Islam, Gibrail (2024) *On the real world practice of Behaviour Driven Development.* PhD thesis.

https://theses.gla.ac.uk/84085/

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission in writing from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses
https://theses.gla.ac.uk/
research-enlighten@glasgow.ac.uk

On the Real World Practice of Behaviour Driven Development

Gibrail Islam

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

School of Engineering
College of Science and Engineering
University of Glasgow

January 2024

Abstract

Surveys of industry practice over the last decade suggest that Behaviour Driven Development is a popular Agile practice. For example, 19% of respondents to the 14th *State of Agile* annual survey reported using BDD, placing it in the top 13 practices reported. As well as potential benefits, the adoption of BDD necessarily involves an additional cost of writing and maintaining Gherkin features and scenarios, and (if used for acceptance testing,) the associated step functions. Yet there is a lack of published literature exploring how BDD is used in practice and the challenges experienced by real world software development efforts. This gap is significant because without understanding current real world practice, it is hard to identify opportunities to address and mitigate challenges. In order to address this research gap concerning the challenges of using BDD, this thesis reports on a research project which explored: (a) the challenges of applying agile and undertaking requirements engineering in a real world context; (b) the challenges of applying BDD specifically and (c) the application of BDD in open-source projects to understand challenges in this different context.

For this purpose, we progressively conducted two case studies, two series of interviews, four iterations of action research, and an empirical study. The first case study was conducted in an avionics company to discover the challenges of using an agile process in a large scale safety-critical project environment. Since requirements management was found to be one of the biggest challenges during the case study, we decided to investigate BDD because of its reputation for requirements management. The second case study was conducted in the company with an aim to discover the challenges of using BDD in real life. The case study was complemented with an empirical study of the practice of BDD in open source projects, taking a study sample from the GitHub open source collaboration site.

As a result of this Ph.D research, we were able to discover: ***(i)*** challenges of using an agile process in a large scale safety-critical organisation, ***(ii)*** current state of BDD in practice, ***(iii)*** technical limitations of Gherkin (i.e., the language for writing requirements in BDD), ***(iv)*** challenges of using BDD in a real project, ***(v)*** bad smells in the Gherkin specifications of open-source projects on GitHub. We also presented a brief comparison between the theoretical description of BDD and BDD in practice. This research, therefore, presents the results of lessons learned from BDD in practice, and serves as a guide for software practitioners planning on using BDD in their projects.

Acknowledgements

First of all, I would like to thank the Almighty God for His uncountable blessings. It has been a long and hard journey, and the person I would like to thank the most is my supervisor. I could not wish for a better supervisor than him. He has been a real inspiration. I thank him from the depths of my heart for keeping me motivated and guiding me throughout this journey.

I would like to thank my father, my mother, and my brother for their love, support, and prayers. I would like to thank my wife from the bottom of my heart who took care of my responsibilities while I did my Ph.D. I can never thank her enough. I would also like to thank my children who had to bear the pain of being away from me.

I would like to thank Dr. Robbie Simpson (Late) for his help and support during the early phase of my Ph.D. research. Lastly, I would like to extend my gratitude to the faculty members and the staff at the School of Computing Science (University of Glasgow), and the individuals within *the Company* who assisted me with my research, particularly with undertaking the interviews.

Declaration

I declare that the research in this study is my own work and no part of this thesis was presented for another degree in this or any other university anywhere. The contents of Chapter 4 of this thesis contributed to the following publication which was co-authored with my supervisor Dr. Tim Storer.

> Islam G, Storer T. A case study of agile software development for safety-critical systems projects. Reliability Engineering & System Safety. 2020 Aug 1;200:106954.

Contents

Abstract **i**

Acknowledgements **ii**

Declaration **iii**

1 Introduction **1**

1.1 Background 1
1.2 Motivation 3
1.3 Thesis Statement 4
1.4 Research Contribution 5
1.5 Thesis Overview 7

2 Research Methodology **10**

2.1 Justification of the Research Approach 10
2.2 Literature Review 12
2.2.1 First Literature Review 13
2.2.2 Second Part of the Literature Review 13
2.3 Case Study 14
2.3.1 Semi Structured Interviews 15
2.3.2 Wengraf's Method 16
2.4 Action Research 18
2.5 Online Experiment on GitHub 21
2.6 Summary 22

3 SLR on Agile Methods in Safety Critical Systems **23**

3.1 Agile Development and Safety-Critical Systems 23
3.1.1 Agile Software Development 24
3.1.2 Safety-Critical Systems 26
3.1.3 Agile Software Development for Safety-Critical Systems 28
3.2 Literature Review Process 29

3.3 Thematic Analysis of the Selected Studies 33
3.3.1 Advocates of Agile Methods 33
3.3.2 Studies Focusing a Particular Aspect of Agile Development of Safety-Critical Systems 35
3.3.3 Suggested Tailoring in Agile Methods 41
3.4 Discussion Based Upon Thematic Analysis of the Challenges Identified from the Studies 47
3.4.1 Statements and Perceptions 47
3.4.2 Organisational Culture and Training 51
3.4.3 Project Management 52
3.4.4 Documentation 54
3.4.5 Regulatory Standards 55
3.4.6 Design and Architecture 57
3.5 Discussion 58
3.6 Threats to Validity 60
3.7 Summary 61

4 Agile in Large Scale Safety Critical Systems 63
4.1 Objectives of the Exploratory Case Study 64
4.2 Interviews 65
4.3 Overview of Software Development in the Company 68
4.3.1 Project Team Structure 69
4.3.2 Development Process 69
4.3.3 Project Customers 70
4.3.4 Requirements Management 73
4.3.5 Product Integration and Certification 73
4.4 Use of Agile Software Development 74
4.5 Discussion of Challenges 79
4.5.1 Pressure for Waterfall (Challenges 1, 2, 3, 4, 5) 79
4.5.2 Coordination amongst Stakeholders (Challenges 6, 7, 9) 86
4.5.3 Documentation and Communication (Challenge 8, 10) 89
4.5.4 Cultural Challenges (11, 12, 13) 92
4.5.5 Agile Methods Tailored to Large-Scale Safety-Critical Systems 93
4.6 Threats to Validity 94
4.7 Summary 95

5 Literature Review and Background: BDD 96
5.1 Requirements Engineering in Regulated systems 96
5.2 Agile Requirements Engineering in Regulated Systems 99

5.3 Background on Behaviour Driven Development 100
5.4 Literature Review Method 105
5.4.1 Use of Natural Language 106
5.4.2 Embrace BDD as a Holistic Approach 107
5.4.3 Role of Experience in Using BDD 107
5.4.4 Maintenance of BDD Specifications 108
5.4.5 Tool Support 108
5.4.6 Quality of BDD Specifications 109
5.4.7 BDD for Hardware 109
5.4.8 BDD for Regulated Systems 109
5.5 Discussion 110
5.6 Research Context 110
5.7 Threats to Validity 111

6 BDD in Practice: A Case Study 113
6.1 Objectives of the Exploratory Case Study 114
6.2 Context of the study 115
6.2.1 Overview of the Project and the Project Team Structure 115
6.2.2 Software Process Overview 116
6.2.3 Development Technology 118
6.3 Action Research 118
6.4 Semi-Structured Interviews 126
6.5 Discussion of the Limitations and Observations 129
6.5.1 Test First Development is Difficult to Apply 131
6.5.2 BDD Lacks Methods and Tools for Identifying and Refactoring Bad Smells 133
6.5.3 Gherkin Lacks Hierarchy of Features and Traceability 134
6.5.4 Identification of Appropriate Level of Abstraction is Difficult 135
6.5.5 Gherkin Does Not Support Multiple Actors in "*As A*" Statements . . . 136
6.5.6 Gherkin Does Not Support Concurrency of Execution 137
6.5.7 Convincing Developer and the Customer to Use BDD 140
6.5.8 Risk of Duplication of Effort in Large-Scale Systems 141
6.6 BDD in Theory vs BDD in Practice 142
6.6.1 Understanding 142
6.6.2 Collaboration 144
6.6.3 Acceptance Testing 146
6.7 Threats to Validity 147
6.8 Summary 150

7 An Analysis of the Practice of BDD on GitHub 152
7.1 Objectives of the Experiment . . . 153
7.2 Experiment Design . . . 154
7.2.1 Definition of exclusion / inclusion criteria: . . . 154
7.2.2 Repository Data Set Preparation . . . 157
7.3 Results . . . 158
7.3.1 Prevalence of BDD on Github . . . 158
7.3.2 Characterisation of Gherkin Projects . . . 159
7.3.3 Gherkin versus Non-Gherkin Projects . . . 166
7.4 Discussion of the Results . . . 173
7.5 Threats to Validity . . . 176
7.6 Summary . . . 177

8 An Analysis of Bad Smells in Gherkin Specification 180
8.1 Objectives of the Experiment . . . 181
8.2 Review of Bad Smells in Gherkin . . . 182
8.2.1 Gherkin Bad Smells identified in Peer-Reviewed and Grey Literature . 183
8.2.2 Mapping Bad Smells . . . 184
8.2.3 Experiment Design . . . 187
8.3 Results . . . 189
8.3.1 Arrange-Act-Assert vs Given-When-Then . . . 191
8.3.2 Multiple Assertions . . . 193
8.3.3 Duplication of Gherkin steps . . . 194
8.3.4 Lazy Steps Data Table: . . . 196
8.3.5 Lazy Scenario Outline . . . 198
8.4 Gherkin Specifications Bad Smells and Other Gherkin Artefacts . . . 200
8.4.1 Relationship with the size of scenarios . . . 200
8.4.2 Relationship with contributors . . . 202
8.5 Threats to Validity . . . 205
8.6 Summary . . . 209

9 Conclusions 211
9.1 Thesis Research Question 1 . . . 211
9.2 Thesis Research Question 2 . . . 214
9.3 Thesis Research Question 3 . . . 220
9.4 Contributions . . . 223
9.5 Interconnection of the Studies . . . 226
9.6 Scope and Validity . . . 227
9.7 Limitations . . . 228

9.8 Research Implications 230
9.9 Future Work 232
9.10 Summary 236

Bibliography **237**

A Interview Questions for use of Agile Methods **281**

B Interview Questions for Investigating Use of BDD **288**

List of Tables

3.1 Challenges reported in selected studies 48
3.1 Challenges reported in selected studies 49
3.1 Challenges reported in selected studies 50

List of Figures

2.1 Flow diagram mapping RQs to methods and chapters 11
2.2 The interview process 17
2.3 RP -> CRQ- > TRQ -> IQ: Wengraf's semi-structured interview model 18

3.1 Search strings 30
3.2 Literature sources for the literature review 30
3.3 Data extraction properties 32
3.4 Potential conflicts between agile principles and DO-178C 56

4.1 Research question construction using Wengraf's method 66
4.2 Summary of Interview Participants 67
4.3 A typical phase of a project from the perspective of the Software Team 71
4.4 Layers of Customers 72
4.5 Summary of the identified challenges 80

5.1 Feature template 102
5.2 Scenario template 103
5.3 Example of a scenario 103
5.4 Example of a code step 103
5.5 Literature Sources for the Second Literature Review 106
5.6 Search Strings 106

6.1 Use Case Template 117
6.2 Format of a Feature in Gherkin 119
6.3 Structure of a Scenarios with AAA violations 120
6.4 Research question construction using Wengraf's method 127
6.5 Summary of Interview Participants 128
6.6 Summary of challenges found during action research and interviews 132
6.7 Feature with multiple actors 137
6.8 Scenario with multiple actors 138
6.9 Scenario concurrent execution of steps 138

6.10 Implement "*When I run 100m"* as a thread . . . 139
6.11 Scenario with a complex step . . . 139
6.12 Scenario with concurrent execution . . . 140

7.1 Pipeline diagram . . . 156
7.2 Cumulative histograms of feature and scenario counts for 493 repositories . . . 160
7.3 Average scenario and step count . . . 161
7.4 Project Given-When-Then share . . . 161
7.5 Change in number of scenarios and their size versus project age . . . 162
7.6 Feature count growth versus project age histogram . . . 163
7.7 Average number of scenarios and steps versus the project feature and scenario count respectively . . . 163
7.8 Share of Gherkin and Non-Gherkin commits and their frequency over time . . . 164
7.9 Percentage of commits and LoC before first feature . . . 165
7.10 Change in practice of introduction of first feature in a project . . . 166
7.11 Comparison of characteristics of Gherkin and non-Gherkin projects . . . 168
7.12 Popular application languages in Gherkin and non-Gherkin projects . . . 169
7.13 Project durations for projects with at least 10 commits . . . 170
7.14 LoC at first feature histogram . . . 171
7.15 Histogram of contributors in Gherkin and non-Gherkin projects . . . 171
7.16 Developer's lifetime and earliest involvement in Gherkin projects . . . 172
7.17 Change frequency and commit history of Gherkin and non-Gherkin projects . . . 173

8.1 Search Strings . . . 183
8.2 Applicable bad smells in the context of BDD specification . . . 186
8.3 Projects' bad smells bar chart . . . 190
8.4 Contributor time to bad smell introduction . . . 190
8.5 Arrange-Act-Assert vs Given-When-Then . . . 191
8.6 Pie charts showing addition and removal of AAA pattern violations . . . 192
8.7 Pie charts showing addition and removal of multiple assertions . . . 193
8.8 Example of Duplication . . . 194
8.9 Pie charts showing addition and removal of clones . . . 195
8.10 Example of step parameters . . . 196
8.11 Example of a step data (text) . . . 196
8.12 Step function for scenario in Figure 8.11 . . . 197
8.13 Example of a step data table . . . 197
8.14 Pie charts showing addition and removal of lazy steps . . . 198
8.15 Example of a scenario outline . . . 199
8.16 Pie charts showing addition and removal of lazy outline tables . . . 199

8.17 Histograms of scenario sizes for repositories with and without selected smells . 201
8.18 Time series of project smell density 203
8.19 Scatter plots of project total smell counts against project gherkin contributor counts 204
8.20 Histogram of time working on a repository for commit authors who do and do not make changes to selected smells for 274 repositories with known history . . 206
8.21 Scatter plots of total smells introduced versus author days in a project for 201 projects 207

9.1 Future work research questions 232

Chapter 1

Introduction

This chapter introduces the background on Behaviour Driven Development (BDD), the motivation that guided this research, the thesis statement, and the research questions pursued during this Ph.D. This chapter is divided into five sections. Section 1.1 provides a background for the research; whereas, Section 1.2 provides the description of the motivation for investigating the challenges of applying an agile process, specifically, Behaviour Driven Development to the development of a project in a large-scale avionics company. Section 1.3 describes the thesis statement and the research questions answered during this research. Section 1.4 discusses the research contribution, and Section 1.5 presents an overview of each chapter in this thesis.

1.1 Background

Typical requirements engineering activities comprise elicitation, analysis, documentation, and review [Dick et al., 2017, Fricker et al., 2015, Kassab, 2015]. A variety of methods have been developed to elicit requirements from and with stakeholders including, prototyping, interviews, focus groups, etc. Similarly, a variety of methods for analysing and documenting requirements have also been proposed in both the peer-reviewed and grey literature, including, for example, business process modeling, gap analysis, requirements templates, user stories, etc. In general, the purpose of these activities and methods is to (i) facilitate communication between the stakeholders; (ii) establish a shared vision of the project among stakeholders; and (iii) inform project management and planning.

Nuseibeh and Easterbrook [2000] and many other researchers [Macaulay, 2012, Rost and Glass, 2011, Fricker et al., 2015, Lehtinen et al., 2014, Iqbal et al., 2020] argue that inadequate requirements are a major cause of project failure. A recent survey [The Standish Group, 2019] on the success of IT projects and project management best practices shows that 83.9% of IT projects partially or completely fail. The top factor in the failure of projects, according to the survey, is incomplete requirements.

These challenges are exacerbated in large-scale complex IT systems because of factors such

as organisational structure, size and nature of the project, the number of teams involved, and involvement of non-agile units [Inayat et al., 2015b, Kalenda et al., 2018]. A large number of stakeholders' involvement also affects the ability to communicate effectively in large-scale development context [Fucci et al., 2018].

To address these challenges, several researchers [Paasivaara et al., 2018, Abrar et al., 2019, Venkatesh and Rakhra, 2020, Kalenda et al., 2018] have suggested the use of agile methods for the development of large-scale systems because of their flexibility, and emphasis on communication and coordination. Despite the attempts to adopt agile methods in large-scale systems development, and their perceived benefits and success in small-scale projects, requirements management, shared understanding and communication among stakeholders consistently appear in the literature as challenges of agile requirements engineering in a large-scale development environment. [Inayat et al., 2015a, Vilela et al., 2017, Uludag et al., 2018, Dikert et al., 2016]. Kalenda et al. [2018] argue that "*scaling of requirements management cannot be avoided when scaling agile*". Kasauli et al. [2021] identify six themes related to the challenges of agile requirements engineering in large-scale systems during their multi-method study involving interviews, focus groups, and cross-company workshops. Four out of the six themes focus on the challenges related to requirements management, shared understanding, and communication. The remaining two themes focus on the process and organisational aspects of a project. In a Systematic Literature Review by Dikert et al. [2016], requirements management, communication, and coordination are among the top challenges of agile requirements engineering in a large-scale context. The recent studies list requirements management and shared understanding of a project among the major challenges in the application of agile methods in large-scale environment [Dikert et al., 2016, Uludag et al., 2018, Steghöfer et al., 2019, Kalenda et al., 2018, Kasauli et al., 2018b].

The nature of the challenges reported in the literature implies that mere adoption of an agile process does not solve the challenges pertaining to the requirements engineering of large-scale systems. A systematic mapping study by Curcio et al. [2018] also identified requirements management in large-scale systems development as a research gap. It is, therefore, important to investigate the challenges related to requirements management, communication, and shared vision in a large-scale development context.

Although it was originally conceived for acceptance testing purposes [North et al., 2006], Behaviour Driven Development (BDD) has recently gained popularity and appeared as an agile method that promotes flexibility and shared vision through ease of communication and requirements management [Oliveira and Marczak, 2018, Wang and Wagner, 2018, Moe, 2019]. The annual industrial survey reports on the state of agile from the last eight years show a gradual increase in the popularity of BDD [CollabNet VersionOne, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]. The survey reports show that more than a fifth of teams are incorporating BDD.

BDD is a test-first approach which focuses on writing acceptance tests for requirements specification before writing production code [Oliveira and Marczak, 2018, Irshad et al., 2021, Smart,

2014]. Advocates of BDD argue that the focus on requirements specification in BDD helps in avoiding waste of effort *i.e., building something that is not required* [Oliveira and Marczak, 2018, Smart, 2014]. A recent survey shows that practitioners believe that BDD provides ease of understanding by employing a ubiquitous language (i.e., Gherkin) for expressing requirements specification in the form of agile user stories [Binamungu et al., 2020].

Gherkin uses a structured natural language to describe requirements specifications in the form of features and scenarios. A feature in BDD is an agile user story, describing a single functional behaviour; whereas, a scenario is a specific example of the feature being performed with a system. Existing literature on BDD [Oliveira and Marczak, 2018, Dees et al., 2013, Smart, 2014, Irshad et al., 2021] recommends writing features and scenarios in a way that focuses on a required functionality instead of delving into how that functionality is achieved. Smart [2014] argues that the use of natural language and obscuration of technical details help the non-technical stakeholders understand *what is going to be built*. According to opinions of BDD practitioners reported in a survey study [Oliveira and Marczak, 2018], requirements specification in BDD bridges the communication gap between technical and non-technical people, encourages communication, and creates a shared understanding of a project across team(s).

1.2 Motivation

Despite the benefits, the BDD process itself requires an investment of time and effort to produce and maintain BDD-related artefacts in addition to all other project-related artefacts as a software project evolves. For example, Storer and Bob [2019] observe that the adoption of BDD necessitates simultaneous maintenance of both Gherkin specifications and corresponding step functions as system requirements and implementation evolve. More generally, the results of a survey focusing on challenges of BDD by Binamungu et al. [2018b], show that the area which needs most attention in the context of BDD is maintenance of BDD specifications. Six out of ten themes of the challenges discussed in the survey are related to the maintenance of BDD specifications. These themes include, for example, difficulty in locating faults in large BDD suites, duplication detection, and difficulty in changing BDD suites. The survey demonstrates that the cost of maintaining BDD specifications is a significant concern for practitioners working in the software industry. The practitioners also believe that the change in BDD requirements becomes difficult as the requirements of a system evolve [Binamungu et al., 2018b].

The results of a survey study [Irshad et al., 2021] on the use of BDD in large-scale system development show that large-scale system development requires more time and effort because of the factors that define large-scale system's context. These factors include communication overhead between multiple teams and a large number of stakeholders, cross functional dependencies, parallel development, and effort estimation issues. The results of the survey show that the practitioners believe that the use of BDD in a large-scale context could be costly because of

the effort required to perform detailed analysis and model a large number of scenarios.

Several researchers [Storer and Bob, 2019, Binamungu et al., 2018b, Irshad et al., 2021, Binamungu et al., 2020] have pointed out the maintenance of BDD specification as one of the research opportunities. The existing research has explored various aspects of the challenges related to the maintenance of BDD specifications e.g., size of a project [Irshad et al., 2021], tool support [Storer and Bob, 2019], quality of BDD specifications [Binamungu et al., 2020], experience of applying BDD [Binamungu et al., 2018b] etc. However, to the best of our knowledge, no study has investigated the technical limitations of Gherkin and BDD specifications writing styles which could also create maintenance issues in BDD specifications.

Several researchers [Binamungu et al., 2020, Oliveira et al., 2019] argue that writing a good BDD test suite is a challenging task. The studies [Binamungu et al., 2020, Oliveira et al., 2019], however, are focused on the quality of BDD specifications. For example, as a result of a survey from BDD practitioners by Oliveira et al. [2019] has proposed a question-based checklist to assess the quality of BDD specifications. The maintenance issues emerging from underlying technical limitations of Gherkin and the ways people write requirements are still undiscovered.

In summary, challenges evident in industry concerning the use of BDD in large-scale systems development, coupled with the lack of research on this topic in available literature is the biggest motivation for this research. A case study in the early phases of this research identified *requirements management in large-scale systems for teams adopting agile process* as a real concern. We, therefore, chose to focus on studying the challenge from their perspective and conducted an action research case study to explore the application of BDD to a project. In order to generalise and compare the observations from the second case study, an experiment was conducted that extended the scope to include open-source projects on GitHub. This research helped in discovering technical limitations of Gherkin and studying bad smells (i.e., structural inflexibilities) in BDD requirements specifications due to the ways people write BDD requirements.

1.3 Thesis Statement

Ultimately, the focus of this research was to explore the practice of Behaviour Driven Development in a real-world software development context. Therefore, the thesis statement for the research described here is as follows:

The practice of Behaviour Driven Development, its associated artefacts, and the intrinsic nature of the associated Gherkin language, incurs significant additional overhead for software engineers in terms of on-going maintenance in real-world software engineering environment.

However, the research presented in this thesis was developed in stages. The research was

initially motivated by the lack of case study research concerning the challenges of applying agile methods generally in large scale safety critical systems. Therefore, a study was developed in partnership with a large avionics company to explore this concern. The company was transitioning to the use of agile methods within their software development teams, so this process provided an opportunity to understand both the benefits and challenges of applying agile as perceived by the practitioners. Semi-structured interviews were conducted with members of several teams at different stages of the transition to explore the challenges faced during the application of an agile method within the company. The following research question was the focus of the first semi-structured interview study:

RQ1: What are the challenges of adopting agile methods in large-scale software development activities, particularly in regulated environments?

During this first study, the engineering and management of requirements in large-scale systems emerged as a key challenge. Based upon the findings of the initial study and in agreement with the case study partner, we decided to explore the application of Behaviour Driven Development (BDD) to requirements engineering within the company's software development process due to its association with agile methods and growing popularity in the industry. We conducted an action research case study in the same company with the aim of exploring the challenges of applying BDD to the development of a project in the company. The study was focused on the following research question:

RQ2: What are the challenges of adopting Behaviour Driven Development for the purposes of requirements engineering and acceptance testing in the early phases of a software project?

The action research case study yielded the discovery of technical limitations of Gherkin (i.e., the language for writing BDD specifications) along with the challenges of applying BDD. However, the study was restricted to a single project in an organisation. We extended the scope of our research and decided to investigate open-source BDD projects on GitHub. The purpose of the study was to identify existing writing practices in BDD specifications that could potentially create maintenance issues in BDD specifications. The following research question was the focus of that study:

RQ3: What are the specification writing practices in the existing open-source BDD projects that could result in maintenance challenges in behaviour driven development software projects?

1.4 Research Contribution

The research presented in the thesis makes several contributions to the body of knowledge. The contributions include:

Exploratory Case Study of Agile Software Development for Safety-Critical Systems Projects This study significantly extends the existing evidence base for the application of agile software development within safety-critical systems engineering by investigating the challenges from the perspective of practitioners. We conducted four semi-structured interviews with employees of the company in a variety of roles in different software projects and with diverse experiences. The extent of the material generated from these interviews allowed us to gain significant insight. Specifically, we reported on how some teams within the company have employed an agile software process (Scrum) within a Waterfall process for the wider systems engineering project. We elaborated on this integration by describing how the teams have made necessary customisations to Scrum to fit within this process.

We described the successes that the teams have experienced in employing and adapting individual agile practices, such as, planning poker, continuous integration, automated static analysis, and code reviews, as well as, discussing where the use of agile software development has led to drawbacks. We also investigated the practices that the teams have not employed, such as pair programming and user stories, and discussed the rationale for this from the teams' perspective. Where appropriate, we related these insights to the available literature. The work, therefore, provides a substantial case study based on evidence from an industry of the real-world challenges of employing agile software development for safety-critical systems and establishes a foundation for future research in addressing these challenges.

Participatory Action Research Case Study of Behaviour Driven Development The action research case study contributed to the body of research in four ways. First, this study was an industrial study. Since there is a lack of empirical studies exploring the challenges and technical limitations of applying Behaviour Driven Development (BDD) in a large-scale organisation, the study significantly extended the evidence base for the empirical studies on the application of BDD.

Second, the action research explored the technical limitations of BDD. The action research study was a *walk through* of the application of BDD to a project. It explored the technical limitations of BDD in representing some of the requirements for example: lack of support for concurrent execution of actions by multiple actors. During the action research, we also discovered that BDD is helpful in the validation of requirements and evaluating their consistency.

Third, the semi-structured interviews presented an overview of the challenges experienced by the practitioners during the application of BDD to a project. Through the semi-structured interviews, we were able to highlight the real-life challenges in the application of BDD due to the factors like experience of the team members, workflow, number of stakeholders, and organisational structure and culture.

Fourth, the overall study enabled us to present a comparison between *BDD in theory* and *BDD in practice*. We learned that the theoretical description of BDD and its steps do not take the

real-life factors into consideration e.g., organisational structure and culture; and team's experience and familiarity with BDD. The outcomes of the steps involved (in the theoretical framework of BDD) are based upon assumptions and ideal circumstances. The comparison between the theoretical framework of BDD and BDD in practice helped us in understanding the practicality of the *recommended steps* in BDD.

Empirical Study of BDD Characteristics in Open Source Projects The study presents the state of BDD in open-source projects on GitHub. The purpose of the study was to give a bird's-eye view of *BDD in practice*. The study presents: *(i)* an overview of the open-source Behaviour Driven Development projects; *(ii)* a comparison between the BDD and non-BDD projects; *(iii)* the evolution of different (BDD) projects related artefacts. The study uses the projects' metadata to present an overview and the growth of different project-related artefacts by analysing the commit history of the projects. The study also draws the statistical relationships between various project-related artefacts.

Online Experiment to Explore the Nature of Bad Smells in Open-Source BDD Project's Specifications To the best of our knowledge, there is not a single study that has investigated the open-source BDD projects to identify the existing BDD specifications writing practices and patterns that could adversely affect the maintainability of BDD requirements specifications. This study is a foundation stone of the evidence base for the empirical studies on the discovery of existing bad smells in open-source BDD projects' specifications.

First, we analysed the existing literature on bad smells (i.e., structural inflexibilities of project artefacts) in software engineering. Then, the bad smells, which appeared applicable to BDD requirements specifications, were identified. Next, we performed a feasibility assessment of the identified *bad smells* to pick out the bad smells for which a further investigation was possible within the time limit of this Ph.D. We selected a number of *applicable bad smells* for further investigation and discussed the rest of the *applicable bad smells* as an opportunity for the future.

This study discussed the impact of each of the finally selected bad smells on BDD specifications. The study also presented the percentage of their existence in the open-source BDD projects. Through this study, we have attempted to identify the existing BDD specification writing practices and patterns that must be avoided.

1.5 Thesis Overview

The objective of this research was to investigate the limitations and challenges during the use of agile, specifically, Behaviour Driven Development (BDD) in practice.

Chapter 2: presents the description of the research methods that were used during this Ph.D research. The research methods include a systematic literature of the relevant studies, exploratory case study, action research and interviews within a large-scale avionics company, and an online experiment.

Chapter 3: serves as a background for Chapter 4. It examines the relevant literature related to the use of agile process in large-scale safety-critical systems. The chapter includes a Systematic Literature Review on the challenges of applying agile methods in a large-scale safety-critical systems development context. The chapter presents the challenges reported in the literature on the use of agile methods for the development of large-scale safety-critical systems.

Chapter 4: is an exploratory case study on the use of agile in a large-scale avionics company. The case study identifies the challenges in the use of an agile process in large-scale safety-critical systems development. The chapter includes a discussion on the organisational hierarchy, team structure, and overall context. The chapter also presents the results from the semi-structured interviews in the form of the challenges related to the use of agile process in the company.

Chapter 5: serves as a background for Chapters 6, 7 and 8. The chapter discusses the concept of Behaviour Driven Development (BDD). The chapter presents an overview of the literature on Behaviour Driven Development (BDD). The chapter also discusses the BDD process.

Chapter 6: is an action research case study for the incorporation of BDD into a project at the company. The chapter discusses the incorporation of BDD; the technical limitations of Gherkin; the context of the study; and the results from the post hoc semi-structured interviews which were conducted to learn from people's experience of using BDD. The chapter presents a summary of the technical limitations of Gherkin and the challenges faced by the team in using BDD. The chapter also presents a comparative analysis of BDD in theory and BDD in practice.

Chapter 7: gives an overview of the open-source BDD projects on GitHub. The chapter discusses the development process of a tool for retrieving the metadata from the open-source BDD projects on GitHub. The chapter presents the bird's-eye view of the open-source BDD projects; their comparison with non-BDD projects; and the growth and evolution of different BDD-related project artefacts.

Chapter 8: discusses different types of applicable bad smells in BDD specifications. The chapter presents a summary of the existing bad smells in the open-source BDD projects on GitHub. The chapter also discusses the implications of such bad smells on BDD specifications.

Chapter 9: presents the conclusions to the research questions discussed in Section 1.3. The chapter also discusses the limitations of this research work and the suggestions for future research.

Chapter 2

Research Methodology

This chapter describes the research methods used during the course of this Ph.D research. These research methods include literature review, case study, action research and (online) experimentation. Section 2.1 provides the justification for the choice of each research method. Section 2.2 through to Section 2.5 describes the use of literature review, case study, action research and (online) experiment in detail. Section 2.6 presents the summary of the chapter.

2.1 Justification of the Research Approach

Figure 2.1 shows the flow of the progress of this research, the relationship between the research questions and the research methods used during this research. The first stage of the research was a systematic literature review in the broad area of agile methods in safety-critical systems. This revealed that a body of the academic literature considers agile methods in their traditional form unsuitable for the development of large-scale safety-critical systems mainly due to the rigour, production of heavy documentation and following of strict procedures required to develop these systems. Following this, it was decided to investigate whether these issues could be confirmed in a real industrial setting. A case study was conducted in a large avionics company to report their experience of using the Scrum agile method. During the case study, requirements management in agile development of large-scale safety-critical systems appeared as a major concern. This concern was further investigated with the help of a literature review on Behaviour Driven Development (BDD) followed by an action research case study on the use of BDD for the development of a project at the (same) company. The choice of BDD was influenced by its apparent popularity for requirements management and enhanced communication. During the study, we learned that the language used for describing BDD scenarios (i.e., Gherkin), provides a lot of freedom which could potentially allow developers to write specifications that are functionally correct but structurally problematic. The lessons learned from the action research study were limited to an early phase of a single case study; therefore, we decided to extend the scope of the study to open-source BDD projects on GitHub.

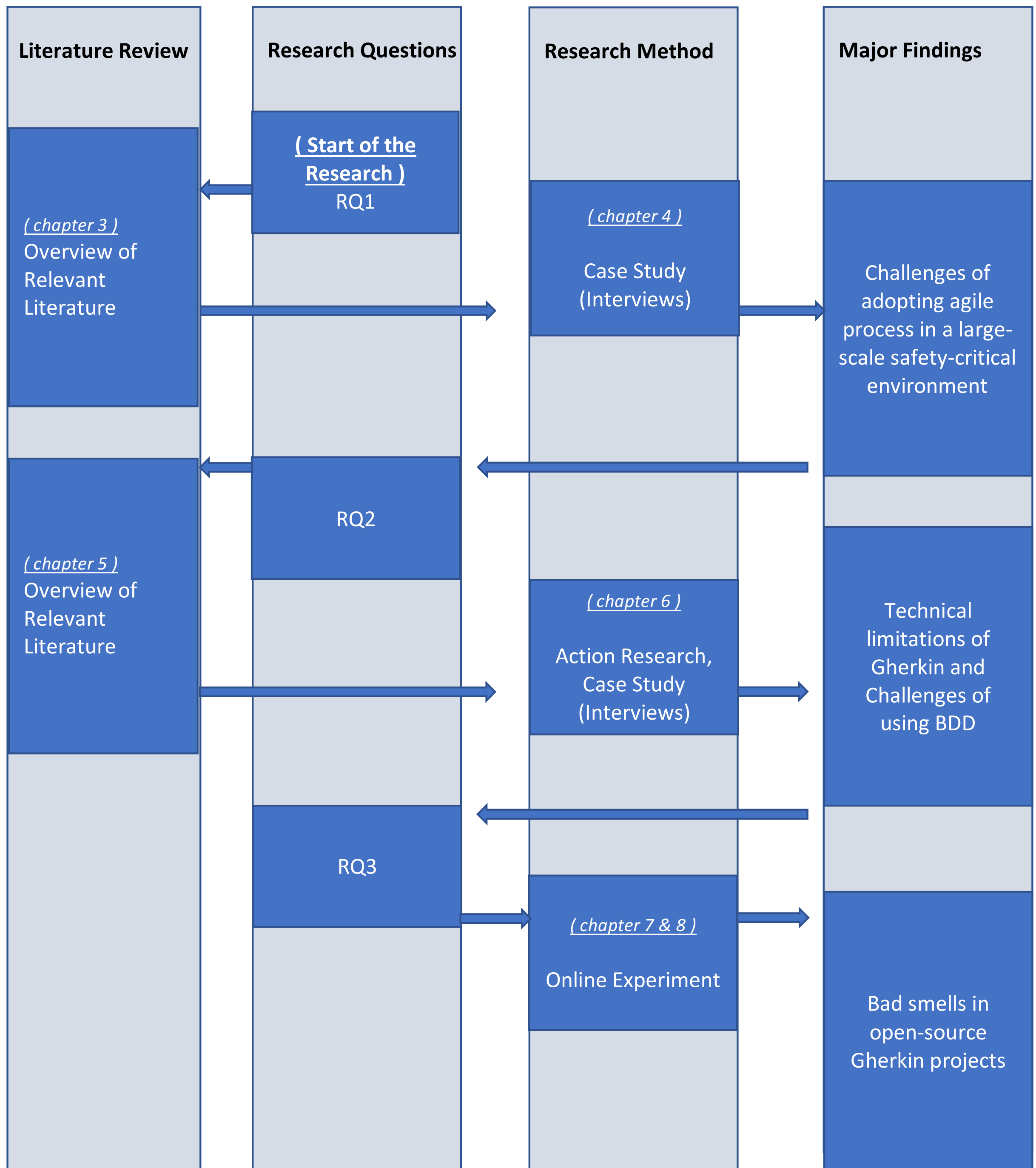


Figure 2.1: Flow diagram mapping RQs to methods and chapters

The choice of research methods during the course of this Ph.D was driven by the circumstances as the programme of studies progressed. The research was based on the knowledge gained from the literature review. The analysis of recent and past literature helped us in laying the foundation of this Ph.D research. The analysis of the relevant literature not only helped us in identifying the research gap but enabled us to construct the objectives of this research. The literature review helped in identifying the lack of empirical research in the area we were initially interested in i.e., *application of agile methods in large-scale safety-critical systems*. The curiosity behind the question "*What happens in real life?*" made us want to conduct our research in an industrial setting. The aim was to report the challenges of applying agile methods to large-scale system development.

The author was fortunate to find an industrial partner for his research. The industrial partner was a large avionics company in the UK, and they were already experimenting with an agile method (i.e., Scrum) for the development in several of their projects. We had an appointment based access to the employees of the company (i.e., the industrial partner). Due to the confidential nature of the organisation's projects, we did not have direct access to the projects, the site of the development or any project related documentation. The data we collected focused on opinions and experiences of applying Scrum in their own context in the form of semi-structured interviews.

The second case study with the same company was focused on learning the impediments in applying Behaviour Driven Development (BDD) to the development of a project at the company. The project team at the company was unfamiliar with the use of BDD, so action research was the best suited approach. Action research is an "*on the spot*" process which aims at taking corrective actions for bringing improvement in a situation [Chu and Ke, 2017]. Following the action research method, we were able to incorporate BDD in the development process of a project by constantly suggesting improvements based upon ongoing evaluations and observations. Conducting the action research allowed us to observe the limitations of using the Gherkin language for BDD in practice

Our choice of an online experimentation was driven by the need to extend the scope of the study and the desire to generalise the findings to other projects. We took the motivation from a few existing studies [Kalliamvakou et al., 2016, Chong and Lee, 2018, Ortu et al., 2018, Munaiah et al., 2017, Cosentino et al., 2017, Borle et al., 2018, Vendome et al., 2017] and decided to investigate a random sample of open-source projects on GitHub to understand BDD in practice.

2.2 Literature Review

A literature review is performed to identify the "*state of the art*" in a field of interest [Walliman, 2017]. A literature review also helps in identifying the gap in research and provides an

understanding of how new work will extend the state of the art [Walliman, 2017]. According to Snyder [2019], a literature review is a way of synthesising research findings and uncovering areas where more research is needed.

Snyder [2019] categorised the literature reviews as *(i)* systematic review: aims to identify all empirical evidence to answer a particular research question or hypothesis, *(ii)* semi-systematic review: explores how research within a selected field has progressed over time, *(iii)* integrative review: assesses or critiques the literature on a research topic in order to propose new theoretical frameworks or suggest improvements to existing frameworks.

The literature reviews during this research were performed in progressive steps. As this Ph.D research progressed, the focus of the literature review also progressed. The initial focus of the literature review was on application of agile in large-scale and safety-critical systems. Then, on the basis of the first case study, the focus of a second literature review narrowed to Behaviour Driven Development (BDD). Two literature reviews formed different sections of Chapter 3 and Chapter 5.

We relied on the online libraries for doing the literature review. The first part of the literature review (i.e., Chapter 3) was a systematic literature review. Whereas, an exhaustive search for literature was performed, and a semi-systematic protocol was adopted for the next part (i.e., Chapter 5).

2.2.1 First Literature Review

According to Shokraneh [2019], irreproducibility of the research is a major concern in all fields of science. The literature reviews, if not documented properly, are seldom reproducible. In order to make the literature review reproducible, the author of this document used the systematic literature review guideline by Kitchenham et al. [2009].

The Kitchenham et al. [2009] guideline for a Systematic Literature Review (SLR) describes a process for conducting a repeatable literature review, which also provides the guidelines on identifying, evaluating and interpreting the relevant research on a particular area of interest. The guidelines also propose methods for conducting a well planned study and provide recommendations for measuring the quality of a literature review and validity of its results.

The systematic literature review was focused on the use of agile methods in regulated environments, in particular, safety-critical systems development. It was helpful in understanding the current state of issue(s) and solutions related to the application of agile in large-scale safety-critical systems.

2.2.2 Second Part of the Literature Review

The first part of the literature review provided a background for the first case study. The result from the first case study suggested that requirements engineering in the large-scale systems was

an issue and needed further investigation. Therefore, our second literature review was focused on agile requirements engineering, specifically, Behaviour Driven Development (BDD). BDD is an agile method which is based upon requirements, their elaboration and their management. The second literature review served as the background knowledge for the second case study.

During an initial review of the literature, we identified a lack of empirical research on BDD. Snyder [2019] recommends a semi-systematic review of the literature where the intention is to present a narrative overview of the research area. We, therefore, adopted this approach prior to conducting our own empirical research on BDD.

2.3 Case Study

Thomas [2015] defines the case study as a research which concentrates on a person, group, institution, country, an event or a period of time. Ridder [2017] calls it investigation of "*a real-life phenomenon in-depth and within its environmental context*". Unlike experimentation, the case study research does not have a controlled environment. Instead, the environmental context is a part of the case study research investigation [Ridder, 2017]. Thomas [2015], while defining the case study, says "*it is not a method in itself... rather it is a focus on one thing*". According to the author, the results from a case study should not be generalised since the purpose of a case study is to investigate a phenomenon in relation to a single entity (i.e. person, group, an organisation etc).

The case study approach was used twice during the course of this Ph.D research. The case study was a suitable method in our context because we wanted to learn from a real life example of the use of an agile method in an industrial setting, conduct the study in an uncontrolled environment, and report the experience and opinions of the people using the method in a real project. The first case study explored the use of agile process in a large-scale environment; whereas, the second case study explored the use of Behaviour Driven Development (BDD). Both case studies were conducted in a large avionics company in the United Kingdom (referred to as "*the company*"*) throughout this document.

The company as a whole was engaged in a variety of projects for external customers, typically comprising both hardware and software development for safety-critical systems. Although they had a previous collaborative relationship with the supervisor of the author, their selection as a research partner was driven by mutual interests. The company was already exploring the use of agile in a large-scale regulated environment which made them an ideal *case* for the research.

Negotiations led to a non-disclosure agreement signed between the parties involved in the research due to the sensitive nature of the work at the company. An unstructured interview was conducted with two employees at senior positions in the company as a preliminary step. The interview was focused on understanding the overall structure, processes and the workflow in the

*name of the company redacted because of the non-disclosure agreement between the research partners

organisation. This interview formed the basis of the objectives and the design of both (i.e., first and the second) case studies.

