/the-14th-annual-state-of-agile-report/, 2020, online; accessed 06 May 2022.

[15] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Characterising the quality of behaviour driven development specifications," in *Agile Processes in Software Engineering and Extreme Programming - 21st International Conference on Agile Software Development, XP 2020, Copenhagen, Denmark, June 8-12, 2020, Proceedings*, ser. Lecture Notes in Business Information Processing, V. Stray, R. Hoda, M. Paasivaara, and P. Kruchten, Eds., vol. 383. Springer, 2020, pp. 87–102. [Online]. Available: https://doi.org/10.1007/978-3-030-49392-9_6

[16] SpecFlow, "How to fix dependancy chains in your scenarios," https://specflow.org/challenges/chain-of-dependent-scenarios/.

[17] Cucumber, "Writing better gherkin," https://cucumber.io/docs/bdd/better-gherkin/.

[18] R. T. Ogawa and B. Malen, "Towards rigor in reviews of multivocal literatures: Applying the exploratory case study method," *Review of Educational Research*, vol. 61, no. 3, pp. 265–286, 1991. [Online]. Available: https://doi.org/10.3102/00346543061003265

[19] A. Santos, S. Vegas, O. Dieste, F. Uyaguari, A. Tosun, D. Fucci, B. Turhan, G. Scanniello, S. Romano, I. Karac, M. Kuhrmann, V. Mandić, R. Ramač, D. Pfahl, C. Engblom, J. Kyykka, K. Rungi, C. Palomeque, J. Spisak, M. Oivo, and N. Juristo, "A family of experiments on test-driven development," *Empirical Softw. Engg.*, vol. 26, no. 3, May 2021. [Online]. Available: https://doi.org/10.1007/s10664-020-09895-8

[20] D. Janzen and H. Saiedian, "A leveled examination of test-driven development acceptance," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 719–722.

[21] M. Pancur and M. Ciglarič, "Impact of test-driven development on productivity, code and tests: A controlled experiment," *Information and Software Technology*, vol. 53, no. 6, pp. 557–573, 2011.

[22] A. Dookhun and L. Nagowah, "Assessing the effectiveness of test-driven development and behavior-driven development in an industry setting," in *Proceedings of 2019 International Conference on Computational Intelligence and Knowledge Economy, ICCIKE 2019*, 2019, pp. 365–370, cited By 0. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85080882684&doi=10.1109%2fICCIKE47802.2019.9004328&partnerID=40&md5=cb86f2f2113de1080df50ace3fc61819

[23] H. Abushama, H. Alassam, and F. Elhaj, "The effect of test-driven development and behavior-driven development on project success factors: A systematic literature review based study," in *2020 International Conference on Computer, Control, Electrical, and Electronics Engineering (ICCCEEE)*, 2021, pp. 1–9.

[24] S. Karpurapu, S. Myneni, U. Nettur, L. Gajja, D. Burke, T. Stiehm, and J. Payne, "Comprehensive evaluation and insights into the use of large language models in the automation of behavior-driven development acceptance test formulation," *IEEE Access*, vol. 12, pp. 58 715–58 721, 2024.

[25] F. Zampetti, A. D. Sorbo, C. A. Visaggio, G. Canfora, and M. D. Penta, "Demystifying the adoption of behavior-driven development in open source projects," *Inf. Softw. Technol.*, vol. 123, p. 106311, 2020. [Online]. Available: https://doi.org/10.1016/j.infsof.2020.106311

[26] H. Lima, K. C. Souza, L. De Paula, L. E. Martins, W. Giozza, and R. De Sousa, "Acceptance tests over microservices architecture using behaviour-driven development," in *2021 16th Iberian Conference on Information Systems and Technologies (CISTI)*, 2021, pp. 1–6.

[27] D. Lübke and T. Van Lessen, "Modeling test cases in bpmn for behavior-driven development," *IEEE Software*, vol. 33, no. 5, pp. 15–21, 2016.