The guidelines by Hancock and Algozzine [2017] and Yin [2011] were followed for execution of the case study research process which involved following activities:

- *Selecting a design*: Objectives are defined and case study design (i.e., exploratory, explanatory, or descriptive [Yin et al., 2003]) is selected.
- *Data collection*: Procedures and protocols for data collection are defined and executed.
- *Analysis*: First, the collected data is organised in a logical and a manageable format. After that, interpretation is performed on the data.
- *Reporting*: The reader is given a comprehensive view of the focal issue and the findings of the case study.

The first case study was an exploratory study. The objective of the case study was to understand and explore the application of agile in the context of an industrial, large-scale regulated environment. A series of semi-structured interviews was conducted to collect qualitative data, and the focus of the interviews was on the discovery of challenges and impediments in the application of agile. The data was analysed using Wengraf [2001] guideline.

The second case study was also an exploratory study. Its objective was to explore the use of BDD in an industrial setting. Four iterations of action research were performed before conducting the semi-structured interviews. The use of action research method not only aligned the BDD workflow with an ongoing development process but was also instrumental in formulating the interview questions for the second case study. The focus of the interviews was on the discovery of impediments in the application of BDD.

2.3.1 Semi Structured Interviews

Different sources of information for collection of evidence during a case study [Yin et al., 2003] include documents, email correspondence, notes, direct observations etc. Interviews are also considered one of the important means for collection of evidence in a Case Study [Holstein et al., 2002, Yin et al., 2003]. Interviews are generally categorised under qualitative research methods, and they are used to explore practices, views and beliefs. Research data from interviews often comprises of lengthy explanations.

Interviews do not have a single standard definition, and a number of different methods to conducting them have been proposed. According to Wang [2015], an interview is interactional communication process between two parties, where one of the parties asks pre-determined questions. Walliman [2017] considers interviews as one of the methods for collecting qualitative data

from people's experiences and recollections. According to Gubrium and Holstein [2001], "*interviews give access to observations of others*". We can say that interview is a form of an interaction which lets the interviewee narrate the account of his/ her experience in his/ her own words.

Semi-structured interviews are *in-depth interviews* where respondents are asked about facts and their opinions on a matter of interest [Yin et al., 2003]. In these types of interviews, the respondents may be considered as "*informants*" [Yin et al., 2003] because they can provide insight and refer to corroboratory sources of evidence. Unlike structured interviews, where the interviewer asks a set of pre-determined questions, semi-structured interviews are flexible and their use encourages reciprocity between the interviewer and the participant [Kallio et al., 2016]. Semi-structured interviews give the freedom of expression to the participants, and open ended questions prompt discussion which helps the interviewer to explore a particular theme and improvise new questions during the interview.

Use of the semi-structured interviews helped us to explore, investigate and learn the practices and the behaviours followed in the company, particularly, the company's experience of using the software development methods studied during this research. Two sets of semi-structured interviews in the company were conducted during the course of this research to gather opinions in the form of qualitative data. The first series of interviews was conducted during the first case study to explore and learn the use of agile methods in the company. The second series of interviews was conducted during the second case study to learn the company's experience of using Behaviour Driven Development (BDD).

Figure 2.2 provides a visual summary of our semi-structured interview process. Wengraf [2001]'s guidelines were used for preparing the questions for the interviews. After carefully reviewing the questions and getting them validated from an independent expert, mock interviews were conducted to estimate the average duration of the interview and adjust the sequence of the questions.

2.3.2 Wengraf's Method

Wengraf [2001] provides a guideline for the development and analysis of a semi-structured interview questions. Figure 2.3 is a graphical representation of the Wengraf's model. The model is a hierarchical approach consisting of four steps. First step is to determine the Goal (i.e., Research Purpose) of the interview. The next step is to determine "*What do you need to know?*" in order to reach the Goal of the interview. To achieve this, RP is refined as one or more objectives called Central Research Questions (CRQs) that encompass the broader aspects of the research purpose. The description of the information required to understand each aspect is documented in the form of a separate objective at this stage.

In the third step, each CRQ is divided into a number of Theory Questions (TQ), specific propositions to be investigated during the conduct of the study. Theory questions "... *are for-*

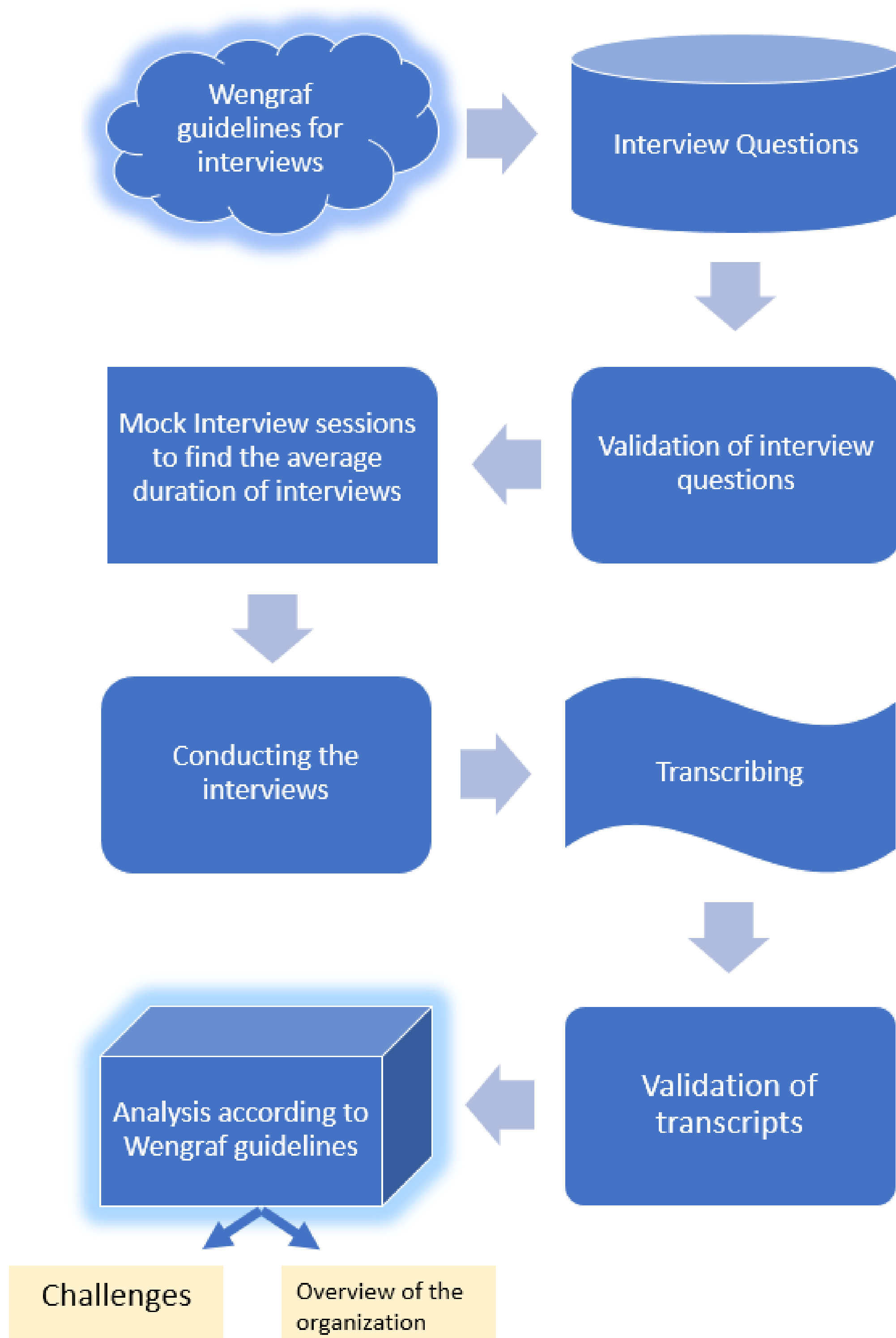


Figure 2.2: The interview process

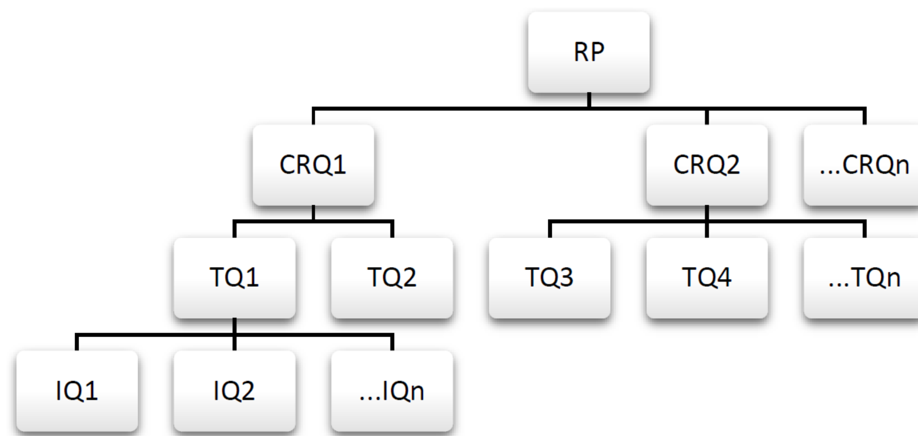


Figure 2.3: RP -> CRQ- > TRQ -> IQ: Wengraf's semi-structured interview model

mulated in the theory-language of the research community" [Wengraf, 2001]. The information required to answer each proposition is further divided into a set of interview questions in the fourth step. In this step, the set of suitable interview questions (IQs) for each TRQ are developed. The interview questions (IQs) are "... *formulated in the language of the interviewee"* [Wengraf, 2001]. This approach provides a logical hierarchy and rationale behind every interview question.

Analysis of the gathered data is also performed by using Wengraf [2001] guideline by answering the questions backwards with the help of the following formula 2.1 where IM is the interview material, ATQ is the answer to each theory question which will answer the Central Research Questions.

$$IM- > ATRQs- > ACRQ \tag{2.1}$$

Answers (IM) to the Interview Questions (IQ), related to a single Theory Question (TQ) are combined to form a story. This story answers a Theory Question. Then, the answers to each Theory Question related to a single Central Research Question (CRQ), are combined to answer the CRQ. The descriptive answer of each CRQ is then combined to describe the purpose of the research.

Once the interview instrument was developed, mock interviews were conducted to estimate the duration of the interviews. Next, the interviews were conducted, transcribed, and the transcriptions were validated with the interviewees in case they wanted to add or omit anything. The analysis was performed using the Wengraf [2001] guidelines through which we reached the "*Goal"* of the interview. The guidelines also assisted in drawing the context of the interviews.

2.4 Action Research

Action research is also called learning by doing [Riley and Moltzen, 2011]. It involves conducting research alongside implementing actions or changes within an environment where circum-

stances require flexibility [Somekh, 2005]. Action research enables the practitioners to study and improve aspects of practice [Tran, 2009, Koshy, 2005]. Understanding the context, planning, and implementing a corrective action are the main purposes of being engaged in action research [Koshy, 2005]. Action research is suitable in settings where practitioners seek to "*select a new initiative, study its practical implications, consider ways of implementing the ideas, and evaluate and make decisions*" [Koshy, 2005]. Gunbayi [2020] categorises the action research into three types i.e.,

- **Technical Action Research:** *The researcher puts an action into practice, and the practitioner follows the instructions of the researcher.*
- **Participatory Action Research:** *The researcher and practitioner put an action into practice together.*
- **Emancipatory Action Research:** *The practitioner is given new knowledge and skills to gain a critical perspective towards the practitioner's own practices.*

The need for action research arose at the start of the second case study when the company wanted to employ a development process which could help them with better management of requirements. Our intention was to enable the team to employ BDD and study the impact of BDD on their development processes. Participatory action research design was adopted to collaborate with the practitioners in order to align BDD with their development processes. Participatory action research is suitable in the situations where the focus is on capacity building [MacDonald, 2012]. It involves the use of self-reflective cycles to achieve a social change [Savin-Baden and Wimpenny, 2007]. The work proceeded in four phases of action research, with data gathered throughout.

We followed the participatory action research guideline described by Kindon et al. [2007]. According to Kindon et al. [2007], (participatory) action research is an iterative method with five main steps: *(i) Diagnosis*, *(ii) Planning*, *(iii) Action*, *(iv) Analysis*, *(v) Reflection*. Please note that from this point onwards wherever we use the term "*action research*", it will mean "*participatory action research*".

Diagnosis concerns the identification of the underlying problem which cultivates an organisation's desire for change [Baskerville, 1999]. According to Danley and Ellison [1999], *diagnosis* in participatory action research identifies the gaps in knowledge and skill of the team members.

The first case study (as described in Chapter 4) in this thesis provided the information which helped in *diagnosing* the problem i.e., requirements management in large-scale systems is a challenge. The company wanted to deliver earlier and improve its project development process by using a method which could help them with their requirements management process.

Planning in action research describes the planning process of the actions that could resolve the issues identified during *diagnosis*. According to Kemmis et al. [2013], planning includes identification of the stakeholders, provision of monitoring the action, shared concerns, possibilities and limitations. Chevalier and Buckles [2019] refer to planning as a blueprint for systematic action.

Planning activities were undertaken in collaboration with representatives from the company throughout the action research. This took place either through email or in-person meetings. For example, after a considerable number of email exchanges and several meetings between the author and the company representatives, it was mutually decided to explore Behaviour Driven Development (BDD) for development of a project because of its apparent popularity for requirements management, and communication through requirements specifications.

Action in this context describes the implementation of the plan for improvement. According to Mertler [2009], action (in this context) is also a process of data collection. The *new state* or change that arises (as a result of this phase) serves as an input (i.e., data) for the next phase (i.e., analysis) in action research.

The *action* in our context was primarily focused on requirements elicitation, refinement and documentation of requirements of the project we agreed to work upon with the company. For example, the development team at the company already had an existing set of user stories before we decided to collaborate with them. In order to incorporate BDD, these user stories were converted into BDD requirements (feature) format as a part of the *action taking* during the first iteration of action research.

Analysis involves understanding and interpretation of the data collected during or as a result of the *action*. According to Johnson [2008], "*...as you collect your data, analyse them by looking for themes, categories, or patterns that emerge. This analysis will influence further data collection [and analysis] by helping you to know what to look for"*.

Qualitative data in form of observations were collected during the four iterations of the action research. For example, during the conversion of the existing requirements into the BDD requirements (feature) format in the first iteration of action research, discrepancies in the existing set of requirements were observed. Removal of those discrepancies revealed a number of limitations of Gherkin. These observations were documented. In addition, more qualitative data in the form of observations regarding BDD and the use of Gherkin were collected during the interviews.

Reflection, in this context, implies the corrective actions needed on the basis of the observations made during analysis. According to Mertler [2009], reflection is a corrective action needed at the end of a particular action research cycle to make improvements to present context (of the action research) to reach better results.

Corrective actions, on the basis of analysis, were suggested at the end of each of the four iterations. These suggestions became the basis for planning for the next iteration. For example, as a corrective action during the first iteration of action research, we suggested removal of the structural anomalies in the existing set of requirements, which led to the need for breaking up the requirements into smaller requirements.

2.5 Online Experiment on GitHub

The design, nature and environment of an experiment is dependent upon the aim of the study [Juzgado and Moreno, 2001]. The aim of our study was to conduct an experiment using *uncontrolled* historical data of open-source projects. To be able to include and access a large number of projects' data, it was decided to use an online platform.

Online platforms are increasingly being used by researchers and analysts for conducting online experiments and collection of data for different purposes [Newman et al., 2021]. Examples of these online platforms include: Prime Panels[†], Study Response[‡] and Prolific Academic[§]. In spite of GitHub[¶] being an online version control system, the meta-data associated with projects has drawn interest of the researchers. Many studies [Kalliamvakou et al., 2016, Chong and Lee, 2018, Ortu et al., 2018, Munaiah et al., 2017, Borle et al., 2018, Cosentino et al., 2017, Vendome et al., 2017, Bao et al., 2019, Sharma et al., 2017, Goyal et al., 2018, Qi et al., 2017] have been conducted that used project related meta-data from GitHub to study the patterns, behaviour of the developers, practices and various other phenomena such as communication patterns [Ortu et al., 2018], effects of test driven development [Borle et al., 2018], and software license usage [Vendome et al., 2017].

GitHub provides a number of resources like *GitHub API*, through which project related meta-data can be collected for analysis. Project related information on GitHub includes number of commits, contributors, development languages, issues, pull requests etc. Researchers and data miners analyse this meta-data to draw various statistical results [Munaiah et al., 2017, Borle et al., 2018].

The scope of the case study (discussed in Chapter 6) was limited to a single project. The focus of this case study was on finding challenges of BDD. To extend the scope of our investigation, we decided to conduct an online experiment using open-source projects on GitHub. The purpose of the study was to develop an understanding of the practice of Behaviour Driven Development in open source software projects. The study used evidence available through the history of changes made to relevant artefacts in project version control repositories, i.e. Gherkin feature files.

[†] https://www.cloudresearch.com/products/prime-panels
[‡] http://www.studyresponse.net
[§] https://www.prolific.co
[¶] https://github.com

Extending the scope of our research to open-source projects on GitHub exposed us to a large number of open-source projects, ranging from new to established projects. Investigation of open-source projects helped us in studying the overall practice of BDD and investigating the relationship between various project artefacts. The term "*experiment*" was inspired from similar studies [Sharma et al., 2017, Ortu et al., 2018, Bao et al., 2019] that used GitHub to collect meta-data. We replicated the steps which were common their [Sharma et al., 2017, Ortu et al., 2018, Bao et al., 2019] experiment design i.e., *(i)* defining the aim of the experiment, *(ii)* devising an exclusion/ inclusion criteria *(iii)* data extraction and *(iv)* drawing the results. The *aim* of the experiment was to study the application of BDD in open-source projects on GitHub.

2.6 Summary

This chapter presented an overview of the research methods used during the course of this Ph.D research. The research methods included two literature reviews, each followed by an exploratory cases study. In addition to the exploratory case study, the second literature review was followed by action research and experimentation.

The first literature review was a systematic literature review, conducted to understand the challenges of applying agile methods to the development of large-scale safety-critical systems. Whereas, the second literature review was a semi-systematic literature review. Its purpose was to understand and explore the state of the art research on BDD. Each literature review was followed by an exploratory case study on the same topic. Whereas, action research was used for aligning the BDD process with the development process of the project, investigated during the second case study.

The online experiment, using open-source projects on GitHub, was an extension of the investigation on BDD. Analysis of meta-data from the open-source project on GitHub helped in understanding the BDD in practice, and studying different project related artefacts and relationship between them.

Chapter 3

Systematic Literature Review on Agile Methods for Large Scale Safety Critical Systems

This chapter presents a systematic literature review on the use of agile methods for development of large-scale safety-critical systems. It also provides a theoretical background for Chapter 4. The focus of systematic literature review was on the challenges of using agile method for development of large-scale safety-critical systems.

Section 3.1 consists of sub-sections which provide an introduction of the concept of agile software development of safety-critical systems. Section 3.1.1 presents a brief overview of agile software development followed by an introduction on safety-critical systems in Section 3.1.2. Section 3.1.3 presents a background on agile software development of safety critical systems. Section 3.2 describes the literature review process and protocol. Section 3.3 consists of sub-sections that divide the selected studies into major themes including: studies advocating the use of agile for safety-critical systems development, studies focusing a particular aspect of agile development of safety-critical systems, and studies that suggest tailoring in agile methods. The challenges found in literature on agile development of large-scale safety-critical systems were also divided into major themes that are discussed in Section 3.4. Section 3.5 consists of the discussion on the findings of the literature review. Section 3.6 discusses threats to validity of this study. Section 3.7 presents a summary of this chapter.

3.1 Agile Development and Safety-Critical Systems

This section provides an overview of agile software development and its relationship to safety critical system development. It serves as an introductory background for the systematic literature review and builds an understanding of the context.

3.1.1 Agile Software Development

Agile software development emerged in the late 1990s and is considered to be a response to the failure of existing plan based software development processes, such as Waterfall [Benington, 1983, Vijayasarathy and Butler, 2016, Wang et al., 2012] and the Rational Unified Process [Software, 2003, Tanveer, 2015] to accommodate the highly volatile nature of requirements for software development projects. A common critique of these methods is that the lifecycle of software delivery is far slower than the pace of change in the problem domain [Schwaber and Beedle, 2001, Tanveer, 2015, Koronios et al., 2015, Abrahamsson et al., 2017]. For example, a typical iteration in the Rational Unified Process is between six and twelve months, during which time, the requirements for the project or the technology available in the market place may have changed considerably.

Proponents of an *agile* approach to software development [Beck et al., 2001a, Abrahamsson et al., 2017] instead advocate for a process model that is based on continual review of progress and requirements through continued close collaboration with the customer. Schwaber and Beedle [2001] and Abrahamsson et al. [2017] explain that this approach is derived from empirical process engineering, in which, rather than attempting to design a software process apriori, process engineers closely monitor and make small, frequent changes to the production process. As a consequence of this approach, a team practising agile software development will still begin work with a broad understanding of the long term objectives for their project, but will avoid detailed planning for all except the most immediate project activities.

Agile methods are a family of software process models that share this common agile philosophy. Examples of agile methods include Lean [Poppendieck and Poppendieck, 2003, Dingsøyr and Lassenius, 2016], Crystal [Cockburn, 2004], Feature Driven Development [Palmer, 2002], Extreme Programming (XP) [Beck and Andres, 2005] and Scrum [Schwaber and Beedle, 2001]. A unifying characteristic of these process models is that they are *iterative* and *concurrent*. Software development takes place within short iterations of typically two or three weeks, but sometimes as short as a single day, punctuated by deliveries to a customer for immediate feedback and review. In further contrast to plan-based methods, within each iteration, multiple software development activities may occur concurrently, including requirements analysis, design, implementation and testing. Each agile method is itself further characterised by a set of practices undertaken to support development work and manage the complexity of the concurrent software process. Examples include backlog grooming, planning poker, sprint planning daily standups and retrospectives from Scrum [Schwaber and Beedle, 2001]; spike prototyping, automated unit testing and refactoring in extreme programming, and value-chain mapping in Lean [Poppendieck and Poppendieck, 2003, Dingsøyr and Lassenius, 2016].

The software industry, as a whole, is witnessing a gradual transition from traditional plan-driven process models to agile software development [Chapman, 2016, Chapman et al., 2017, Glas and Ziemer, 2009, Paige et al., 2011, Wils et al., 2006]. A 2018 survey of software indus-

try practitioners found that 97% of respondents reported using agile methods [CollabNet VersionOne, 2019]. In addition, the survey found that 78% of respondents reported that the teams in their organisation continued to use a mix of agile and plan-based methods and practices. Advocates of agile software development contend that plan-driven software processes lack the flexibility to respond to rapidly changing business requirements [Beck and Andres, 2005, Beck et al., 2001a, Schwaber and Beedle, 2001].

Agile software development addresses this demand for flexibility by emphasising the organisation of work into small co-located teams, short development cycles punctuated by deliveries of software releases to customers for review and feedback, encouraging frequent informal communication amongst software team members and the exclusion of practices that do not demonstrably contribute value to the project customer, often including formal documentation [Black et al., 2009, Rayside et al., 2009]. Such values are embodied in a number of agile methods, such as Feature Driven Development [Palmer, 2002], Extreme Programming (XP) [Beck and Andres, 2005] and Scrum [Schwaber and Beedle, 2001]. Each agile method may also be characterised by a number of agile methods, such as daily standup in Scrum or pair programming in XP. Methods may also be customised by the addition of supplemental practices, or practices themselves may be customised to meet the demands of the project context.

According to industry surveys, Scrum and XP are the most frequently reported methods employed by software teams for organising an agile software development process [Wang et al., 2012, CollabNet VersionOne, 2019]. Schwaber and Beedle [2001], and Lei et al. [2017] state that the Scrum process works well for small teams of between three and nine members. Key roles within Scrum include the Scrum master, responsible for facilitating team activity and the product owner, responsible for managing the relationship between the customer and the team. The Scrum process comprises of short iterations called sprints, typically lasting 1-3 weeks. Each sprint begins with a planning meeting during which new requirements are transferred from the *product backlog* to the *sprint backlog*. The sprint begins once the requirements are agreed upon for the sprint backlog. Communication between team members is maintained through a daily meeting, called a *stand-up*, during which each team member briefly reports progress, plans and any issues that have arisen. At the end of a sprint, the team holds a *review meeting* during which progress is compared against the goals of the sprint.

The XP process, as described by Beck et al. [2001a] and Wang et al. [2012], has a similar focus on short iterations punctuated by releases to the customer. Similar practices to Scrum are also advocated for project management, such as a daily stand-up meeting and release planning for an iteration. However, in contrast to Scrum, XP practices focus on the lower level activities associated with software engineering. For example, XP advocates the use of user stories developed in user story workshops for requirements gathering; test driven development for both new features and bug fixes; and refactoring as an explicit practice to maintain code quality. Other practices are also recommended to foster team communication through pair programming. For

example, Schwaber and Beedle [2001] argue that the two methods are complementary and can co-exist in a single team with Scrum providing a *wrap around* for the practices within XP.

3.1.2 Safety-Critical Systems

Sometimes a certain characteristic of a software system is mandated by law or an organisational policy. These characteristics are so important for the operation of a software that without them, often the software system is deemed unusable. Such characteristics are considered critical to the business and consequently, the software. These software systems are referred to as *critical systems*. Regulations play a key role in constraining the specification, design, assurance and maintenance of a wide variety of *critical* software systems. Examples include: railway/ aircraft operating and control systems, electric power grid systems, first responder communications systems etc.

According to Pal and Karakostas [2021] critical software systems are "*software whose failure would impact safety or cause large financial or social losses*". Development of critical systems have to demonstrate compliance to a regulation if their malfunction or unintended use involves a risk of resulting in an outcome which is prohibited by law or a regulation [Heeager, 2014]. For example, the UK government's recent guidelines* on standalone medical apps and softwares which involve diagnosis, treatment or management of patients state that "*Standalone software and apps that meet the definition of a medical device are required to be CE, CE UKNI or UKCA marked in order to demonstrate that they are acceptably safe to use and perform in the way the manufacturer/developer intends them to*".

Safety-critical systems are one of the examples of critical systems. According to Knight [2002], "*Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment.*". Examples include nuclear systems, medical devices, air traffic control, avionics, railway control systems and automotive control systems. Due to the involvement of physical risks, development of safety-critical system development is typically undertaken within respect to particular generic or domain specific standards or other regulatory constraints [Heeager and Nielsen, 2018]. Such standards may impose considerable structure on the software development process including the selection and ordering of activities. Furthermore, standards may specify artefacts that must be produced during the development to show conformance. For example, DO-178C is a standard for development of airborne software. Similar standards exist for other domains, such as IEC 62304 for development of Medical devices, ISO 26262 for automotive and IEC 61513 for nuclear.

The purpose of following these regulatory standards is to demonstrate that a sufficient amount of rigour was applied during the development to ensure that the computer system is safe to use

*https://www.gov.uk/government/publications/regulatory-status-of-software-including-apps-used-in-the-diagnosis-treatment-and-management-of-patients-with-coronavirus-covid-19

[Lee et al., 2014]. For example, under the law of negligence† in the United Kingdom, the manufacturers (of safety-critical systems) owe a *duty of care* to ensure that the computer systems they supply are not likely to cause physical injury [Davis, 1995]. This implies that an uncertified safety-critical computer system (i.e., a safety-critical computer system which does not fulfil the legal requirements of safety) lacks the demonstration of element of *duty of care* as mandated by the law. Depending on the regulatory framework, supplying a safety-critical system which lacks the appropriate certification could place considerable legal liabilities on the vendor of the computer system [Myers et al., 2012, Lee et al., 2014].

Bell [2017] provides a historical perspective on the development of the field of safety critical systems. He argues that the field emerged as a distinct discipline in the 1970s, with initial focus on hardware. For nearly a decade, the focus of the experts of safety-critical systems was on hardware and mechanical components. As the role of software became increasingly important in the chemical process sector in early 1980, increasing attention was paid to the role of safety in programmable electronic systems [Parry, 1993]. From 1980-1990 guidelines (i.e., HSE PES guidelines) related to the safe use of programmable electronic systems were published, leading to the publication of IEC 61508 (regulatory standard for functional safety) in 1998.

Achieving certification for safety-critical systems may require performing additional activities during the development e.g., hazard analysis. Certification could also require production of significant amount of documentation as a means of demonstrating compliance with the regulatory framework. Due to the additional activities involved in the development of the safety-critical systems, it takes considerably longer to complete safety-critical system development [Kasauli et al., 2018a].

Several researchers [Winkler et al., 2012, Notander et al., 2013, Hatcliff et al., 2014] argue that there is a dominance of traditional approaches and sequential models in safety-critical systems development. NASA's Software Safety Guidebook [NASA] specifically recommends against using agile methods for safety-critical software development because of their "*low rigour*". Since traditional development models tend to deliver the product near the end of the project, the customers have to wait a long time before they can use the system [Myklebust et al., 2015, Abrahamsson et al., 2017]. There are two main problems with delivering a system after a long period of time. Firstly, the needs and wants of the customer change with the change in needs of the business; therefore, the delivered system may no longer fulfil the current business needs. Secondly, late delivery delays feedback, increasing the cost of fixing mistakes [Myklebust et al., 2015]. Therefore, there is increasing demand to adopt more frequent delivery cycles, or use of processes that enable more frequent delivery, such as those derived from agile principles.

Agile methods are based upon agile principles whose main focus is to replace heavy-weight time consuming procedures with light-weight activities and subtract the element of delay. Quick delivery, strong communication and accommodation of change are the core values of agile. Ag-

†https://www.legislation.gov.uk/ukpga/2006/29/notes/division/3/1

ile methods are widely praised for their benefits such as ability to adapt to changes [Mohammad et al., 2013, Khan and Beg, 2013, Paetsch et al., 2003, Coram and Bohner, 2005], quick development in short releases and progress visibility.

Despite the benefits, Agile methods are criticised for reasons such as inadequate support for large-scale and safety-critical projects, lack of "*proper*" documentation, and discipline [Martins and Gorschek, 2016, VanderLeest and Buter, 2009]. Development of safety-critical software requires detailed analysis in each phase that consists of long and careful procedures [Kasauli et al., 2018a]. Heavy documentation and certification is a mandatory part of the process in safety-critical system development [Notander et al., 2013, Heeager and Nielsen, 2018]. Therefore, it is important to know *if or not agile methods are suitable for use in regulated environments*.

3.1.3 Agile Software Development for Safety-Critical Systems

Like any other software systems organisation, the developers of safety-critical systems also faced the pressure of delivering earlier and faster. To the best of our knowledge, the first study which recognised this issue and proposed the use of agile methods for the development of safety-critical system was the study by Boehm [2002]. The author, while theoretically arguing the suitability of agile methods for the development of safety-critical systems, proposed using a hybrid (i.e., Agile-Planned) approach for the development of safety-critical systems.

This idea of *using agile methods for development of safety-critical systems* faced a wide criticism during the last 20 years. Studies by Lindvall et al. [2002] and Turk et al. [2014] were amongst the earliest studies that criticised agile methods and discussed the challenges of using agile methods for development of safety-critical systems. Lindvall et al. [2002] presented a theoretical analysis of the empirical evidence discussed in an eWorkshop on importance of highly skilled developers in agile teams. Eighteen Agile experts participated in the eWorkshop who acknowledged the widespread criticism of agile methods and argued that agile methods are unsuitable for the development of the safety-critical systems. Turk et al. [2014] conducted a theoretical analysis of the limitations of agile methods. Their findings seem to coincide with Lindvall et al. [2002]. According to the authors, the quality control mechanisms offered by agile are not adequate for the development of safety-critical systems.

Both studies [Lindvall et al., 2002, Turk et al., 2014], however, highlighted the criticism of agile development without citing any other study which implies that these studies were among the initial studies on this topic. This assumption was confirmed when we were unable to find studies published before the above mentioned studies on the *challenges of agile development of safety-critical systems*.

To understand the reason behind this criticism, we need to understand the context i.e., the regulatory standards. The development of safety-critical systems is constrained by the regulatory standards. Regulatory standards can be classified by their scope i.e., generic vs. domain specific [Gruber et al., 2010, Notander et al., 2013]. However, Notander et al. [2013] divide the regu-

latory standards into two categories *(i) means-prescriptive*: the software development method is either required or recommended and *(ii) objective-prescriptive*: defines what objectives the resulting system artefacts must satisfy without stating how the objectives are achieved.

According to Notander et al. [2013], means-prescriptive standards dictate traditional life cycles, making the accommodation of agile software development much more difficult. On the other hand, objective-prescriptive standards, such as DO-178C may offer fewer restrictions. Since objective-prescriptive standards such as DO-178C do not prescribe the development life cycle, it is important to know *if we can use agile methods to develop safety-critical systems - if they are suitable then what practices are applicable and which of them are unsuitable for the development of a safety-critical project?*.

There are several experiences of applying agile methods to safety-critical systems reported in the literature [Stålhane et al., 2012, Myklebust et al., 2016, Wang and Wagner, 2016b, Hanssen et al., 2016]. These studies focused different themes and reported different challenges related to the use of agile methods for development of safety-critical systems. For example, Myklebust [2008] identified a conflict between quality assurance activities practiced by agile and the quality assurance requirements mandated by the regulatory standards. Shenvi [2014] and Kuchinke et al. [2014] argued that the agile methods' lack of support for the documentation mandated by the regulatory standards makes agile a weaker candidate for the development of safety-critical systems.

We also came across studies that advocated use of agile methods. For example, Gary et al. [2011] demonstrated successful use of an agile method for the development of image guided surgical toolkit, a safety-critical system. Fitzgerald et al. [2013] also illustrated a successful application of an agile method in a large scale regulated environment.

During the initial informal review of the literature at the start of this Ph.D research, we discovered that the extent of the use of agile methods in regulated environments in literature is unclear, and the researchers seem to have diverse opinions on the topic. Therefore, in order to have a holistic view of the problems and challenges of using agile in safety-critical systems context, we decided to conduct a systematic literature review with a focus on *challenges of using agile methods for the development of safety-critical systems*.

3.2 Literature Review Process

In order to conduct the Systematic Literature Review (SLR), we developed a search protocol and an inclusion/exclusion criteria to find the relevant literature on *the challenges of applying agile in safety-critical systems* published between the years 2000 and (March) 2021.

(((agile) AND regulated) AND software)
(agile and safety standards)
agile large-scale mission critical systems
(agile* challenges* large-scale safety-critical)
(agile challenges in large-scale safety-critical)
agile in safety-critical systems
conflicts between agile and regulatory standards
agile success factors and regulatory standards

Figure 3.1: Search strings

Database	URL	Number of Matches
ACM	http://dl.acm.org/	2480
IEEExplore	http://ieeexplore.ieee.org/Xplore/home.jsp	2711
Scopus	https://www.scopus.com	198
Science Direct	http://www.sciencedirect.com/	9752
Web of Science	http://apps.webofknowledge.com	178
	Total	15319

Figure 3.2: Literature sources for the literature review

Search String Strategy

The search strings described in Figure 3.1 were formed following the Population, Intervention, Comparison, Outcomes and Context (PICOC) criteria suggested by Petticrew and Roberts [2008]. Keywords were identified from the *purpose* of the SLR i.e., *what major challenges in the application of agile methods in safety-critical systems are reported in the literature*. The synonyms of important search terms were used to improve the search. The search strings were formed by using AND and OR Boolean operators. The databases were iteratively searched i.e., keywords and search terms were changed and refined.

Preliminary Searches

Figure 3.2 lists the repositories used for searching the studies using various combinations of search strings. At the time of the search, the databases listed in Figure 3.2 were among the top 10 databases for searching software engineering research according to `google.com`.

Inclusion/Exclusion criteria

The following criteria were used to accept or reject the studies.

Include:

1. Study must be accessible in full text.
2. Study highlights the focus on the application of agile in regulated environment.
3. Study which explicitly or implicitly reports challenges, success factors and limitations of application of agile in regulated environment.

4. All studies from 2000 to March 2021.

Exclude:

1. Study which does not focus on the use of agile in regulatory environment.
2. Study which is older than year 2000. The studies older than the establishment of the agile manifesto were excluded.
3. Study which is not from the field of computer science e.g., Biology etc.
4. Non English language studies.

Identification of primary studies

Selection of primary studies for this SLR is performed in number of steps.

1. First step was to use keywords to search and identify potential sources. 15319 Studies were identified using search strings from different online repositories, listed in Figure 3.2.
2. All BibTex files were exported to a tool called Jabref.
3. After filtering the search results and removing 5407 duplicates, there were 9912 studies.
4. The studies that were older than the establishment of agile manifesto i.e., older than the year 2000, were removed. 733 Studies were removed, and the remaining studies were 9179.
5. Another 5841 irrelevant studies i.e., the studies from non-computer science/software engineering fields e.g., Biology etc., were removed.
6. There were 89 duplicates that were not detected by Jabref and were removed manually.
7. Studies removed on the basis of keywords, title and abstract were 3140 and the remaining studies were 109.
8. After applying inclusion/exclusion criteria, 52 studies were selected, and 57 studies were removed because they did not meet the inclusion/exclusion criteria.

Snow ball sampling

Snowball sampling (also called backwards Snowball sampling) refers to checking the reference lists of the included studies for additional references [Jalali and Wohlin, 2012]. We used the backward snowball sampling technique and included four more studies.

Category	Properties
General Information	Author, Year, Title, Date of publication
Aim of the Study	Purpose of research
Methodology	Experiment, Case Study, Tool Proposal, Theoretical Argumentation
Sample size	Can the results be generalised?
Result	Outcome
Critical analysis	Evaluation of the quality of the study
Challenges reported	Limitations about application of agile
Success factors reported	Benefits, Strengths

Figure 3.3: Data extraction properties

Data extraction and synthesis strategy

A spreadsheet was created for the extraction of properties of the 56 finally selected studies in the SLR. Figure 3.3 lists the properties which were extracted from the studies. The data was then synthesised by performing thematic analysis on the extracted data using guidelines by Cruzes and Dybå [2011].

Thematic analysis

Two thematic analyses were conducted during the SLR.

1. Thematic analysis of the selected studies,
2. Thematic analysis of the challenges of agile identified from the studies.

Thematic analysis of the selected studies: The selected literature included studies based upon theoretical argumentation, experience reports, case studies, interview surveys, review papers, experimentation, toy examples, industrial assessment and systematic literature reviews. We observed that the selected studies were not homogeneous i.e., studies differ from each other with respect to sample space, research methods, publishing venues, rigour, outcomes and focus of the study. Due to the heterogeneous nature of studies, we decided not to draw collective statistical inferences from the studies such as frequency of a reported challenge. Instead, we decided to group the studies into following themes based upon the primary focus of each study.

- studies advocating the use of agile methods,
- studies which focus a particular aspect of agile development of safety-critical systems
- suggested tailoring in agile methods.

Thematic analysis of the challenges of agile identified from the studies: The purpose of this SLR was to get an overview of the challenges and the limitation of the agile methods, when used in regulated environments. Each of the 56 studies was read, and the properties described in the Figure 3.3 were extracted for each study. These properties were documented in a single excel

sheet. One of the fields in the excel sheet was *Challenges reported*, under which the challenges reported in each study were listed.

We realised that some of the challenges were inter-related; whereas, others were just general statements and perceptions e.g., *Adoption of agile in safety-critical systems is very slow*. Therefore, we decided to group the findings into themes and sub-themes. These themes were based upon the nature of the challenges identified from the studies e.g., challenges specific to the regulatory standards were grouped together.

3.3 Thematic Analysis of the Selected Studies

As discussed before, the selected studies had diverse perspectives, and it was important to devise a mechanism which could present an overall picture. Therefore, we decided to group the selected studies into themes based upon the overall narrative of each study. This section presents a brief overview of the 56 selected studies. Each of the following sub-sections explain these themes in detail.

3.3.1 Advocates of Agile Methods

We found studies that used various research methods to demonstrate that agile methods could be used for the development of safety-critical systems. These studies demonstrated that agile methods could be used for the development of safety-critical systems. These studies did not report any challenges of using agile for safety-critical system development, however, argued that agile needs to be tailored according to the demands of the project.

For example, Fitzgerald et al. [2013] conducted a case study at QUMAS (a leading supplier of regulatory compliance management solutions). The authors discussed the perceived tension points of agile e.g., time to market, lack of formal planning, adherence to regulatory standards, verification and validation and lack of attention to documentation. The authors, through this study illustrated how an agile approach was implemented successfully in a regulated environment. The study demonstrated a successful application of agile and reported no limitations of agile. However, the authors argued that agile needs to be tailored for use in regulated environments.

Mango [2016] discussed a project at NASA in which agile software development process was used to develop the ground and flight application software in a space exploration rocket. Although the issues were not reported in this study, the key message in the study was that agile has to be tailored in for use in regulated environment.

The study by Huang et al. [2012] was a case study which discussed the use of flexible style of agile systems engineering for complex hardware and software projects. The authors incorporated innovations in a project and presented the lessons learned which included need for

interactive design reviews. The authors recommended tailoring of the agile process according to the requirements of a project.

Rasmussen et al. [2009] presented a comparison of two medical device projects i.e., one developed using agile and the other without use of agile. Both projects required FDA approval. According to the findings of the study, use of agile appears to be beneficial but agile needs to be tailored to fit into the context of regulated environment.

The aim of the case study by Gary et al. [2011] was to challenge the assumption that agile methods are inappropriate for the development of safety-critical software and demonstrate that agile methods are flexible enough to incorporate right amount of ceremony i.e., heavy documentation and incorporation of activities mandatory for safety-critical system development. The authors elaborated on their experience of using agile for the development of IGSTK (image guided surgical toolkit). They adopted ten of the *best practices* from agile including constant communication, continuous building and testing, and focus on the current set of requirements. Through the development of IGSTK, the team demonstrated that agile can be used for the development of the safety-critical systems. However, the authors did not elaborate on the challenges in the application of agile.

Browning and Heath [2009] discussed the case of Lockheed Martin's production system for F-22 aircraft. The authors studied the relationship between the lean implementation and production cost. According to the authors, elimination of tasks does not guarantee cost reduction because Lean/Agile is affected by other factors such as novelty, complexity and instability. The study presented eleven propositions e.g., *if implemented at the wrong time, even lean practices can be wasteful*. However, the propositions were not validated with empirical evidence.

Stelzmann [2012] and Kruchten [2013] investigated the context in which use of agile methods is feasible. Stelzmann [2012] investigated the context in which agile is feasible by interviewing 20 people. The author deemed agile to be unsuitable for the development of safety-critical systems. However, the author did not provide details of the underlying reasons behind the reported challenge. Kruchten [2013] further elaborated on the issue in an experience report which discussed the context in which agile methods could be used. The author discussed a project where the development team thought that a solid architecture will emerge after iterations and refactoring, but it did not. According to the author, detailed documentation clashes with agile which also implies that agile is not suitable for safety-critical systems with high level of criticality. The authors also acknowledged that adaptations must be made to agile methods for use in safety-critical system development.

3.3.2 Studies Focusing a Particular Aspect of Agile Development of Safety-Critical Systems

We identified several studies which focus a particular aspect of the agile development of safety-critical systems such as:

- documentation [Shenvi, 2014, Kuchinke et al., 2014, Hajou et al., 2014, Heeager, 2014, Rottier and Rodrigues, 2008, Carpenter and Dagnino, 2014, Wang et al., 2017a],
- conflicts between evolutionary design offered by agile and upfront design required by the regulatory standards [Abdelaziz et al., 2015, Rottier and Rodrigues, 2008, Carlson and Turner, 2013, Ge et al., 2010], recertification challenges [Gallina et al., 2018, Myklebust et al., 2014b],
- conflicts between agile practices in XP and the requirements of the regulatory standards [Jonsson et al., 2012, Mehrfard et al., 2010, Mehrfard and Hamou-Lhadj, 2011, Paige et al., 2008],
- conflict between testing activities in agile and the requirements of the regulatory standards [Jonsson et al., 2012, McBride and Lepmets, 2016, Kuchinke et al., 2014, Baron and Louis, 2021, Heeager and Nielsen, 2018, Hajou et al., 2014, Carlson and Turner, 2013, Notander et al., 2013, Górski and Lukasiewicz, 2012, Doss and Kelly, 2016],
- lack of use of formal methods in agile development [Wolff, 2012],
- developers' behaviour towards development of safety-critical systems [Lenberg et al., 2020],
- need for software estimation methods in agile process [Rottier and Rodrigues, 2008, Hajou et al., 2014, Alleman et al., 2003, Koski and Mikkonen, 2015].

These studies argue the unsuitability or challenges of incorporating agile methods in their traditional form for the development of safety-critical systems.

Documentation

We found various studies [Shenvi, 2014, Kuchinke et al., 2014, Hajou et al., 2014, Heeager, 2014, Rottier and Rodrigues, 2008, Carpenter and Dagnino, 2014, Wang et al., 2017a] which use different research methods to point out agile methods' lack of support for the documentation requirements of safety-critical systems. Shenvi [2014] performed a theoretical analysis of three regulatory standards for medical software including ISO 13485, IEC 62304, US FDA QSR 820 and argued that agile is weak on documentation aspects, therefore, not favoured by

the regulatory agencies. Results of a survey conducted with four groups of academic developers [Kuchinke et al., 2014] also suggest that agile methods are short on documentation when it comes to regulatory environments. Hajou et al. [2014] conducted a systematic literature review with a primary focus on the challenges faced by the pharmaceutical industry. The authors highlighted the lack of evidence and "*not much research*" in the area. After discussing the dynamics of pharmaceutical environments, the authors listed five challenges identified from the analysis of 49 selected studies. The compatibility of documentation with agile practices was one of those challenges. The authors argued that there is no room for creating less documentation or performing less quality assurance activities.

Heeager [2014] presented two case studies to demonstrate the implementation of a hybrid approach in the regulatory environment. The results of the study implied that agile methods in their original form do not support the documentation needed by the regulatory standards and agile needs to be adapted for the regulatory environments. Study by Rottier and Rodrigues [2008] was their experience report of a project that was developed using agile (Scrum) by a medical device company. The authors reported various organisational and process challenges in adoption of agile. Their findings included the conflicts between minimal documentation in agile and extensive documentation required by the regulatory standards. The study suggested that Scrum needs to be adapted for the development of medical device software. However, the nature of adaptations remained unclear in the study. Carpenter and Dagnino [2014] reviewed the relevant literature in space based system engineering and argued that Test Driven Development (TDD), Extreme Programming and Scrum are suitable agile practices. Whereas, the non-applicable agile practices included evolutionary requirements, minimal documentation and refactoring.

Wang et al. [2017a] intended to improve the safety related communication by improving the safety related documentation in a Scrum development environment. The authors investigated three types of safety related documentation patterns in the agile development i.e., safety epic, safety story and the agile safety plan. Safety story and safety epic were found to be beneficial in improving the safety related communication. According to the authors, although the agile safety plan provides an overview of the process, there is a gap between the high level plan and the concrete development; therefore, the agile safety plan has a little positive effect on the communication.

Evolutionary Design offered by Agile

Several studies [Abdelaziz et al., 2015, Rottier and Rodrigues, 2008, Carlson and Turner, 2013, Ge et al., 2010] highlighted the conflict between evolutionary design offered by agile and upfront design required by the regulatory standards. For example, Abdelaziz et al. [2015] in a study based upon theoretical argumentation, argued that the *evolutionary design* offered by agile conflicts with the regulatory standards because the standards require upfront design (offered by the traditional models) which serves as an input for the hazard analysis. According to Rottier

and Rodrigues [2008], hazard analysis is a mandatory part which needs to be accomplished at product definition and then updated at major milestones. Risks cannot be identified on the features that are not yet well-defined. Carlson and Turner [2013] presented non-software agile case studies for lessons that are potentially applicable to the aircraft systems' integration process. The authors presented five case studies and the lessons learned from them. Lessons learned from three out of the five case studies emphasise on the need of an early architecture.

The study by Ge et al. [2010] is based upon a toy example which used theoretical argumentation for the construction of a model and then illustrated the model with help of a the toy example. The authors pointed out the difficulty in using agile with the regulatory standards by discussing the overall characteristics of the regulatory standards and the activities mandated by the standards e.g., need for an upfront design and production of heavy documentation. According to the authors, detailed design and plan serve as an input to hazard analysis which in turn produces the safety requirements and initiates certification process. The authors proposed construction of an upfront design which is sufficient for hazard analysis. The approach allowed modular construction of the safety-arguments. However, evaluation of the model revealed that the model is unable to assess the quality of the safety argument.

Need for Recertification

Gallina et al. [2018], in a study based upon theoretical argumentation, elaborated on the issue by pointing out the need for recertification with the change in requirements. According to the authors, regulatory standards require early validation of a process plan which includes the requirements definition. Whereas, the concept of continuous improvement in agile encourages accommodation of change in the software which consequently requires re-validation of the plan by a regulatory authority. According to the authors, the re-validation of the plan incurs rework, additional cost and time.

On the contrary Myklebust et al. [2014b] performed a theoretical assessment of Scrum and the documentation required by the standard IEC61508 with the help of five experts (i.e., document writers). The study aimed at optimising the effort spent in the documentation by identifying the documents that can be auto-generated, combined, reused, and needed to be reproduced in case of recertification. Their analysis suggests that only five documents are needed to be reproduced when performing recertification using IEC61508. According to the authors, the existing conflicts between Scrum and IEC61508 are not a consequence of the standard's certification requirements "*... but are related to what the individual assessor will accept as a proof of conformance (PoC) for an activity*" [Myklebust et al., 2014b].

Inapplicable Practices

Jonsson et al. [2012] presented a theoretical analysis of agile practices in the context of software development in the railway sector regulated by the EN50128 regulatory standard and validated

the findings with two researchers. The authors argued that XP's practice of using simple cards to document requirements (stories) is unlikely to be accepted in a regulated environment because the regulatory standards dictate that the requirements must be placed in a document or a tool. The authors argued that XP recommends waiting until the end of the development to create the design and architecture related document; whereas, these documents are created early in the regulated environments to enable verification and validation. Theoretical mapping of XP and the FDA regulations [Mehrfard et al., 2010, Mehrfard and Hamou-Lhadj, 2011] also show that XP needs to be extended before its use with FDA regulations. According to Mehrfard et al. [2010], Mehrfard and Hamou-Lhadj [2011], XP does not support FDA activities e.g., documentation and design review. Paige et al. [2008] conducted a case study and developed a high integrity software using XP. According to the authors, agile processes are not designed to be used for the development of safety-critical systems. The authors pointed out that agile methods need to be modified for the use in safety-critical systems development.

Testing

Jonsson et al. [2012] argued that EN50128 (regulatory standard for railway software) requires the implementer and tester to be separate people; whereas, in Test Driven Development (TDD), the developers write the tests themselves. McBride and Lepmets [2016] also emphasised on the importance of an independent testing body. In a study based upon theoretical argumentation, the authors discussed the potential problem of confirmation bias which can appear due to the agile practice of allowing the development teams do the testing. The authors emphasised on the importance of an independent testing body. The results of a survey conducted with four groups of academic developer by Kuchinke et al. [2014] also showed that there is a lack of quality assurance guidelines (required by regulatory standards) in agile, and there is a need to train the developers in order to help them understand regulatory compliance. Baron and Louis [2021] reviewed the relevant literature on certification of safety-critical avionics software. The authors argue that the way agile principles are interpreted is not compatible with the certification process. According to the authors, adaptations are necessary to ensure that compliance is still met.

Heeager and Nielsen [2018] conducted a Systematic Literature Review (SLR) to identify the disputes in agile development of safety-critical systems. The results of the study suggested that requirements, documentation, life-cycle and testing are the four problem areas in agile development of safety-critical systems. Hajou et al. [2014] also pointed out the lack of fundamentals in agile methods on which the quality of the safety-critical systems should be based. According to the authors, agile methods lack the method of ensuring quality assurance required by the regulatory standards.The authors argue that the regulatory complexity cannot be altered as it is a mandatory element of the pharmaceutical environment; therefore, the existing agile methods need to be tailored to meet the need of the regulatory environment. However, the exact adapta-

tions remain unclear in the study. According to Carlson and Turner [2013], incremental testing is effective and speeds up the iteration pace. A strong change process is needed.

Notander et al. [2013] interviewed five engineers from four different domains with a focus on challenges in flexible safety-critical software development. The authors identified four themes in terms of the challenges to the flexible development i.e., *(i)* Human Factors: there is a need for investment in training people to understand special nature of safety-critical system development, *(ii)* Requirements and verification: need to improve quality assurance activities in agile (for safety-critical environment), *(iii)* agile development: common belief is that pure agile conflicts with the requirements of safety standards, *(iv)* Variants and components: how to use reusable components to optimise the process. They report the lack of evidence as their own experience from reading the literature.

The case study by Górski and Lukasiewicz [2012] discusses several agile models in the field of safety-critical systems. Their analysis of the literature shows that none of the models was validated properly, and a lack of explicit guidance on the application of agile in the regulated environments still exists. The authors proposed an approach called assurance argument patterns. According to the authors, the agile methods should be regarded as complementary to the plan driven practices instead of being a replacement. Extensive testing and good identification of requirements are vital, and communication with domain experts and potential users is crucial. The authors argued that the safety assurance should be incremental.

Doss and Kelly [2016] presented a proposal about the research on integration of assurance case with Scrum. The authors [Doss and Kelly, 2016] argued that there is a reluctance to adopt agile methods within safety-critical system development, and used this argument as a basis for the study. The authors [Doss and Kelly, 2016] proposed to apply 4 + 1 safety assurance principle [Kelly, 2014] to the Scrum process and interview the practitioners to investigate the current concerns and opportunities voiced by the safety-critical systems professionals regarding the use of agile development methods, integration of incremental assurance case development and evaluation within the existing "*Scrum*" methodology. They also proposed to investigate the changes that the Scrum process has to undertake in order to become compliant with the safety standards. As a results of this study, the authors [Doss and Kelly, 2016] proposed additional activities for Scrum to enable its use in the development of safety-critical software.

Use of Formal Methods

Wolff [2012], in a study based upon theoretical argumentation on the use of formal methods in agile development of the safety-critical systems, highlighted the reliance of agile methods on the informal evaluation techniques as a major problem. The author argued that the informal methods are insufficient for establishing the quality of a safety-critical system; therefore, agile is rarely used in regulated environments.

Developers' Behaviour

Lenberg et al. [2020] discussed developers' behaviour towards development of safety-critical systems. The authors conducted interviews with six software engineers. They identified four themes which linked behaviour of the developer to the safety-critical systems development i.e., *(i)* awareness and alignment, *(ii)* norms over standards, *(iii)* domain knowledge, and *(iv)* organisational trust and stress. Their analysis suggested that the safety-critical development " *imposes stress on the software engineers and that to reduce such pressure it is critical to enhance the organisational trust"*. The authors pointed out the non-compatibility of the regulatory standards with the approach proposed by agile.

Estimation

We already discussed the study by Rottier and Rodrigues [2008]. The study was an experience report of a project that was developed using agile (Scrum) by a medical device company. The authors point out the disparity between using an agile method in the software department versus using a waterfall process throughout the rest of the organisation. The authors also argue that the estimations in agile process are often problematic and un-realistic. Hajou et al. [2014] also pointed out the need for a method for estimation of software development projects when using agile methods.

Alleman et al. [2003] acknowledged that there is no schedule variance process for XP; therefore, XP is unable to forecast the future cost and schedule. The authors described the experience of using the earned value analysis in conjunction with the agile development on a mission critical government project. Earned value analysis is a way of predicting future schedule and cost variance. This study demonstrated that the earned value management (EVM) system can be used with XP i.e., XP complies with the EIA748 (standard for EVM). Selected practices and activities of XP are used in this study [Alleman et al., 2003], but the rationale for selecting a practice is not provided.

Study by Koski and Mikkonen [2015] presented an overview of the major issues which were faced during a multi-million euros mission critical information system project for emergency services. The scope, duration and the price were fixed by the signed contracts, but the costumer was willing to collaborate with the developers in an iterative and incremental development environment. XP as a model was followed for the development. The authors acknowledged that the fixed price contracts and the traditional way of estimation do not anticipate the cost of change clearly. They also suggested few improvements for the future which include:

- Direct access to real customer for discussion, feedback and validation
- Keeping the feedback loop as short as possible with the ability to scale the loop when required

- Creating an environment which supports collaboration among all the players e.g., testers, development teams, managers etc.
- Improving automated testing

However, the recommendations were not validated by the authors.

3.3.3 Suggested Tailoring in Agile Methods

Several researchers [Stålhane et al., 2012, 2013, Hanssen et al., 2016, Wang and Wagner, 2016b, Wang et al., 2017b, Lukasiewicz and Górski, 2016, 2018, McHugh et al., 2013, 2014, Trektere et al., 2016, Clarke et al., 2014, Özcan-Top and McCaffery, 2018, Stephenson et al., 2006, Cordeiro et al., 2007] have extended agile methods to propose various new frameworks e.g., SafeScrum [Stålhane et al., 2012], AgileSafe [Lukasiewicz and Górski, 2016] etc. The following sub-sections discuss these frameworks in detail.

Adaptations to Existing Agile Process

There seems to be a major consensus on the need for adaptations in agile methods in order to enable them for the use in the development of safety-critical systems. Several researchers [Siddique and Hussein, 2014, Wils et al., 2006, Axelsson et al., 2016, Goncalves et al., 2015, Martins and Gorschek, 2016] have suggested adaptations to the agile process. For example, Siddique and Hussein [2014] interviewed twenty one practitioners from twenty one different organisations to present an insight into the choice of development methodology in large and complex software projects in Norway. The findings suggested that the agile methodologies are not the preferred choice of the large-scale safety-critical systems development organisations. Based upon the data gathered from the interviews, the authors suggested the use of hybrid models (i.e., agile combined with waterfall).

The study by Wils et al. [2006] is an industrial assessment report. The authors reported the findings of their study conducted at Barco (a major Belgian avionics equipment supplier). Barco adopted XP to benefit from strengths offered by the agile methods. The company wanted to improve the time-to-market and respond quickly to the change in requirements. It turned out that XP did not bring the expected improvements because the project was dependent upon hard to control external factors e.g., automated testing was taking too long, hardware co-development etc. Following recommendations are made by this study:

1. Add more communication and feedback to the process.
2. Limit amount of changes at the later stages
3. Auto generate documents, use version control, and keep track of the dependencies and the changes in the documents.

4. Use agile document preparation practice for the documents that cannot be auto generated e.g., RaPiD7.

5. Use automated testing, pair programming reviews.

Axelsson et al. [2016] presented an overview of the topics discussed at a seminar in Stockholm in 2014 on the current state of agile, its applications to the safety-critical systems and the consequences of innovations in large organisations. Need for quality assurance activities expected by regulatory standards, thorough design reviews, and compliant documentation is highlighted in the study. The authors also acknowledged the lack of evidence and practical guidance on agile, therefore, emphasised on the need for more research in the area.

Study by Goncalves et al. [2015] is an experience report about an academic project in which five scrum teams i.e., around 70 students, worked together on a micro-satellite system and successfully delivered the project in four sprints of four weeks each. They used Scrum and its best practices to produce a prototype as a proof of concept. However, the study [Goncalves et al., 2015] does not report the limitations.

Martins and Gorschek [2016] performed a systematic literature review of requirements engineering in the domain of safety-critical systems. One of their main conclusions was the dominance of the traditional approaches i.e., the well-established methods (both in terms of analysis techniques and overall project management) are widely used. Whereas, the newer approaches are often introduced and then abandoned i.e., the "*early mortality of new approaches*". While agile and lean methodologies were not the primary focus of their study, one of their suggested research questions for the research community, based on their analysis of existing work, was: "*to what extent may the lean and agile requirements engineering approaches improve the integration amongst safety, requirements, test and certification teams?*". They pointed towards the need to investigate the use of agile methods for the development of safety-critical systems.

SafeScrum

Stålhane et al. [2012] performed a theoretical assessment of conformance of Scrum with IEC 61508 certifiable environments. The analysis was performed in two iterations by three experts in the area of software development, certification, and agile development respectively. Based on their assessment, the authors proposed an extended version of Scrum and called it "*Safe-Scrum*". The authors performed a manual analysis of Scrum and IEC61508, and found fifteen issues where adaptations were needed. These issues were mostly related to documentation and planning.

In the later years, more studies [Stålhane et al., 2013, Hanssen et al., 2016] were conducted to perform evaluation of SafeScrum. Stålhane et al. [2013] performed a theoretical analysis of the challenges of using SafeScrum with three different regulatory standards i.e., IEC 61508, IEC 60880 and IEC 50128. The authors argued that agile (e.g., Scrum) is not well adapted

for safety-critical systems. According to the authors, the regulatory standards are developed for the plan driven approaches, which makes the accommodation of change difficult. However, the authors contradict themselves later by saying "... *IEC 61508 and IEC 60880, EN 50128 explicitly allows iterative development...*". This means that although the regulatory standards allow iterative development, the existing agile methods in their traditional form are unsuitable for development of safety-critical systems.

The aim of the study by Hanssen et al. [2016] was to report the lessons learned from the trial run of SafeScrum in a company. The company used SafeScrum in one of their projects which required IEC 61508 compliance. The authors gathered the data by observing the sprint reviews, analysis of the documentation, and conducting the interviews and discussions with the scrum team. During this study, the authors learned that the quality assurance offered by Scrum is insufficient for a regulated environment. Based upon the analysis of IEC 61508 (regulatory standard), the discussion with an independent assessor and working with the scrum team, the authors identified the necessary additional tasks for the quality assurance and the need for a QA role in agile. The authors argued that the scrum quality assurance is thought to be embedded in the process itself e.g., Pair programming etc. According to the authors, the Quality Assurance mechanisms in Scrum and agile are insufficient for the regulated domain. Also, agile has a quality assurance mechanisms but no explicit QA role, which is against the "*independent testing*" required by the regulatory standards Hanssen et al. [2016].

Wang and Wagner [2016a], Myklebust et al. [2014a] proposed adaptations to SafeScrum. Wang and Wagner [2016a] theoretically integrated a novel systematic safety analysis technology STPA into SafeScrum. The authors argued that the current safety analysis technologies are inadequate for agile. Myklebust et al. [2014a] presented a theoretical description of change impact analysis in safety-critical software when using SafeScrum. The authors provided recommendations for integrating SafeScrum with the change impact process in safety-critical software. However, a critical evaluation of those recommendations was not discussed in the study. A similar study is performed by Stålhane and Myklebust [2015].

S-Scrum

Wang and Wagner [2016b] extended SafeScrum by integrating SafeScrum with a safety analysis and verification approach based upon STPA (System-Theoretic Process Analysis), a safety guided design technique by Leveson [2016]. The authors referred to this extended version as S-Scrum. The authors [Wang and Wagner, 2016b] validated their model using a toy example of airbag system. The authors argued that safety should be approached from the agile standpoint rather than combining agile with a plan driven approach. They also pointed out the difficulty in using agile methods in the regulated environments due to the activities mandated by regulatory standards e.g., need for an upfront design.

Wang et al. [2017b] applied and validated the S-Scrum method proposed earlier. The study

was conducted in three episodes i.e., Scrum was used in the first episode, S-Scrum in second and optimised S-Scrum was used in the last episode of the study. The authors suggested few additional activities and improvements to optimised S-Scrum.

AgileSafe

Lukasiewicz and Górski [2016] proposed a new methodology called AgileSafe for the development of safety-critical systems. The approach employed evidence-based arguments which followed recommendations on assurance cases from IEC 15026 regulatory standard. The objective was to help the SMEs (Small/medium size enterprises) developing safety-critical systems and increase their profit. The authors argued that accommodation of change will result in the changes in safety evidence collected during the development which may affect the scope and the structure of the certification process. Therefore, change in safety-critical environment is a complicated and potentially costly operation. The authors claimed that their approach was validated but there was no evidence for it nor the results of the validation were reported in the study. A concrete justification for the need of this new approach was also not provided.

Lukasiewicz and Górski [2018] further extended the method by proposing AgileSafe Use Case - a two step approach. In the first step, the system was decomposed into the use cases and the regulatory constraints for the use cases were identified. In the second step, the AgileSafe was improved by updating the knowledge base of the method through identifying patterns, and practices emerged from the system under development. The authors demonstrated the use of the method with the help of a case study of continuous glucose monitoring-enabled insulin pump system. According to the authors, in order to the improve AgileSafe use case knowledge base, the user should be an expert on agility in addition to being a person with good knowledge of the standards and the safety aspects of the software development. This implies that the use of agile methods requires considerable training and investment.

Agile-V Model

The study by McHugh et al. [2013] is focused on use of agile methods for development of medical device software. The authors proposed the Agile-V model by integrating agile practices applicable to development medical software with a plan driven V model. They conducted interviews and then mapped findings of the interviews with the literature by conducting a systematic literature review. Accommodation of change in the requirements was identified as a main issue with the plan driven life cycles during the interviews. Participants of the interviews suggested different measures to counter this problem including detailed upfront planning and preventing the customer from introducing the changes once the project enters the development phase.

Another study [McHugh et al., 2014] was conducted as an extension of the previous study. In this study, the authors conducted an experiment for implementing and validating the model. The authors argued that no agile method is sufficiently comprehensive in producing the regulated

deliverables. The authors of the study demonstrated that the hybrid model works better than a plan driven model in a safety-critical environment but failed to report the shortcomings of the framework they had proposed.

MDevSPICE®

Clarke et al. [2014] introduced MDevSPICE by describing it as a framework to facilitate the production of a medical device software. The framework was developed by Regulated Software Research Centre (RSRC) in Ireland. In this study, however, the authors did not describe the framework itself but discussed the regulatory standards the framework could meet. In another study Lepmets et al. [2015] described MDevSPICE as a framework that "... *integrates generic software development best practices with medical device standards' requirements enabling consistent and thorough assessment of medical device processes"*. The authors described the MDevSPICE framework which consists of a process reference model, a process assessment model, an assessment method, and training and certification schemes. According to the authors, the model was validated by five different organisations. However, the authors did not elaborate on the limitations of the framework.

Trektere et al. [2016], in their study, demonstrated that MDevSPICE could be tailored by introducing agile practices into the framework. The authors theoretically argued the development of mobile medical applications using MDevSPICE, which combined agile with the reduced V-model. The authors did not discuss the limitations of the approach. Özcan-Top and McCaffery [2018] performed a theoretical mapping of MDevSpice with the Scrum and XP activities to assess the extent to which the regulatory requirements defined in MDevSPICE meet the activities of Scrum and XP. The study showed that using XP and Scrum practices for development of a medical device software may meet nine processes in MDevSPICE. The authors described 14 additional processes in order to show conformance to medical regulations. This shows that tailoring is essential for agile methods if they are to be used in the medical device software domain. According to the authors, XP (as compared to Scrum) showed a limited support for the development of medical device software.

Other Frameworks

Stephenson et al. [2006] proposed a four steps framework to introduce agility in the safety-critical systems development by using the following as input:

- The original agile security architecture definition.
- Component-based safety modelling techniques such as Cecilia/OCAS and HiP-HOPS.
- Recommended practice documentation ARP 4754 [Landi and Nicholson, 2011] and ARP 4761 [SAE, 2017].

- Expertise within the safety analysis community.

The authors performed a theoretical analysis of the model with respect to increments, testing, infrastructure, guidance and training, independence, and integration. However, the analysis lacks detail.

Cordeiro et al. [2007] used theoretical argumentation for the construction of a model. The authors conducted an experiment to validate the model. However, the limitations of the model were not reported in the study. The results of the study implied that agile methods need to be adapted to the needs of a project.

Commonalities and Differences

Further analysis showed that we can group the research suggesting tailoring of agile methods for safety-critical systems development into two i.e., *(i)* research which suggests combining agile methods with another method, *(ii)* research which suggests integration of safety aspects and improvements in various activities of agile methods.

The research which suggests combining other methods includes using a hybrid method (i.e., agile combined with waterfall) [Siddique and Hussein, 2014]. Other researchers propose combining other traditional methods with agile for example; McHugh et al. [2013] propose Agile-V model by integrating agile practices applicable to development medical software with a plan driven V model. Trektere et al. [2016] propose combining MDevSPICE framework in a setting where an agile method is already in use in combination with V-model. The study by Özcan-Top and McCaffery [2018] shows that using XP and Scrum practices may meet some of the processes described by MDevSPICE. However, the authors described 14 additional processes to show conformance to medical regulations.

The second group of studies suggests improvements in existing agile methods and integration of safety aspects. For example, Wils et al. [2006] recommend improvements such as automated testing in XP. Axelsson et al. [2016] suggested the need for quality assurance activities expected by regulatory standards. Stålhane et al. [2012] integrate safety aspects in Scrum and call it SafeScrum. The authors suggest improvements in documentation and planning. However, Wang and Wagner [2016a] and Myklebust et al. [2014a] suggest integration of STPA and change impact process in safety-critical software development, respectively, into SafeScrum which implies that SafeScrum is not sufficient for the development of safety-critical systems on its own. In a later work, Wang et al. [2017b] suggest further improvements in their earlier work in which they combined STPA with SafeScrum (and named it S-Scrum). Lukasiewicz and Górski [2018] propose AgileSafe which employs evidence-based arguments that follow recommendations on assurance cases from IEC 15026 regulatory standard. According to the authors, the use of the method requires considerable prior experience in agile.

3.4 Discussion Based Upon Thematic Analysis of the Challenges Identified from the Studies

In this section, we analyse the challenges found during our Systematic Literature Review (SLR) in the light of the evidence presented in the studies selected in this SLR. The challenges reported in the selected studies are divided into themes using Figure 3.1. Each theme listed in Figure 3.1 either describes the nature of the challenge or focuses a particular aspect of a project.

3.4.1 Statements and Perceptions

Some of the arguments made in some of the selected studies listed in Figure 3.1 seem to be the "*general perceptions*" of the researchers. In this section, we have discussed and analysed the evidence presented in the studies in support of such arguments and the statements.

Our experience from reading the literature is that *agile is unsuitable for the development of safety-critical systems* used to be a perception because of the *non-ceremonial* facade of agile methods. If we look at the research published during the last decade in this area, we notice a gradual transition from traditional plan-based software development life cycles to the use of agile methods. Attempts to employ agile methods in the development of safety-critical systems are being made in the regulated environments e.g., avionics [Goncalves et al., 2015, Wils et al., 2006, VanderLeest and Buter, 2009], railways [Jonsson et al., 2012], medical device software [Trektere et al., 2016, McHugh et al., 2014, Shenvi, 2014, Rottier and Rodrigues, 2008] etc. Studies show that agile is not unsuitable for the development of safety-critical system but needs to be adapted according the requirements of the projects [Trektere et al., 2016, McHugh et al., 2014, Carpenter and Dagnino, 2014, Huang et al., 2012, Stålhane et al., 2013, Górski and Lukasiewicz, 2012, Mehrfard and Hamou-Lhadj, 2011, Shenvi, 2014, Mehrfard et al., 2010, Cordeiro et al., 2007, Jonsson et al., 2012, Rottier and Rodrigues, 2008, Alleman et al., 2003, Goncalves et al., 2015, Wils et al., 2006]. However, the exact nature of adaptations remains unclear.

Slow adoption of agile in regulated environments is also a perception. There is no evidence to support this argument in the studies reporting this observation [McHugh et al., 2014, Kuchinke et al., 2014, Doss and Kelly, 2016, Stephenson et al., 2006]. As discussed before, we can clearly see a rise in the number of publications on *the use of agile in the regulated context* in the last ten (10) years. We cannot say anything conclusive about the pace of adoption of agile in the regulated environment unless a point of reference and the difference between *fast* and *slow*, in this context, are defined.

Many of the selected studies argue that agile needs to be tailored for its use in regulated environments [Trektere et al., 2016, Wang and Wagner, 2016b, McHugh et al., 2014, Myklebust et al., 2014a, Huang et al., 2012, Stålhane and Myklebust, 2015, Ge et al., 2010, Abdelaziz et al., 2015, Stålhane et al., 2013, Górski and Lukasiewicz, 2012, Lukasiewicz and Górski, 2016, Cordeiro et al., 2007, Fitzgerald et al., 2013, Rasmussen et al., 2009, Hajou et al., 2014,

Table 3.1: Challenges reported in selected studies

Theme	Challenges	References
Statements and Perceptions	Agile is not suitable for safety-critical systems and systems with high level of criticality.	[Siddique and Hussein, 2014, Kruchten, 2013, Stelzmann, 2012, Stålhane et al., 2013, Górski and Lukasiewicz, 2012, Paige et al., 2008, Mehrfard and Hamou-Lhadj, 2011, Shenvi, 2014, Wolff, 2012, Rottier and Rodrigues, 2008, Doss and Kelly, 2016, Wils et al., 2006, Fitzgerald et al., 2013, Hajou et al., 2014, McBride and Lepmets, 2016]
	Adoption of agile in regulated environments is very slow	[McHugh et al., 2014, Kuchinke et al., 2014, Doss and Kelly, 2016, Stephenson et al., 2006]
	Agile has to be tailored in order to be used in regulatory environments.	[Gallina et al., 2018, Baron and Louis, 2021, Gallina et al., 2018, Baron and Louis, 2021, Trektere et al., 2016, McHugh et al., 2014, Carpenter and Dagnino, 2014, Heeager, 2014, Siddique and Hussein, 2014, Huang et al., 2012, Stålhane et al., 2013, Górski and Lukasiewicz, 2012, Mehrfard and Hamou-Lhadj, 2011, Shenvi, 2014, Mehrfard et al., 2010, Cordeiro et al., 2007, Jonsson et al., 2012, Rottier and Rodrigues, 2008, Alleman et al., 2003, McHugh et al., 2013, Wils et al., 2006, Rasmussen et al., 2009, Hajou et al., 2014, Mango, 2016]
	Lack of evidence and guidance to support the use of agile in regulatory environments; and Slow adoption of agile in regulated environment	[Wang and Wagner, 2016b, Notander et al., 2013, Carpenter and Dagnino, 2014, Heeager, 2014, Siddique and Hussein, 2014, Huang et al., 2012, Axelsson et al., 2016, Ge et al., 2010, Abdelaziz et al., 2015, Shenvi, 2014, Jonsson et al., 2012, Browning and Heath, 2009, McHugh et al., 2013, Stephenson et al., 2006, Carlson and Turner, 2013, Fitzgerald et al., 2013, Stålhane et al., 2012, Wang and Wagner, 2016a, Hajou et al., 2014, McBride and Lepmets, 2016]

Table 3.1: Challenges reported in selected studies

Theme	Challenges	References
Organisational culture and training	Lack of investment in training people (e.g., developers) i.e., to develop understanding of the criticality involved in regulated systems.	[Notander et al., 2013, Kuchinke et al., 2014, Jonsson et al., 2012, Rottier and Rodrigues, 2008, Carlson and Turner, 2013, Hajou et al., 2014, Carlson and Turner, 2013, Koski and Mikkonen, 2015, Jonsson et al., 2012, Lenberg et al., 2020]
Project management	Agile vs Fixed Contracts and traditional way of estimating of system development projects. Agile is unable to forecast future cost and schedule.	[Rottier and Rodrigues, 2008, Alleman et al., 2003, Koski and Mikkonen, 2015, Hajou et al., 2014]
Documentation	Non-compatibility of certification documentation with agile practices and methods.	[Trektere et al., 2016, Notander et al., 2013, McHugh et al., 2014, Heeager, 2014, Siddique and Hussein, 2014, Kruchten, 2013, Mehrfard and Hamou-Lhadj, 2011, Kuchinke et al., 2014, Shenvi, 2014, Mehrfard et al., 2010, Jonsson et al., 2012, Koski and Mikkonen, 2015, Myklebust et al., 2014b, Hajou et al., 2014]
	Certification is expensive, and certification procedure is usually performed on a complete system. "*Many releases*" makes certification lengthy, and it also significantly increases cost.	[Notander et al., 2013, Lukasiewicz and Górski, 2016, Jonsson et al., 2012]
Regulatory standards	Standard lack guidance on specific context. Terms used are ambiguous.	[Stålhane et al., 2013, Mehrfard et al., 2010]

Table 3.1: Challenges reported in selected studies

Theme	Challenges	References
	Standards dictate a sequential plan driven approach.	[Trektere et al., 2016, Wang and Wagner, 2016b, Notander et al., 2013, Ge et al., 2010, Abdelaziz et al., 2015, Stålhane et al., 2013, Mehrfard et al., 2010, Jonsson et al., 2012, Rottier and Rodrigues, 2008, Stålhane et al., 2012, Hajou et al., 2014]
	Agile lacks quality assurance activities required by regulatory standards i.e., tight collaboration between development teams and test teams is in contrast to independent test teams	[Wang and Wagner, 2016b, Notander et al., 2013, Siddique and Hussein, 2014, Axelsson et al., 2016, Ge et al., 2010, Abdelaziz et al., 2015, Górski and Lukasiewicz, 2012, Kuchinke et al., 2014, Shenvi, 2014, Fitzgerald et al., 2013, Wang and Wagner, 2016a, Hajou et al., 2014, Hanssen et al., 2016, McBride and Lepmets, 2016, Myklebust, 2008]
Design and architecture	Many safety analysis techniques e.g., FTA, FMEA need upfront architecture which is unlike agile.	[Wang and Wagner, 2016b, Ge et al., 2010, Abdelaziz et al., 2015, Rottier and Rodrigues, 2008, Stephenson et al., 2006]
	Agile offers evolutionary design vs traditional upfront design. There is a need for explicit guidelines on how to do periodic design reviews.	[Kruchten, 2013, Huang et al., 2012, Axelsson et al., 2016, Ge et al., 2010, Abdelaziz et al., 2015, Mehrfard and Hamou-Lhadj, 2011]

Mango, 2016]. Several studies [Ge et al., 2010, Górski and Lukasiewicz, 2012, Cordeiro et al., 2007] have also proposed models and adaptations to the agile process to enable its use in the safety-critical system development, but very few were validated.

Most of the selected studies recommend the use of a hybrid approach i.e., agile practices shall be combined with a plan driven life cycle [Trektere et al., 2016, Wang and Wagner, 2016b, McHugh et al., 2014, Myklebust et al., 2014a, Huang et al., 2012, Stålhane and Myklebust, 2015, Ge et al., 2010, Abdelaziz et al., 2015, Stålhane et al., 2013, Górski and Lukasiewicz, 2012, Lukasiewicz and Górski, 2016, Cordeiro et al., 2007]. For example, Trektere et al. [2016] combined agile with reduced V-model and the authors [Stålhane et al., 2013] suggested use of an extended version of Scrum i.e., SafeScrum. Lukasiewicz and Górski [2016] proposed a new methodology called AgileSafe that uses evidence based arguments. Mehrfard et al. [2010] extended XP to meet FDA requirements. Despite of researchers' advocacy for a hybrid approach, the nature of "*tailoring*" and the criteria for such tailoring are unclear.

The above are few of the many examples of different adaptations to the agile methods to enable their use in regulated environments. Every other study proposes a different hybrid approach which raises two questions i.e., What adaptations need to be made to agile methods in order to enable their use in regulated environments? (ii) What is the criteria for making such adaptations? The lack of knowledge on how to implement a hybrid approach is also pointed out by Heeager [Heeager, 2014]. Mehrfard and Hamou-Lhadj [2011] argue that the trade-off that balances agility and auditability, needs to be investigated.

Lack of evidence and guidance to support the use of agile in safety-critical systems is reported by many studies [Wang and Wagner, 2016b, Notander et al., 2013, Carpenter and Dagnino, 2014, Heeager, 2014, Siddique and Hussein, 2014, Huang et al., 2012, Axelsson et al., 2016, Ge et al., 2010, Abdelaziz et al., 2015, Shenvi, 2014], but none of the selected studies provides any evidence to support this argument. For example, low percentage of empirical research is interpreted as "*lack of evidence*" by McHugh et al. [2013]. To support their argument, the authors [McHugh et al., 2013] refer to a statement made by VanderLeest and Buter [2009]. This is a weak deduction because unless someone can define what number of empirical studies is considered "*sufficient*", this argument has no basis.

We believe that the "*lack of evidence and guidance to support use of agile in regulatory environments*" is a general perception among the researchers. We also believe that most of the available literature lacks detailed guidance on specific issues. One of the reasons could be the confidential nature of regulated systems projects due to which the researchers are unable to publish the project related details.

3.4.2 Organisational Culture and Training

Resistance to *change the ways of working* has been reported by several studies [Martins and Gorschek, 2016, Miler and Gaida, 2019, Cinite and Duxbury, 2018]. People feel more confi-

dent about using a development method which they have the past experience and guidance for [Islam and Storer, 2020b]. A "*successful*" method of the past is, sometimes, used as a standardised method for the future projects as well. The fear of failure, and lack of confidence and guidance on *how to apply* a method often impede its application [Islam and Storer, 2020b]. The researchers [Notander et al., 2013, Kuchinke et al., 2014, Jonsson et al., 2012, Rottier and Rodrigues, 2008, Carlson and Turner, 2013, Hajou et al., 2014] agree that one of the biggest challenges behind the adoption of agile process in the companies is the resistance to its adoption.

According to Shimoni [2017], using a new process is like *moving from the known to the unknown* which could produce frustration and anxiety that manifests resistance. According to Erwin and Garman [2010], not believing in the effectiveness of a new process is one of the reasons behind change resistance in a company. This finding coincides with the findings of various other studies on the use of agile in safety-critical system development [Hajou et al., 2014, Carlson and Turner, 2013, Koski and Mikkonen, 2015, Jonsson et al., 2012, Lenberg et al., 2020].

The analysis of the existing literature [Jonsson et al., 2012, Rottier and Rodrigues, 2008, Carlson and Turner, 2013, Hajou et al., 2014, Carlson and Turner, 2013, Koski and Mikkonen, 2015, Jonsson et al., 2012, Lenberg et al., 2020] reveals that the risks which prevent the companies from experimenting with a new process like agile include: the lack of guidance on the practical use of agile in safety-critical system development, consequences of "*improper*" development of safety-critical systems, and the time and effort involved in their development. Introduction of a new process such as agile, into the companies developing safety-critical systems, requires convincing and educating people about its potential benefits [Notander et al., 2013, Kuchinke et al., 2014, Jonsson et al., 2012, Rottier and Rodrigues, 2008, Carlson and Turner, 2013, Hajou et al., 2014, Carlson and Turner, 2013, Koski and Mikkonen, 2015, Jonsson et al., 2012, Lenberg et al., 2020].

According to Miler and Gaida [2019], the adoption of an agile process requires changing the mindset. Miler and Gaida [2019] emphasise that the agile way of working is a *particular attitude* or a *way of thinking* of the entire team. According to Notander et al. [2013], the organisations must invest in agile trainings and focus on changing people's attitude towards the agile procedures. This also includes educating the customer about working in a collaborative agile environment [Jonsson et al., 2012]. One of the recommendations made by Kuchinke et al. [2014] is training the developers to support validation and maintenance of the safety-critical systems in an agile context.

3.4.3 Project Management

This section presents an analysis of the potential contracting issues with the projects developed using agile methods. Please note that this section does not discuss the role of the customer. Focus

of this section is specifically on the issues related to the contracting process of agile projects.

The development life cycle of a project that is initiated and developed internally by an organisation is usually controlled internally. The project related decision making takes place at the organisation's own expense e.g., schedule negotiation, delays, change accommodation etc. Adding more work automatically increases the cost and the schedule of the project. Normally, when the projects are initiated and developed within the organisation, there is no external organisation to sign a contract with. However, outsourcing or having an external customer typically involves signing contracts at organisational level.

Outsourcing a software development task to a sub-contractor often requires a clear definition of what is required [Turk et al., 2014]. The sub-contractor has to submit a clear plan for completion. The agreed plan for completion with formally agreed terms and conditions between the parties, takes shape of a formal contract.

The traditional contracting environment, including the government contracts, follows a linear model [Alleman et al., 2003]. It involves a step-wise completion of tasks, usually, without the possibility of going back to the previous phase [Mergel et al., 2018]. Any change in the functionality usually results in re-negotiation of the contract [Gerster and Dremel, 2019]. Also, in a traditional model, the customer has to wait too long before the product delivery is made [Mergel et al., 2018]. Agile software development approaches, however, "... *involve creating, testing, and improving technology products incrementally in short, iterative sprints"* [Mergel et al., 2018].

Agile approaches seem like a perfect solution for accommodation of change and early delivery, but there are problems with contracting a project which is to be developed using an agile development model. Estimation of effort is one of the biggest challenges in agile [Rottier and Rodrigues, 2008]. "*It appears difficult to estimate effort through the 'unstructured nature' of agile methods. The amount of effort for delivering a feature seems to be underestimated"* [Alleman et al., 2003]. Also, since agile does not follow a formal plan, there is no alternative to reporting progress-to-plan [Alleman et al., 2003].

We have noticed a lack of research on the topic of "*agile contracts for the development of safety-critical systems"*. Very few [Russo et al., 2018, Turk et al., 2014, Baron and Louis, 2021] have discussed this issue briefly and suggested adaptations to the contracting process to incorporate agile development of safety-critical systems. Russo et al. [2018] recommend using Function Point Analysis techniques like Simple Function Points (SiFP) for effort estimation in an agile project. They suggest Sprint-based contracts for calculating the economic value of the effort determined by the Function Point Analysis. Baron and Louis [2021] seem to agree with Russo et al. [2018] and also recommend using iteration-based contracts. Turk et al. [2014] has suggested dividing a contract supporting agile development into two parts i.e., *(i) fixed part:* the activities that must be carried out by the sub-contractor. It includes the criteria for accepting or rejecting the modification; *(i) variable part:* the requirements that can vary within the scope and

boundaries defined in the "*fixed part*".