[28] S. Rodriguez, J. Thangarajah, and M. Winikoff, "A behaviour-driven approach for testing requirements via user and system stories in agent systems," in *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems*, 2023, pp. 1182–1190.

[29] V. Garousi, M. Felderer, and M. V. Mäntylä, "Guidelines for including grey literature and conducting multivocal literature reviews in software engineering," *Information and Software Technology*, vol. 106, pp. 101–121, 2019.

[30] P. Raulamo-Jurvanen, M. Mantyla, and V. Garousi, "Choosing the right test automation tool: a grey literature review of practitioner sources," in *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 2017, pp. 21–30.

[31] C. Solís and X. Wang, "A study of the characteristics of behaviour driven development," in *37th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA 2011, Oulu, Finland, August 30 - September 2, 2011*. IEEE Computer Society, 2011, pp. 383–387. [Online]. Available: https://doi.org/10.1109/SEAA.2011.76

[32] A. Scandaroli, R. Leite, A. H. Kiosia, and S. A. Coelho, "Behavior-driven development as an approach to improve software quality and communication across remote business stakeholders, developers and QA: two case studies," in *Proceedings of the 14th International Conference on Global Software Engineering, ICGSE 2019, Montreal, QC, Canada, May 25-31, 2019*, F. Calefato, P. Tell, and A. Dubey, Eds. IEEE / ACM, 2019, pp. 95–100. [Online]. Available: https://doi.org/10.1109/ICGSE.2019.00030

[33] L. Pereira, H. Sharp, C. R. B. de Souza, G. Oliveira, S. Marczak, and R. Bastos, "Behavior-driven development benefits and challenges: reports from an industrial study," in *Proceedings of the 19th International Conference on Agile Software Development, XP 2019, Companion, Porto, Portugal, May 21-25, 2018*, A. Aguiar, Ed. ACM, 2018, pp. 42:1–42:4. [Online]. Available: https://doi.org/10.1145/3234152.3234167

[34] T. Storer and R. Bob, "Behave nicely! automatic generation of code for behaviour driven development test suites," in *19th International Working Conference on Source Code Analysis and Manipulation, SCAM 2019, Cleveland, OH, USA, September 30 - October 1, 2019*. IEEE, 2019, pp. 228–237. [Online]. Available: https://doi.org/10.1109/SCAM.2019.00033

[35] "Specflow," https://specflow.org/, online; accessed 11 January 2020.

[36] "Jbehave," https://jbehave.org/, online; accessed 13 January 2020.

[37] "Rspec," https://rspec.info/, online; accessed 11 January 2020.

[38] "Nbehave," https://github.com/nbehave/NBehave, online; accessed 11 January 2020.

[39] "Mspec," https://github.com/machine/machine.specifications, online; accessed 10 January 2020.

[40] "Storyq," https://github.com/wforney/storyq, online; accessed 10 January 2020.

[41] "Cucumber," https://cucumber.io/, online; accessed 12 January 2020.

[42] A. Okolnychyi and K. Fögen, "A study of tools for behavior-driven development," *Full-scale Software Engineering/Current Trends in Release Engineering*, p. 7, 2016.

[43] "Concordion," https://concordion.org/, online; accessed 10 January 2020.

[44] "Spock," https://code.google.com/archive/p/spock/, online; accessed 08 January 2020.

[45] "Easyb," http://easyb.io/, online; accessed 06 January 2020.

[46] A. Chandorkar, N. Patkar, A. Di Sorbo, and O. Nierstrasz, "An exploratory study on the usage of gherkin features in open-source projects," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022, pp. 1159–1166.