3.4.4 Documentation

According to the researchers [Trektere et al., 2016, Notander et al., 2013, McHugh et al., 2014, Heeager, 2014, Siddique and Hussein, 2014, Kruchten, 2013, Mehrfard and Hamou-Lhadj, 2011, Kuchinke et al., 2014, Shenvi, 2014, Mehrfard et al., 2010, Jonsson et al., 2012, Koski and Mikkonen, 2015, Myklebust et al., 2014b, Hajou et al., 2014], agile methods do not support heavy documentation, especially the documentation mandated by the regulatory standards. The agile principle [Beck et al., 2001a] "*working software over comprehensive documentation*" is commonly quoted as the basis of this conflict between agile methods and the regulatory standards. This principle is often interpreted as "*documentation is discouraged in agile*" [Ramesh et al., 2010, Turk et al., 2005, Baron and Louis, 2021].

Agile gives priority to the *important stuff*, and the delay caused by the unnecessary things such as documentation could be the reason behind the interpretation of this agile principle. Since the pace of change in business is faster than the pace of updating the documents [S. Bose, 2010], updating the document, every time a change occurs, causes delay. The issue becomes more evident in the case of safety-critical systems certification which involves production of comprehensive documentary evidence. Heavy documentation makes the work "*less agile*" [Jonsson et al., 2012]. The formal way of requirement specifications is also not considered "*agile friendly*" [Rayside et al., 2009, Martins and Gorschek, 2016].

Several researchers [VanderLeest and Buter, 2009, Wils et al., 2006, Jonsson et al., 2012] also draw a contrast between the iterative nature of agile and the certification artefacts which require looking at the system in its entirety. For example, DO-178C requires early completion and approval of the Plan for Software Aspects of Certifications (PSAC). Later changes require updating the PSAC and its re-approval by an FAA Designated Engineering Representative (DER) [VanderLeest and Buter, 2009]. According to VanderLeest and Buter [2009], late introduction of change in requirements in agile is no worse than waterfall. This problem is also reported by Wils et al. [2006], Jonsson et al. [2012].

In reality, the agile principle "*working software over comprehensive documentation*" promotes quick development and accommodation of change. Agile is not against documentation and this principle is often misinterpreted [Baron and Louis, 2021]. Agile only discourages the *wasteful* documentation [Baron and Louis, 2021]. Whereas, in a safety-critical system development context, certification is a part of the *working software* without which, a safety-critical system is unusable.

We believe that the agile principle "*working software over comprehensive documentation*" has been misinterpreted for a long time. There is evidence of use of agile in the development of airborne software [Chapman, 2016, Glas and Ziemer, 2009, Paige et al., 2011, Rayside et al., 2009, Turk et al., 2005, Black et al., 2009, Ramesh et al., 2010, Cawley et al., 2010]. Even

formal specification, with few adaptations, is compatible with agile [Rayside et al., 2009, Black et al., 2009]. A study by Baron and Louis [2021] discusses the internal surveys of over a hundred industrial certification audits by a technical authority. The study [Baron and Louis, 2021] is focused on the continuous integration of certification requirements in the software development process. Baron and Louis [2021] argued that with agile, it is possible to maintain concise records that ensure traceability. The authors recommend automating the documentation process. According to Chapman [2016], automated documentation can reduce delay.

Chapman [2016] suggests building a high integrity deployment pipeline. The author describes this pipeline which implements an agile environment using four points. The first point discusses the use of principled requirements engineering [Jackson, 2000], focusing initially on non-functional requirements development of architecture, specification and associated satisfaction arguments. The second point involves use of formal language for requirements. Third point describes use of an evidence engine, *"... combining static verification, continuous regression testing, automated generation of documents and assurance evidence, and a cloud of virtualized target platforms for integration and deployment testing"*. The fourth point describes use of well planned early iterations while the plans for later iterations are left open to accommodate changes.

3.4.5 Regulatory Standards

Before discussing the applicability of agile in the regulatory environment and the potential conflicts between them, we need to understand the difference in the nature of the regulatory standards. Usually, the regulatory standards are classified by their scope i.e., generic vs. domain specific, but a better and a logical categorisation is provided by Notander et al. [2013]. The authors [Notander et al., 2013] divide regulatory standards into two categories (i) means-prescriptive, (ii) objective-prescriptive.

A means-prescriptive standard, e.g., ISO61508, focuses on the achievement of certain high-level safety goals and typically provides the lists of methods and suggestions that the developers would be forced to include in their development process. An objective-prescriptive standard on the other hand, e.g., RTCA/DO-178C, defines the (low-level) objectives that should be reached, but does not provide a description of *how to reach them*. High-level safety goals are achieved when the objectives are fulfilled [Stålhane et al., 2013].

According to Notander et al. [2013], the means-prescriptive standards dictate traditional life cycles, and the accommodation of agile is much more difficult in means-prescriptive standards. Whereas, the objective-prescriptive standards do not put any restrictions on the use of agile methods.

Several studies [Trektere et al., 2016, Abdelaziz et al., 2015, Stålhane et al., 2013, Mehrfard et al., 2010, Jonsson et al., 2012, Stålhane et al., 2012, Wang and Wagner, 2016b, Notander et al., 2013, Ge et al., 2010, Stålhane et al., 2013, Rottier and Rodrigues, 2008, Hajou et al., 2014], in

Agile Principle	DO-178C Principle
Individuals and Interactions	Processes and Tools
Working Software	Comprehensive Documentation
Evolving Requirements via Customer Collaboration	Rigorous Requirements Specification
Responding to Change	Following a Plan

Figure 3.4: Potential conflicts between agile principles and DO-178C (Reproduced from [Coe and Kulick, 2013])

this systematic literature review, used the statement "*standards favour plan-driven approaches*" as one of the motivations for their studies, but none of them provides any empirical evidence for this assertion. However, some of the above studies [Abdelaziz et al., 2015, Stålhane et al., 2013, Mehrfard et al., 2010, Jonsson et al., 2012] discuss the potential conflicts between agile methods and different regulatory standards.

Means-prescriptive standards need no further discussion since they dictate the processes and the methods, and there is no possibility of incorporating *other* development methods than the prescribed ones. However, we do need to take a look at the objective-prescriptive standards e.g., DO-178C and see if there are conflicts between agile methods and the objective-prescriptive standards.

DO-178C [RTCA], is a regulatory standard for airborne software. Since it is an objective-prescriptive standard, it does not favour a particular software development life cycle [Cawley et al., 2010, Wils et al., 2006]. DO-178C provides a list of (total 71) objectives which need to be met during the development of airborne software [Coe and Kulick, 2013].

Coe and Kulick [2013], in Figure 3.4, provide a list of potential conflicts between agile and DO-178C [RTCA] certification requirements. Same conflicts were identified by other researchers [Chapman, 2016, Glas and Ziemer, 2009, Ramesh et al., 2010, Turk et al., 2005, Martins and Gorschek, 2016, Wils et al., 2006, Cawley et al., 2010, Coe and Kulick, 2013, Cawley et al., 2015, Marques and Cunha, 2013, Chenu, 2009, Boehm and Turner, 2003, Vuori, 2011].

The conflicts between the agile process and regulatory standards imply that agile methods in their pure form and the regulatory standards do not go hand in hand [Stålhane et al., 2013]. Therefore, agile needs to be tailored and used in combination with other approaches e.g., Agile-Planned [Boehm, 2002, Coe and Kulick, 2013, Boehm and Turner, 2003]. Cawley et al. [2010] argue that the agile in combination with traditional approaches (e.g., waterfall) is the most used and recommended approach.

Researchers [Chapman, 2016, Martins and Gorschek, 2016, Wils et al., 2006, Cawley et al., 2010, Coe and Kulick, 2013, Cawley et al., 2015, Marques and Cunha, 2013, Chenu, 2009, Boehm and Turner, 2003, Vuori, 2011] have also proposed various adaptations to agile for its use in safety-critical systems development, but the methods proposed in the literature have failed to gain acknowledgement at the industrial level despite having some evidence of success [VanderLeest and Buter, 2009]. One of the main reasons behind early mortality of these methods

could be the lack of empirical evidence to prove generalisability of these methods. Lack of validation of studies is also reported by Martins and Gorschek [2016] in their systematic literature review and recognised as the main reason behind early mortality of new the proposed methods. Since the recommendations suggested by these studies are not validated nor the effectiveness of these recommendations is reported, we are unable to analyse the usefulness of the results of these studies.

Stålhane et al. [2013] point out the ambiguity in the language of the regulatory standards. According to Stålhane et al. [2013], many important terms are used loosely in the standards e.g., Phase, Risks etc. A word in one standard means one thing and another in the other standard. Different interpretations of a term can often mean different things. A study by de la Vara et al. [2016] also reports this issue. According to de la Vara et al. [2016], the text in the safety standards can be ambiguous and inconsistent. The terminology varies across standards e.g., "*work products"* in ISO 26262 vs. "*data item"* in DO-178C [de la Vara et al., 2016]. The inconsistencies can hinder the comparison of the standards "*...especially since there is often incomplete conceptual overlap between safety-critical domains"* [de la Vara et al., 2016]. Ambiguous terminologies is one of the reasons behind the newer version of regulatory standard for airborne software i.e., DO-178C [Spitzer et al., 2017]. Various terminologies e.g., "*guidelines"* were unclear in DO-178B which were clarified and rephrased in DO-178C.

Several studies [Wang and Wagner, 2016b, Notander et al., 2013, Siddique and Hussein, 2014, Axelsson et al., 2016, Ge et al., 2010, Abdelaziz et al., 2015, Górski and Lukasiewicz, 2012, Kuchinke et al., 2014, Shenvi, 2014, Fitzgerald et al., 2013, Wang and Wagner, 2016a, Hajou et al., 2014, Hanssen et al., 2016, McBride and Lepmets, 2016, Myklebust, 2008] point out the conflict in the quality assurance activities of agile and the quality assurance requirements mandated by the regulatory standards. Agile does not draw a distinction between the testers and developers. The testing is performed by the developers in the agile teams [Wang and Wagner, 2016b, Notander et al., 2013, Siddique and Hussein, 2014, Axelsson et al., 2016]. Whereas, the regulatory standards require the testing team to be independent. This not only conflicts with the regulatory standards, but it is very difficult for an independent team to perform the testing without the inherited knowledge of the project [Islam and Storer, 2020b].

3.4.6 Design and Architecture

Traditional methods favour end to end planning; therefore, a lot of effort is spent in creating a long-term view while following traditional development life cycles. In the traditional development life cycles, the accommodation of change is hard and expensive. The later the need for change is discovered, the expensive it is to accommodate those changes. With a sequential approach, the point of design review (the stage where the potential changes to the design are reviewed) comes very late.

Agile methods, on the other hand, believe in taking small steps. Agile methods are flexible

because of their notion of dynamic planning. Agile offers evolutionary design while the regulatory standards mandate an upfront design which also serves as an input for hazard analysis [Ge et al., 2010, Abdelaziz et al., 2015, Rottier and Rodrigues, 2008] in safety-critical systems. If we look at the requirement of an upfront design, waterfall seems more suitable lifecycle for regulated environments but Fitzgerald et al. [2013] believe that long-term view is a perception which is often not fulfilled. We need to look at different dynamics of safety-critical systems, including hazard analysis before considering agile methods.

Researchers [Myklebust et al., 2014a, Ge et al., 2010, Fitzgerald et al., 2013] argue that the design should be "*sufficient enough"* to enable the hazard analysis. The question is: how much upfront work is "*sufficient"* for a particular project [Ge et al., 2010, Chapman, 2016]?

Different hazard analysis techniques require different level of input information [Ge et al., 2010, Abdelaziz et al., 2015]. "*FMEA or HAZOP analysis, engineers not only need to know the system structure, but also the information of the effect of a failure on other components, which sometimes comes from a detailed design of components"* [Ge et al., 2010, Abdelaziz et al., 2015, Stephenson et al., 2006].

However, most of the information comes from experience of the safety engineers [Ge et al., 2010, Abdelaziz et al., 2015]. Considering iterative nature of agile development, we could suppose that we start with a minimum architectural layout and then update the hazard analysis at the major milestones. This approach is considered inappropriate by Rottier and Rodrigues [Rottier and Rodrigues, 2008] because the risk assessment (i.e., the second stage of safety engineering) cannot be performed on the system components which are not well defined.

From the above information, we can infer that the *(i)* level of detail required in the design to proceed with the hazard analysis in agile is not clear; and *(ii)* since the output of popular hazard analysis techniques (e.g., FTA, HAZOP, FMEA etc) also involves creativity and experience of the safety engineers, a complete upfront design does not guarantee a thorough hazard analysis.

3.5 Discussion

Several studies which use various research methods, on the use of agile in regulated environment, have been conducted in the past years. In a recent survey of the field, Heeager and Nielsen [2018] reviewed 51 studies published over two decades (2001 – 2018). Heeager and Nielsen found that of those studies, 10 were based on case studies and a further 5 were considered to be experience reports, such as Gary et al. [2011]. Another experience report not listed by Heeager and Nielsen [2018] is the work by Chenu [2012].

Relatively few studies have developed conclusions based on detailed interviews with practitioners. Of the existing research, McHugh et al. [2013] conducted interviews with practitioners working on the development of medical devices. Notander et al. [2013] interviewed five engineers at four different companies to understand the impact of increasing demands for flexibility

on established safety-critical development. Siddique and Hussein [2014] interviewed 21 individuals, each in different companies in Norway, to understand the practical choices made by software engineers in choosing a development method. Reporting on then on-going interview-based research, Stelzmann [2012] proposed a classification scheme for different safety-critical contexts in which agile software development is being considered or applied. Hajou et al. [2015] conducted 14 interviews with software developers in the pharmaceutical industry to understand the reasons for the lack of adoption of agile software development in that context. In particular, the authors concluded that the perceived risk of agility mitigated against its adoption.

A common theme in the work on applying agile software development in a safety-critical context has been the need for adaptation of agile methods and practices to fit within the constraints of safety standards. For example, McHugh et al. [2013] suggested that incorporating agile methods with existing plan-driven methods is the most favourable choice in the software organisation they studied. To facilitate this, McHugh et al. propose a hybrid V model which incorporates aspects of agile methods and activities from plan-driven methods.

A more extensive investigation of the integration of agile software development with safety-critical systems has been developed in the SafeScrum method [Stålhane et al., 2012]. The original motivation for this work was the integration of the Scrum method with the IEC 61508, a high level standard for safety-critical systems. The key intuition in the approach is that safety requirements change far less frequently and are far more certain than product requirements. To accommodate this, the SafeScrum method (a) focuses only on software development within the overall system engineering process; and (b) maintains separate Scrum backlogs for functional and safety requirements.

Later work on SafeScrum extended the assessment of its compatibility with a variety of other safety standards, such as in the petrochemical industry [Myklebust et al., 2016]. Other authors have also considered extensions to the original SafeScrum method, including the integration of change impact analysis into the agile change request lifecycle [Stålhane et al., 2014], safety analysis [Wang and Wagner, 2016b] and configuration management [Stålhane and Myklebust, 2015].

A limitation of much of the work on SafeScrum is the lack of case studies or experience reports, evaluating the method through industrial experience. However, Hanssen et al. [2016] undertook a two year case study of applying SafeScrum to the development of a fire detection system. As a consequence of the case study, the authors discovered the need to augment SafeScrum with an embedded quality assurance role within the development team. The duration of Hanssen et al.'s case study demonstrates the difficulty of conducting real world evaluations of methods for safety-critical systems. Equally, the work demonstrates the importance of doing so in order to identify necessary adaptations to theoretical process models.

The published research [Jonsson et al., 2012, McHugh et al., 2013, Wils et al., 2006, Chenu, 2012, Cordeiro et al., 2007, Wang and Wagner, 2016b] shows that there have been multiple

attempts to use agile methods in safety-critical environments. Many of these studies have discussed challenges and benefits of the use of agile methods in safety-critical system development. However, much of the published literature on the application of agile software development to safety-critical systems work is speculative, suggesting considerable uncertainty amongst practitioners concerning how best to proceed in applying and adapting agile software development in the context of safety-critical systems development. Also, the nature of adaptations to the agile process [Siddique and Hussein, 2014, Wils et al., 2006, Axelsson et al., 2016, Goncalves et al., 2015] differs from each other, showing that the researchers have been unable to reach a consensus on the number and nature of additional activities in agile for the development of safety-critical systems.

3.6 Threats to Validity

This section discusses the threats to validity of this systematic literature review. In this section, we have discussed three types of validity threats i.e., construct, reliability, and external validity.

Construct validity threat

Construct validity is the appropriateness of the operationalisation of concepts being studied. Construct validity threats for this systematic literature review on challenges related to the use of agile methods in safety-critical system development may include:

Limited representation of studies: The selected sample of studies may not represent the wide variety of industries, domains, or safety-critical systems in which agile methods are applied. Limited representation of studies can potentially undermine the generalisability of the results.

Ambiguous definition of agile and safety-critical systems: The concepts and definitions of "*agile methods*" and "*safety-critical systems*" may differ across the diverse range of published literature. These *terms* may be interpreted in various ways which can introduce ambiguity.

Bias towards certain methodologies or perspectives: The inclusion/exclusion criteria used during the study selection process may introduce a bias towards certain agile methodologies or perspectives on safety-critical systems. This bias could limit the range of challenges or lead to an over-representation of certain opinions.

Measurement bias: Different studies may have used different criteria to measure and report the challenges related to the application of agile methods in safety-critical system development. The inconsistencies amongst the selected studies in terms of studied samples, methods, analysis, and conclusions can make it difficult to compare and synthesise the findings accurately.

Reliability Validity Threat

Reliability validity threats for the results from this systematic literature review may include:

Inter-rater reliability: The involvement of more than one reviewer in the study selection process could lead to different interpretations of the findings and exclusion/inclusion criteria. This can introduce inconsistencies and potentially biased results.
Data extraction errors: Erroneous or partial data extraction from the selected studies can compromise the reliability of a systematic literature review. Discrepancies in the extraction process can impact the validity and reliability of the findings. Moreover, different opinions on the quality of a study can lead to inconsistencies in the evaluation of the selected studies.

External Validity Threat:

There may be a risk of publication bias, where studies with significant or positive results are more likely to be published or easily accessible compared to studies with negative results. In this case, if the systematic literature review only includes published studies, it may not capture the complete range of challenges related to the application of agile methods in safety-critical system development. This bias can impact the general understanding of challenges related to the use of agile methods in safety-critical systems.

Addressing Threats to Validity

To address these threats, we ensured a comprehensive search strategy that included multiple online databases. The concepts of agile methods and safety-critical systems were clearly defined at the start of the review and the search string used for searching the online databases was extracted from these concepts. The exclusion/inclusion criteria were defined clearly, so that the study selection process is transparent and replicable. An exclusion criteria was defined to identify the studies with non-significant results.

There were multiple reviewers involved in the selection process (i.e., the author of this thesis, his supervisor, and another colleague). A consensus was reached together with improved inter-rater reliability. Additionally, conducting regular meetings and discussions among reviewers to clarify any uncertainties or inconsistencies enhanced the reliability of the study selection. Careful attention was paid to the data extraction process, ensuring that all relevant studies were included accurately. Documenting the selection and assessment processes in detail further supported the reliability and reproducibility of the systematic literature review. By doing so, we enhanced the construct validity of the review and provided more robust insights into the challenges in this context.

3.7 Summary

This chapter provides a theoretical background for the next chapter. This chapter consists of a systematic literature review on the use of agile development for large scale safety-critical

systems. Fifty-six studies were selected for the literature review. From the literature review, we concluded that the software industry is in transition from the traditional development life cycles to flexible and earlier development. Increasing demand for earlier delivery of a working software has encouraged the organisations to consider agile methods for development. However, agile methods need to be adapted according to the need of the system under development. For this purpose, different researchers have suggested different adaptations for example, hybrid model.

Despite the attempts to use agile methods for the development of large scale safety-critical systems, there is a lack of empirical research and guidance on the said topic. The published research literature suggests uncertainty amongst the research community over the use of agile methods for the development of safety-critical systems. Therefore, as a next step, we decided to explore the use of agile methods for the development of safety-critical systems in an industrial setting. The next chapter contains the details of our exploratory study in the industry.

Chapter 4

A Case Study of Agile Software Development for Safety-Critical Systems Projects

A key finding from the previous chapter was lack of empirical research in the use of agile methods in large scale safety-critical systems. This led us to explore the use of agile in safety-critical systems in a real world setting. This chapter explores the introduction of agile software development within an avionics company engaged in safety-critical system engineering. There is increasing pressure throughout the software industry for development efforts to adopt agile software development in order to respond more rapidly to changing requirements and make more frequent deliveries of systems to customers for review and integration. This pressure is also being experienced in safety-critical industries, where release cycles on typically large and complex systems may run to several years on projects spanning decades. However, safety-critical system developments are normally highly regulated, which may constrain the adoption of agile software development or require adaptation of selected methods or practices. To investigate this potential conflict, we conducted a series of interviews with practitioners in a company, exploring their experiences of adopting agile software development and the challenges encountered. This chapter also explores the opportunities for altering the existing software process in the company to better fit agile software development to the constraints of software development for safety-critical systems. Please note that the contents of this chapter were published in a journal paper*.

This chapter is structured as follows: Section 4.1 provides an overview of the objectives of this chapter. Section 4.2 describes the research method for this study including the design of the semi-structured interview instrument and validation of the findings in a review workshop with the company. Section 4.3 provides an overview of the company, and how it approaches systems engineering, giving an understanding of the context in which agile software development is employed. Section 4.4 summarises the use of agile software development, including specific

*https://www.sciencedirect.com/science/article/abs/pii/S0951832018308597

practices, to date within the company, and how these have been fitted into the existing software development process. Section 4.5 discusses the challenges discovered from the interviews. Section 4.6 discusses the threats to validity of this study. Section 4.7 presents the summary of this chapter.

4.1 Objectives of the Exploratory Case Study

The practice of adapting and customising methods and practices to suit local needs has been reported for other software domains [Fitzgerald et al., 2006, Wang and Wagner, 2016b, Conboy, 2009]. However, there has been a very little reported in the literature of the experience of practitioners who have applied necessary adaptations to agile methods or practices in the context of safety critical system development i.e., empirical research on application of agile methods for the development of safety-critical systems. Therefore, there are many open questions about the selection of particular adaptations and their efficacy in different contexts.

To continue to address this gap, we conducted a series of semi-structured interviews with software engineers working for a large avionics company in the United Kingdom (referred to as 'the company'). The company as a whole is engaged in a variety of projects for external customers, typically comprising both hardware and software development for safety critical systems. The purpose of the study was to learn about the company's experiences in the application of agile software development to safety-critical systems projects and to gain a deeper insight into the difficulties experienced. Therefore, the objectives within the context of the exploratory case study in this chapter were:

- Explore agile methods and practices employed in the context of software development for safety-critical systems.
- Explore the challenges in employing agile methods and practices in the context of software development for safety-critical systems.

Addressing the first objective provides an understanding of the use of agile software development within the company. Addressing the second objective allows for an exploration of the impact of agile software development from the perspectives of the practitioners. We also seek to understand what challenges *they* encountered when employing different practices within agile methods, which practices were rejected and adapted, and the rationale for doing so. Due to the exploratory nature of the research, a case study approach was taken [Runeson and Höst, 2009]. An initial interview with stakeholders at the company was conducted as a scoping exercise. Following this, a semi-structured interview instrument was developed following Wengraf's (2001) method to ensure traceability between research questions and data gathered. Findings from this stage were validated in a full-day workshop with wider group of participants.

4.2 Interviews

The company that is the focus of this study is a large multi-national that develops products in the avionics sector. The company is engaged in a number of projects concerning the design and development of safety-critical systems, comprising both hardware and software. As discussed above, the company had begun to experiment with the use of elements of the Scrum process and other agile practices. During this period, the researchers were invited to conduct interviews with a number of the company's employees who had been involved in this transition process. The purpose of this study was to explore and understand the application of agile software development to the development of software for safety-critical systems from the perspective of practitioners. The study sought to identify both: the benefits recognised by practitioners in using agile methods and practices in this context and the challenges and limitations experienced. We conducted a series of interviews with practitioners at the company.

Since this was an exploratory study, and the researchers did not have prior experience of the company's work, the first stage of the research process was an unstructured interview (Interview 0) with two senior employees of the company. One of these participants, who also participated in all the following interviews, was the team lead of a systems team, which was responsible for elaborating requirements and disseminating these to other teams within a larger project. The other participant was the Head of Software Engineering, who was responsible for the overall software development function of the company. The interview meeting continued for 90 minutes. This interview was conducted in person, with one of the researchers taking extensive notes during the interview. A memo was prepared summarising the answers to the questions asked. This memo was validated by one of the interviewees during a follow-up discussion. The answers to this initial interview provided guidance to help scope the next stage of our research.

Following this stage, semi-structured interviews were used to gather data. This approach offers freedom of expression to the participants, and open-ended questions prompt discussion aiding the interviewer to explore a particular theme. Following McHugh et al. [2013], Wengraf's guidelines were used to construct the interview instrument [Wengraf, 2001]. Figure 4.1 illustrates how Wengraf's method was applied to the design of the semi-structured interviews.

The *Research Purpose* (RP), in this case: "*Learn about application of agile software development to software development for safety-critical systems and to gain a deeper insight into difficulties experienced when developing avionics systems using agile methods and practices.*". In the current work, the RP is refined into two CRQs. Each CRQ is divided into a number of *Theory Questions* (TQ), specific propositions to be investigated during the conduct of the study. For example, CRQ1 is refined into two TQs, including "*TQ1.1 What agile methods are employed in practice?*". To answer each TQ, a number of interview questions that will be presented to the participants are defined. The figure shows a sample of interview questions for TQ1, with the full interview instrument available for review [Islam and Storer, 2020a]. This approach provides a traceable hierarchy and rationale behind every interview question.

Research Purpose	Central Research Questions	Theory Questions	Example Interview Questions
Learn about the application of agile software development to software development for safety-critical systems and to gain a deeper insight into difficulties experienced when developing avionics systems using agile methods and practices.	1. What aspects of agile methods and practices are being employed in the context of software development for safety-critical systems?	1. What agile methods and practices are employed ?	Customer Involvement 6. Are multiple releases delivered to the customer during a project?
		2. What customizations have they made to the method and practices they are employing?	Requirements 9. How are requirements managed during elaboration/change/evolution?
	2. What are the challenges in employing agile methods and practices in the context of software development for safety-critical systems?	3. What benefits did they expect from agile software development?	Requirements 4. What proportion of the requirements specification requires change?
		4. What benefits were they able and not able to achieve?	Requirements 10. How often are requirements reviewed? How is this done?
		5. What are the potential conflicts of agile software development with regulatory standard(s) (i.e. DO-178C) ?	Quality Assurance 4. How does certification drive quality assurance practices?

Figure 4.1: Research question construction process following Wengraf's method [Wengraf, 2001] for the interview instrument used in the Avionics Company.

Participant and Role	Experience of Agile
P1: Lead software engineer	Using agile and practices within current team; experience of using agile on previous projects
P2: Lead software engineer	Experience of using agile software development in previous projects; Considering the use in current project
P3: Deputy lead software engineer	Using waterfall
P4: Lead software engineer	Using waterfall
P5: Systems team lead	Using a hybrid model (water-scrum-fall)

Figure 4.2: Summary of Interview Participants

Once an initial version of the interview instrument was prepared, it was validated by an independent academic expert who did not have any involvement in the research. The validator was contacted by email to arrange a teleconference during which all questions in the interview instrument were reviewed. The validator advised altering the order of questions to facilitate the interview process but did not recommend changing the content of any questions. A series of mock interviews were also conducted with non-participants in the study to familiarise the researchers with the structure of the interview instrument and to test the timing and duration of the interviews.

Four interviews were conducted during four sessions. Our intention was to gather data from multiple perspectives within the company, creating a broader understanding of the context of this chapter. Interviews were conducted with five practitioners (Participants P1-P5) with different experiences, expertise, and roles. These experiences included acting as a project manager, requirements engineer, software developer and a member of an integration team. The fifth participant, P5, was a systems team lead and participated in all the interviews. The first four interviewees were working on three different projects within the company. The first team had some experience of employing agile software development within their projects whereas the second software team was considering its use because they wanted to be able to deliver more frequent releases. In both cases, the participants interviewed had used an agile method and associated practices in their *previous* projects within the company. However, the third software team was reluctant to adopt agile software development and wanted to retain their existing plan based process, which resembled Waterfall [Benington, 1983]. The third team felt that they worked effectively within this process and although aware of the use of agile software development elsewhere within the company, did not see the need to begin introducing an agile method or practices to their own software process. All the participants, including the ones with experience of agile software development within the company, worked on avionics related projects requiring D178-C certification. A summary of the interview participants is presented in Figure 4.2.

The approximate duration for each interview was 90 minutes. Interviews were transcribed and sent to the participants for validation, permitting participants to make additions or clarifications. After getting verbal permission from the participants, the transcripts were used for analysis. The transcripts from the interviews were then analysed to answer the theory questions. The analysis of the gathered data is also performed by using Wengraf [2001]'s guidelines, using a bottom-up approach to answer the questions at each level.

For the analysis, answers to the questions were gradually aggregated at each stage in the hierarchy. A table was created similar to Figure 4.1 for this purpose. Answers to every interview question from all participants were pasted in the Answer column next to the respective interview question. Answers to every group of IQ relating to each Theory Question were then merged to form a story. The group of Interview Questions relating to each Theory Question was deleted such that each Theory Question had a descriptive answer. The same process was repeated again to find answers to CRQs.

The descriptive answers to each CRQ were reviewed by the authors independently, and the issues reported in them were highlighted. The notes were compared afterwards in a meeting to discuss the discovered issues. Eleven challenges were identified during this data analysis. These results were presented to a group of people from the company for validation. The participants in the workshop validated all the challenges identified during the interviews, with the exception of one. In addition, the participants of the meeting raised three new challenges which were not discovered during semi-structured interviews. All fourteen of these challenges are discussed in Section 4.5. As a result, we also gained an understanding of the factors that directly or indirectly affect and contribute to the actual and perceived benefits of agile software development within the company. At the end, the findings from the interviews were mapped to findings in the literature. Note that where we use quotations below to illustrate a challenge it is sometimes necessary to anonymise some of the topics to preserve confidentiality. All the work described in this section took place between March 2017 and March 2018.

4.3 Overview of Software Development in the Company

This section draws on the analysis of the answers to the interview questions to develop a description of the structure and process for software development used by the company. The description below provides the context for the discussion of challenges which were identified during the interviews and discussed in Section 4.5. Each theme discussed below was identified in the interviews as having an impact on the introduction of agile practices to the software teams. The Section begins with an overview of the a typical project team structure, organised to accommodate both hardware and software development processes. The section then describes the *overall* software development process within the company and where agile software development has been adopted within individual sub-teams. Next, the section describes the relationship between

a typical project in the company and a complex network of project customers. The next section reviews the requirements management process, showing how requirements derived for the overall project are communicated to the software teams and sub-teams. Finally, the process of delivering and certification for products according to safety standards is described.

4.3.1 Project Team Structure

The size of project teams within the company varies considerably, typically between 50 and 200 people. Within a project, a software development team (SDT) itself typically comprised of 20 to 35 people, with the rest of the project team working on different other components or functions within the project, including the systems integration team, hardware, firmware, software, safety, flight trials, configuration and the management team.

The SDT has its own organisational structure. The overall team has a small management unit, comprising a lead software engineer, deputy lead software engineer, program manager and coordinator. The lead software engineer and deputy lead software engineer share technical and managerial responsibilities for the overall project. These include the overall software lifecycle, comprising requirements, definition, design, software implementation, quality assurance, certification and delivery. The lead software engineer is also responsible for customer liaison and has sign-off authority for documentation and software changes. The lead software engineer is also responsible for assigning responsibilities to individual software sub-teams. The software program manager has responsibility for project planning within the software team and resource allocation. Finally, the software coordinator is responsible for maintaining documentation, for example, meeting minutes.

A software team is typically divided into a number of sub-teams, which specialises in a particular functional aspect of the software project and consists of either four or five people. Each sub-team has a sub-team leader, who is expected to be able to run a full lifecycle including high level design and requirements analysis within their area of expertise. The sub-team leads also act as *functional champions* because of their expertise in some area of functionality. The sub-team leaders typically have 15 to 30 years of experience. Other members of the team have different level of experience, from recent graduates to 20-30 years of experience.

4.3.2 Development Process

Most of the projects within the company, including the participants' current projects, are planned to run for several years and are divided up into a number of *phases* with each phase intended to deliver further new functionality on the product, as agreed with the customer(s). The duration of a *phase* varies from project to project. In some projects, a phase is between four (4) and six (6) months and in others, a phase is between one (1) year and eighteen (18) months. Each phase is allocated a number of requirements to be implemented, agreed with the project customer. At the

end of each successful phase, a delivery is made to the customer comprising (in the ideal case) the features of the requirements that were originally agreed upon.

A typical phase is illustrated in Figure 4.3. Requirements are created in the IBM DOORS documentation tool by the systems team and later exported into the IBM Rhapsody modelling tool used by the requirements analysis sub-team within the SDT. The requirements analysis team translates the requirements into a high level software architecture. During this process, the software team and systems team are in constant communication, due to the need to further negotiate and clarify the requirements. Once the requirements and architecture are agreed upon, they are allocated to different sub-teams by the requirements manager. Within each sub-team, the company allows some flexibility with regard to the software process, for example, with some sub-teams using a Waterfall software process within a single phase and others applying the Scrum method. Consequently, one participant (P2) called their software process "*water-scrum-fall*", as Scrum was inserted into the middle of the company's overall project lifecycle. Towards the end of a phase, different functions of the software are packaged into an integrated software release. The software is delivered to the integration team to develop an overall delivery release to the project customer.

There is a set practice of having a weekly technical and management meeting and a monthly software team meeting. Minutes and actions are captured at the meetings and distributed only to the relevant people. Other than the formal meetings, spoken/face-to-face communication is the main type of interaction that takes place between the software team and other teams. Within each sub-team, members are co-located and interviewees report that the culture within the company encourages workplace interaction.

4.3.3 Project Customers

From the perspective of a project software team, the relationship with the project customer was viewed as complex, with the project actually having several 'layers' of customer (Figure 4.4). The systems team acts as the most immediate customer for the software team, providing the requirements specification (recall Figure 4.3). In turn, the systems team manages the relationship with the project's immediate external customer. The systems team is therefore responsible for gathering requirements from the external customer. As the company may be part of a larger project consortium, the external customer may itself also have a further external customer who will have a significant influence on the direction of the project. Alternatively, the system under development may have several direct customers. In all these cases the software team may find themselves interacting less frequently with these stakeholders, or doing so through informal communication mechanisms, indicated by the dashed arrows in Figure 4.4.

One of the interview participants (P5) described this as "*a very complex stakeholder relationship in terms of lots of people with different views and influences.*" The customer has a certain delivery schedule which has the main influence over the overall schedule. The interview

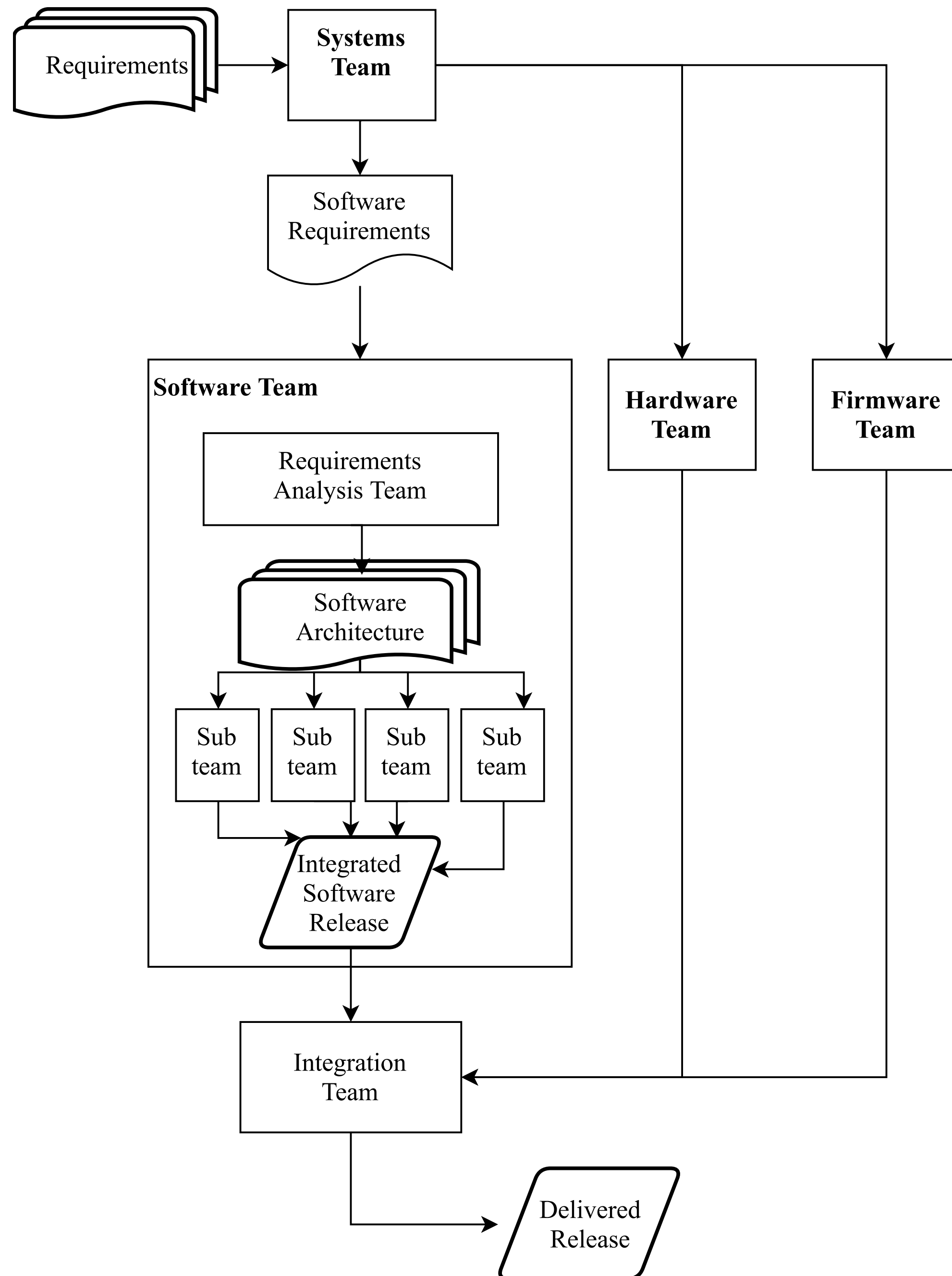


Figure 4.3: A typical phase of a project from the perspective of the Software Team

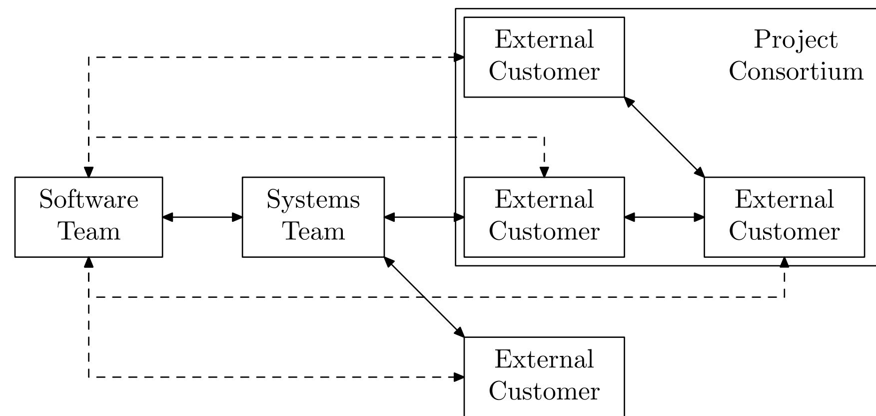


Figure 4.4: Layers of Customers. Solid arrows represent formal lines of communication. Dashed arrows represent informal or infrequent lines of communication.

participants reported that in the past, the overall project management team decided the project schedule, but now the software team also gives their input on tasks and schedule. Although the wider project management team sets the major milestones in agreement with the external customer, the software teams set their own milestones within these boundaries. This gives the team members a sense of ownership and responsibility. Agreed delivery dates are then passed onto the external customers. Normally, the software team would involve more people if there is a risk of missing the delivery date, but if the schedule needs to be changed, it is done after negotiation with the external customer. Final decision about changes to a schedule is made by the Software Function lead.

For the software team, the "*customer*" is primarily the project's systems team, who partitions and allocates requirements to teams within the project. Consequently, the systems team is usually one or two delivery phases ahead of a software team. For example, the systems team will be preparing requirements for the second or third phase while the software team is working on the first phase. The main involvement of a systems team is in the beginning (elaborating requirements) and at the end (completing integration) of each phase. A systems team does not participate in the feedback reviews regularly, but if there is a very complex task (a complex algorithm to be implemented, for example), they would get involved. The systems team also provides inputs for acceptance testing.

The interview participants reported that in the past, their software team has had ready access to the systems team, who can be approached on a needs basis. However, there is no pre-defined way of soliciting feedback from the respective systems team. Rather, it is mostly informal, whenever needed. Conversely, gate reviews and interim reviews are formally performed with the external customer (representatives). Normally, it takes more than six weeks to get feedback on a delivery as the customer requires this time to test the new features on the integrated system. Certification also delays delivery sometimes.

4.3.4 Requirements Management

Requirements are analysed and refined at the start of each iteration. At the end of requirements analysis phase of each iteration, the requirements are reviewed by a panel which involves the software team lead and software engineers. Requirement specifications are delivered to the software teams in textual form with some supporting UML diagrams to help the engineers understand the requirements. Requirements are managed through the IBM requirements management application, DOORS. The interview participants reported that requirements analysis and decomposition is a challenge and depends on an engineer's familiarity and experience with the nature of task to be performed well. There is no typical number of requirements for a phase. The average number of requirements per iteration is unknown because it depends upon the amount of work required to meet a particular requirement, due to the unequal size of requirements.

One software team had experimented with converting requirements into more formal structured text. They converted the requirements from free text into a structured Z notation. However, one participant (P1) reported that this turned out to be a "*disaster*". According to P1, the customer reported their displeasure with the transformed requirements because they were less readable than the original.

The interview participants reported that requirements change was experienced in all projects. One participant estimated that 10% of the requirements changed throughout the software lifecycle. Changes were reported due to a variety of sources, including requests from customers, the discovery of conflicts between the architecture and requirements during implementation or the need for further requirements elaboration or additional scope. The need for a change in the requirements can be discovered at any stage from requirements analysis to delivery. Participants also reported that the discovery of requirements changes often necessitated rework or coordination with other teams in the project to assess impact, particularly the project's systems team. It was also observed that requirements tended to stabilise towards the end of the project.

4.3.5 Product Integration and Certification

Integration and certification is performed iteratively, beginning within the software team, before an entire product release is provided to the customer. Certification occurs when a *formal release* is due to be delivered to the customer. Also, an integral part of the integration process is the preparation of supplementary documentation to support certification processes. This documentation includes requirements specifications, risk management plans, accomplishment summaries, release information and high level and subsystem design documents.

Software teams manage all their documentation and design models locally using the Serena Dimensions configuration management tool and generally only have visibility of other teams' documentation during the integration and certification process. Documentation is reviewed whenever a significant change is made as well as during the certification process. Documen-

tation is formally reviewed during a lifecycle in the appropriate phase. For example, test reports will be reviewed in testing.

More recently, projects have used a practice of delivering *engineering releases* as well as the end of phase *formal* releases. Although these are releases that are provided to the customer, they are done so in order to generate feedback and do not undergo the whole certification process.

The participants reported that some visibility of progress is lost during the integration process. This happens because during the integration process, there are many other ways of tracking progress, and it is possible that software team members do not update internal issue tracking (such as Jira) because this creates duplication of work. Moreover, if a problem arises in integration, it is recorded via a project wide defects recording tool, and the respective software team involves the people they need immediately in the task. Thus the benefits of internal progress tracking within the team are lost during integration.

4.4 Use of Agile Software Development

This section discusses the extent to which the company has so far used agile practices, building upon the Section 4.3 to meet the first objective of this chapter. Each team has some flexibility in choice of software process, depending on the nature of the overall project, with the final selection of lifecycle being made by a team's lead software engineer. The company has developed a series of questions that guide the selection of a software process. Historically, teams have typically employed Waterfall or an iterative process because of the duration of the projects.

Two of the interviewees had previously worked in software teams that employed agile methods. In their current projects, one participant had also begun employing elements of Scrum, several months prior to the interviews. Several motivations for this were given during the course of the interviews:

- The need to speed up delivery times and produce a series of phased releases for the customer. The second team reported that this goal had not been reached yet, although the first team found employing aspects of agile methods had resulted in significant benefits. One participant (P5) commented that they wanted to be "*...giving the customer many more releases"*. Another participant (P3) with no experience of using agile software development, while expressing his expectation from its adoption, emphasized the need to deliver more frequently "*...we would be able to provide the customer with more frequent deliveries of the software"*.

- Improving communication within the software team. One interviewee (P1) reported that "*...we wanted more visibility in the project i.e. who is doing what?, how many tasks have been completed?, estimates, performance and list of completed jobs etc."* Tools like Jira Kanban boards were reported as helpful in this regard.

- Improving team member engagement with the coordination of the software project. Freedom to select one's own tasks has prompted a sense of responsibility among team members. While expressing benefits of using Scrum, a participant (P1) said "*The level of engagement of some of my engineers is much better... That is the massive difference, my teams are working much better.*"

- Earlier discovery of problems. The interviewees reported that problems were often discovered late during integration, requiring more costly rework. One participant (P1) while talking about reasons of adopting agile software development commented "*...not letting things get too far before realising its gone wrong. It's that visibility thing. It's about knowing about problems sooner"*. Another participant (P3) who did not have any experience of using agile software development, while discussing the reasons seen for using agile in other projects, said "*...so we get feedback earlier.*"

The Scrum method itself had been selected by these teams for this part of the process because it was perceived as the de facto industry standard, and within the scope of a sub-team, did not require senior management support to allow the experiment. At the time when interviews were being conducted, the organisation had not undertaken significant Scrum training for its personnel. Rather, individual teams had chosen to adopt agile methods and practices within their own parts of a wider project.

Each team has a scrum master responsible for coordination of activity. Project planning is organised into a series of *sprints* with associated planned releases, with each sprint typically lasting one or two weeks. The team creates a plan at the start of each sprint, using a Jira or Kanban issue board to track progress. The scrum master begins by calculating the available effort in terms of *story points* in the sprint based on team size and availability. The teams do a "*T-shirt size"* estimation of the tasks and record this on Jira boards. Numerical information is extracted with the help of a formula from T-shirt estimation and entered into a Microsoft Project plan for long term planning. Items from the backlog are then selected for completion and allocated to the sprint.

The interviewees reported that the first and second teams follow the daily stand-up ceremony to facilitate communication. In the second software team, the lead software engineer acts as the product owner, so the team also conducts customer demonstrations. However, the first software team does not have customer demonstrations because they do not have a product owner within the team. Customer demonstrations are also interpreted as something that induces a sense of failure or inability to finish the task on time, by the first software team we interviewed. According to the first software team lead (P1) "*...at times the teams don't feel failure, and I know that meeting (customer demonstration) helps with the feeling of failure, which would be nice sometimes...It helps with building reasonable pressure on the team member.*" The interviewee suggested that the first software team lead would rather have a 'mock' meeting with another

internal team than the external customer because they have a very formal relationship with the customer with whom the team feels unable to discuss delays. Neither of the two software teams currently conduct retrospectives. It was stated that the first team does not see the value in it because they are already monitoring progress through Jira boards. Therefore, they do not see the need of having a separate meeting for looking at previous performance. Separately, the second team reported that they had experimented with retrospectives. They reported finding the number of potential process improvements to consider to be overwhelming and so had abandoned them until additional Scrum training could be completed. However, the available literature advocates conducting retrospectives. For example, Kasauli et al. [2018a] performed a mapping study on the use of agile in safety-critical systems development. The authors conducted a workshop with experts from six large Swedish product development companies to identify critical aspects in this topic which were to be further investigated. Then as a next step, the authors performed a systematic literature review during which they selected 34 studies for further analysis. According to one of their findings, sprint planning combined with retrospective helps in improving estimation in safety-critical system development.

The team members are encouraged to communicate and help each other, and use this as a means of learning. Pair programming is viewed only as a form of mentoring in the organisation and people have different opinions about it. Pair programming is found to be ineffective and a time wasting activity by one of the participants because it has been observed that the weak member does not learn from it, and mostly, the stronger member takes the keyboard. According to the lead software engineer (P1) "*...someone always takes a back seat while the stronger member takes the keyboard.*" Others take pair programming purely as a way to "*help each other out.*" However, this negative opinion about pair programming strongly contradicts with what is reported in the literature. The researchers [Williams and Kessler, 2003, Chong et al., 2005, Hannay et al., 2009, Jones and Fleming, 2013] agree that pair programming not only improves productivity and code quality but provides a learning opportunity to the "*pair*". According to Williams and Kessler [2003], pair programming involves assumption of roles i.e., a driver and a navigator. They switch roles periodically. Results from an experimental study by Jones and Fleming [2013], show that pair programming improves technical skills along with production of high quality code in a relatively shorter amount of time. In order to understand the circumstances which led to interviewees' negative opinion about pair programming, we need to observe their pair programming sessions in a future study perhaps.

Despite the perceived benefits, participants P1 and P2 also reported drawbacks of employing agile software development. First, the team has discovered that applying an agile philosophy to design is creating rework because short term design decisions are later discovered to be incompatible with the overall product design, "*...this is because the teams think, since they are working agile, they are concerned (only) with the part they are working on.*" The Lead Software Engineer (P1) for the first team further suggested that the team needs some "*forward thinking*", for

example, to anticipate the need for extension points in design. The first software lead engineer felt that the Waterfall process helps with this issue by encouraging a more holistic approach to design.

The third team was using a Waterfall process, rather than an agile method. When we asked them if they would consider applying agile software development in the future, several reasons were given for not doing so. First, the third team believed that agile methods and practices are not suitable for projects where requirements are uncertain or volatile. One of the Lead Software Engineers (P4) said "*I think one of the reasons we've not gone agile is experience. You know we're experienced with the lifecycles that we follow.*" This was a common reply from the team members who were reluctant to use agile software development. The project they worked on was for a new product, but they based their software process on that for a long standing (more than 20 years) project within the company, "*...it used fairly similar processes all the way through. So for us on the new product it made sense to stick with the non-risky strategy of going with what we'd done previously. We know that process works, we know, what we're going to get out of it (P4).*" We believe that there were two main factors behind the teams' reluctance to use agile i.e., *(i)* general misconception about agile that it lacks focus on documentation, *(ii)* lack of experience and guidance in using agile methods.

Kasauli et al. [2018a] reported challenges of using agile methods in safety-critical system development in their mapping study. According to the authors, agile methods often reported to suffer from lack of focus on documentation "*..as extensive documentation will diminish advantages of agility"*. This perception is incorrect and probably comes from false interpretation of agile manifesto. Agile principle "*working software over comprehensive documentation"* is seen as minimum or no documentation as reported by Sharp and Robinson [2010]. In safety-critical system development, major portion of documentation is used to demonstrate regulatory compliance without which the system cannot be used legally. Therefore, in safety-critical system development, documentation is part of a "*working software"*. The focus of this agile principle is on avoiding waste of effort. It does not mean that documentation should be avoided [Wagenaar et al., 2018]. Kasauli et al. [2018a] also report that, in the context of safety-critical systems development, agile methods are thought to have lack of guidance on project monitoring and control. Also, according to the findings of the authors, there are doubts in sufficient level of testing mechanisms offered by agile. Authors report that agile's "*unstructured nature"* lead to lack of trust in agile. Also, the regulatory standards suggest upfront planning which is against the recommended workflow of agile.

In particular, the team believed that the requirements for the project were relatively well understood and stable, so the team was able to plan Waterfall phases of 9 - 12 months duration, "*So I guess it was a sort of macro-agile process... but the sprints were just incredibly long...But each of those was separate in a way (P4).*" The team used waterfall but incorporated sprints in the development life cycle to build the system in increments, but these sprints were much longer

than the normal (week or two weeks) duration recommended by agile. However, the duration of these "*sprints*" is unclear.

Second, the third team believed that applying an agile method only within the software team would create difficulties for coordination with the other teams in the project (hardware, firmware, integration etc.). A software team does not work in isolation as said by one of the participants (P4) "*...we do work very closely with the systems team to define our requirements, we work very closely with the hardware and firmware teams to integrate software, so those, you know would all need to be working to the same schedule, and the same set of sprints.*" Consequently, there was a concern that applying an agile method and practices within the software team would complicate this as different teams work at their own pace and the schedules between different teams often mismatch, "*...they may not be working to the same schedule as we are...that's not something we're very good at (P4).*"

More widely, the third team believed that the company as a whole lacks guidance on how to adopt agile software development when developing software for safety-critical systems and are uncertain about the suitability, "*...there's always been a fear of certification... and how an agile development would affect that?.*" The Lead Software Engineer (P4) for the third team also pointed towards the need to change the mindset of the people saying "*...within the business there is a fear or a concern that doing something an agile way means doing it in a scrappy way,.. you know or doing it in a careless way.*"

These concerns were also reflected in the experiences of the first two teams in employing agile software development. The participants from these two teams reported both internal and external obstacles, both in convincing team members of the benefits of change and in engaging with the project customer. They found that the application of an agile method was constrained by the customer's desire for a form of contract that encouraged a plan-driven software process. For example, the requirements phase is associated with a milestone in the contract for delivering a full requirements specification before design and implementation work proceeds. Further, the participants believed that the regulatory framework also dictated a plan driven process. Both software team leads argued that "*...regulatory standards do not let us choose our own method (P1).*" In addition, these regulatory standards require production of a lot of documentation.

As a consequence, both participants that had experience of agile software development picked up the parts of Scrum and agile practices that they thought were beneficial and could be applied without conflicting with regulatory standards of project contracts. These included the use of Kanban boards in Jira, sprints, daily stand-ups, sprint planning and product backlogs. Further, both teams anticipated employing more agile practices, such as the specification of requirements as user stories, in the future. Conversely, both software teams wanted to re-instate the Gate Reviews they were conducting while using Waterfall but they had not reached an agreement yet on how to do this within their agile software development process. Both these teams expected that employing agile software development would enable them to deliver smaller, in-

cremental improvements of the overall system more frequently to the customer over the lifecycle of the project, compared with their existing plan-driven process.

4.5 Discussion of Challenges

This section discusses the challenges in implementing agile software development within the Company, building upon Section 4.3 and 4.4 to meet the second objective of this chapter. Figure 4.5 presents the key challenges elicited during the interviews. We discuss these challenges under the distinct themes of Pressure for Waterfall, Coordination amongst Stakeholders, Documentation Demands and Cultural Challenges, below. For each theme, we present and discuss extracts from our interview transcripts where relevant observations of interest are made. For each theme, we also identify relevant literature and discuss the implications of the findings. At the end of this section, we discuss how the current agile methods tailored to large-scale systems could contribute to solve some of the challenges reported in this study.

4.5.1 Pressure for Waterfall (Challenges 1, 2, 3, 4, 5)

Challenges 1, 2 and 3 reflect the difficulties of implementing agile software development in a wider software development culture where the Waterfall process has become embedded. All the participants except one said that regulatory standards are one of the main hurdles in use of agile software development. They anticipated that Waterfall imposed by standards would prevent the use of an agile method (Challenge 2). For example, "*Our standard says that we use waterfall... it doesn't say that we can pick our method (P1)"* Further, the participants stated that the company's internal standard, which conforms with DO-178C prevents the use of agile methods, and that customers are also wary of such an approach. However, the other participant (P4) argued that "*...No, I don't think there are any conflicts...I can't see really why it would be a problem."*

Reflecting on the emphasis on Waterfall in the standards, the participants also reported that the use of Waterfall is often mandated by the (external) customer which restricts them from using agile software development (Challenge 3). Within the company, contractual agreements are the primary driving force of a project. Plans, milestones, term, and conditions of a project greatly impact the development lifecycle of a project, "*the customer is saying, we want you to use waterfall... because of the way we get a set of contractual requirements and we must complete all those contractual requirements, rather than create a set of requirements then cut dead at a certain point (P1)."* This perspective reflects the culture within safety-critical systems development of defining the full requirements at the beginning of the project because of the need to understand the full features of the software, and how it will integrate with the hardware. As a consequence, most participants believed that use of agile software development in full was not practical because this would require a different relationship with the customer in which requirements were continually refined and renegotiated at the beginning of each sprint or release.

	Challenge
1	Agile software development advocates incremental design, but safety standards require upfront design as necessary input for hazard analysis.
2	Regulatory standards are perceived as mandating Waterfall and not permitting agile methods.
3	The prevalence of fixed price contracts for pre-agreed requirements in safety critical systems projects is not readily compatible with agile software development.
4	The actual time taken to complete the tasks always turns out to be more than it is estimated in the beginning, particularly due to integration complexity in safety-critical systems projects.
5†	Requirements are difficult to modularise in safety-critical projects because the functionalities are so interdependent that it is very hard to separate them.
6	Software teams lose visibility during the integration phase. Agile methods lack guidance on integration with hardware.
7	There is a complex network of customers that obstructs agile ceremonies such as the Sprint Review
8	Face-to-face informal contacts dominate communication, causing project related information to be lost.
9	Software, hardware, firmware and other teams in safety-critical systems work function independently according to their own schedule causing plans to become mismatched.
10	Frequent releases increase overheads and costs because they must be accompanied by supplemental documentation to achieve certification.
11	The Software team has no practical example to follow for applying agile methods and they lack the resources to experiment.
12†	The teams need guidance on how to scale agile methods for use in large multi team context.
13†	The organisational mindset require convincing about the benefits of agile software development.
14‡	Independent testing required by standards conflicts with the practice of developer created tests advocated by agile software development.

Figure 4.5: Summary of challenges identified during this study. Unmarked challenges were discovered during the semi-structured interviews and confirmed in the validation workshop. Challenges marked †were discovered within the validation workshop. The challenge marked ‡was discovered during the semi-structured interviews, but rejected during the validation workshop. All challenges (including 14) are reported for completeness.

In the literature, VanderLeest and Buter [2009] argue that "*Contractual models in aerospace expect firm-fixed estimates of large complex projects with little room for change. The agile approach of using client-driven adaptive planning at the start of each iteration faces the hurdle of dealing with the potential contractual changes that result from such frequent planning.*" Limited support for subcontracting is a connected limitation of agile software development reported by Turk et al. [2014]. Sub-contracted tasks are usually well defined, and the milestones are clearly laid out [Turk et al., 2014] which already gives a limited freedom to the development team and the remaining "*flexibility"* is constrained by regulatory standards.

There are mixed opinions about use of agile software development for software development for safety-critical systems in the literature. For example, VanderLeest and Buter [2009], Cawley et al. [2010] and Wils et al. [2006] all argue that DO-178C does not favour a particular software development lifecycle, but rather provides process guidelines and (in total 71) objectives for development of airborne software [Coe and Kulick, 2013]. Wils et al. [2006] argued that a reasonable re-interpretation of agile principles would mean they are compatible with certification. In particular, Wils et al. contend that working software in this context comprises both the implementation and the documentation, because the documentation is necessary for the software to be certified as safe to enable use.

Conversely, Winningham et al. [2015] argue that agile methods and practices are not developed for safety-critical systems. In order to be used for safety-critical systems such as avionics, the software process has to conform to process standards i.e. DO-178C in the context of the current study [RTCA]. Several authors have identified and discussed specific conflicts. Relevant to our work, agile principles discourage the development of detailed designs that anticipate future requirements prior to implementation work. Beck and Andres [2005], for example, allude to the 'you ain't gonna need it principle' and argue that the expectation of requirements change means that any effort dedicated to design for future implementation could well be wasted. However, Chapman [2016], Chapman et al. [2017], Cawley et al. [2010], Wils et al. [2006], Chenu [2009], Glas and Ziemer [2009], Boehm and Turner [2003] and Coe and Kulick [2013] all contend that this principle conflicts with most safety-critical standards that mandate the development of a sufficiently detailed design to act as input to certification processes. Changes to the design may invalidate the certification status of the product and require an extensive rework of assurance related artefacts.

One particular impact of this emphasis on Waterfall reported by participants is the extent of detailed requirements analysis, specification, and design that take place before implementation work proceeds (Challenge 1). These processes are accompanied by gate reviews to evaluate the quality of work before permitting a project to proceed to the next phase. Our participants said, for example "*The design itself, we tend to come up with fairly stable architectural designs quite early on... Specifically because we don't want to be changing them all the time (P4).*"

This issue was explored further with the participants. During discussion, it emerged that

several of the participants *preferred* to engage in substantial upfront design, regardless of the constraints imposed by the standard. This preference was justified by the scale of the system development and the need to accommodate future planned features within the existing design, "*I think you need to be forward thinking as to what your design needs to be (P1).*" One of the participants also stated that adopting an agile approach to design increased costs of this aspect of the work overall because the team did not design with future requirements in mind and so created substantial additional rework, "*What I guess was not anticipated was the amount of rework that agile is creating for me... I think you need to be forward thinking as to what your design needs to be (P1).*" The participant goes on to explain that this anticipatory design is necessary because of the interdependence between the different teams in the overall project. The team needs to be aware of the expectations of other teams on the software they are working on and anticipate this in the design.

Despite the preference for upfront design, all of the participants noted the tendency for the software projects to undergo substantial requirements and consequent design changes once implementation begins, with estimates ranging from between 10% and 20% although one participant estimated that *deviations* from the original plan could reach 80%. Our participants also reported that these changes could come from the customer or from the software process itself (such as the need for further elaboration) and occur throughout the software process.

VanderLeest and Buter [2009] quote findings from different studies suggesting that a typical project may experience 25% change in requirements, increasing to 35% for a a large project. These estimates suggest that there is considerable variability within 'safety-critical' projects as to the degree of certainty in the project requirements and plan, and thus the feasibilty of applying a plan-driven process. On the one hand, the extent of volatility in requirements for safety-critical systems suggests that adopting an agile method or practices would be appropriate for requirements engineering in this context. However, there is a need to understand how agile methods and practices can be adapted to accommodate the need for continual certification against standards. As discussed above, SafeScrum [Stålhane et al., 2012] is an indication of the interest in this area. There is also a need to extend agile methods and practices to mitigate changing requirements across software, hardware and other developments, as discussed concerning Challenge 6 below.

Related to this, one participant in the validation phase workshop identified a further challenge with the modularisation of requirements (Challenge 5), stating "*We get over 8000 pages of requirements and it becomes really difficult for us to isolate a sub-set of requirements from a big pool of requirements (Validation Workshop).*" The sheer amount of detail in the fully elaborated requirements document makes it difficult for the software team to allocate packages of functionality to the different sub-teams. Later design and implementation work reveals interdependencies between functions that were not anticipated during the requirements analysis phase. This requirements complexity would appear to be a significant challenge for the implementation of agile software development, since requirements cannot readily be divided into modular,

manageable features.

As a result of these constraints, the participants reported that they feel the time and amount of work needed is nearly always underestimated, and that delays occurred due to the additional effort needed to better understand or implement altered requirements (Challenge 4). One participant (P3) commented, "*there is a high level of, what we call punt... in our system level requirements which then obviously impacts us downstream.*" The fixed price approach to contracts and the estimation process does not anticipate this cost of change. In particular, one participant noted that even though change occurs in project requirements or plans, due to requests by the customer, this does not always get integrated into the estimates for the overall plan "*...but a lot of this time, changes come in that are not considered (P3).*" We believe that along with other factors, fixed contracts also contribute to the issue of estimation in this context. Agile methods, by nature, have a short-term vision. Agile advocates accommodation of change and can handle uncertain situations; therefore, some researchers [Turk et al., 2014, Alsaqqa et al., 2020, Richardson et al., 2020] propose using short-term contracts to match the short-term planning nature of agile methods.

However, in some cases, the participants did report being able to rely on historical data from previous projects to produce reliable work estimates, "*it was a fairly mature, although [it]'s a new [product], our ... product line is very mature, you know. So the requirements, eighty percent of them probably were very well understood at the beginning of the project (P4).*" Also, to process historical data, Duszkiewicz et al. [2022] developed a tool in collaboration with a Danish software development company that employs Natural Language Processing algorithms to find past similar user stories and retrieve the time spent on them. However, the evaluation of the tool shows that different phrasings and wordings can impact the similarity scores.

Similar challenges have been identified in the literature. For example, Wils et al. [2006] reported the finding of their study of implementing XP, conducted at Barco (a major Belgian avionics equipment supplier). The company employed XP in order to reduce time-to-market and respond quickly to change in requirements. However, during the study, it was found that the software project was dependent upon external factors that were hard to control, such as delays in automated testing and mismatched hardware development schedules.

Large systems engineering projects often depend on significant upfront design as a means of coordinating effort between different sub-teams working on software, firmware and hardware elements [Chapman, 2016, Chapman et al., 2017]. In addition, requirements and design documentation serve as inputs for hazard analysis and other safety certification processes which begin while software implementation is still underway. For example, DO-178B/C requires early completion and approval of Plan for Software Aspects of Certifications (PSAC). Later changes are difficult because the PSAC has to be updated and re-approved [VanderLeest and Buter, 2009]. Therefore, some level of detailed design documentation is required for this purpose.

However, many regulatory standards, such as DO-178C [RTCA] do not prevent changes

to software design because having a rigid upfront system design that cannot be revisited and changed is unrealistic. The concern here then is how much upfront design do is needed and how much change to a design can be accommodated by safety analysis processes. Ge et al. [2010] demonstrate that design can be simple but detailed enough to allow preliminary hazard analysis. Ge et al. have used the term "*sufficient design*" to refer to the level of detail in the initial design without explaining the minimum level of detail needed to conduct preliminary hazard analysis. Critically, there is a need to develop a design process that copes with both evolution and satisfies the needs of existing hazard analysis techniques, or develop a hazard analysis technique that copes with evolutionary design.

Advocates of agile software development, such as Beck and Andres [2005], advise against undertaking detailed software design work prior to implementation, arguing that without sufficient information about the problem domain and associated constraints, any proposed designs will be subject to change once implementation begins. One potential direction to address this problem may be to extend the practice of system metaphor definition in the XP agile method to encompass the need for some *anticipatory* design desired by the participants.

Despite these challenges, the participants reported considerable experience experimenting with agile software development, making adaptations to fit their needs. Winningham et al. [2015] note that agile methods were not developed for safety-critical systems and that consequently, many practices within agile methods need to be compliant with standards, such as DO-178C [RTCA]. Coe and Kulick [2013], Boehm [2002], Boehm and Turner [2003] suggest that methods such as Agile-Planned that combine elements of both philosophies show promise in this context. This selection and adaptation of elements was reported by the participants. As one of the participants (P1) described it "*We follow some bits of agile that are of interest to us..*" The participants reported that their teams participated in a variety of Scrum 'ceremonies' including sprint planning, daily standups, customer demonstrations and retrospectives, although all participants reported adaptations, or the non-use of a ceremony, which we examined further.

In particular, two of the participants reported conducting frequent retrospectives, reflecting the use of Scrum within their teams, whereas, the other two participants reported undertaking less frequent "*lessons learned*" within their projects, typically following the delivery of a release to the customer. When discussing the practicality of employing retrospectives, one participant (P2) noted the difficulty of making frequent change to their software process, due to the risk that a change to the process might be disruptive, "*we've got pretty fluent software development delivery system...we're being encouraged to stick to schedule...it would be unwise to inject too many silly ideas into how to change that at this point in time. So we also encourage people to, like, to sort of like story-board their ideas and just to put them to the side.*" Instead, the participants reported collecting ideas for changes to the software process (on a Trello board, for example) that could be reviewed at less frequent meetings. This practice shows the company adapting agile practices to match the tempo of a safety-critical project, and avoiding the risks of

frequent small changes.

Several of the participants reported extensive use of quality assurance associated with agile software development, including automated static analysis, refactoring, automated unit testing, test driven development, code review and pair programming. Automated static analysis in particular was used extensively within the company. One participant (P2) confirmed that a key goal of employing static analysis was to achieve conformance with MISRA C standards "*it's for MISRA, I think level coding standards.*" In a follow up discussion, it was revealed that the company had found that the application of static analysis within a continuous integration pipeline had transferred well to an agile software development approach without the need for adaptation. In fact, the transition had led to enhanced benefit from the use of static analysis. The teams found that applying the tooling more frequently led to the production of reports with fewer but more meaningful warnings, "*As the delivery frequency increased...As the maturity of the product became higher, the easier it was to run static analysis as large swathes of code were unchanged from delivery to delivery. (P5)*"

In other cases, these practices were adapted to fit within the constraints of safety-critical system development when appropriate. In the case of pair programming, two of the participants were very emphatic that they did not practice pair programming despite all the participants reporting that informal mentoring of newer members of the company was strongly encouraged. One of the participants (P1) made the distinction between pair programming and mentoring, "*... I think that it's much better giving people a little bit of help and then dropping back and then reviewing their changes and giving them some feedback but making them do the task. Really to use the adage teach someone to fish so that the next time they can fish. There is always a ...when you do pair programming, there is always a stronger member and they will always take the keyboard... and that's not what you want.*" One participant (P2) suggested that the "*demographics*" of the company was partly a cause of this approach. Many employees have worked for the company for considerable periods of time and have become experts in particular domains of the development work. Therefore, the participants felt that these engineers would not benefit from pair programming with a younger graduate, but that the graduate would benefit from a mixture of demonstration and peer review. As one participant (P1) described it, "*I think it wastes budget. I don't think we get the value from that task.*"

A final challenge within this theme was identified during the semi-structured interviews concerning quality assurance practices within the company (Challenge 14). Safety-critical standards, such as DO-178C advocate or even require the use of independent teams to develop test procedures. However, agile methods, such as XP advocate the development of tests by the development team themselves, partly as a form of documentation of the application software [Beck and Andres, 2005]. When we investigated this conflict with the participants, a complex picture emerged, with some participants contending that this conflict was "*an ongoing problem. No, I don't think we have eliminated it. (P1)*" However, different perspectives amongst the team

and within the validation workshop ultimately led to this Challenge being rejected by the participants, because the established compromise described below was considered to be sufficient. However, we report the discussion from the interviews for completeness.

The issue emerged when one of the participant stated that the DO-178C standard they were working towards did not require complete independence, instead, the testing procedures are independently *witnessed*, "*We satisfy that by having all of our v & v witnessed or signed off by our QA people. So, all of our document reviews and things like that would have input from the QA department. All of testing is actually witnessed, you know we have someone sitting there writing things down, so that that gives us our independence. (P1)"* However, the participants also recognised that this situation is the result of a tension between the desire for independence of testing and the need to have domain expertise concerning the software under development in order to test it effectively, "*it's an interesting tension there, between needing to know exactly the details of the component you're testing. (P2)"*. What emerged from the following discussion was that the deliberate physical distance of the QA team to ensure independence had made it very difficult for them to gain a sufficient understanding of the system to develop effective tests "*There was too much inherent knowledge that the guys in these teams have about the internals of the software. (P2)"*. One possible avenue here, proposed by the participants was a compromise in which the QA team remained independent, but engaged in closer cooperative work with the development team, "*[if] we had got a v & v team in much earlier it would have worked a lot better. "*

4.5.2 Coordination amongst Stakeholders (Challenges 6, 7, 9)

The participants reported several aspects of the software team's work specific to safety-critical software development connected with coordination with external (to the software team) stakeholders that presented challenges to the use of agile software development (Challenges 6, 7 and 9). Agile principles emphasise the close involvement of an identifiable customer as critical to a project success [Chapman, 2016, Chapman et al., 2017]. Providing the team with ready access to the customer enables better communication, allowing uncertainties with regards to requirements and design to be resolved more quickly [Schwaber and Beedle, 2001]. However, the projects reported by the interview participants experience a far more complex relationship with the project customers (Challenge 7). The participants described various customer structures, for example, "*joint systems team meeting ... happens on a sort of two monthly basis .... And that involves our direct customers and members of.. their direct customers (P3)"* and "*...there's certain customers could be viewed as being the END USER they are the end users. They are type of customers. But then there are people who are little bit closer like COMPANY, then we get little bit closer again.. which are the people who are involved as product owners (P1)."* From the perspective of a software team, the immediate customer is the project's systems team who allocates the requirements. The whole project may have several different customers, each with

slightly different needs. These customers may, in turn, be procuring the product as a component to be integrated into one or more larger systems for their own customers. This network of stakeholders is characteristic of safety-critical systems projects, and so "*Agile use in these environments is restricted by the elements that define these environments*" [Hajou et al., 2014]. However, from the participant's perspective, the influence of external customers is difficult to manage, because they only have direct access to the systems team in order to demonstrate their work and receive feedback "*For me, I would say that customer demonstration ... would be the demonstration of how things work when it gets to the TEST ENVIRONMENT (P2).*"

Chapman [2016] and Chapman et al. [2017] notes that requirements engineering in agile software development is dependent on close customer involvement in the project to the extent that the customer may be viewed as an additional member of the project team. However, as Chapman et al. [2017] notes, this may not be practical in the scenario described above, where there are many different types of customers with different perspectives on and commitments to the project, such as procurers, end users, industry regulators and independent auditors. Ensuring close involvement of a larger number of customers on an on-going basis is difficult due to practical considerations such as time availability. In addition, these customers may have very different views on the requirements for the project, but there is very little guidance available on decision making, where the customer relationship is inevitably more complex [Chapman, 2016]. One possibility is the suggestion by Paige et al. [2011] to use a "*Stakeholder consortium*" to mitigate this problem. However, Chapman [2016] and Chapman et al. [2017] suggest that achieving consensus within the consortium may not be practical and that establishing "*rules of engagement*" and use of tools to automate communication and documentation can counter this problem.

The other teams within the overall project are all also effectively external stakeholders for the software team and coordination here also presents challenges. The different teams within the overall project have their own pace of completing tasks (Challenge 9). Deadlines and milestones are defined in the contracts for the whole project, but individual teams choose their own development lifecycles within this framework, creating a "*silo effect*" [VanderLeest and Buter, 2009]. Members of the software teams interviewed report being unaware of the details of activities and current status of tasks in other teams. Participants also reported that schedules across teams often do not match. For example, "*they're working on their own bunch of things at their own priorities, with their own pace dictated by the number of resources that those have, and it's often when it gets to the point where the crunch is coming that we start to understand that we've, we're misaligned in terms of priority (P2).*" The teams also have their own interpretation of when tasks are considered complete, as one participant (P2) observed, "*when I say hardware guys I mean the guys who produce the actual circuits, and then the firmware guys who bring that to life so we can use it for software development. Their definition of what finished is, so that we can put the capability of software on it, tends to be separate from what we think the done thing is.*"

In early work in the field of Global Software Engineering, Herbsleb and Mockus [2003] recognised the challenges of coordinating work across loosely coupled or distributed sub-teams. These challenges remain an active area for Software Engineering research, as illustrated by the recent study by Ebert et al. [2016]. Turk et al. [2014] suggests frequent and informal communication to overcome the lack of visibility but informal and face to face communication poses a risk of important project related information getting lost. VanderLeest and Buter [2009] also emphasize the importance of tools to improve communication and coordination among teams. In our case study, one participant described a project where all the teams were compelled to strictly follow the same schedule using a single Microsoft Project plan. According to the lead software engineer interviewed, this approach worked well. However, it is unclear whether this approach can be imposed on all projects in the company.

The lack of visibility also causes problems at integration between software and hardware (Challenge 6), a challenge that Stelzmann argues is characteristic of safety-critical system developments [Stelzmann, 2012]. Integration between hardware and software is often done towards the end of a project release, due to the components only being available at this stage. All participants agreed that this arrangement caused problems, "*typically when we get to integration. We'll find that something that the hardware is doing either isn't as we understood it to be, or it's not working (P2).*" Although the allocation of tasks and designs is well understood by the different teams at the start of the release, it was difficult for the team members to stay up to date with "*what is happening in other teams.*" One participant (P1) said "*Once we get into the integration phase, we found that the boards don't always stay up to date.*" Several interview participants suggested this was because different teams run their own development lifecycles, for example "*We've got software people working in the software plan and hardware people and firmware people working in the firmware plan. So it often becomes dissected. (P2).*" Due to the late-stage integration, it was suggested that a software team tends to focus predominantly on their own tasks, and so lose visibility of changes that are occurring elsewhere in the project. This phenomenon affected both the team that followed Waterfall and the team that had recently employed aspects of agile software development. One participant (P1) also reported that the benefits of employing a Kanban board in Jira had been lost once the project moved to an integration phase, as other tools were used for tracking progress on integration "*Once we get into the integration phase, we found that the boards don't always stay up to date... I believe, that the reason for that is we have got other methods of tracking our problems and the guys see it as duplication.*"

To partly address this challenge, one of the participants (P4) described how they had adapted their software process to incorporate a weekly *integration meeting* during the integration phase of the project, "*during our integration process, you know a lot of people had to work quite closely together so we were having weekly meetings. Once we got through that process they stopped becoming useful.*" As described above, this demonstrates how the company is employing the

principles of agile, such as frequent informal communication, but adapting the specific practices to fit with the needs of safety-critical system development. The weekly integration meeting allowed issues to be aired and resolved frequently between the different sub-teams, in a similar way to a product planning meeting within a single team.

The difficulties in employing continuous integration are also reported in the literature. Jamissen [2012] argues that DO-178C does not conflict with the concept of continuous integration in agile software development, however, Ge et al. [2010] and Kaisti et al. [2013] note that continuous integration of embedded systems is challenging. Kaisti et al. [2013] report a scarcity of evidence on the use of continuous integration in embedded systems. According to Douglass [2016], most of the literature concerning agile software development is focused on software application development, not embedded systems. This lack of guidance on Hardware and Software co-development and integration is recognised by many researchers [Chapman, 2016, Chapman et al., 2017, Kaisti et al., 2013, Douglass, 2016]. For example, in their study, Wils et al. [2006] found that the software-hardware integration phase inevitably slows down development efforts. This stage is also where the discovery of required changes can frequently arise and be the most problematic.

One proposal in the literature is to use simulators and emulators to help reduce problems at integration [Ard et al., 2014, Schooenderwoert and Morsicato, 2004, VanderLeest and Buter, 2009]. A key challenge in this approach is to ensure that emulators, simulators or test equipment have the exact specification of the target equipment [Ard et al., 2014]. While testing a system using emulators, changes made to software and hardware should also be kept in mind [Ard et al., 2014]. All the interview participants told us that the equipment for testing is not updated and often its specification does not match the target hardware. This suggests that there is a challenge in maintaining up to date test harness implementations.

4.5.3 Documentation and Communication (Challenge 8, 10)

Two related challenges were reported by participants concerning the use of agile documentation and communication practices. The Agile Manifesto [Beck et al., 2001a] advocates the delivery of "*working software over comprehensive documentation.*" Several authors have argued that this principle makes agile software development incompatible with the development of software with certification requirements [Ramesh et al., 2010, Turk et al., 2005, Martins and Gorschek, 2016, Rayside et al., 2009]. This conflict was reflected in the interviews, with one participant (P3) commenting that "*..the process documentation that we have at the moment doesn't adhere to agile sort of development process*" (Challenge 10). Critically, certification standards for safety-critical systems (DO-178C, for example) mandate the generation of documentation to demonstrate that both the delivered product and development process conform with standards and is safe to use. Certification is a very expensive and time consuming activity since it is performed on the complete system for delivery, as one participant (P1) described "*the standards*

require us sometimes on producing a hell of a lot of documentation... a lot of overhead in that respect." Certifying the system each time a change had been made would be prohibitively expensive, so the company normally only certifies the system for each "*formal delivery"* to the customer. Participants also identified the need for maintenance of documentation as a cause of delays in the project schedule, having an additional impact on Challenges 4 and 5 discussed above.

Despite this apparent conflict, there are a number of studies which demonstrate the use of agile software development in the development of formal specifications, for example, Rayside et al. [2009], Black et al. [2009]. Several of these authors emphasise on the need to adapt agile methods and practices according to the need of safety-critical system development. For example, Rayside et al. [2009] argue that traditional and agile methods are separated by limitations of current technology rather than by fundamental intellectual differences. They believe that the use of a "*mixed interpreter that executes mixed programs, comprising both declarative specification statements and regular imperative statements"* [Rayside et al., 2009] can mitigate many of the problems. Black et al. [2009] suggest that if requirements can be expressed in a formal notation they can then be machine checked for inconsistencies, effectively extending the automation of quality assurance processes to requirements documentation, in a similar manner to the Behaviour Driven Development (BDD) practice [North et al., 2006]. Bowen et al. [2023] describe two approaches for transforming informal specifications into formal notations. One of these approaches consist of derivation of first-order-logic (FOL) predicate from BDD specifications by incorporating formal notations in BDD's textual specifications. However, the authors do not evaluate the readability of such specifications in the study.

The company in the current study has also adapted its practice with respect to certification to achieve more frequent deliveries. The participants reported having employed a practice of making non-certified intermediate deliveries available to the customer, called "*engineering deliveries or releases."* One participant (P3) stated "*we have moved to the philosophy of ... there would be all engineering releases and at certain points in development we would take an engineering release and do the formalities on it."* An advantage of this approach is that the customer is able to begin integrating the product into their own system development efforts earlier. A subsidiary benefit is that the engineering releases do not require the demonstration of quality assurance processes demanded by many safety-critical standards [Chapman, 2016, Chapman et al., 2017, Cawley et al., 2010, Boehm and Turner, 2003, Vuori, 2011]. As one participant (P3) stated, "*certainly when we come to formal release if you like... that's where our testing level moves up."* Another potential option to mitigate the costs of document production is the use of automated techniques, which can reduce delay [Chapman, 2016, Chapman et al., 2017]. In addition, the approach implies that there is an *expectation* that an engineering release may eventually become a formal release, which as a consequence imposes the quality assurance standards for developing a formal release, but without the accompanying documentation to demonstrate it.

Similarly, agile software development advocates frequent face-to-face communication in small groups to ensure that critical information is circulated effectively. However, it is well understood that this approach does not necessarily scale effectively to larger multi-team projects with different lifecycles and cultures (Challenge 8). In particular, communication in agile software development is reliant on the retention of tacit knowledge, which can be difficult to recover in large-scale projects [Boehm, 2002, Glas and Ziemer, 2009, Ramesh et al., 2010]. We discussed the challenge of managing communication in large scale projects with the participants and a number of different perspectives were identified. The participants reported that a mixture of approaches to documenting information were taken, with some teams relying predominantly on an informal approach, "*I would say that large majority of them are not recorded. There is very few... where in the meeting someone minutes the meeting. (P3)*", whereas, others stated that formal documentation was used extensively for communiciation, either through email or design documents, "*know have a face to face chat and then email out the outcome of that discussion and any action points, what was agreed, and distribute that to the rest of the team (P4).*"

Several of the participants stated that an informal approach had led to mis-communications, with one participant (P2) suggesting for example, that the informal communications needed to be 'snooped' on to ensure the information wasn't lost "*we could get someone to snoop the conversations, and figure out how much we lost.*" However, another participant (P4) reported that the project teams could often rely on the tacit knowledge of individual members because of the demographics of the company. "*I don't think we really suffered as a result of that. Because we had a good group of people and a lot of very experienced people. If it was a less mature project with you know, less experienced engineers then I think it would have been a problem.*" These two different perspectives illustrate the need to not just adapt agile practices to safety-critical systems, but to adapt them to the specific context of the project.

There was some discussion about the impact that adopting agile software development had on this problem. One participant (P2) commented that "*I'm not, at the moment I should be at the ten o' clock stand-up in the roof lab. If someone doesn't come and tell me what happened or what I'm meant to do or any of the other information then that could be lost.*" However another participant (P1) described how agile software development had assisted in retaining some aspects of information that might otherwise be lost because the team became *more* disciplined about recording information in the project team's tracking tool "*Since we have employed the boards and they understand more about what's going on.*" Again, this suggests that there is potential for agile software development to be adapted to allow teams working on large scale, safety-critical systems projects to identify and maintain the documentation that is valuable to them. Spijkman et al. [2022]'s work is also focused on conversational requirements engineering techniques in agile methods to address the issue of *information being lost during informal conversations*. The focus of the study is on pre-requirements specifications to trace the source of requirements. The authors propose a prototype tool called TRACE2CONV which makes use of

NLP techniques to link the relevant part of a transcribed conversation to a user story. However, according to the authors, the tool requires further development and validation.