[47] S. Kamalakar, S. H. Edwards, and T. M. Dao, "Automatically generating tests from natural language descriptions of software behavior," in *ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering, Angers, France, 4-6 July, 2013*, L. A. Maciaszek and J. Filipe, Eds. SciTePress, 2013, pp. 238–245. [Online]. Available: https://doi.org/10.5220/0004566002380245

[48] M. Soeken, R. Wille, and R. Drechsler, "Assisted behavior driven development using natural language processing," in *Objects, Models, Components, Patterns - 50th International Conference, TOOLS 2012, Prague, Czech Republic, May 29-31, 2012. Proceedings*, ser. Lecture Notes in Computer Science, C. A. Furia and S. Nanz, Eds., vol. 7304. Springer, 2012, pp. 269–287. [Online]. Available: https://doi.org/10.1007/978-3-642-30561-0_19

[49] M. Diepenbeck, M. Soeken, D. Große, and R. Drechsler, "Towards automatic scenario generation from coverage information," in *8th International Workshop on Automation of Software Test, AST 2013, San Francisco, CA, USA, May 18-19, 2013*, H. Zhu, H. Muccini, and Z. Chen, Eds. IEEE Computer Society, 2013, pp. 82–88. [Online]. Available: https://doi.org/10.1109/IWAST.2013.6595796

[50] N. Li, A. Escalona, and T. Kamal, "Skyfire: Model-based testing with cucumber," in *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 393–400. [Online]. Available: https://doi.org/10.1109/ICST.2016.41

[51] A. Z. H. Yang, D. A. da Costa, and Y. Zou, "Predicting co-changes between functionality specifications and source code in behavior driven development," in *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, M. D. Storey, B. Adams, and S. Haiduc, Eds. IEEE / ACM, 2019, pp. 534–544. [Online]. Available: https://doi.org/10.1109/MSR.2019.00080

[52] L. P. Binamungu, S. M. Embury, and N. Konstantinou, "Detecting duplicate examples in behaviour driven development specifications," in *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests, VST@SANER 2018, Campobasso, Italy, March 20, 2018*, C. Artho and R. Ramler, Eds. IEEE, 2018, pp. 6–10. [Online]. Available: https://doi.org/10.1109/VST.2018.8327149

[53] C. Sathawornwichit and S. Hosono, "Consistency reflection for automatic update of testing environment," in *2012 IEEE Asia-Pacific Services Computing Conference, APSCC 2012, Guilin, China, December 6-8, 2012*. IEEE Computer Society, 2012, pp. 335–340. [Online]. Available: https://doi.org/10.1109/APSCC.2012.49

[54] ——, "Consistent testing framework for continuous service development," in *2014 Asia-Pacific Services Computing Conference, APSCC 2014, Fuzhou, Fu Jian, China, December 4-6, 2014*. IEEE Computer Society, 2014, pp. 8–15. [Online]. Available: https://doi.org/10.1109/APSCC.2014.13

[55] M. Irshad, J. Börstler, and K. Petersen, "Supporting refactoring of BDD specifications - an empirical study," *Inf. Softw. Technol.*, vol. 141, p. 106717, 2022. [Online]. Available: https://doi.org/10.1016/j.infsof.2021.106717

[56] G. Oliveira and S. Marczak, "On the empirical evaluation of bdd scenarios quality: Preliminary findings of an empirical study," in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, 2017, pp. 299–302.

[57] G. Oliveira, S. Marczak, and C. Moralles, "How to evaluate BDD scenarios' quality?" in *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019*, I. do Carmo Machado, R. Souza, R. S. P. Maciel, and C. Sant'Anna, Eds. ACM, 2019, pp. 481–490. [Online]. Available: https://doi.org/10.1145/3350768.3351301

[58] N. Borle, M. Feghhi, E. Stroulia, R. Grenier, and A. Hindle, "[journal first] analyzing the effects of test driven development in github," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 1062–1062.