4.5.4 Cultural Challenges (11, 12, 13)

The final theme which emerged from the interviews was the need to change the culture within the company. Three particular challenges emerged in this context. First, the software teams in the company had no prior experience of using agile software development on a large scale and lacked guidance from elsewhere in the literature (Challenges 11 and 12). At the moment, software teams are using the Scrum method and other agile practices within individual software sub-teams, but expressed a strong desire for guidance on how to scale these for use in large multi-team context, "*if we can get...the other functions who work in those projects like firmware and hardware, if we can get them simply to follow the water-scrum-fall, that might be as good as what we can achieve (P2).*" However, there is relatively little guidance in the academic or practitioner literature on this, an issue also reported by Fitzgerald et al. [2013]and Cawley et al. [2010]. However, there are studies which report the successful use of agile software development in safety critical systems [Fitzgerald et al., 2013, VanderLeest and Buter, 2009, Gary et al., 2011, Cawley et al., 2010]. A commonly reported point in the literature is that agile methods and practices have to be adapted according to the requirements of a project [Fitzgerald et al., 2013, VanderLeest and Buter, 2009, Gary et al., 2011, Cawley et al., 2010].

The participants reported feeling confident about using Waterfall because they have plenty of practical examples from the past. The company finds it difficult to experiment with something new, given the safety-critical nature of their projects and with very little or no prior example to follow. Also, there is relatively less guidance available in the literature about the use of agile software development in safety-critical systems, particularly in the avionics industry [Ge et al., 2010, Paetsch et al., 2003, Wang and Wagner, 2016b, Carpenter and Dagnino, 2014, Heeager, 2014, Huang et al., 2012, Axelsson et al., 2016].

As a consequence, the company has a well documented and understood software development process, which is reflected in the organisational culture. The participants, therefore, identified the need to change the mindset of their colleagues (Challenge 13), as the Waterfall process has been in practice for years in the company. As one participant (P4) said "*it would be quite difficult to have an Agile process that spanned this whole organisation, without a fairly fundamental paradigm shift.*" Fitzgerald et al. [2013] also report this issue in their study. Fitzgerald et al. found that agile methods and practices are "*developer-centric*", therefore, they are typically easily accepted by the development team, whereas, management requires some convincing about the benefits of agile software development. One of the reasons behind the resistance by the management is the perception of "*short termism*" about agile software development [Fitzgerald et al., 2013]. Management usually prefers an upfront complete plan, whereas, the agile philosophy advocates short term sprints and a "*plan as you go*" approach.

4.5.5 Agile Methods Tailored to Large-Scale Safety-Critical Systems

Edison et al. [2022] compared main methods for large-scale agile development namely SAFe, LeSS, Scrum-at-Scale, DAD, and the Spotify model through a Systematic Literature Review (SLR). Although the focus of this SLR is not safety critical systems, many of the challenges associated with application of agile methods in large-scale systems found during the SLR overlap with the challenges found in this study. This shows that some of the challenges found during our study are not specific to safety-critical system development and they exist in large-scale system development in general. For example, challenge number 11 and 12 in Figure 4.5 stating "*The Software team has no practical example to follow for applying agile methods and they lack the resources to experiment*" and "*The teams need guidance on how to scale agile methods for use in large multi team context*" overlaps with a challenge in the SLR. According to one of the findings of the SLR there is a lack of guidance for adoption of agile. Similarly, challenge number 13 i.e., "*The teams need guidance on how to scale agile methods for use in large multi team context*" overlaps with a finding in the SLR. According to the authors, one of the common reasons for agile methods' failure is adopting agile methods without adopting agile thinking. According to the authors, focus is on "*doing agile*" instead of "*being agile*" is one of the causes of failure of agile methods.

However, the study by Steghöfer et al. [2019] focuses on challenges of scaled agile for safety-critical systems. The authors conducted a focus group with three experts from automotive industry to collect challenges in their daily work. According to the authors, frameworks like SAFe or LeSS do not provide explicit support for development of safety-critical systems because they lack activities for ensuring safety such as risk management, safety analysis, and certification. The authors found from the literature that there are two agile approaches that cover the entire development lifecycle for safety-critical systems i.e., *(i)* R-Scrum and *(ii)* SafeScrum. The study however, proposes no explicit solution for the challenges presented in this study except challenge number 10 i.e., "*Frequent releases increase overheads and costs, because they must be accompanied by supplemental documentation to achieve certification*". The study suggests incremental compliance i.e., incremental update of safety case (i.e., documentary showing regulatory compliance). However, incremental compliance is not discussed in detail in the study. Nevertheless, the study explicitly confirms two of our observation i.e., challenge numbers 11 and 12 stating that no practical guidance is available for the team to adopt agile in their context especially in large multi team context. The authors acknowledge that neither R-Scrum nor SafeScrum provide guidance on how work on safety should be divided between teams. According to the authors, agile lacks guidance on several aspects of the development of safety-critical systems.

4.6 Threats to Validity

This section discusses the threats to the validity of this exploratory case study. In this section, we have discussed three types of validity threats i.e., construct, reliability, and external validity.

Construct validity threat

Construct validity threats for this exploratory study on discovering the challenges related to the application of an agile method in a safety-critical system development organisation may include:
Limited representation of industry: The selected organisation may completely represent the wide range of industrial domains, or safety-critical systems where agile methods are practiced. This limited representation of industry can undermine the generalisability of the findings of the case study.
Bias towards specific methodologies or perspectives: The interviewees used scrum as a method that may introduce a bias towards certain agile methods or perspectives on safety critical systems. This bias can affect the construct validity by limiting the range of challenges considered or leading to an over-representation of specific point of views.

Reliability Validity Threat

Reliability validity threats for the results from this case study may include:

As mentioned at the beginning of this thesis, a research paper was in a reputable journal using the text from most of the sections of this chapter. The research paper had two authors which can lead to varying interpretations of the interview answers during the analysis process. This can introduce inconsistencies and potentially bias the results. Incomplete data extraction from the interviews can also impact the validity and reliability of the findings.

External Validity Threat:

One potential external validity threat is the generalisability of the findings. Since the case study only focuses on the findings from one avionics company, the findings may not apply to other avionics companies or industrial domains, such as railways or medical software. Another potential threat is the small sample size and selection of the participants that may not represent the broader population. Furthermore, personal biases such as preconceptions or personal experiences may influence the interview questions or interpretation of the data.

Addressing Threats to Validity

To address these threats, we clearly described the context and characteristics of the case study, allowing readers to determine the applicability of the findings to their own context. To enhance external validity, it was important to ensure a diverse and representative sample of participants

from various roles and levels within the organisation. We used an already established guideline called Wengraf [2001]'s guideline to develop a set of questions for the interview sessions. This set of interview questions was validated with the help of an independent researcher. Later on, the analysis was also conducted using the Wengraf's guideline. We maintained transparency throughout the research process, acknowledging any biases and taking steps to minimise their impact. The analysis process and the findings were discussed and validated with the organisation.

4.7 Summary

The purpose of this exploratory study was to investigate agile practices and their employment challenges in the context of large-scale safety-critical systems development. A series of semi-structured interviews were conducted with four employees of an avionics company which led to the identification of 13 challenges. This study presents a detailed analysis of the challenges relating to the application of agile in a large-scale safety-critical development context.

The challenges found were grouped into three categories: the influence of traditional systems engineering processes on agile software development, complex interactions with multiple external stakeholders, and the documentation required to meet the required regulatory standards. One of the key findings of this study was the difficulty in managing and maintaining systems' requirements while following agile development. We, therefore, decided to explore the use of an agile method called Behaviour Driven Development (BDD) which focuses on the requirements of a system. The next chapters in this thesis elaborate on our investigation of Behaviour Driven Development.

Chapter 5

Literature Review and Background: Behaviour Driven Development

One of the key findings from the exploratory study in the previous chapter was the difficulty in managing requirements in safety-critical systems when using agile as a development method. We decided to explore the use of Behaviour Driven Development (BDD) (i.e., an agile practice) because of its focus on requirements management. The reason for adopting a semi-systematic literature review was that it explores how research within a selected field has progressed over time [Snyder, 2019] and we wanted to explore how research on challenges of BDD has progressed over time. This chapter presents a theoretical background of BDD and an overview of the available literature on the challenges of BDD in large-scale regulated environments. This chapter also serves as a theoretical background for the subsequent chapters.

Section 5.1 provides a brief introduction to the concept of requirements engineering in regulated systems. The literature discussed in the section shows that requirements engineering of regulated systems requires performing additional activities. Section 5.2 discusses the concept of agile requirements engineering in regulated systems with a focus on safety-critical systems. Section 5.3 provides a theoretical background of BDD followed by a description of the BDD process. Section 5.4 discusses the method adopted to collect and analyse the literature. Also, the section contains various themes that emerged from dividing the studies found during the literature review. The context for this Ph.D research is described in Section 5.6. Section 5.7 discusses the threats to the validity of this literature review.

5.1 Requirements Engineering in Regulated systems

Requirements engineering is the process of discovering, developing, tracing, analysing, qualifying, communicating, and managing the requirements which define the intended working of a system [Dick et al., 2017]. A typical requirements engineering process includes activities such as: *(i)* understanding the problem and the need, *(ii)* elicitation of requirements, *(iii)* analysis

and modeling of the system through the requirements, *(iv)* documentation of requirements, *(v)* validation of the requirements, and *(vi)* management of the requirements [Macaulay, 2012].

Comparative analysis of guidelines [Dick et al., 2017, Macaulay, 2012] on requirements engineering and literature on requirements engineering in regulated systems [Martins and Gorschek, 2017, Gallina et al., 2018] shows that the requirements engineering process for the development of regulated systems involves additional activities. For example, a typical requirements elicitation activity in requirements engineering involves requirements gathering through focus groups, interviews, observations, etc. Whereas, for the development of a regulated system such as safety-critical system, the requirements elicitation process involves additional activities such as hazard analysis for elicitation of safety requirements [Vilela et al., 2017]. The process which covers these *additional activities* is called safety analysis, and its purpose is to identify potentially hazardous software faults [Medikonda and Ramaiah, 2014].

The safety analysis of software requirements detects safety issues in the requirements [Hansen et al., 1998]. During this process, incomplete and hazardous requirements are identified. The safety analysis is typically performed by safety engineers as a separate process [Leveson, 2016]. which takes requirements as an input [Vilela et al., 2017]. This means that this process is performed after obtaining a set of requirements. This could delay the detection of incomplete and hazardous requirements which could lead to repeating the requirements engineering activities and a complete safety re-analysis of software requirements i.e., rework [Leveson, 2016, Vilela et al., 2017]. According to Leveson [2016], this separation of requirements engineering and safety engineering is "*...almost guaranteed to make the effort and resources expended a poor investment"* because the concept of safety might be isolated from the developers building a system.

To enable integration between requirements engineering and safety analysis process, several researchers [Martins and Gorschek, 2016, Vilela et al., 2017, Mhenni et al., 2018, Vilela et al., 2020] have stressed on the importance of communication between team members, especially between requirements engineers and safety analysts. Martins and Gorschek [2016] conducted a Systematic Literature Review (SLR) to investigate the usefulness of the approaches proposed to elicit, model, specify, and validate safety requirements in the context of safety-critical systems development. Their findings show that the focus of a large percentage of studies was on integration between requirements engineering and safety engineering, traditional safety engineering approaches, and the need for more industrial validation of research on safety-critical systems. The authors also suggested a need for the establishment of a communication process between requirements engineers and safety analysts. The findings of the study by Vilela et al. [2017] coincide with the findings of Martins and Gorschek [2016]. Vilela et al. conducted a Systematic Literature Review (SLR) on the integration between requirements engineering and safety analysis. They selected 57 studies for their SLR. The authors discussed the benefits and challenges of the integration techniques in detail. While pointing out the lack of empirical research in the

integration between requirements engineering and safety analysis, they emphasised the need for formal guidelines for requirements engineers to derive and communicate safety requirements from the safety analysis.

The findings of several studies also showed that there is a need to establish communication mechanisms between requirements engineers and safety analysts. For example, Raatikainen et al. [2011] conducted a case study in the nuclear energy domain in Finland. Their study was focused on finding challenges in the requirements engineering process in safety-critical systems, specifically, safety-related automation systems of nuclear power plants. The challenges found during the study included: regulatory requirements, communication, aging of the system, representation, and the tool for requirements management. The authors argued that there is a need to establish practices to communicate and collaborate. In a systematic mapping study, Vilela et al. [2019] used 60 (selected) studies (out of 1164) to investigate the integration and requirements communication among different stakeholders when developing safety-critical systems. The authors analysed factors associated with safety requirements such as challenges, needs involved, application context, evaluation methods, languages and tools used to specify safety requirements. According to their analysis, model-based collaboration is the most used form of communication; whereas, face-to-face verbal communication is among the least used forms of communication.

A number of researchers proposed frameworks to integrate requirements engineering and safety analysis activities. For example, Mhenni et al. [2018] proposed a framework by extending SysML, a systems modeling language. The authors integrated safety engineering with SysML and named the framework SafeSysE. This seven step framework was demonstrated using a case study. Vilela et al. [2020] followed design science methodology to propose a safety maturity module for Unified Requirements Engineering Process Maturity Model (Uni-REPM) [Svahnberg et al., 2015], a light-weight model presenting the maturity of requirements engineering process through sets of activities divided into seven areas such as organisational support, requirements process management, elicitation, requirements analysis, release planning, documentation and requirements specification, and requirements validation.

The authors [Vilela et al., 2020] proposed addition of fourteen safety-related activities including safety knowledge management, safety communication, and human factors, spread across the seven areas of Uni-REPM. The authors conducted a static validation of the extended version of Uni-REPM (i.e., Uni-REPM SCS) with two practitioners and nine academic experts. The authors implemented a software tool to support the usage of Uni-REPM SCS. The aim of the tool was to reduce the gap between requirements engineering and safety-critical systems by focusing on safety actions that should be covered in the requirements engineering process. The authors conducted a theoretical comparison between Uni-REPM SCS, +SAFE-CMMI-DEV (an extension to CMMI for Development (CMMI-DEV) that covers safety management and safety engineering) and ISO 15504-10 (a framework for the assessment of processes including safety).

The comparison shows that Uni-REPM SCS is more descriptive and detailed because of its focus on safety in requirements engineering and its comprehensive assessment instrument.

5.2 Agile Requirements Engineering in Regulated Systems

Agile methods have been the focus of the research for many years, yet we see *challenges of requirements engineering in agile methods* among the recent topics in research [Inayat et al., 2015b, Schön et al., 2017a,b, Curcio et al., 2018, Kasauli et al., 2021].

According to participants of the study by Kasauli et al. [2021], the standard conformance could only be integrated with agile methods if the development is planned systematically. Several researchers [Vilela et al., 2017, Martins and Gorschek, 2016] have also argued the importance of the need for integration of requirements engineering and the safety analysis process in safety-critical systems development. The idea behind this is that in order to be able to use agile for the development of safety-critical systems, the activities mandated by the regulatory standards need to be made part of the life cycle. Different researchers have proposed different ideas for doing this. Antinyan and Sandgren [2021] propose automating several steps of safety analysis to reduce administrative effort. Maqsood et al. [2020] have proposed two sets of patterns for agile development of safety-critical systems: *(i)* for tracing safety requirements and *(ii)* performing automated testing.

Several researchers [Hughes et al., 2017, Dick et al., 2017, Laplante, 2017] agree that incomplete requirements are considered the major cause of project failure. Agile's advocation for the accommodation of change in requirements is built upon the narrative that requirements are volatile (i.e., they tend to change). Agile methods deal with the volatility of requirements by focusing on the high-priority requirements related to the immediate development life cycle i.e., the iteration. Agile teams focus on delivering the system in small iterations by implementing small sets of *known* requirements.

The accuracy of these requirements is improved through the emphasis on communication through practices like customer involvement and face-to-face communication. In a Systematic Literature Review (SLR) on agile requirements engineering, Schön et al. [2017b] argue that agile relies on continuous communication and collaboration to involve the stakeholders in the requirements engineering process. According to Martins and Gorschek [2016], the most important factor for successful agile requirements engineering is the intensive communication among the stakeholders.

Although the literature [Inayat et al., 2015b, Schön et al., 2017a,b] implies that the communication challenges are mitigated by agile, yet the existing challenges reported in the literature on agile requirements engineering in safety-critical systems relate to communication and knowledge management [Kasauli et al., 2018b, Martins and Gorschek, 2017, 2016]. The *challenges in communication* is a common theme that appears both in traditional and agile requirements engi-

neering in safety-critical systems. Kasauli et al. [2018b] reported the requirements engineering related challenges of large-scale agile safety-critical system development based upon 20 qualitative interviews, 5 focus groups, and 2 cross-company workshops. Their findings are summarised in four sets of conclusions, all of which relate to the communication and information between the stakeholders. In a similar study Kasauli et al. [2021] the authors state that "*...our results suggest, it is crucial to establish suitable exchange and management of knowledge throughout large-scale agile system development*".

Various studies [Badampudi et al., 2013, Heikkilä et al., 2017, Martins and Gorschek, 2016, 2017, Kasauli et al., 2021] investigated the issue of communication in large scale agile projects. Many of them suggested improvements in various areas of requirements engineering; whereas, some suggested frameworks. For example, Fægri and Moe [2015] suggested a conversation model of software development. The work by Spijkman et al. [2022] is also aimed at conversational requirements engineering. The authors introduced a tool that establishes backward traceability from requirements to one or more relevant transcript segments in a requirements conversation. The tool matches the speakers' turn to the requirements using tokenization and lemmatization techniques. Although the authors recognised the need for further development in the tool for use in practice, their preliminary results showed the feasibility of the overall approach. Medeiros et al. [2020] proposed an approach called Requirements Specification for Developers (RSD) to create an SRS that provides information closer to development needs. The RSD approach adopted conceptual modeling, mockup modeling, and specification of acceptance criteria to create a requirements specification for the developers.

Among other methods and frameworks, Behaviour Driven Development (BDD) is an agile method that claims to improve communication and knowledge sharing among stakeholders through requirements. We have discussed and analysed the literature on BDD in this chapter. The reason for our interest in BDD was due to the popularity of BDD. It is a well established process as compared to the frameworks proposed by the studies discussed before. The annual survey for agile has been including BDD in the list of most used agile processes for many years [CollabNet VersionOne, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020].

5.3 Background on Behaviour Driven Development

BDD was originally conceived for the purpose of acceptance testing. While discussing agile user stories, North et al. [2006] says "*A story's behaviour is simply its acceptance criteria: if the system fulfills all the acceptance criteria, it's behaving correctly; if it doesn't, it isn't.*" According to Smart [2014], Behaviour Driven Development (BDD) is an agile practice for writing and automating the execution of acceptance tests for requirements with a focus on the behaviour of a system. The goal of BDD is to create a shared understanding of a system through requirements. It is a process of elaboration and automated testing of the behaviour of a system [Pereira

et al., 2018].

In spite of its initial conception, various researchers recognise BDD as an effective method for improving communication and requirements engineering as well rather than a mere way of testing [Pereira et al., 2018, Oliveira and Marczak, 2018, Wang and Wagner, 2018, Scandaroli et al., 2019, Moe, 2019, Nascimento et al., 2020a, Smart and Molak, 2023]. According to Smart and Molak [2023], BDD works well for requirements analysis. Aslak Hellesøy, the founder and creator of Cucumber (a tool for automating BDD test suites), in one of his blogs [Hellesøy, 2020] says that BDD is not a testing technique. According to him, BDD is not test automation; it is collaborative requirements analysis combined with formulation and automation of behaviour testing.

Scandaroli et al. [2019] reported the lessons learned from applying BDD using two different case studies. According to the authors, the benefits of BDD include effective communication between team members through a shared understanding of requirements. Pereira et al. [2018] also reported improved collaboration and communication through requirements as benefits of BDD during their industrial case study on finding benefits and challenges of BDD. The majority of BDD activities theoretically described by Smart [2014] also consist of requirements elicitation and decomposition. This set of activities also includes writing and testing of the acceptance criteria in addition to the requirements engineering activities. This shows that BDD is much more than purely acceptance testing and that it supports communication and requirements engineering activities.

The requirements in BDD are organised into agile user stories. The user stories in BDD are called features where each feature is written and stored in a separate text file with a ".feature" extension. The features in BDD are written in Gherkin: a non-technical and human readable language for documenting writing requirements specifications in BDD. Gherkin uses natural language sentences to describe the desired behaviour of a system.

An agile user story in a Gherkin feature file template is described in Figure 5.1. The structure of a typical feature in Gherkin comprises four parts i.e.,

i. **Title:** describes the name of the feature;

ii. **Rationale** (i.e., the phrase starting with "*In order to...*"): describes the benefit of performing the function;

iii. **Actor** (i.e., the phrase starting with "*As a...*"): provides the information about the stakeholder needing the ability to perform the function;

iv. **Goal** (i.e., the phrase starting with "*I want ...*"): describes the ability to perform a function.

Altogether, these four parts of a feature help in defining the scope of functionality and facilitate a common understanding between technical and non-technical stakeholders.

```
Feature: <title>
In order to <achieve a business goal or deliver business value>
As a <stakeholder>
I want <something>
```

Figure 5.1: Feature template

A typical feature file contains a feature and several formalised examples of how a feature works [Smart, 2014]. These formalised examples are called Scenarios and each scenario represents a unique variation of the sequence of behaviour during the performance of a feature. There are two main artefacts in BDD i.e., (i) requirements i.e., features (including the associated scenarios); and (ii) the associated code steps.

However, according to North [2019], a requirement in BDD consists of a feature and its associated scenario, whereas the associated code steps for automating the testing of acceptance criteria in BDD is complementary to BDD and is inherited through Test Driven Development [North, 2021]. Improvement in the understanding of the requirements is one of the main benefits of BDD identified during an industrial study by Irshad et al. [2021]. This shows the primary goal of BDD is to get the requirements right through communication and collaboration. It could also mean that automation of acceptance testing in BDD is a part of the requirements engineering process. There are several studies [dos Santos and Vilain, 2018, Maciel et al., 2019, Bjarnason et al., 2016, Bjarnason and Borg, 2017] which seem to support this concept.

dos Santos and Vilain [2018] experimented with 18 students in the last year of their Computer Science bachelor's degree at the University of Santa Catarina, Brazil. The experiment compares the applicability of two acceptance testing techniques that complement requirements engineering (Fit tables and Gherkin language). The authors did not find sufficient evidence to show that one of these techniques is easier to use of better to communicate software requirements. However, the evidence showed that the mean time to specify test scenarios using the Gherkin language is lower than fit tables. The study by Maciel et al. [2019] is also based upon combining testing with requirements to add clarity to the requirements. The authors propose a model driven driven approach to promote test specification at a very early stage and combine them with requirements specifications. To demonstrate the applicability of the approach, the authors use ITLingo RSL to support requirements and test specifications, and the Robot language to specify test scripts. The approach uses model-to-model transformation, such as test cases into test scripts which then are executed by the Robot test automation framework. However, the approach was not evaluated by the study.

Bjarnason et al. [2016] performed an iterative case study at three companies and collected data through 14 interviews and focus groups to investigate the idea of using test cases as requirements to understand how test cases can support requirements activities. The results suggest that frequent communication enforced by using tests as requirements supports elicitation, validation, and management of customer requirements. The authors discovered five variants of tests

```
Given <a context>
When <an action>
Then <you expect some outcome>
```

Figure 5.2: Scenario template

```
...
Scenario: Basic DuckDuckGo Search
    Given the DuckDuckGo home page is displayed
    When the user searches for "panda"
    Then results are shown for "panda"
...
```

Figure 5.3: Example of a scenario

as requirements specification i.e., de facto, behaviour-driven, story-test driven, stand-alone strict and stand-alone manual. The study by Bjarnason and Borg [2017] is based upon theoretical argumentation. The authors highlight three practices and argue that they can provide effective alignment of requirements engineering and testing i.e., using test cases as requirements, harvesting trace links, and reducing distances between requirements engineers and testers.

Figure 5.2 describes the format of a scenario in Gherkin. Each natural language sentence in a scenario is called a step of which there are three types i.e., *(i)* Given: represents the pre-condition, *(ii)* When: represents the action on the target system, *(*iii) Then: represents the post-conditions or the desired state of the system as a result of the action. Altogether, these three parts describe one or more outcomes as a result of one or more specific actions in a specific context. Every step is linked to the system under test through the implementation of step definition functions, sometimes informally referred to as *glue code*, such that every step has a piece of code associated to it. This is why requirements specification in BDD is called executable specification.

Figure 5.3 is an example of a scenario (re-produced from a tutorial* on BDD). The scenario describes searching a term on the homepage of a website called DuckDuckGo. The *Given* step in the scenario describes a pre-condition i.e., the homepage of the website is already displayed. Figure 5.4 describes the associated code step for the *Given* step in the scenario. The first line of a step function annotates the function with the text in the respective Gherkin step from Figure 5.3.

*https://automationpanda.com/2018/10/22/python-testing-101-pytest-bdd/

```
...
@given('the DuckDuckGo home page is displayed')
def ddg_home(browser):
    browser.get(DUCKDUCKGO_HOME)
...
```

Figure 5.4: Example of a code step

The function fulfils the behaviour specified in the text i.e., in this case, displays the homepage for DuckDuckGo.

A wide range of tools is available to support BDD for different programming languages. For example, Cucumber works with some major programming languages including, Java, Ruby, and many popular web application programming languages. SpecFlow is a BDD tool for .NET platform, Behave works with Python, JBehave is a Java framework whereas Behat is for PHP. Most BDD frameworks have mechanisms for generating empty code steps from scenario steps; whereas, some BDD tools embed the scenarios directly in test code (i.e., no separate feature file). The exact nature of the code depends on the implementation framework. For example in Behave (python), code steps are functions that are annotated with the string representing the corresponding scenario step. Similarly, in JBehave (Java), the code steps are methods collated in a Steps class, again annotated with the corresponding scenario step. In this section, we have explained how BDD is practiced using Cucumber.

BDD Process

To the best of our knowledge, the books by Smart [2014] and Wynne et al. [2017] are the most cited books on Behaviour Driven Development (till December-2021). They proposed a number of steps for the production of BDD features and scenarios. We extracted the following notable BDD activities from the above cited books to provide a theoretical overview of the BDD process. Please note that this process is a collection of theoretical recommendations by Smart [2014] and Wynne et al. [2017].

Determination of business goal: Since the aim of Behaviour Driven Development (BDD) is to provide value, it is very important to understand the business goal of a project. For example, the business goal of a project which aims to provide an online service to the customers could be to *attract more customers* by providing easy access to a service.

Define and document set of major features and determine the relative value of proposed features: The high-level functionality of the system is discussed and transcribed in the form of features (user stories) described in Figure 5.1. A set of useable features and the value associated with them is discovered and discussed during this activity. The high level functionality elicited during this step helps in reaching the business goal. Considering the above example, several useable features like "*create account*" or "*perform an online transaction*" could help the business owners reach the business goal i.e., *easy access to a service*.

Illustrate the stories with examples: Discuss the concrete and real-life examples with the customer (Or the product owner) such that every scenario is an end-to-end representation of the corresponding feature. These examples serve as the acceptance criteria at the later stages.

Describe examples as BDD scenarios: Examples are transcribed using the BDD scenario template in Figure 5.2. Each example describes a unique sequence of events corresponding to the respective feature. This will serve as the basis for the acceptance criteria. Acceptance testing

is used to demonstrate the high-level, end-to-end behaviour of an application.

Write the code: A scenario in BDD contains Given-When-Then steps that altogether express the behaviour of the system described in a scenario. During this step, a piece of code that satisfies the description of each step in a scenario is written. The complete code for a scenario is organised in a unit test which implements the complete behaviour described by the scenario. In other words, the unit test will build up the components which demonstrate the behaviour described in the scenarios [Smart, 2014].

Since activities in Behaviour Driven Development process emphasise communication and collaboration between stakeholders, it is pertinent to know who should take part in these activities. One of the recommendations on who should take part in these activities is the "*three amigos meeting*" where the developer, tester, and business analyst or the product owner get together to discuss the features and associated examples. Three amigos meeting is a recommended practice for all activities in the BDD process [Smart, 2014, Wynne et al., 2017]. Participation of different roles in the activities helps in presenting different views of a functionality.

5.4 Literature Review Method

Behaviour Driven Development (BDD) has gained increasing popularity in recent years as an agile method. Annual surveys on the state of agile list BDD as one of the most employed techniques in the industry. The annual surveys on the state of agile published during the last eight years [CollabNet VersionOne, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020] show an annual increase in the use of BDD by agile practitioners i.e., from 10% in 2013 to 19% in 2020. In a study, Binamungu et al. [2018b] surveyed 75 BDD practitioners from different parts of the world, out of which, 20% made BDD a mandatory tool; whereas, 61% used it as an optional tool. The results of the study also suggested that BDD is in active use in the industry. Rahman and Gao [2015] presented a reusable automated acceptance testing architecture to address concerns associated with reusability, audibility, and maintainability. Rahman and Gao [2015] pointed out that there has been a recent increase in the number of people embracing the methodology. Zampetti et al. [2020] analysed 50,000 popular open-source projects written in five programming languages on GitHub. Their results show that BDD tools are used in 27.25% of 50,000 open-source projects. According to the survey by Rahman and Gao [2015], the main reason for this growing interest in BDD is the ease of understanding of requirements.

Although there has been a growing interest in Behaviour Driven Development in recent years [Storer and Bob, 2019, Silva et al., 2020a], several researchers [Egbreghts, 2017, Binamungu et al., 2018b, Zampetti et al., 2020, Pereira et al., 2018] have pointed at the scarcity of empirical research on application and challenges of BDD in real projects in large-scale environments. These studies were based upon different research methods such as interviews [Pereira et al., 2018], experimentation [Diepenbeck et al., 2018, Storer and Bob, 2019], case study [de Souza

Database	URL
ACM	http://dl.acm.org/
IEEExplore	http://ieeexplore.ieee.org/Xplore/home.jsp
Scopus	https://www.scopus.com
Science Direct	http://www.sciencedirect.com/
Web of Science	http://apps.webofknowledge.com
Google Scholar	https://scholar.google.com/

Figure 5.5: Literature Sources for the Second Literature Review

(((Behaviour Driven Development) AND regulated) AND software)
(Behaviour Driven Development and safety standards)
(Behaviour Driven Development* challenges* large-scale safety-critical)
(Behaviour Driven Development challenges in large-scale safety-critical)
(Behaviour Driven Development challenges)

Figure 5.6: Search Strings

et al., 2017], toy examples [Gómez, 2018, Zaeske et al., 2021], online surveys [Zampetti et al., 2020, Binamungu et al., 2018b, 2020], literature review [Solis and Wang, 2011, Egbreghts, 2017], and expert panel research [Nascimento et al., 2020b]. We divided the studies into various themes to present an overview of the topics on BDD in the literature.

The relevant literature was reviewed from a variety of online sources including relevant books, white papers from the industry, journals, conference proceedings, and experience reports. An exhaustive search for the literature on *BDD* and *application of BDD in large-scale environments* was performed. The repositories listed in Figure 5.5 were used for searching the studies using the combinations of search strings in Figure 5.6. The keywords were identified from the *purpose* and synonyms of the important search terms. The databases were searched iteratively by using the search strings that were formed by using AND and OR Boolean operators. The purpose of the second literature review was to find the studies on *BDD and its application in large-scale environments*.

5.4.1 Use of Natural Language

Researchers [Storer and Bob, 2019, Carrera et al., 2014, Keogh, 2010, Smart, 2014, Pereira et al., 2018] have acknowledged that the use of BDD encourages communication and forces elaboration of poorly understood requirements. The use of BDD bridges the gap between stakeholders and written specifications in a way that not only improves understanding and communication but draws both technical and non-technical stakeholders into the process [Wynne et al., 2017]. Ease of understanding in BDD is due to the use of Gherkin language which uses a structure similar to the natural language [Binamungu et al., 2018b, Borgenstierna, 2018, Pyshkin et al., 2012, Solis and Wang, 2011, Silva et al., 2020a, Diepenbeck et al., 2018, Sarinho, 2019].

Borgenstierna [2018] performed a comparison of two frameworks i.e., Behave (for BDD) and PyUnit (for TDD). The comparison was performed from a tester's perspective using Yubikey (i.e., USB NFC authenticator) as a case. The results show that the Gherkin language used in Behave is easier to read. Nascimento et al. [2020b] investigated the potential benefits and challenges of teaching BDD to software engineering students. They conducted an expert panel research with 28 active learning experts from four countries. Results of the study suggested that BDD has a greater positive impact in the requirements phase than other stages of software development. Solis and Wang [2011] analysed the relevant literature and the available tools. They discussed six main characteristics of BDD which, according to the authors, cover a range of software development activities including requirements elicitation, analysis, design, and implementation. According to the authors, BDD workflow initiates discussion and provides a good starting point to communicate with the customer. Ease of communication because of the use of natural language and automation of requirements specifications are considered strengths of BDD.

5.4.2 Embrace BDD as a Holistic Approach

According to Solis and Wang [2011], BDD is a combination of characteristics that include the use of natural language, TDD, and automated acceptance testing. The authors emphasised that these characteristics are interlinked and should be embraced in a holistic way to get the full benefit of the BDD approach.

On the contrary, Egbreghts [2017], in a literature review of BDD, highlighted the characteristics of BDD from the literature. According to the findings of the study, BDD is not fully embraced in the projects because it is not a well-defined agile method. According to the author, different concepts of BDD are not interlinked as they are in traditional agile development.

5.4.3 Role of Experience in Using BDD

Pereira et al. [2018] interviewed 24 participants from different companies and varying levels of expertise in agile and BDD. The focus of their research was on discovering the benefits and the challenges of BDD. According to Pereira et al. [2018], scenarios are the strength of BDD and investing in training and tools pays off. Their results suggested that lack of experience in BDD can lead to production of poorly written scenarios. BDD starts to show benefits once a company overcomes the initial learning curve.

Gómez [2018] conducted an experiment using a toy example with twenty (20) students to compare three agile software development methods i.e., Incremental Test-Last (ITL), Test Driven Development (TDD), and Behaviour Driven Development (BDD). The study did not present any conclusive results and mainly discussed the data gathering and potential ways of evaluating the data. The study showed a decrease in productivity because of a lack of expe-

rience in BDD. Zampetti et al. [2020] also argued that the use of BDD in a project requires experience and training.

5.4.4 Maintenance of BDD Specifications

Binamungu et al. [2018b] conducted a study by using an online survey for gathering data. They investigated the extent of the use of BDD in the industry, perceived challenges and benefits, and the discovery and management of duplicates in BDD specifications. An online survey was filled out by seventy-five (75) practitioners from all over the world. As a result of the study, the authors learned that BDD tests suffer from the same maintenance challenges that test suites in automated testing face. Maintenance and duplication in BDD specification are the major challenges according to the respondents of the survey in the study. According to Nascimento et al. [2020b], the use of BDD requires performing additional activities that can increase delivery time.

The basis of the study by Storer and Bob [2019] was the difficulty in maintaining BDD tests. In case of an update, the changes need to be made at two places i.e., the scenario and the corresponding code steps. To address this issue, the authors proposed automatic generation of code steps. According to the authors, excluding the underlying implementation logic makes the scenarios implicit. On the other hand, if scenarios are made explicit by placing some implementation details in the scenarios, the scenarios become long and harder to understand. Hence, there is a need for a guideline to write tests in Gherkin.

5.4.5 Tool Support

The goal of the study by Zampetti et al. [2020] was to investigate the adoption of BDD in 50,000 open-source projects developed in five programming languages. The authors also surveyed 31 practitioners with high level of testing experience. Findings of the study indicated that BDD tools are used in 27.25% of 50,000 open-source projects. The authors identified six different tools that were used in the majority of the 27.25% BDD projects. The percentage of Ruby projects was highest among the BDD projects. The results of the study showed that the adoption of BDD tools in the projects is still low, and in many cases, they are used for activities such as unit testing. The authors also observed that the majority of the developers write the code before writing the tests. According to the authors, the developers are still skeptical about the use of BDD.

Article by de Souza et al. [2017] reported the findings of a case study in which BDD was combined with Scrum for the development of an educational learning and management system at a Brazilian university. They reported a considerable improvement in communication between team members and product owner because of the use of BDD. The study showed that existing tools mainly focus on the "... *implementation phase, providing limited support to the require-*

ments gathering, analysis, and design phases of software life cycle". According to Solis and Wang [2011], the existing tool kits for BDD lack support for the planning and analysis phases of the software development.

5.4.6 Quality of BDD Specifications

Nascimento et al. [2020b] suggested that more investigation is needed on the quality of scenarios and the role of experience in writing them. Binamungu et al. [2020] presented the results of a survey on views of the BDD practitioners on BDD suite quality. They proposed four principles of BDD suite quality and asked respondents to respond according to their level of agreement with them. The four principles of BDD suite quality were: *(i)* Principle of Conservation of Steps, *(ii)* Principle of Conservation of Domain Vocabulary, *(iii)* Principle of Elimination of Technical Vocabulary, *(iv)* Principle of Conservation of Proper Abstraction. At least 75% of respondents voted in support of each of the four principles. The Importance of writing reusable scenarios and the readability and clarity of the resulting specification was highlighted by the respondents.

5.4.7 BDD for Hardware

A study by Diepenbeck et al. [2018] was on adaptation of BDD to the needs of the verification-centric hardware design flow. They extended BDD methodology by combining formal properties with test-driven development design. The authors implemented a test bench in Verilog to test their approach. While pointing towards the implicit nature of the scenarios, the authors argued that the lack of implementation semantics and implicit environment assumptions, such as restrictions on the data range of certain inputs, shows that Gherkin does not share the underlying complexity of the code.

5.4.8 BDD for Regulated Systems

A study by Zaeske et al. [2021] discussed the application of BDD to an example of an avionics application that comes under DO-178C regulated standard for airborne software. The example system was a Class C Terrain Awareness and Warning System, developed using Rust language with Rust BDD infrastructure. The study showed that the high-level requirements of a system regulated by DO-178C can be captured and formalised using Gherkin. However, the authors do not report any challenges about the application of BDD to an avionics system regulated by DO-178C.

5.5 Discussion

Despite the increasing popularity of BDD, there is a lack of empirical studies on the discovery and resolution of the challenges associated with BDD. This scarcity of literature is also mentioned in various studies [Binamungu et al., 2018b, Borgenstierna, 2018, Binamungu et al., 2018a, Solis and Wang, 2011, Pereira et al., 2018, Zampetti et al., 2020]. The *lack of empirical evidence* also means that there is less practical guidance available on BDD, which as a consequence, can lower the confidence of the people attempting to use BDD. In the previous study (in Chapter 4), we have seen that the lack of practical guidance on agile methods is one of the biggest hurdles in its adoption. According to Julian et al. [2019], there has been relatively little investigation on how agile is adopted and used in industry. While pointing at the lack of availability of empirical studies, the authors [Julian et al., 2019] argue that there is very little literature available on "*how agile is used in practice?*" as compared to the existing literature discussing "*what is agile?*".

The need for practical guidance comes from the gap between theory and real life. The theoretical models and processes are usually based upon certain assumptions and tend to ignore real-life constraints such as size and structure of the organisation, unavailability of the customer, the experience of the team members, etc. The real-world factors can limit the application of the theoretical models. The theoretical description of BDD also presents an ideal situation where everyone is thought to be involved in the development. We do not know anything about the practicality of the activities of BDD. There is a need for guidance on what activities to follow, their outcomes, and how to handle real-life situations that can impede the speed of development e.g., unavailability of the customer, benefits of adopting certain activities, or implications of ignoring them.

Investigating the use of BDD in the real-world could bring clarity and confidence in the use of BDD. It is important to know the impediments in the application of BDD so that the industry can take guidance from such knowledge. Lack of practical guidance on a method often hinders the adoption of the method e.g., agile [Islam and Storer, 2020b]. There is a need to investigate the constraints and the real-life challenges in the application of BDD. Learning from the experience of the practitioners keeping in mind the circumstances they applied BDD in, can serve as a guide for the people attempting to use BDD.

5.6 Research Context

This research is focused on the challenges of applying Behaviour Driven Development. Effective management of project requirements in large-scale organisations appears to be a challenge because of the factors such as ineffective communication due to the size of the team and project [Fucci et al., 2018, Konrad and Gall, 2008]. Since agile methods are popular for close collab-

oration, communication, accommodation of change, and shared vision of the project, several researchers [Paasivaara et al., 2018, Abrar et al., 2019, Venkatesh and Rakhra, 2020, Kalenda et al., 2018] have proposed use of agile methods in large-scale environments. Unfortunately, the use of agile methods in large-scale environments does not solve the issues related to communication, shared vision, and requirements management since the same issues appear as challenges of agile requirements engineering in large-scale environments in the literature [Inayat et al., 2015a, Vilela et al., 2017, Uludag et al., 2018, Dikert et al., 2016]. In an attempt to solve the issues related to communication, shared vision, and requirements management, we decided to explore the use of Behaviour Driven Development (BDD).

Despite the increasing popularity of BDD, several researchers [Binamungu et al., 2018b, Nascimento et al., 2020b, Irshad et al., 2021] have pointed at the lack of knowledge and evidence on the challenges of BDD. As a part of this research, we have explored the challenges related to the use of BDD in an industrial environment which helped us in discovering the technical limitations of Gherkin along with exploring the challenges of using BDD in a large organisation.

Since Gherkin provides a lot of freedom for writing requirements specifications, it is easy to write an over complicated set of requirements which could create maintenance issues in BDD requirements specification. In this research, we also identified several requirements specification writing styles and practices that could lead to inflexible requirements specification and, consequently, bad smells. The open-source BDD projects on GitHub were examined for this purpose.

Highlighting the limitations of Gherkin, exploring the challenges of BDD, and identifying the existing bad smells in requirements specifications of open-source BDD projects will serve as a guide for people and organisations intending to adopt Behaviour Driven Development (BDD).

5.7 Threats to Validity

This section discusses the threats to the validity of this literature review. In this section, we have discussed three types of validity threats i.e., construct, reliability, and external validity.

Construct validity threat

A construct validity threat for this semi-systematic literature review on Behaviour Driven Development could be the lack of clarity in defining BDD and its associated concepts. Different studies can use varying definitions for concepts such as "*adoption of BDD*", "*effectiveness of BDD*", or "*challenges in implementing BDD*", which can make the comparison of the results difficult.

Reliability Validity Threat

A reliability validity threat for this semi-systematic literature review on Behaviour Driven Development could be the lack of replicability of the review. The reliability of a literature review may be compromised if the literature review is not reproducible. Also, a reviewer can have his/her own interpretation of the data. This can result in inconsistent findings and conclusions drawn from the reviewed studies.

External Validity Threat

An external validity threat for this semi-systematic literature review on Behaviour Driven Development could be the limited generalisability of findings. The reviewed studies may not represent the entire population of the studies on BDD.

Addressing Threats to Validity

To address the validity threats, the BDD concept and the activities of a BDD process were described upfront. We presented a conceptual framework that outlines the specific aspects and dimensions of Behaviour Driven Development. The studies on the challenges of BDD already appeared to be scarce so we did not apply inclusion/exclusion criteria for selecting the relevant studies, ensuring that no studies were excluded. Regular meetings and discussions with other academics and a series of email exchanges with Dan North (creator of BDD) addressed the uncertainties during the study selection.