[59] M. Diepenbeck, M. Soeken, D. Große, and R. Drechsler, "Behavior driven development for circuit design and verification," in *2012 IEEE International High Level Design Validation and Test Workshop, HLDVT 2012, Huntington Beach, CA, USA, November 9-10, 2012*. IEEE Computer Society, 2012, pp. 9–16. [Online]. Available: https://doi.org/10.1109/HLDVT.2012.6418237

[60] M. Diepenbeck, U. Kühne, M. Soeken, D. Große, and R. Drechsler, "Behaviour driven development for hardware design," *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 11, pp. 29–45, 2018. [Online]. Available: https://doi.org/10.2197/ipsjtsldm.11.29

[61] P. Morrison, C. Holmgreen, A. Massey, and L. A. Williams, "Proposing regulatory-driven automated test suites for electronic health record systems," in *Proceedings of the 5th International Workshop on Software Engineering in Health Care, SEHC 2013, San Francisco, California, USA, May 20-21, 2013*, J. Knight and C. E. Kuziemsky, Eds. IEEE Computer Society, 2013, pp. 46–49. [Online]. Available: https://doi.org/10.1109/SEHC.2013.6602477

[62] F. Giorgi and F. Paulisch, "Transition towards continuous delivery in the healthcare domain," in *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, H. Sharp and M. Whalen, Eds. IEEE / ACM, 2019, pp. 253–254. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00035

[63] M. Schneider, B. Hippchen, S. Abeck, M. Jacoby, and R. Herzog, "Enabling iot platform interoperability using a systematic development approach by example," in *2018 Global Internet of Things Summit, GIoTS 2018, Bilbao, Spain, June 4-7, 2018*. IEEE, 2018, pp. 1–6. [Online]. Available: https://doi.org/10.1109/GIOTS.2018.8534549

[64] P. Rathod, V. Julkunen, T. Kaisti, and J. Nissila, "Automatic acceptance testing of the web application security with ITU-T X.805 framework," in *Second International Conference on Computer Science, Computer Engineering, and Social Media, CSCESM 2015, Lodz, Poland, September 21-23, 2015*. IEEE, 2015, pp. 103–108. [Online]. Available: https://doi.org/10.1109/CSCESM.2015.7331876

[65] V. T. Sarinho, ""bdd assemble!": A paper-based game proposal for behavior driven development design learning," in *Entertainment Computing and Serious Games - First IFIP TC 14 Joint International Conference, ICEC-JCSG 2019,*

Arequipa, Peru, November 11-15, 2019, Proceedings, ser. Lecture Notes in Computer Science, E. D. V. der Spek, S. Göbel, E. Y. Do, E. Clua, and J. B. Hauge, Eds., vol. 11863. Springer, 2019, pp. 431–435. [Online]. Available: https://doi.org/10.1007/978-3-030-34644-7_41

[66] I. Papadopoulos, D. G. Kogias, C. Z. Patrikakis, and C. Marinou, "Enhancing the student's logical thinking with gherkin language," in *2017 IEEE Global Engineering Education Conference, EDUCON 2017, Athens, Greece, April 25-28, 2017*. IEEE, 2017, pp. 1543–1547. [Online]. Available: https://doi.org/10.1109/EDUCON.2017.7943054

[67] TestQuality, "Cucumber and gherkin language best practices," https://www.testquality.com/blog/tpost/v79acjttj1-cucumber-and-gherkin-language-best-pract, 2022, online; accessed 11 July 2022.

[68] E. Hartill, "Cucumber best practices evaluated," https://www.lithichor.com/2020/06/cucumber-best-practices-evaluated.html, 2020, online; accessed 11 July 2022.

[69] M. Honkanen, "Cucumber best practises- push how down," https://honkanen.io/blog/cucumber-best-practises-push-how-down, 2013, online; accessed 10 July 2022.

[70] C. Helm, "Top 5 cucumber best practices," https://www.cloudbees.com/blog/cucumber-best-practices, 2013, online; accessed 10 July 2022.

[71] F. Toledo and L. Zambra, "A guide to cucumber best practices," https://dzone.com/articles/a-guide-to-good-cucumber-practices, 2019, online; accessed 08 July 2022.

[72] J. Sheffer, "Cucumber bdd best practices," https://www.youtube.com/watch?v=nrggIRWK6qo, 2017, online; accessed 07 July 2022.

[73] P. Shah, "Bdd — best practices," https://priyank-it.medium.com/bdd-best-practices-61f01098ed95, 2016, online; accessed 07 July 2022.

[74] M. Thorn, "Implement bdd with cucumber and specflow," https://www.slideshare.net/TechWellPresentations/implement-bdd-with-cucumber-and-specflow, 2018, online; accessed 10 July 2022.

[75] L. Jorente, "6 best practices for gherkin," https://blog.avenuecode.com/gherkin-best-practices/, 2021, online; accessed 08 July 2022.

[76] R. Azevedo, "Bdd best practices," https://azevedorafaela.com/2015/12/12/bdd-best-practices/, 2015, online; accessed 09 July 2022.

[77] A. Ghahrai, "Bdd guidelines and best practices," https://devqa.io/bdd-guidelines-best-practices/, 2016, online; accessed 09 July 2022.

[78] T. Hamilton, "Gherkin language format and syntax and gherkin test in cucumber," https://www.guru99.com/gherkin-test-cucumber.html, 2022, online; accessed 12 July 2022.

[79] N. Khunteta, "Best practices for using given/when/then/and/but gherkin keywords in feature file," https://www.youtube.com/watch?v=2vUHh6nFyFI, 2020, online; accessed 07 July 2022.

[80] J. Topcu, "Behavior-driven development from scratch," https://beyondxscratch.com/2019/05/21/behavior-driven-development-from-scratch/, 2019, online; accessed 09 July 2022.

[81] Specflow, "Writing effective behaviour-driven test cases," https://specflow.org/gherkin/writing-effective-behaviour-driven-test-cases/, 2022, online; accessed 08 July 2022.

[82] S. John, "Bdd vs tdd : Highlighting the two important quality engineering practices," https://www.cuelogic.com/blog/bdd-vs-tdd, 2021, online; accessed 10 July 2022.

[83] A. Knight, "Better behavior-driven development: 4 rules for writing good gherkin," https://techbeacon.com/app-dev-testing/better-behavior-driven-development-4-rules-writing-good-gherkin, 2018, online; accessed 06 July 2022.

[84] H. Paul, "Top 6 cucumber best practices for selenium automation," https://www.lambdatest.com/blog/cucumber-best-practices/, 2020, online; accessed 07 July 2022.

[85] SmartBear, "Best practices for scenario writing," https://support.smartbear.com/cucumberstudio/docs/tests/best-practices.html, 2022, online; accessed 06 July 2022.

[86] "Github," https://github.com/, online; accessed 06 September 2020.

[87] "Selenium," https://www.selenium.dev/, online; accessed 07 September 2020.

[88] "Behave," https://behave.readthedocs.io/en/latest/, online; accessed 12 January 2020.

[89] "Java," https://www.java.com/en/, online; accessed 07 September 2020.

[90] M. Leotta, B. García, F. Ricca, and J. Whitehead, "Challenges of end-to-end testing with selenium webdriver and how to face them: A survey," in *IEEE Conference on Software Testing, Verification and Validation, ICST 2023, Dublin, Ireland, April 16-20, 2023*. IEEE, 2023, pp. 339–350. [Online]. Available: https://doi.org/10.1109/ICST57152.2023.00039

[91] J. Unadkat, "Understanding selenium webdriver," https://www.browserstack.com/guide/selenium-webdriver-tutorial, 2023, online; accessed 06 May 2024.

[92] T. M. Foundation, "Element: getboundingclientrect() method," https://developer.mozilla.org/en-US/docs/Web/API/Element/getBoundingClientRect, 2023, online; accessed 06 July 2024.