Chapter 6

BDD in Practice: A Case Study

Chapter 4 reported on the experience of adopting an agile method within a company engaged in large-scale, complex software and hardware systems development. The company's established development process was based on sequential models. The reasons for following sequential models included stringent documentary requirements, project size, team size and setting, software and hardware co-development, and interdependence of hardware and software components of a project. The change in business needs (i.e., the desire to deliver earlier and continuously) pushed the company to adopt agile development. The company undertook the transition gradually, with attempts to introduce agile development in some of the new and ongoing projects and sub-projects.

The focus of our previous study (as explained in Chapter 4) with the company was to understand the impediments to the adoption of an agile development process. We learned that adoption of agile in large-scale safety-critical system development is constrained by its context i.e., highly regulated safety-critical environment. Using an agile process in a large-scale safety-critical environment often requires adaptation of various methods and practices. In particular, there is a need to understand how agile software development can be adapted in the context of large-scale, complex systems engineering which includes the development of both software and hardware components on projects that may last many decades.

We also learned that requirements management in large-scale systems is a challenge. We learned that factors such as pace of development, nature, and size of a project, number of teams involved, and organisational and regulatory requirements could impact an organisation's ability to manage requirements. We, therefore, decided to explore an agile process, Behaviour Driven Development (BDD), which is primarily hinged upon requirements engineering. The decision to explore the use of BDD was based upon its apparent popularity and emphasis on communication and collaboration through requirements specification.

We conducted a literature review in Chapter 5 on BDD, its challenges and its application in large-scale environments. We detected a scarcity of literature on BDD, especially, application of BDD in large-scale environments. This motivated us to conduct action research to explore

the use of BDD for development of a project in an industrial setting.

This chapter discusses the challenges faced during the application of BDD within the company and a number of limitations of Gherkin (i.e., language for writing requirements specifications in BDD). This study also presents a comparison between BDD in practice and the theoretical process of BDD as proposed by Smart [2014] and Wynne et al. [2017]. This chapter is organised as follows: Section 6.1 elaborates on the objectives of the study. Section 6.2 describes the context of the project studied in this chapter. Section 6.3 explains the use of action research in the early phases of this study. Action research helped in aligning the BDD process with the development process of the project discussed in this study. Section 6.4 describes the use of semi-structured interviews in this study. The semi-structured interviews were used at the end of this study to report the experiences and opinions of the people who applied BDD during the project discussed in this study. Section 6.5 discusses the observations regarding the limitation of Gherkin (i.e., the language for writing BDD specifications). These observations were made during the action research and the interviews. Section 6.6 presents a comparison of BDD in practice with the BDD process. Section 6.7 discusses the threats to the validity of this study and the summary of the chapter is presented in Section 6.8.

6.1 Objectives of the Exploratory Case Study

As discussed earlier in Chapter 1, the annual industrial survey reports from the last eight years on the state of agile show a gradual increase in the popularity of BDD [CollabNet VersionOne, 2013, 2014, 2015, 2016, 2017, 2018, 2019, 2020]. Some recent studies [Degutis, 2018, Wang and Wagner, 2018, Zaeske et al., 2021] show that BDD is also being considered for the development of complex systems. Yet there has been very little investigation reported in the literature on the experience of practitioners who have applied BDD (as explained in Section 5.4) which implies that there is a scarcity of knowledge on the use of BDD in academic literature. There are, therefore, many open questions about the application, challenges, and adaptations to BDD in various software domains.

To begin to address this gap, we conducted a study with software engineers working for a large avionics company in the United Kingdom (referred to as 'the company'). As described in Chapter 4, the company, as a whole, is engaged in a variety of projects for external customers, typically comprising both hardware and software development for safety-critical systems. The purpose of this study was to learn about the company's experience in the application of Behaviour Driven Development (BDD) to software development for a project and to gain a deeper insight into the difficulties experienced. Therefore, the research objectives within the context of the case study in this chapter were:

- Explore the feasibility of applying Behaviour Driven Development to the development and maintenance of a project.

- Identify the benefits and difficulties experienced when developing a system using Behaviour Driven Development.
- Analyse BDD in practice versus its portrayal in theory (as described in section 5.3).

A combination of research methods (i.e., action research and case study interviews) was used to reach the objectives of this study. The study was conducted in two phases. In the early phase, we wanted to enable the project development team to use BDD for the development of a project. Action research was used for this purpose. The use of action research in the early phase of this study helped us align the development process of the project with BDD and enable the project development team to adopt BDD for the development of the project.

In the latter phase, we wanted to learn and report the experiences and opinions of the people who used BDD for the development of the project discussed in this study. A series of post hoc semi-structured interviews was conducted for this purpose. The use of semi-structured interviews helped us understand and report the opinions and experiences of the development team using BDD.

6.2 Context of the study

BDD was used for the development of a sub-part of an ongoing project. This sub-part was a tool internally used and developed at the company for the automation and management of large data sets. As described in Chapter 4, we are unable to discuss specific details (including the requirements specification) of the sub-project (i.e., the tool) because of the confidentiality agreement between ourselves and the company.

The description below provides context for the discussion of the challenges identified during action research and interviews conducted during this study. Each theme discussed below gives an overview of the context in which BDD was applied and adopted. This section describes the *overall* project structure where BDD was adopted as a software development process. The section also discusses the project team structure, and the software process and technology used to develop the project.

6.2.1 Overview of the Project and the Project Team Structure

The project discussed in this study was initiated by the product owner since he worked in an area where he had felt the need for the tool. As he was familiar with the use and need of the system, he was one of the key people in specifying the initial concept of the project. The project discussed in this study was part of a larger project which had a completion time of three years and comprised a six-member Scrum Team. Three out of those six people, including the product owner and scrum master, in addition to working on the larger project, worked on the project discussed in this study.

The project (discussed in this study) was a tool developed for internal use for a specific group of engineers in the company with an estimated completion time of 12 months. It was purely a software project, and also, one of the projects that the company wanted to develop using agile. For the development of this project, the company adopted the basic agile team settings recommended in the literature [Mergel et al., 2020, Dybå and Dingsøyr, 2008, Stadler et al., 2019, Conboy and Carroll, 2019]. The adopted agile team settings included having a small team and the creation of a co-located environment. Therefore, the team was limited to three people including the developer, the product owner, and the scrum master. All team members were co-located for frequent and informal communication.

The team members had a varying level of professional experience. The product owner, with over seven years of experience, served in different roles within the company e.g., product owner, and system integrator of hardware, firmware and software. He used agile methods for four or five years before working on the said project. Whereas, the software developer was an industrial placement student and had no prior experience of using agile methods in a professional environment. The scrum master, however, had four years of working experience, with a couple of years of experience in test-driven development.

Before starting the project, a prototype was developed to get feedback on the usability of the tool. The initial users of the tool were seven employees of the company who assumed the role of the customers for this project. They included a mix of software engineers and experts from other fields who did not have a background in software development. The experts were able to give feedback on the usability; whereas, the software engineers gave feedback on the requirements and functionality.

Feedback from the users helped in the preparation of an initial set of requirements in the form of fifty use cases. Each use case was documented using the company's prescribed requirements template. The template was in a tabular form consisting of several fields describing the use case and can be seen in Figure 6.1. These fields could be divided into (i) use case information i.e., title, description, owner and primary actor, and status of the use case; (ii) precondition and post-condition; (iii) sequence and description of steps; (iv) sequence and description of steps for alternate path; (v) assumptions and constraints; (vi) demonstration and validation criteria.

6.2.2 Software Process Overview

This section presents an overview of the software process used in the case study, which will be described in more detail in the phases of the action research presented below. The life cycle model followed for the development of the overall project was Scrum. The sub-part focused in this study was managed within the wider project's Scrum process; therefore, there were no separate Scrum activities for the sub-part. However, Behaviour Driven Development was only applied in the sub-part (i.e., the project discussed in this study).

Application of BDD to the project consisted of transcription of features and scenarios, ac-

Title

<table>
<tr><td colspan="2">Description</td><td colspan="2">Owner
Primary Actor(s)</td><td>Status
ACTIVE / INACTIVE / SHIPPED Target Release</td></tr>
<tr><td colspan="5">System Context</td></tr>
<tr><td colspan="2">Preconditions</td><td colspan="3">Describe the state of the system prior to the first event in the use case</td></tr>
<tr><td colspan="2">Post-Conditions</td><td colspan="3">Describe the state of the system after the use case has taken place</td></tr>
<tr><td colspan="5">Sequence of Execution</td></tr>
<tr><td>Steps</td><td colspan="3">Description of Actor Step</td><td>Description of the System Response</td></tr>
<tr><td>1</td><td colspan="3"></td><td>Detailed description of how the system interacts with the actor</td></tr>
<tr><td>2</td><td colspan="3"></td><td></td></tr>
<tr><td colspan="5">Sequence of Execution (Alternate Path)</td></tr>
<tr><td>Steps</td><td colspan="2">Description of Actor Step</td><td>Alternate Path Trigger</td><td>Description of the System Response</td></tr>
<tr><td>1</td><td colspan="2"></td><td></td><td>Detailed description of how the system interacts with the actor</td></tr>
<tr><td>2</td><td colspan="2"></td><td></td><td></td></tr>
<tr><td colspan="5">Assessment</td></tr>
<tr><td colspan="2">Assumptions</td><td colspan="3"></td></tr>
<tr><td colspan="2">Key Constraints</td><td colspan="3">Description of constraining requirements</td></tr>
<tr><td colspan="5">Validation</td></tr>
<tr><td colspan="2">How is this demonstrated?</td><td colspan="3"></td></tr>
<tr><td colspan="2">How is this validated?</td><td colspan="3"></td></tr>
</table>

Figure 6.1: Use Case Template

ceptance testing, and implementation. The features and the associated scenarios were elicited in separate meetings for the project. Whereas, implementation of both the application and the tests was performed by the developer. Several additional meetings also took place for minor clarifications on the project. These additional meetings included *(i)* face-to-face conversations between the developer and the product owner for clarifications on the nature of input data; and *(ii)* the telephone conversations between the author (of this thesis) and the product owner or the author and the developer for further clarifications on the elicited features and scenarios.

The overall project (i.e., the larger project and its sub-part) was developed in sprints, and the duration of each sprint was two weeks initially. Afterwards, the team felt that some of the issues they were putting into their backlog were either too big or the priority of the tasks kept on changing because of the pressure from the customer (i.e., the users of the tool). Consequently, at the end of two weeks, the team would end up working on requirements different than they planned for that particular sprint. So, within the first few months of the project, the team moved to running one-week sprints. However, after running a one-week sprint for several weeks, the team moved back to a two-week sprint duration once again because the team felt that there was not much to discuss after one week.

Up till the point of the interviews, there were fifteen two-week sprints and ten releases of the overall project. At the end of each sprint, the team held a back-to-back review and a retrospective meeting on the same day. There was no separate sprint planning for the project (discussed in this study). Therefore, the tasks for a sprint for the project were part of the list of tasks for the overall project.

The team had daily stand-up and review meetings. In the review meetings, the team dis-

cussed the feedback on the overall project with the stakeholders and also reviewed the backlog. In the daily stand-ups, the team typically discussed the tasks they were working on at that point and the tasks they did on the last work day. They also introduced a weekly backlog refinement meeting to talk about the backlog, and what certain stories or tasks would involve.

Testing of the overall project was performed by the product owner, the developer, and the potential users of the application. The product owner and the developer were involved in unit testing and integration testing with the rest of the platform; whereas, the users provided feedback after manually using the application.

6.2.3 Development Technology

Initially, the team wanted to develop the project using Matlab. Just before starting the development, it was decided that the project would be developed using Python/Django and not Matlab. The decision was made because of the technological limitations and licensing cost of Matlab (i.e., the team had a limited budget).

The project was developed in Python using the Django framework*. The data was stored in Apache HBase† - an open-source database, and Behave‡ was used for automating the specifications. The other tools for supporting the project included Cucumber§ for BDD tests management, Jira for requirements management, and Confluence for drawings and documentation. The team used a local version control system because of the confidentiality of the projects.

6.3 Action Research

A decision was made to implement an action research study following the guidelines and process described in Section 2.4. By the time we decided to collaborate with them and incorporate BDD into the project, the team had already started some initial development work using the initial set of use cases. As described in Chapter 4, the company was in transition towards agile. Their biggest motivation for adopting agile was early delivery and effective requirements management. Due to our research interest in the application of agile methods in large-scale environments, we agreed to collaborate on a project.

The first meeting with two senior employees from the company, regarding the project (discussed in this study), took place on 11th October, 2018. Since the previous study described in Chapter 4 highlighted requirements management as a challenge of agile development in a large-scale environment, it was mutually decided to explore the application of Behaviour Driven Development (BDD) to the project. After the first meeting, we were handed over the fifty use

*https://www.djangoproject.com
†https://hbase.apache.org
‡https://behave.readthedocs.io/en/stable/
§https://cucumber.io/

```
In order to <achieve a business goal or deliver business value>
As a <stakeholder>
I want <something>
```

Figure 6.2: Format of a Feature in Gherkin

cases for conversion into BDD features. We went through four iterations with varying numbers of stakeholders, mostly the development team and the users.

First Iteration

Planning: All the use cases were examined and discussed between the author and his supervisor. The author established a strategy for conversion. Rules were defined for interpreting different fields of the use case template in Figure 6.1. To generate a raw suite of user stories (features) and scenarios compatible with the Gherkin language, we decided to:

1. Transcribe 'Pre-conditions' as (Given steps in the) 'Background'.
2. Transcribe 'Sequences of Execution' steps as 'When' statements.
3. Transcribe 'Description of the System Response' as 'Then' statements.
4. Transcribe 'Post-conditions' as 'Then' statements.
5. Alternative triggers may describe alternative user actions ('When'), causes of system failure (annotations to 'Then' statements or alternative scenario labels).

Action: The author studied each use case and transcribed the system's intended behaviour in the Gherkin feature format described in Figure 6.2. The first line in Figure 6.2 (starting with *In order to*) describes the rationale of the functionality. The second line describes the intender who wishes to perform the action or the functionality described in the third line of Figure 6.2.

After writing a feature for each use case, scenarios were transcribed for each feature while following the rules we described earlier. During this iteration, fifty use cases were converted into Gherkin features.

Analysis: During the conversion, we identified that the emerging scenarios were complex and contained interleaved When/Then statements. Figure 6.3 shows the structure of one such feature and the corresponding scenarios. The text describing the actual use case is redacted due to project-related confidentiality. The Figure 6.3 shows that the transcription of the use cases resulted in scenarios with interleaved When/Then statements.

It was observed that Given, When and Then statements are synonyms for the Arrange, Act and Assert stages of a unit test, according to the AAA pattern, proposed by Bill

```
Feature: < description >
        In order to <achieve a business >
        As a <stakeholder>
        I want <something>

Background:
# Pre-conditions
        Given <a context>
        And < additional context related information >
        And < additional context related information >

Scenario: < description of a scenario>
# Steps and responses
        When < an action >
        Then < expected outcome >
        When < an action >
        Then  < expected outcome >
        When < an action >
        Then  < expected outcome >

# Post-conditions
        And < additional expected outcome >
        And < additional expected outcome >

Scenario:  < description of a scenario>
# Steps and responses
        When < an action >
        Then < expected outcome >
        When < an action >
        Then < expected outcome >
```

Figure 6.3: Structure of a Scenarios with AAA violations

Wake¶. AAA pattern is a common way of structuring and organising a unit test into three functional sections (i.e., Arrange, Act, Assert) [Ma'ayan, 2018, Sundelin et al., 2018]. Following this pattern creates a clear separation between a unit test's setup (i.e., Arrange), its operation (i.e., Act), and results (i.e., Assert). Sundelin et al. [2018] argue that following the AAA patterns makes a unit test easier to maintain and understand. Therefore, the interleaving of when and then steps was a potential indicator of complexity in the BDD test scenario because it violates the AAA pattern.

Violation of AAA pattern (i.e., use of interleaved Given-When-Then statements) in a scenario is an indication that the scenario is attempting to test *too much functionality*. According to Khorikov [2020b], testing too much functionality in a single unit test makes a test "*exceed the realm of a unit test and become an integration test*". Oliveira and Marczak [2018] argue that scenarios with interleaved Given-When-Then statements are difficult to read and understand. This implies that the Gherkin scenarios with AAA pattern violations (i.e., use of interleaved Given-When-Then statements) are difficult to read and understand even if they are functionally correct. We refer to the violation of AAA pattern as one of the bad smells in unit testing.

Reflection(corrective action): The resulting scenarios showed us that it may be possible to translate use cases into BDD scenarios. The translation strategy we devised was unknown until we attempted it and then performed a corrective action i.e., evolved our strategy. However, the accuracy of the strategy we applied for the translation of the use cases into BDD scenarios was unknown.

The process of translating the use cases into BDD scenarios allowed us to detect the violation of the AAA pattern which was not evident in the use case format. At this stage, we observed that we needed a means of breaking up the associated features into smaller features. In the process of searching for tools and support on Google, we found that BDD lacked tools and methods for identifying and refactoring scenarios with bad smells.

Second Iteration

Planning: The scenarios with the AAA pattern violations were presented in a meeting to the two senior members of the project team on 10th January, 2019. It was agreed during the meeting that breaking the scenarios is inevitable which indicated that the features also need breaking up. Six use cases were prioritised for conversion during the meeting. Among those six use cases, four were from the previous set of fifty use cases while two of them were new.

Action: Since we did not find a tool support for removing AAA pattern violations, we had to

¶`https://xp123.com/articles/3a-arrange-act-assert`

do it manually. First, the two new use cases were also converted into Gherkin features according to the rules we described earlier. Then each of the six features was discussed in detail between the author of this thesis and his supervisor. Each feature was broken down into smaller features based on this discussion. There were seven scenarios in six feature files before removing the AAA pattern violation. The removal of AAA pattern violations led to the breaking up of the features and scenarios which resulted in a total of nineteen scenarios and twelve feature files. We presented the resulting features, scenarios and the corresponding code steps in a validation meeting to two experts from the company.

Analysis: During the process of conversion and removal of AAA pattern violations, we made five more observations about the limitations of Gherkin as summarised below:

- During the conversion of the scenarios, it was observed that Gherkin has a non-hierarchical requirements organisation. Gherkin does not provide a means to define hierarchy or dependencies between features. Each Gherkin feature file is a text file with no indication of the feature it was derived from. Gherkin also lacks a means of expressing dependencies across features and scenarios.
- During the second iteration of the action research, we observed that sometimes the scenarios and steps are simple to express in Gherkin but turn out to be complex in implementation. The appropriate level of abstraction in Gherkin steps is unknown [Binamungu, 2020]. Gherkin scenarios and steps do not express the details embedded in implementation.

  The documentation for Cucumber (a tool used for writing Gherkin) [CucumberStudio, 2019b] advocates writing declarative Gherkin. Declarative scenarios hide implementation details and focus on the goal without specifying how the goal is achieved. Whereas, imperative scenarios include the details of the user's interaction with the system, often UI. The rationale behind avoiding writing imperative scenarios discussed in the documentation [CucumberStudio, 2019b] is that imperative scenarios include UI details which can often change. This can make the scenarios brittle and hard to maintain. However, our finding does not relate to a user's UI interaction. We observed that an apparently simple step like "*firing a space rocket*" can have an extensive amount of implementation in the background which is hard to imagine at the scenario writing stage.
- We noticed that most of the feature files consisted of scenarios with multiple assertions. Having multiple assertions in a scenario is an indication that multiple behaviours are tested in a single scenario, and it is a violation of *Cardinal Rule* of BDD[‖]. According to the cardinal rule in BDD, every scenario must be focused on

[‖] https://automationpanda.com/2018/02/03/are-gherkin-scenarios-with-multiple-when-then-pairs-okay/

testing a single behaviour. Multiple assertions could mask bugs e.g., if an assert fails in a test, the status of the remaining part of a test becomes unknown since the test stops execution at the first failing assert. As discussed before, we did not find tool support to identify and remove *assertion roulette* bad smell in Gherkin scenarios.

- Each feature in Gherkin is seen from a single user's perspective. Gherkin does not support describing the feature from multiple users' perspective i.e., a feature has a single "As a" statement. Gupta [2019] recommends writing a separate user story for each user but the problem is that Gherkin does not provide a way of expressing dependency between features in case the features are separated. Also, the features which mandate participation of more than one actor cannot be expressed using Gherkin.
- Gherkin does not support concurrency of execution of scenario steps. There is a lack of support in Gherkin for the situations where two or more steps are to be executed together to successfully execute a scenario. For example: In a situation where two actors need to perform the action specified in the scenario together for successful execution of the scenario.

Reflection (corrective action): After the removal of (AAA violations) bad smells, the twelve features were presented to the four members of the (larger) project team (including the two senior members from the previous meetings) on 23rd January, 2019. This meeting lasted four hours. Each use case was discussed separately in detail during the meeting. The participants observed that although the proposed features corresponded with the use cases, they did not express the functionality intended for the system. The participants stated that the conversion of the use cases into user stories had made the documented requirements clearer, allowing them to identify where they were incorrect.

The strategy to translate use cases into BDD scenarios showed that it may not be possible to directly translate the use cases into user stories i.e., there could be intermediate step(s) in the middle. This discovery was not made until after we tried converting the use cases to Gherkin scenarios. One of the reasons could be the difference in basic structure of a use case and a BDD scenario. Use cases tend to contain a multiple paths of execution with no distinction between action and assertion, whereas a scenario describes a particular instance in a specific context.

Third Iteration

Planning: At the end of the meeting, we decided to conduct a User Story workshop at the company which would enable them to create a new set of requirements written as user stories following a Behaviour Driven Development process from the outset. A date was decided for conducting the User Story workshop.

Action: The workshop was conducted with four senior members of a development team for the wider project (i.e., web application for managing large data set generation) on 1st May, 2019 at one of the company's premises. It was pre-decided that the workshop will be focused on the features for an upcoming iteration.

The project was outlined and all the main stakeholders were identified at the beginning of the workshop. After giving team members a tutorial on writing the user stories, they were asked to write user stories. Each user story was written on a sticky note and pasted on a white board. While the participants were pasting user stories (sticky notes) on the board, the author grouped them into different functions such as search, upload, and analysis. All non-functional requirements for example, security and user-interface (UI) requirements, were grouped together under non-functional requirements.

Analysis: The author of this thesis inspected the user story board after the completion of user story writing session. Many duplicates were identified in the user stories.

Reflection (corrective action): In a following session, all duplicates and out of scope user stories were discarded after discussing each user story with the participants of the user story workshop. At the end of the session, we had a total of 41 user stories out of which, seventeen were non functional requirements. After the final selection of the user stories, it was decided that the author will convert the user stories into BDD features and later discuss the scenarios for each feature with the participants.

Fourth Iteration

Planning: Right after the User Story workshop, the author (of this thesis) started working on converting the user stories, elicited during the workshop, into BDD features. In the first week of July 2019, we were told that an individual had joined the company who would work as a software developer for the project discussed in this study. A meeting was scheduled for 1st August, 2019 to share project materials with him and have a discussion on BDD and its application in the project.

Action: Four individuals participated in the meeting on 1st August, 2019. The participants included the software developer, a senior member of the project team (from the previous meetings), the author(of this thesis) and his supervisor. The senior member of the project team assumed the role of the product owner for the project focused on this study.

At the meeting, the participants discussed the role of BDD, its importance in the project and the *progress so far*. It was decided at the meeting that the author will collaborate with the development team on refining the scenarios while the development team will start implementing the tests for the scenarios. After the meeting, the project team set up a GitHub repository to collaborate with the author. The features were put in Jira, and the

features were divided into four functional categories. The code was also uploaded to the GitHub repository.

Clarification was required around some aspects of terminology in the scenarios to be developed, so an additional meeting with the team was scheduled on the 23rd August, 2019, lasting for four hours. All three team members were present during the meeting, along with the researcher. The purpose of the meeting was to clarify some of the technical details and elicit examples of data for the scenarios. During the meeting, thirteen features out of a total of 41 were reviewed and clarified. As a consequence of the discussion, a further sixteen additional features were identified that required elaboration. It was agreed that a further meeting would be scheduled to review the remaining features.

Analysis: At this stage, it was apparent to us that the pace of development became faster than the pace of updating the features and scenarios. This was later confirmed in interviews. The reason for this was the time pressure for completing the project. The development team started lagging behind in meeting the project schedule because *getting the scenarios right* was taking longer than the team expected. The development team was not sure about when to stop refining the Gherkin specifications. They did not know *when is a particular scenario good enough?* i.e., it completely represents the user's intention.

We also observed that Gherkin gives a lot of freedom of expression, and there are no restrictions on the way a feature or the corresponding scenarios are described, which is one of reasons it is very difficult to maintain the quality of the scenarios. Studies [Lucassen et al., 2016, Prakash and Prakash, 2017] show that the practitioners mostly rely on the experience of the user story writer to ensure the quality of the user stories. According to the studies [Oliveira and Marczak, 2018, Smart, 2014], badly written scenarios can negatively impact the ability of the tests to reflect the system coverage and the team confidence in them.

Oliveira and Marczak [2018] define a set of five quality attributes for the quality of BDD scenarios. Unfortunately, the quality attributes defined by Oliveira and Marczak [2018] are descriptive and subject to the understanding and experience of the person applying them. For example, one of the quality attributes *Small* is hard to determine and requires judgement of the person writing BDD scenarios. Prakash and Prakash [2017] argue that determining *Small* in the context of user stories is not apparent.

Reflection (corrective action): To keep the pace of refining the scenarios with the development, we scheduled another meeting on 9th September, 2019. The participants of this meeting were: the product owner, the developer, and the author of thesis and his supervisor. The aim of this meeting was to seek clarity on few of the unclear terminologies in the scenarios in four feature files.

> In this meeting we discussed scenarios the developer was working on. Some of the features were prioritised for development. The development team could not discuss the actual example of the data due to the confidentiality of the project. Therefore, the unclear scenarios were completed using dummy data. Also, we refactored some of the feature files to improve readability.

Overall, the action research phase of the study comprised eight separate meetings with the development team spanning more than 32 hours, as well as considerable offline work to define and revise features. In addition to the (in-person) meetings, the research team (i.e., author and his supervisor) and the development team exchanged more than a hundred emails over the period of twelve months. These emails were focused on clarifications and updates. Telephone calls were also scheduled for minor clarifications and to discuss updates. The work in this section was carried out between October 2018 and November 2019.

The action research revealed several limitations of the BDD practice (as described in theoretical accounts, [Binamungu et al., 2018a,b, Oliveira et al., 2019]) and the Gherkin language. These limitations could potentially result in complexity in requirements specifications or difficulty in maintaining specifications. To confirm these findings, follow-up interviews were conducted with the development team to understand the practice of BDD from their perspective.

6.4 Semi-Structured Interviews

Post hoc semi-structured interviews were conducted with the development team for the project (discussed in this study). The aim was to gather qualitative data in the form of experiences and opinions of the people who used BDD for the development of the said project. As explained in Section 2.3.1, semi-structured interviews offer freedom of expression to the participants, and open-ended questions prompt discussion, aiding the interviewer to explore a particular theme. Wengraf's guidelines [Wengraf, 2001] (as explained in Section 2.4) were used to construct the interview instrument. Figure 6.4 illustrates how Wengraf's method was applied to the design of the semi-structured interviews.

Wengraf's guidelines [Wengraf, 2001] is a top down approach for developing a semi-structured survey instrument. Wengraf follows an iterative process for refining the questions derived from the overall research purpose into a number of smaller questions. The *Research Purpose* (RP) in this case is: "*Learn about the feasibility of applying Behaviour Driven Development to the development and maintenance of a system developed in a large-scale environment; to gain a deeper insight into the benefits and difficulties experienced when developing and maintaining a system using Behaviour Driven Development.*".

In the current study, the RP was refined into three central research questions (CRQs) included in Figure 6.4 for completeness. Each CRQ was divided into a number of *Theory Questions* (TQ), specific propositions investigated during the study. For example, CRQ2 is refined into three

Research Purpose	Central Research Questions	Theory Questions	Example Interview Questions
Learn about the feasibility of applying behaviour driven development to the development and maintenance of a system developed in a large-scale environment; to gain a deeper insight into the benefits and difficulties experienced when developing and maintaining systems using behaviour driven development.	1. What is the extent of the application of behaviour driven development to development and maintenance of the systems in the company?	1. What is the level of the team's familiarity and experience with agile methods and BDD?	When did you personally start using behaviour driven development?
		2. How was BDD incorporated in the overall project structure and team setting?	Who was involved in writing BDD specifications?
	2. Is it feasible to apply behaviour driven development to the development and maintenance of a system developed in a large-scale environment?	3. How does BDD help in the development of a system in a large-scale environment?	What tools are you using for automating BDD specifications? Have you found any problems with the tools?
		4. How does BDD help in the maintenance of a system developed in a large-scale environment?	What steps did you follow to accommodate and manage a change or update the specifications?
		5. Can you record all the requirements with BDD?	What were the types of requirements that were difficult to document in BDD?
	3. What benefits and difficulties were experienced when using behaviour driven development?	6. What are the perceived benefits of applying BDD?	In which phases of software development, BDD helped? How?
		7. What difficulties were experienced when the project was developed using BDD?	Are there any problems that you expected and faced after application of BDD OR expected but did not face?

Figure 6.4: Research question construction process following Wengraf's method [Wengraf, 2001]

Interview	Role	Responsibility
P2	Industrial Placement Student	Software Developer
P3	Lead System Engineer	Product Owner
P4	Deputy Lead Software Engineer	Scrum Master

Figure 6.5: Summary of Interview Participants

TQs, including *"TQ2.1 How does BDD help in the development of a system in a large-scale environment?"*. To answer each TQ, a number of interview questions were defined. Figure 6.4 shows a sample of interview questions, with the full survey instrument available for review (in Appendix B). This approach provides a traceable hierarchy and rationale behind every interview question.

Full interviews were conducted during four sessions with three members (Participants P2-P4) of the development team: the Software Developer, Product Owner, and Scrum Master (as described in Figure 6.5). These interviews were conducted in person at the company's premises. The first interview was conducted with the Software Developer on 10th December, 2019. Rest of the interview sessions were conducted with the Scrum Master and the Product Owner on 27th February, 2020. The aim was to gather data from multiple perspectives within the project. Interview participants had different experiences, expertise, and roles.

Figure 6.5 presents a summary of the information on interview participants. These experiences included acting as a product owner, scrum master, and software developer. The second and third (P3 and P4) participants had some experience of adopting agile methods within their projects; whereas, the first participant (P2) was using the agile process for the first time in a professional environment.

The approximate duration of each interview was 90 minutes. However, the interview with the product owner was conducted in two sessions because of his other commitments on the interview day. The duration of the two interviews with the product owner was 70 minutes each. All the interviews were transcribed and sent to the participants to make additions or clarifications. After getting verbal permission from each participant, the transcripts were used for analysis.

Analyses of the transcripts were performed using Wengraf's guidelines [Wengraf, 2001], using a bottom-up approach to answer the questions at each level. Every question was answered at each stage in the hierarchy by starting from the bottom i.e., Interview Questions.

A table similar to Figure 6.4 was created for this purpose. Answers to each interview question from all participants were pasted in the Answer column next to the respective interview question. All the answers to each interview question were then merged to form a story. Different stories for every group of IQ relating to a Theory Question were then merged to answer each TQ. The group of Interview Questions for each Theory Question was deleted such that each Theory Question had a descriptive answer. The same process was repeated to find answers to CRQs.

The descriptive answers to each CRQ were reviewed by the author and his supervisor, and

the issues reported in them were highlighted. Four limitations were identified during interview analysis, out of which two overlapped with the five limitations identified during action research.

Figure 6.6 lists the challenges and limitations of Gherkin discovered during the action research and the post hoc interviews. Their discovery gave us an understanding of the limitations of Gherkin that directly or indirectly affected the actual and perceived benefits of Behaviour Driven Development within the company. Note that where we use quotations below to illustrate a challenge, it is sometimes necessary to anonymise some of the topics to preserve confidentiality. The interviews discussed in this section were carried out between December 2019 and February 2020.

6.5 Discussion of the Limitations and Observations

This section reviews the observations made during the application of Behaviour Driven Development (BDD) while using two different research methods throughout the course of this study. This section discusses the adoption of BDD; expectations, achievements and hurdles in practicing BDD in the project (discussed in this study). This section draws on the analysis of the answers to the interview questions and observations made during the action research to develop a description of the use of BDD in the project.

It must be recognised that the adoption of BDD/Gherkin in the company was their first attempt using this approach in their context. Therefore, it is possible that the approach adopted by the team was sub-optimal and that the challenges identified might be overcome through adaption and learning as the engineers gained more experience with the approach. Nevertheless, the research does demonstrate the challenges faced by a reasonably well-resourced software team in adopting BDD in the described context.

The company is a large organisation having sub-divisions e.g., systems engineers, software engineers, and engineering department. Different sub-divisions are responsible for performing different tasks in the company e.g., the systems engineers write the specification while the software engineers do the implementation. The adoption of BDD was driven by the need to improve communication and a common understanding of the requirements. Its adoption was expected to bridge the gap and improve the communication and coordination between different sub-divisions as the product owner (P3) said "*...I hoped that it would close the seams between the specification and the implementation*".

After a couple of iterations of action research, it was revealed that the initial set of user stories was inconsistent and did not represent the requirements adequately. So, the team decided to discard the initial set of user stories and elicit the requirements in a user story workshop (as discussed in Section 6.3). Initially, 41 features were elicited for development later evolved to 57 features and 64 scenarios. Each feature had at least one scenario. Alternate scenarios for all the features were not clear at this point.

After the elicitation of the user stories (features) in the user story workshop, each feature was discussed individually, and the ones which were finalised for development were put in Jira. A Jira ticket was created for each requirement. Every ticket had a description and acceptance criteria. When we inquired about the synchronisation between acceptance criteria in the Jira tickets and the acceptance criteria in the scenarios, we were told by the product owner (P3) "*...Well, there's a disconnect, but they should all...I think, we were mindful to make them related"*. We learned that they wrote the Jira tickets manually. This sometimes caused confusion among the team members because of the duplication of the acceptance criteria i.e., writing it twice (once in the scenarios and once in the Jira tickets).

One of the main reasons for this duplication was that the team had to stay consistent with the internal process of the company along with the attempt to write and execute a BDD test suite for this project. The purpose of the acceptance criteria in the Jira tickets in the company's internal process was to build an understanding of the functionality and to express the rationale behind each functionality. The internal process of the company required them to write tickets in Jira. As the product owner (P3) mentioned "*...we write down acceptance criteria on all of our issues, not just related to this"*.

In addition to the Jira tickets, vision-level drawings were created in Confluence. Some of these drawings were related to a specific set of features to communicate " *before and after"* i.e., this is what the users can do now; this is what they will be able to do later. These drawings were presented to the stakeholders to communicate the development tasks. The drawings helped in setting the expectations of the stakeholders. The product owner (P3) pointed out that most of the customers were non-technical; therefore, " *...we communicated... almost totally through drawings, like some of it ... whiteboard drawings and then photographs, some of it electronic... drawings"*.

At the end of each iteration, the team had a retrospective meeting. The retrospective meetings were focused on the progress and evaluation of the overall project. The team assessed its performance in the retrospective meetings, and the meetings' outcomes and details were recorded in Confluence. Things like (P3) " *...What we could improve, what we could keep, what we could drop, what we could add in terms of team activities"* were discussed at the retrospective meetings. As discussed before, there were no separate retrospective meetings, separate stand-ups or processes for the sub-part. As one of the participants *(P3)* said "*...we treated this as part of those bigger sessions and just sort of tack it on.."*.

The BDD process being used in the sub-project was not discussed in the meetings for the overall project. Only a brief summary of the developed features was discussed. One of the problems at the retrospective meetings was that the team members, who were not involved in the development of the sub-part, experienced difficulty in understanding the discussion about it.

Apart from the challenges in applying BDD, it helped the team to communicate the requirements within the team and encouraged them to discuss each requirement in detail. The team

believed that the BDD workflow enforces communication as one of the participants (P3) mentioned "*...you kind of have to have the conversations; otherwise, you can't build the tests"*.

The findings from four iterations of action research and four sessions of semi-structured interviews are combined and summarised in Table 6.6. The third column of Table 6.6 indicates the research method, during the use of which, each observation was made. It must be kept in mind that the challenges listed here are practitioners' portrayal of the BDD process. Please note that where examples are provided to illustrate a point, they are based on real findings in the action research or interviews, but here they are replaced with toy examples due to the confidential nature of the project.

6.5.1 Test First Development is Difficult to Apply

We observed during the action research and the interviews that some of the functionality was developed before specifying the BDD tests. The literature on BDD discourages this practice i.e., coding before writing BDD tests [Wang and Wagner, 2018, Barus, 2019, Moe, 2019, Solis and Wang, 2011, Smart, 2014]. When we inquired about the reason behind this practice in the project, we were told that the developer was not familiar with the technology and the tools. Also, this was the first time he was practicing BDD. As the software developer (P2) mentioned " ... *it was kind of... it was a little bit experimental start, kind of just seeing what we could do... I am still kind of learning how to use the framework....just because I wasn't sure if that's what I wanted to do. It's kind of like experimenting, and then, if it worked well, I could write a BDD test and fully implement it"*.

According to the product owner, many times the software developer was not clear about the functionality until he developed it. As he (P3) said "... *he was using the code to think through the workflow, and then he writes ... like that seems sensible"*. The software developer (P2) also admitted " *...I want to see how it worked first. So, I would implement it, see how it worked. And if what I've done, I'd like too, so I'd write BDD test after..."*.

Also, we felt that the software developer was not entirely convinced about the use of BDD for the development. The software developer (P2) had some concerns about writing BDD tests for User Interface (UI) as well. As he (P2) pointed out "... *why we're actually doing behaviour testing... a lot of behaviours are through using.... interactive... UI. But we can't really test that. So, then that's where behave, BDD and the... application kind of clashes with what we want to test"*. One possible approach to address this issue is the use of Domain Specific Languages (DSLs). A DSL is a language tailored to a specific domain and provides ease of use and understanding while keeping the required formalism [Fowler, 2011, Rocha Silva, 2022]. For example, Silva et al. [2019a] propose an approach to identify various types of inconsistencies UIs. In another study, Rocha Silva [2022] propose a DSL similar to Gherkin for the specification of consistent and testable user requirements for web-based graphical user interfaces.

While talking about the benefits of BDD, the product owner (P3) pointed out that since the

	Challenges	Explanation	Action research/Semi-structured Interviews
1	Test first development is difficult to apply	Because of lack of clarity it is difficult for a developer to always imagine what a requirement will look like after development. Therefore, it is hard to write tests before development.	Action research and Interviews
2	BDD lacks methods and tools for identifying and refactoring bad smells	Complex scenarios, indicated by an interleaving of When/Then statements result in bad tests and they mask bugs. There is a need for a means of breaking up the associated feature into smaller features i.e., a tool that identifies bad smells in Gherkin scenarios such as AAA pattern.	Action research
3	Gherkin lacks a hierarchy of features	We can't express within Gherkin a relationship between epics, features, user stories, etc. It is important to know and document what evolved from what. Gherkin has no mechanism for showing the hierarchy of features.	Action research and Interviews
4	Identification of appropriate level of abstraction is difficult	for example, Gherkin lacks implementation details. Instead, the code has to be embedded in implementation, making the Gherkin implicit, rather than explicit. This makes it hard for the developers to imagine the system in advance.	Action research and Interviews
5	Gherkin doesn't support multiple actors in "*As A*" statements	Several of the use cases examined identified multiple actors, either with complementary or even identical roles. There is no mechanism within the Gherkin semantics for capturing these relationships.	Action research
6	Gherkin doesn't support concurrency of execution	This should really be critical for a requirements language for modern distributed systems.	Action research
7	Convincing the developer and the customer to use BDD	Like any other new method, it has been observed, that convincing people to adopt BDD is a challenge.	Interviews
8	Risk of duplication of effort in large-scale systems	When a lot of people are involved in a project and everyone is limited to their own set of tasks, it is very difficult to know if someone has already written the steps similar to what someone else was writing in which case, a test already exists.	Interviews

Figure 6.6: Summary of challenges found during action research and interviews

software developer was a novice in the company, he did not understand the vision, and what the product was meant to do. BDD gave the team a method to communicate the vision. According to the team, the use of BDD helped them define the project at an appropriate level which is understandable by the users without implementation.

Although *test first development* in BDD is inherited from Test Driven Development (TDD), the *test first development* in TDD is different from *test first development* concept in BDD. In TDD, the focus of *tests* is on the development of small units called unit tests, whereas the focus of *tests* in BDD is on acceptance testing or a particular behaviour of a system. This *behaviour* can consist of multiple small *units*. So, we can say that TDD is a bottom up *test first development* approach in which the smallest entity is a unit test, whereas in BDD, the smallest unit is a behaviour that can consist of various unit tests making BDD a top-down approach. According to the documentation on Cucumber (a tool used for BDD) "*.... TDD test asserts the result of a specific method, while the BDD test is only concerned about the result of the higher level scenario"*. This is why the studies show that the sequencing (i.e., the order in which test and production code are written) has no important influence [Fucci et al., 2017]. Fucci et al. [2017] analyse 82 data points collected from four workshop sessions conducted with 39 professionals from two companies about unit testing and TDD. The results of the study show that the order in which test code and production code is written has no impact on productivity in TDD. Vu et al. [2009] conducted an experimental study with undergraduate students in which the students designed, implemented, deployed, and maintained a software system to meet the requirements of an industry sponsor. One group of students applied a Test-First (TDD) methodology, while the other group applied a traditional Test-Last methodology. The results show that the Test-Last team was more productive and wrote more tests than their Test-First counterparts. However, the productivity in the study is measured by counting the *number of tests* writing which is much easier when the functionality is clear. This confirms our observation about the difficulty in applying test first development without knowing how the functionality will *look like* after development. This issue can further "*inflate"* when an approach involving testing behaviour through testing of multiple units at once is used i.e., BDD.

6.5.2 BDD Lacks Methods and Tools for Identifying and Refactoring Bad Smells

This problem emerged while converting the user stories, we received from the company, into Gherkin features during the first iteration of action research (explained in Section 6.3). The user stories we received turned out to be lengthy and had interleaving Given-When-Then. This interleaving Given-When-Then is a violation of the AAA pattern and, consequently, a bad smell.