[93] B. J. Oates, M. Griffiths, and R. McLean, *Researching information systems and computing*. Sage, 2022.

[94] G. G. Gable, "Integrating case study and survey research methods: an example in information systems," *European journal of information systems*, vol. 3, pp. 112–126, 1994.

[95] "The unicorn hub," https://www.theunicornhub.com/, online; accessed 13 May 2021.

[96] "React," https://reactjs.org/, online; accessed 07 September 2020.

[97] "Flask," https://flask.palletsprojects.com/, online; accessed 06 September 2020.

[98] "Docker," https://www.docker.com/, online; accessed 06 September 2020.

[99] "cloc," https://github.com/AlDanial/cloc, online; accessed 05 September 2020.

[100] "Django," https://djangoproject.com/, online; accessed 12 November 2021.

[101] V. Egele, L. Kiefer, and R. Stark, "Faking self-reports of health behavior: a comparison between a within- and a between-subjects design," *Health Psychology and Behavioral Medicine*, vol. 9, pp. 895–916, 2021.

[102] G. Charness, U. Gneezy, and M. Kuhn, "Experimental methods: between-subject and within-subject design," *Journal of Economic Behavior Organization*, vol. 81, pp. 1–8, 2012.

[103] A. Bathke, S. Friedrich, M. Pauly, F. Konietschke, W. Staffen, N. Strobl, and Y. Höller, "Testing mean differences among groups: multivariate and repeated measures analysis with minimal assumptions," *Multivariate Behavioral Research*, vol. 53, pp. 348–359, 2018.

[104] T. Macfarland, "Student's t-test for independent samples," *Using R for Biostatistics*, 2020.

[105] K. Jankowski, K. Flannelly, and L. Flannelly, "The t-test: An influential inferential tool in chaplaincy and other healthcare research," *Journal of Health Care Chaplaincy*, vol. 24, pp. 30 – 39, 2018.

[106] J. de Winter, "Using the student's "t"-test with extremely small sample sizes." *Practical Assessment, Research and Evaluation*, vol. 18, p. 10, 2013.

[107] E. Livingston, "Who was student and why do we care so much about his t-test?" *The Journal of surgical research*, vol. 118 1, pp. 58–65, 2004.

[108] G. Islam and T. Storer, "A dataset of repositories on github containing gherkin feature files, v1.0," Available at https://git.dcs.gla.ac.uk/cornichon/cucumber-bdd-on-github-study/cucumber-bdd-repositories-on-github-data-set, November 2022.

[109] "Prestashop," https://github.com/PrestaShop/PrestaShop/tree/develop/tests/Integration/Behaviour/Features/Scenario, 2022, online; accessed 05 May 2022.

[110] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[111] D. Sjoeberg, J. Hannay, O. Hansen, V. Kampenes, A. Karahasanovic, N.-K. Liborg, and A. Rekdal, "A survey of controlled experiments in software engineering," *IEEE Transactions on Software Engineering*, vol. 31, no. 9, pp. 733–753, 2005.

[112] J. Piasecki, M. Waligora, and V. Dranseika, "Google search as an additional source in systematic reviews," *Science and engineering ethics*, vol. 24, pp. 809–810, 2018.

[113] K. Godin, J. Stapleton, S. Kirkpatrick, R. Hanning, and S. Leatherdale, "Applying systematic review search methods to the grey literature: A case study examining guidelines for school-based breakfast programs in canada," *Systematic reviews*, vol. 4, p. 138, 10 2015.

[114] G. Islam, "On the real world practice of behaviour driven development," Available at https://theses.gla.ac.uk/84085/1/2024IslamPhD.pdf, January 2024, online; accessed 20 May 2024.

[115] D. I. K. Sjøberg and G. R. Bergersen, "Construct validity in software engineering," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, pp. 1374–1396, 2023.