Bad smells are certain structures in software artefacts that indicate design problems hindering the evolution and maintenance of the software artefacts [Fontana et al., 2012, Suryanarayana

et al., 2014]. Bad smells usually do not stop the software from executing, but their existence makes the system inflexible for the accommodation of future changes. They can exist in various forms in different software artefacts and usually require refactoring. Examples of the bad smells in different project artefacts include; code duplication and strong coupling between methods in the code [Fontana et al., 2012, Garousi and Küçük, 2018], and ambiguous phrases and nonverifiable terms in requirements specification [Femmer et al., 2017]. During this study, we found that bad smells can also appear in Gherkin specifications mainly because of the way BDD requirements are written.

We discovered two bad smells during action research i.e., (i) AAA pattern violation during the first iteration and (ii) assertion roulette during the second iteration. According to Ciliberti [2017], the AAA pattern improves readability by structuring a unit test into three distinct phases (Arrange-Act-Assert). It helps in separating what is being tested from the setup and results. BDD replicates this concept for structuring the specifications. Smart [2014] argues that organisation of natural language specification into distinct phases cleanly defines the context of a test and improves readability. According to the survey by Oliveira and Marczak [2018], the practitioners believe that mixing Given-When-Then step order impacts readability of the scenario.

As discussed in Chapter 5, a BDD scenario is implemented in the form of a unit test in a BDD test suite. This means that multiple assertions in a BDD scenario are also reflected in a unit test. These multiple assertions in a unit test is a bad smell known as *assertion roulette* [Bavota et al., 2012, Peruma et al., 2020, Grano et al., 2020]. We are unaware of any tools that detect these bad smells in BDD specifications. Other potential examples of bad smells in Gherkin specifications could include; combining two or more independent phrases in a single step [Smart, 2014] and duplication of steps across scenarios [Suan, 2015].

To the best of our knowledge, there is a lack of tool support for detecting and fixing structural discrepancies (i.e., bad smells) in Gherkin scenarios. We made this observation during the first and second iteration of the action research while searching online for a tool support for automatically detecting and refactoring bad smells in Gherkin scenarios.

6.5.3 Gherkin Lacks Hierarchy of Features and Traceability

While looking at the traditional requirements engineering process in the company, we learned that their requirements engineering was a hierarchical process. There were layers to it, and people used to engage at the layer that was appropriate for them to operate at. For example, a systems engineer would deal with high-level requirements; whereas, a software engineer would only look at low-level requirements.

The team, while discussing the difficulty with BDD, pointed out that there was no hierarchy in the requirements, and it was difficult for the team members to work at different layers of requirements. During the second iteration of action research, we observed that Gherkin does not provide a mechanism for showing which higher-level feature a certain feature was derived from.

This observation was later confirmed during the interviews. We observed that BDD features provide a single-layer modeling. A single-layer requirements model is built at the same level [He et al., 2013]. The set of features (in BDD) has a non-hierarchical structure with no indication of features they derive from i.e., there is no way of expressing that some features are refined from a higher level, more abstract requirement. Such a model lacks traceability.

The traceability of requirements is significant to ensure consistency among project artefacts [Duarte et al., 2016]. The record of requirements evolution is a part of the history of the project which shows the trail of decision making. Regulatory standards such as DO-178C and EN-50129 also have strict traceability requirements.

Another related issue in this context was the interdependence of requirements. As the product owner (P3) said "*... understanding how different scenarios or different features relate to one another... understanding how those things are related seems to be quite hard"*. The team also pointed out that getting enough detail in BDD is difficult. While talking about the advantages of BDD, one of the participants (P3) said "*Maybe the value of it's the same thing that makes it difficult... getting the detail in them is tricky. There can be lots of steps"*. The team also stated that there is no way of understanding how different scenarios or different features relate to one another in BDD.

According to Smart [2014], a feature is a functionality that can be delivered independently of other features. It implies that the concept of interdependence of requirements is foreign to BDD. According to Trkman et al. [2016], a lack of awareness about requirements dependencies can lead to missing information about a project and its domain. Requirements dependencies present information, like *which previously developed user stories are required by the new user stories?*, and *how the completion of a user story impacts another user story?* [Trkman et al., 2016].

Although the BDD process, explained in Section 5.3, describes activities that involve decomposition of requirements that consequently create some sort of hierarchy, this hierarchy cannot be reflected through features (i.e., user stories) or Gherkin. Also, the interdependence of requirements cannot be expressed through user stories or Gherkin.

The INVEST criteria by Bill Wake** recommends having no dependencies between user stories. The assumption behind this criterion is that user stories can be written independently. The action research and interviews suggest that this may not be practical, and it is useful to know the relationships between various requirements to ensure traceability.

6.5.4 Identification of Appropriate Level of Abstraction is Difficult

During both action research and interviews, we observed a lack of guidance on the level of abstraction in a scenario i.e., *how much implementation detail should be there in BDD specifications?*. BDD using Gherkin provides a single level of abstraction between the Gherkin

**https://xp123.com/articles/invest-in-good-stories-and-smart-tasks/

language and its implementation. This means that there are only two places that can accommodate complicated implementation details. If we remove all the implementation details from Gherkin scenarios then only the code will have all the implementation details. This issue is also highlighted by Binamungu et al. [2020]. The authors surveyed BDD practitioners to understand their opinions on quality criteria for BDD test suite. One of the findings of the survey shows that practitioners find it difficult to decide how much implementation detail should be there in a scenario.

This issue also relates to the inherent nature of user stories. The declarative nature of the user story helps the stakeholders to communicate amongst themselves without reference to implementation details [Smart, 2014, Wynne et al., 2017]. *"User stories encourage deferring details"* [Cohn, 2004]. If a typical user story is written following guidance such as by Beck et al. [2001b], it lacks details of user interaction and important factors like performance requirements; therefore, a user story is open to interpretations. Avoiding misinterpretations is one of the reasons agile recommends that customer should be a part of an agile team. The available guidance on BDD (as discussed in Chapter 5) emphasises on describing the requirements in terms of; *"...what a feature should do, not how it should do it"* [Smart, 2014]. As the scrum master (P4) said *"...the tricky part there is the implementation of each step can be a bit arduous"*.

However, unlike user stories, scenarios contain more information since they are real-life examples of the associated user stories. Even though this practice is discouraged by practitioners [CucumberStudio, 2019b, Smart, 2014, Wynne et al., 2017], various levels of implementation details can be embedded in the scenarios making them imperative. The issue here is not whether the Gherkin scenarios should or should not contain implementation details (i.e., declarative versus imperative scenarios). The issue is what appropriate level of detail should be there in a scenario. Completely declarative scenarios are easily understandable but hide the amount of implementation effort as well as UI details. Imperative scenarios are brittle [CucumberStudio, 2019b] and require constant maintenance which can create significant overhead.

6.5.5 Gherkin Does Not Support Multiple Actors in *"As A"* Statements

During the second iteration of action research, we observed that one of the limitations of a user story is its lack of support for the multiple actors i.e., it is difficult to describe a feature from multiple actors' perspective. The atomic nature of a user story and the assumption of a single actor forces a developer to view the problem in isolation from the perspective of a single actor i.e., a feature has a single *As a* statement.

Although the content of the user story block of a Gherkin feature is only provided for documentation purposes and is not leveraged when executing the test suite, Gupta [2019] recommends writing a separate user story for each user. According to the author, having each user describe the system from his/ her own perspective creates multiple conceptual models and these models can be used for a better description of the system. The BDD process (as described by

```
Feature: unlock the safety deposit box
In order to access the content of a security box
As a bank manager and a customer
We want to unlock the security system
```

Figure 6.7: Feature with multiple actors

Smart [2014] and Wynne et al. [2017]) also recommends the use of a single actor in a feature.

In BDD, the scenarios pertaining to a feature are further elaboration of a user story in real real-life context. Since the narrative of a user story is from a single person's perspective, the scenario also represents the perspective of a single person. Division of scenarios with respect to users in a feature is not supported by Gherkin. Nevertheless, different users can have different needs and perspectives of a system [Rosson and Carroll, 2002], therefore it is important that this singularity is maintained by Gherkin.

Having a separate user story for each user can only happen when users using a feature do not require each other's participation for successful execution of the functionality expressed by a feature. In this case, a single feature used by various users can be separated into various features such that each feature will have a single actor i.e., copies of the same feature with different actors. This will, however, increase the size of the test suite and involve a risk of duplication of features.

This also means that the features which mandate the participation of more than one user may not be expressed using Gherkin. This limits the ability to express features that mandate more than one user. For example, Figure 6.7 describes *unlocking* feature for safety deposit boxes in a bank. The security system of the vault requires two keys i.e., the bank manager's key and the customer's key to unlock a safety deposit box. Both keys must be turned together to unlock the safety deposit box. Unfortunately, the features that require the participation of more than one actor are not supported by Gherkin.

6.5.6 Gherkin Does Not Support Concurrency of Execution

This limitation of Gherkin was observed during the second iteration of action research. As we developed our feature suite, we noticed that the ordering of steps in some scenarios was arbitrary and could be re-ordered with the same post conditions applying. In principle, it would be desirable that the system did not discriminate between different combinations of step sequences of such scenarios, and considered one test code for multiple combinations of step sequences of the same scenario.

However, in practice, the order of steps in Gherkin scenarios is important. Each combination of the sequence of steps of a scenario is treated as a different scenario by the test suite. The only way to express a scenario in which the step order is arbitrary using the Gherkin language is to create multiple, very similar scenarios leading to increased cloning as is there no way of

```
Scenario: open bank safe
Given a locked steel safe
And a customer with a key
And a manager with a key
When the customer turns the key
And the manager turns the key
Then the safe is unlocked
```

Figure 6.8: Scenario with multiple actors

```
Scenario: race to the finish
Given a race line 100m away
When I run 100m
And my friend cycles 100m
Then my friend finishes first
```

Figure 6.9: Scenario concurrent execution of steps

denoting in the existing language the ordering that applies to the groups of steps.

Similarly, the simultaneous execution of two steps in scenarios is not supported by Gherkin. Concurrency in the context of software development is a phenomenon in which two or more processes cooperate with each other to complete a task [Axford, 2002, do Rocio Senger de Souza et al., 2011]. There could be a number of execution orders between concurrent processes e.g., the processes could start together but end at different times, start at different times but end together, or start or end one after the other, etc. To test all possible execution orders is very difficult [Radnoci, 2009] and would require a number of unit tests, each for testing a particular execution order.

Figure 6.8 gives an example of this problem. The figure is an example of a scenario for *unlocking safety deposit box* feature example in Section 6.5.5. A bank locker is only unlocked when the customer and the bank manager turn their keys at the same time. There could be a number of execution orders for the above scenario. For example, the manager and customer could turn the key at the same time, start turning the key one after the other, or finish turning the key at the same time, etc. Ideally, the safety deposit box should unlock for all of these concurrent execution orders. This means that the order of execution is of no concern in the above scenario. In a scenario where execution of steps can occur in any order, we need to have several unit tests, each testing a particular execution order to demonstrate that we get the same output for each execution order. There could be other examples where the order of execution matters.

Consider another example in Figure 6.9 where two people start the race together. In this example, two steps run sequentially so the scenario does not reflect the desired behaviour. As a first proposed solution, we implement "*When I run 100m*" as a thread in Figure 6.10. The conceptual basis of Gherkin is that each step should describe a single behaviour, be atomic, and executed sequentially. A step sentence should also be an explicit description of the actual behaviour. If "*When I run 100m*" is implemented with a thread it is not explicit, because we can

```
...
def run_100m():
   while i < 100:
      sleep 10:
      metres += 1

threading.Thread(target=run_100m).start()
...
```

Figure 6.10: Implement "*When I run 100m"* as a thread

```
Scenario: race to the finish
Given a race line 100m away
When I run 100m and my friend cycles 100m
Then my friend finishes first
```

Figure 6.11: Scenario with a complex step

no longer guarantee when the thread is going to terminate. If we reword this to "*When I start running 100m"*, we still cannot be sure when the step will terminate (if ever). Moreover, a step should always be atomic i.e., a step should not depend on the prior execution of another step [Smart, 2014]. So the step "*When I run 100m"* at the same time that "*my friend cycles 100m"* will still contain a bad smell because it references another behaviour that must be executed. Also, a step must be fully completed before the next step starts, so "*When I start running 100m"* is smelly because it is vague - it does not describe a behaviour with a fixed termination.

A second solution is to to execute two logically different steps concurrently in the implementation glue code as shown in Figure 6.11. However, a step should not describe a composite behaviour. Smart [2014] argues that combining two or more behaviours in a single Gherkin step could lead to writing a method that tests multiple behaviours within a single method. This is a type of bad smell known as eager test [Garousi et al., 2018]. In an eager test, in case of a failure of any of the asserts, the method stops execution which prevents the remaining parts of the test from executing. According to Garousi et al. [2018], eager tests make it hard for the developer to understand the fault during the execution of the test.

Unfortunately, Gherkin does not support any of these concurrency relationships. It is important to note that we are not discussing any particular execution order. We are discussing a limitation of Gherkin language that it does not facilitate concurrent execution of steps in a scenario irrespective of their execution order. Instead, steps in a scenario execute in a sequence. Also, this is a finding from a real-life project which shows how BDD is practiced in reality. A way to fix this issue is to extend Gherkin semantics so that there is an explicit mechanism for asserting that two (or more) atomic, single behaviours run concurrently as shown in Figure 6.12.

```
Scenario: race to the finish
Given a race line 100m away
Concurrently:
    When I run 100m
    And my friend cycles 100m
Then my friend finishes first
```

Figure 6.12: Scenario with concurrent execution

6.5.7 Convincing Developer and the Customer to Use BDD

Resistance to change is considered a common behaviour in software organisations [Ashbacher, 2010, Gandomani et al., 2014, Jyothi and Rao, 2011], and the reasons could be political, technical or mere uncertainty [Gandomani et al., 2014]. Stray et al. [2020a], in a systematic literature review on agile coaching and the role of agile coach, describe resistance to change and difficulty in understanding and implementing agile methods at scale as one of the important reasons for employing an agile coach. A study by Amorim et al. [2021] guided by design science methodology aimed to eliminate some known challenges of COBIT 5 adoptions by providing a Scrum-based methodology. COBIT 5 is a framework for guidance in evaluating, directing, and monitoring an enterprise's use of IT. One of their findings showed that the use of an agile methodology by itself is not enough to reduce the resistance to change. Mantovani Fontana and Marczak [2020] conducted industrial surveys to discover challenges faced by public sector software organisations in the adoption of agile methodologies in Brazil. Based upon the results from 167 responses, the authors concluded that cultural change and resistance to change are the main challenges still faced by Brazilian government IT organisations.

Sometimes, the new methods and tools, the developers are not familiar with, are forced upon the developers by the senior management, and the developers are not entirely convinced about their use. However, in this case the use of BDD was proposed by the development team in order to improve communication and better understanding and management of requirements. As the scrum master (P4), while talking about the adoption of BDD at the company, said "... *if you're trying to get people to understand behaviour driven development... it's quite a foreign concept for them"*. The interview participants and some recent studies [Pereira et al., 2018, Barbosa, 2020, Smart, 2018] identified "*convincing people to use BDD"* as a challenge.

During the course of this study, we learned that BDD serves as a tool for communication for those who are actively involved in the project. In case of the lack of engagement from the customer, BDD becomes the responsibility of the development team which often results in poorly defined scenarios due to the lack of details in the requirements [Scandaroli et al., 2019, Barbosa, 2020]. BDD requires the customer to express the desired behaviour in a certain format which not only familiarises the customer with the workflow and purpose of BDD but creates a common point of interaction between the customer and development team.

When asked about *who was involved in discussing examples and writing scenarios?*, the

product owner (P3) answered "... *we already knew quite a lot about what the flow of the application was going to be... I think the development team, I'd say... so not really the customers"* According to the findings of Scandaroli et al. [2019], "*...when the adoption of BDD is mostly bound to the technical team, BDD scenarios can become too technical, which removes the benefits of proper understanding of features across all contexts (business and technical)"*. Similar challenges were faced by the interview participants who said that it was difficult to communicate the project-related information using the feature files. It was hard for the people who were not actively involved in the development of the project to understand and negotiate the requirements using feature files. Instead, the participants used hand-drawn whiteboard diagrams to communicate with the stakeholders who were not actively involved in the development.

During this project, the product owner acted as the customer. His role was limited to giving feedback on the development and clarification of BDD scenarios. We believe that one of the reasons behind the lack of participation was the size and the culture of the organisation. The product owner was involved with many teams and in many different projects at the same time. Although the organisation was transitioning towards agile, the long-standing waterfall culture and the segmentation between the departments were still reflected in the projects.

6.5.8 Risk of Duplication of Effort in Large-Scale Systems

In a large software project setting, where team members are too focused on their own tasks and have limited communication with other teams, there is a risk of duplication of effort. The team had believed that they needed to understand a lot about the whole system if they wanted to write a good set of Gherkin features; otherwise, it was very difficult to know if someone had already written the steps similar to what someone else was writing, in which case, a test already existed.

Being unaware of the dependencies between tasks, or not knowing if a certain task has already been completed by someone, can lead to the duplication of tasks. This in turn calls for refactoring and creates rework. As the product owner (P3) said "... *it feels like I need to understand a lot about the whole system if I want to write a good set of Gherkin because ... I don't want to have thousands of, or hundreds of thousands of tests... I want to have the right number of tests... cause you almost want to know how someone's written the steps similar to this, so you've already got a test"*.

BDD projects involve additional artefacts which already require additional effort. Not knowing if a task is completed and can be reused would mean duplication of tasks and effort. This duplication of task effort will be more than the duplication of tasks and effort in an identical non-BDD project. For example, there is a possibility of having duplicate scenarios and acceptance tests in addition to the duplication of effort in the code. Duplication of effort in BDD suites is also discovered as one of the challenges in a survey study on finding challenges of BDD by Binamungu et al. [2018b].

6.6 BDD in Theory vs BDD in Practice

According to Julian et al. [2019], there is very little literature available on *how agile process is used in practice?* as compared to the existing literature discussing *what is an agile process?*. This section presents the overview of the BDD workflow in theory vs *what happened in reality*. While defining the theoretical models and processes, the real-world factors that can limit the application of a process are often ignored. These factors, in the context of software systems, include the availability of the customer, organisational culture and structure, size of the project, etc. Ignoring these real life factors creates a disconnect between *what happens in reality*, and *how it is described in the literature*. In this section, we have discussed the practicality of the theoretical BDD workflow in light of our experience during this study.

During the study, we observed that the theoretical workflow of BDD [Smart, 2014, Wynne et al., 2017] is based upon certain assumptions. To explore these assumptions and bridge the gaps between BDD process and its application in real life, we have discussed the use of BDD in this study. We observed that there is a need for guidance on BDD activities, their outcomes, and how to handle real-life situations that can impede the speed of development e.g., the unavailability of the customer, and benefits of adopting certain activities or implications of ignoring them. After reviewing some of the studies on BDD [Binamungu et al., 2018b, North et al., 2006, Wynne et al., 2017, Rahman and Gao, 2015, Smart, 2014, Zampetti et al., 2020, de Souza et al., 2017, Storer and Bob, 2019] we summarised the BDD process discussed by Smart [2014] and Wynne et al. [2017] under three broad concerns: i.e., Understanding, Collaboration, and Acceptance Testing.

This categorisation of concerns by us was based upon the "*rationale behind using BDD*" discussed in the studies. For example, all the studies listed above emphasise that the main goal of BDD is to create a shared understanding of the system under construction. When we look at the activities in the BDD workflow (discussed in section 5.3) which help in creating this "*shared understanding*", we see that *determination of business goal* helps in creating a shared understanding. Similarly, collaborative practices like *three amigos meeting* in BDD process show that BDD process emphasises collaboration. We categorised activities like scenario writing and writing glue code under acceptance testing.

6.6.1 Understanding

In this sub-section, we review the application of BDD workflow activities that primarily focus on understanding the business, problem, requirements specification and the establishment of the logical connection between all the requirements. We present a summary of the activities of the theoretical workflow, and discuss how and if those activities were performed in our context.

BDD Process

One of the activities of BDD theoretical workflow, which is also considered a starting point in the BDD workflow, is the identification of a business goal [Wynne et al., 2017, Smart, 2014]. Determination of a business goal establishes the importance and the need for a project. It also helps in understanding the strategic benefits of a project to the business. This activity is identical to the primary activity of the theoretical model of Goal Driven Development (GDD) [Schnabel and Pizka, 2006, Park et al., 1996], and its purpose is to shift the focus from requirements to the business goal.

A business goal tends to be more stable than the requirements i.e., change in a broader business goal is less likely to take place than in a set of requirements [Schnabel and Pizka, 2006, Park et al., 1996]. According to Schnabel and Pizka [2006], requirements are not the best source of information for initial understanding of the system due to their volatile nature.

Determination of a business goal in BDD is followed by the definition of the major features and the determination of the relative value of each feature. The purpose of this activity in BDD is to define the major capabilities and their importance towards the achievement of the business goal. Smart [2014] describes this step as an activity where features are *"injected"* to determine how exactly a system is expected to deliver business value. *"This is what's called hunting the value. The aim is to understand the business value that lies behind a feature so that you can objectively decide which features are worth creating"* [Smart, 2014]. The outcome of this activity is a set of major requirements. The major requirements are then broken down into smaller requirements. These smaller requirements are documented along with their real-life examples called scenarios.

Although BDD is an agile way of development, analysis of the BDD activities discussed in Section 5.3 shows that the BDD process is a sequential process where successful execution of an activity is dependent upon the understanding and execution of the preceding activity. For example, BDD process recommends that the features should be defined after understanding the business goal [Wynne et al., 2017, Smart, 2014]. Similarly, the scenarios should be described after defining the corresponding features. It was, therefore, logical for us to review the activities of the workflow in this sequence.

In Practice

Contrary to the theoretical workflow, the starting point for the project at *the company* was the elicitation of requirements. As a participant *(P3)* explained "*..we did a few iterations with the prototype... that helped to settle ....the scope of the tool "*. We observed that the development of a project using BDD was not very different from the development of the other non-BDD projects at *the company*. The development process started from the elicitation of requirements, and team members seemed to be unaware of the underlying business goal. We believe this happened because the requirements specifications are thought to be the primary source of understanding a

software system in the industry [Krüger et al., 2018, Wiegers and Beatty, 2013]; therefore, the organisations see requirements elicitation as a starting point in any software project.

As the project was a novel product, the features were not clear in the beginning. One of the participants *(P3)* said "*...we found it quite difficult to get an appropriate level of specification"*. This means that it was difficult for the team to imagine all the scenarios in advance. While referring to the developer, a participant *(P3)* said "*...he didn't understand the vision, or what the product was meant to do"*. Many times, while writing the scenarios, either a feature was broken down, re-written or discarded. At times, a requirement did not become clear until it was implemented. In which case, the requirement was implemented before its scenarios and underlying tests were written. Even after being able to write a feature and the primary scenario in a reasonable form, the team struggled to define alternate scenarios for a feature. It was hard for them to imagine the alternate scenarios before the implementation of a feature.

Examination of the process of documentation of features and their associated scenarios shows that the process is based upon the assumption that the requirements of a project are known by the person writing the requirements. Without knowing *what is required*, it is difficult to imagine or document how the requirement will execute. The requirements for the project were elicited and documented at the user story workshop without determining the business goal and major features. The features were elicited and documented directly without performing the preliminary activities of the theoretical framework of BDD. According to Smart [2014], the determination of the business goal and major features that will satisfy the business goal, help in the elicitation of requirements. Elicitation of the major features is followed by refinement of the major features. Next, the resulting features are documented along with their associated scenarios. BDD process seems to build a conceptual background of the project requirements before the elicitation of features and their associated scenarios.

We believe that the lack of complete understanding of the requirements, which reflected in the team's struggle to document the features and associated scenarios, was due to the lack of requirements refinement and elicitation activities. However, we cannot say anything conclusive about the effectiveness of the initial activities of the BDD process. We believe that more research is needed to determine the importance of these activities.

6.6.2 Collaboration

In this section, we have looked at how different BDD-related meetings took place during the project, to understand the limitations and impediments of real life.

BDD Process

The BDD workflow recommends collaborative practices like the "*three amigos meeting"* for all BDD activities including elicitation of requirements and acceptance criteria [Smart, 2014]. The

three amigos meeting is a BDD activity where three people (i.e., business analyst, developer, and the tester) meet to discuss how the product will be developed [Wang and Wagner, 2018, Northwood, 2018, Kudryashov, 2015]. The ***business analyst or the product owner*** describes the problem that must be solved; the ***developer*** discusses how the solution will be implemented; and the ***tester*** discusses the criteria for testing the product. The purpose of involving all the stakeholders is to get the requirements right and bring everyone on the same page.

Looking at a feature from three different angles adds clarity and helps in establishing a clear understanding and defining the acceptance criteria. This meeting aims to produce examples (called scenarios) that define the acceptance criteria and the testing strategy. These examples are converted into executable specifications by writing a piece of code for each step of every scenario. The associated code for a scenario demonstrates if a scenario is doing what it was supposed to do [Smart, 2014, Wynne et al., 2017]. Ideally, all the stakeholders, including the customer, should participate in the feature elicitation and scenario writing, but *does the participation of all the stakeholders guarantee productivity?*.

In Practice

The BDD workflow presumes a team of permanent members who are involved from the very start of the project and are familiar with the features of the project under development. We believe that the activities of the theoretical workflow are based upon certain assumptions i.e., *(i)* everyone is involved and available, *(ii)* everyone is experienced in writing BDD scenarios and *(iii)* BDD workflow does not take the time constraints of a project into consideration.

Real-world constraints, such as unavailability of team members and customers, development of a novel project, and situations where a new team member joins the team, are not taken into consideration by the BDD workflow. We have experienced a similar situation during the project discussed in this study. The features were elicited by the product owner, potential users, and the scrum master in a user story workshop, which were then handed over to the developer for scenario writing who joined the team after the features were elicited. The developer was not part of the requirements elicitation process which reflected in his struggle to understand the requirements.

Also, there were no specific roles in a team in *the company* i.e., there was no separate testing or quality assurance team. Everyone in the team was expected to be able to run a complete lifecycle. Too much emphasis on *who should participate?* was probably one of the reasons for not being able to do the *three amigos meeting*. One can argue that the existence of different "*roles*" in BDD does not mean that different people should play these roles; the same person can wear different "*hats*", thus playing different roles along the project.

This assumption could be rejected by looking at the very description of the three amigos meeting by Smart [2014] which says in three amigos meeting "... *three team-members - a developer, a tester, and a business analyst or product owner - get together to discuss a feature*

and draw up the examples". This clearly shows that this meeting is between three individuals having three specific roles.

We observed that the theoretical workflow of BDD is for a specific project setting where all the stakeholders are involved in the development process, and there are specific roles, such as tester, developer, and business analyst. The real-life impediments are not taken into account which makes the implementation of the theoretical framework of BDD unfeasible in the project settings that conflict with the underlying assumptions of activities of the theoretical framework. It is important to mention that the project was dealt with as a part of the main project, which means the processes and practices used for the large project encompassed the project discussed in this study. There were no separate processes for the sub-project.

6.6.3 Acceptance Testing

This section reviews the acceptance testing activity in the project discussed in this study.

BDD Process

An acceptance test is a description of the behaviour of a software product in a specific context. Acceptance test is usually described as an example or a usage scenario while acceptance testing is a process of verifying the execution of that example. Miller and Collins [2001], while discussing the importance of acceptance testing, refer to the acceptance test as a contract between a developer and the customer. The authors further argue that an acceptance test captures a requirement in a directly verifiable way where successful execution of the acceptance test means that there has been no breach of contract.

BDD incorporates acceptance testing and takes it one step further by associating specifications with acceptance tests. A piece of code is written for each step in a scenario which tests the behaviour of the system described in the respective step. We can say that there are two main components in BDD acceptance criteria i.e., *(i)* scenario and *(ii)* acceptance test. A scenario is an example of the acceptance criteria for a requirement, and the associated code is the acceptance test for a scenario.

BDD process recommends writing acceptance tests for specifications before implementation [Smart, 2014]. However, the same recommendation is made by Beck [2003] as a practitioner of Test Driven Development (TDD). Therefore, we can say that the BDD practice of "*writing test before code"* is inherited from TDD. The purpose of this practice is to eliminate the waste of effort by implementing only what is specified and to have a clear understanding of the requirement and its acceptance criteria before implementing the requirement. One of the assumptions behind "*writing acceptance test for the specification before implementation"* is that requirements will become clear while writing the acceptance criteria.

In Practice

Writing the acceptance criteria for a requirement, its acceptance test, and then implementing the code seems like a simple and logical flow of activities; however, a requirement is not always clear until the implementation starts. The same happened in our context.

The project was new, and the developer and the users were not able to imagine the acceptance criteria in advance. As the developer *(P2)* said "*....it was... being kind of developed experimentally in a way. I don't think they really knew that this kind of thing was being developed"*. Many times, the requirements became clear only after the developer had implemented them and got feedback from the users of the system. In which case, the developer had to re-write the scenarios and the acceptance criteria after making changes in the implementation.

This seemed like an overhead, and the team's desire to get the scenarios and the acceptance test right was taking too much of the implementation time. So, the pressure to get the product ready shifted the focus from BDD to development. This shift in focus made the pace of maintenance of the feature files slower than the development. To discuss this further, we need to look at Test First Development (TDD) since BDD's test-first approach is inherited from TDD. Unfortunately, we were unable to find published research on measuring the practicality of test first approach in TDD. However, we were able to find a blog [Vlad, 2019] which discusses *trial and error* situation in TDD i.e., when it is difficult to know how something should work and the problems one might face. In such situations, TDD can initially lead to several rounds of productive failure. *Trail and error* situations can be more difficult in BDD because, unlike TDD, scenarios are written for whole features behaviours rather than individual function behaviours.

The team found prototyping to be more effective than BDD because of the lack of clarity in the requirements. The assumption that the requirements should always become clear while writing the acceptance criteria proved wrong in our context. The requirements were vague which produced vague acceptance criteria which, consequently, led to writing acceptance tests that were based upon assumptions about the system. Most of the requirements did not become clear until they reached the implementation, which naturally made the team rely on manual testing and feedback on the implementation from the users. However, automated testing is less time-consuming than manual testing [Musliu and Jashari, 2021].

6.7 Threats to Validity

This section discusses the threats to the validity of this exploratory study. In this section, we have discussed three types of validity threats i.e., construct, reliability, and external validity.

Construct validity threat

The construct validity threats for this exploratory study on challenges related to the use of BDD may include: *effectiveness of qualitative measurement of the results during action research* and *effectiveness of interviews to capture the results*.

Four cycles of action research were conducted during each of which a few observations were made of a qualitative nature. It is possible that some important observations went unnoticed. It should be noted that we have presented the challenges described in this study from the development team's perspective who attempted to use BDD for the first time in the early phase of a small sub-project. Therefore, we acknowledge that some of the challenges may not have been discovered. Also, since there was no guidance available on the conversion of use cases into BDD scenarios, it is possible that the strategy devised by us for the conversion of the given use cases into BDD scenarios was erroneous and use cases cannot be directly translated into BDD features.

There is another construct validity threat called the Hawthorne effect [Sedgwick and Greenwood, 2015] that relates to action research. This threat refers to the potential bias that arises when individuals modify their behaviour due to the awareness of being observed. This can lead to an inaccurate understanding of the phenomenon under investigation.

Various interview sessions were conducted with the team members of the project discussed in this study. It is possible that the interview instrument used during this study did not cover all aspects of the purpose of this study. Also, the project and the team may not adequately represent the diverse range of industrial domains, or systems in which BDD is applied. This can limit the validity of the construct. The challenges studied and investigated during the study may also introduce bias toward certain perspectives on the use of BDD. This bias could limit the range of challenges considered or lead to an over-representation of opinions.

Reliability Validity Threat

Inaccurate or incomplete data extraction from the collected data can compromise the reliability of the results. One such threat is observer bias. Observer bias occurs when the researchers or observers have preconceived biases that may influence their observations or interpretations of the data. Another reliability validity threat may be the interviewer bias. This occurs when the interviewer's personal beliefs and opinions influence the interviewing process.

External Validity Threat:

Lack of experience in the development of applying BDD may have an impact on what the team considers as a *challenge*. The specific context and setting in which the study took place may have unique characteristics that may have limited the generalisability of the findings. For example, the study was focused on a small project with a small team who were inexperienced in BDD.

The results may not apply to other organisations settings such as a large-scale project and a large team experienced in the use of BDD.

Addressing Threats to Validity

To address these threats, we took several steps. For example, constant communication with the team members including face-to-face meetings, telephonic conversations, email exchanges, and sharing of documents enhanced the expressiveness of the team members. They were able to discuss the issues in applying BDD openly. With the use of action research as a research method, we were able to study the challenges faced by the team as they progressed in the project along with the improvements in practical application of BDD. In addition, we introduced confidentiality by keeping the views of each individual confidential during the lifetime of the project.

Before attempting to translate the use cases into BDD scenarios, we studied the structural concept behind use cases. A use case describes a specific sequence of of events in a given context which is also a characteristic of BDD scenarios. Various fields in the use case template were mapped to Given, When, Thens in the BDD scenarios after carefully reviewing each field in the use case template. However, to the best of our knowledge, the strategy devised by us for translating use cases into BDD scenarios was attempted the first time and the validity of this strategy is unknown.

Interviews were conducted as a secondary source of data to validate the findings and further explore the phenomenon. By employing a secondary source of data and cross-referencing the results, it became possible to develop a more comprehensive understanding of the phenomenon, reducing the influence of the Hawthorne effect. To mitigate the interviewer bias, open-ended questions were asked during the interviews to enable the team members to express their views in detail which helped us in discovering unidentified aspects. In addition, we followed Wengraf [2001]'s guidelines to devise the interview questions and perform the analysis of the data from the interviews. This helped us in the systematic extraction of the data and production of results.

At the start of the project, the team was introduced to the concept of BDD and was given the time to learn BDD and the tools used for it. The project was a part of a larger project internally developed in a large avionics company which is one of the top 10 avionic companies in the UK according to Google. The participants had various levels of industrial experience. However, we acknowledge that it was beyond our control to conduct a study with an additional team to compare the results. We also acknowledge that BDD was applied in a small project with a small team therefore enabling the readers to assess the similarities and differences with their own context.

6.8 Summary

This chapter describes the use of action research for the incorporation of BDD into a project in an avionics company. The action research identified some technical limitations of Gherkin. It is important to note that the findings reported in this chapter are from the perspective of the team. The collection of practices, technology, culture, and mindset manifest these challenges. The findings documented in this chapter as challenges and limitations manifest themselves when BDD is "*practiced*" within this case study.

The action research also revealed that the use of BDD is helpful in assessing the validity and consistency of the requirements. The chapter also presents the results from the post hoc semi-structured interviews with the development team using BDD. The interviews helped in drawing the context of the project. The interviews identified challenges in the application of BDD.

Four out of eight limitations of Gherkin in Figure 6.6 refer to the limitations of Gherkin. According to the project team's feedback from practicing BDD, challenge number 3, 4, 5, and 6 show that Gherkin is unable to express various ways of user interaction with a system and that Gherkin is not very well suited for describing the rich variety of ways that users interact with systems.

This chapter presents an analysis of the practicality of the theoretical steps of BDD. A comparative analysis between BDD in theory vs BDD in practice revealed that the theoretical framework is based upon ideal circumstances and does not consider the real-world factors (e.g., the experience of the development team). At the moment, there is no guidance available on how to take the real-world factors into account. Many activities of the theoretical workflow of BDD are based upon assumptions. These assumptions include: availability, participation, and experience of the stakeholders. Whereas, BDD process does not take real-life circumstances into consideration, such as vague requirements, unavailability of the customer, organisational structure, and the type of the product. The time taken to overcome these impediments is also not considered by the theoretical workflow. There is a need for guidance and training on BDD. Unless the people understand the purpose of using BDD, and have practical knowledge and guidance on how to use BDD for communication and development, it will be seen merely as a way of writing requirements.

Because of the lack of quality criteria, it is easy to not know when to stop refining scenarios, especially, when the requirements are not clear. Writing the acceptance tests is a time-consuming activity, and many times, it is hard to imagine the scenarios in advance in which case, the attempts to get the scenarios right consume a lot of development time. Also, BDD does not (at the moment) specify quality criteria for the scenarios which means it is hard to know if a scenario is good enough to form the basis for the acceptance test for a requirement.

Although BDD is an agile practice, and must not suppose to be used in isolation as a fully-fledged development process, the progress and the quality of artefacts in BDD is dependent upon the clarity in the requirements. Also, BDD is not a technique for requirements elicitation. It has

indeed to be used in conjunction with other practices (and processes). Various activities in BDD process have different purposes. The requirement elicitation activity in BDD is a *three amigos meeting*. During this study, we observed that although BDD is primarily a development method, and the requirements elicitation in BDD is weak. It has to be used in conjunction with the other requirements elicitation methods such as prototyping.

Upon observation during the action research, we detected that the content of the user story block of a Gherkin feature is only provided for documentation purposes and is not leveraged when executing the test suite. Only the scenarios that describe the acceptance criteria play a role in leveraging test execution.

Chapter 7

An Analysis of the Practice of BDD on GitHub

In Chapter 6, we explored the application of Behaviour Driven Development (BDD) in a large-scale organisation and discussed the lessons learned from it. The focus of the previous chapter was on elaborating the challenges faced during the application of BDD in a single commercial project. During the previous study, we learned that, in BDD, developers sometimes transcribe the requirements in ways that introduce bad smells in the requirements (i.e., make the requirements inflexible). These requirements could be functionally correct but they negatively impact the evolution of a system. In this chapter, we extend the research by conducting an empirical analysis of BDD in practice in open-source projects.

This chapter presents an overview of open-source projects that contain BDD artefacts. To do this, a sample of projects that contain BDD artefacts were identified on the GitHub collaboration platform. The contents and meta-data of these projects was explored in order to characterise BDD-containing open-source projects. In addition, the evolution of BDD-containing projects was studied. Finally, a second sample of non-BDD-containing projects was obtained to compare between BDD and non-BDD-containing projects.

To gather the described samples, a GitHub repository sampling tool was implemented to query the GitHub API. Exclusion/ inclusion criteria were defined and implemented in the tool to filter sampled repositories. The tool then gathered relevant meta-data from each project before cloning to a local filesystem. Once cloned, commits on the mainline of each project were checked out and inspected using a number of automated metrics.

The next section provides a more detailed overview of the objectives of this chapter. Section 7.2 explains the experiment design and provides a description of the steps followed during the experiment. The results from the experiment are discussed in Section 7.3 which includes an overview of Gherkin projects, a discussion on the relationship between BDD-related project artefacts, a comparison between Gherkin and non-Gherkin projects, and a discussion on the evolution of BDD artefacts. The section compares the meta-data fields and the contents of

Gherkin and non-Gherkin projects to show similarities and differences between them. Section 7.4 consists of the discussion based upon the results of this study, whereas Section 7.5 discusses the threats to the validity of this study. Section 7.6 provides a bird's eye view of the BDD in open-source projects and a summary of this chapter.

7.1 Objectives of the Experiment

Scope of the existing studies [de Souza et al., 2017, Solis and Wang, 2011, Gómez, 2018, Binamungu et al., 2018b, 2020, Egbreghts, 2017, Zaeske et al., 2021, Oliveira and Marczak, 2018, Irshad et al., 2021] on BDD in practice is limited to a certain organisation or the opinions of the practitioners. We have extended the scope of the study to the open-source BDD projects to create a general overview of the *current state of BDD and its adoption*. Looking at the *bigger picture* helped us in understanding the overall adoption and growth of BDD in practice in open-source projects. Lessons learned from this chapter represent a broader trend in the adoption, growth and evolution of BDD and its artefacts.

The objectives of this chapter are:

- Present an overview of BDD related artefacts and study their evolution over project life time.
- Compare open-source BDD projects with non-BDD projects and understand the differences between them.

For this purpose, the open-source BDD repositories on GitHub were investigated. GitHub has over 37 million repositories and has become one of the major sources of software artefacts on the internet [Kalliamvakou et al., 2016, Chong and Lee, 2018, Ortu et al., 2018]. The projects' data available on GitHub gives researchers and data miners an opportunity to explore *what is being done in practice?* [Kalliamvakou et al., 2016, Chong and Lee, 2018]. Researchers mine GitHub repositories to retrieve the data regarding different matters of interest [Kalliamvakou et al., 2016, Chong and Lee, 2018, Ortu et al., 2018]. GitHub offers a number of means (e.g., GitHub API) through which one can fetch the information someone is interested in.

This chapter is based upon results and statistics from studying the meta-data drawn from the open-source BDD projects on GitHub. The meta-data was programmatically extracted at every commit from a random sample of BDD projects. It not only helped in studying the growth of BDD-related artefacts but also facilitated in drawing the correlation between different project artefacts.

7.2 Experiment Design

The focus of the experiment was to study the *real* projects. The experiment design was replicated from previously published studies [Bao et al., 2019, Sharma et al., 2017, Ortu et al., 2018] which used the meta-data from repositories on GitHub to investigate different phenomena. The steps that were common in their experiment design were replicated i.e., *(i)* establishment of a primary selection criteria; *(ii)* devising a criteria for filtering the repositories; and *(iii)* collection of the results.

The primary selection criteria for the projects in this experiment was the *use of Gherkin language in a project*. According to GitHub documentation*, GitHub provides information on the languages used in a repository. GitHub uses the open-source Linguist library to determine the languages used in a file. BDD specifications are written in Gherkin therefore, presence of Gherkin among the languages used in a project indicate the incorporation of BDD. Repositories where Gherkin was listed as one of the development languages, were considered for selection.

The next step was to devise a filtration criteria for the repositories. Assignments, practice or dummy projects could skew our results. So, we used the guidance provided in the literature [Munaiah et al., 2017, Kalliamvakou et al., 2016, Borle et al., 2018, Leotta et al., 2019, Roehm et al., 2019] to devise an exclusion/ inclusion criteria for removing the dummy projects from our dataset.

7.2.1 Definition of exclusion / inclusion criteria:

Kalliamvakou et al. [2016] conducted a study that was aimed at understanding the potential perils when mining GitHub for research purposes. Although many repositories are being actively used for development on GitHub, according to Kalliamvakou et al. [2016], most of them are simply personal or inactive repositories. According to Munaiah et al. [2017] the proportion of noise (homework assignments etc) in a random sample of repositories could skew the results and may lead to inaccurate conclusions. In order to remove noise in our data, we used the guidance provided in the literature [Munaiah et al., 2017, Kalliamvakou et al., 2016, Borle et al., 2018, Leotta et al., 2019, Roehm et al., 2019] to devise the following exclusion/ inclusion criteria for the selection of GitHub repositories.

1. **Evidence of Sustained evolution: Remove the repositories with less than six commits.** Less number of commits show a lack of evidence of sustained evolution which could mean either a project was a dummy repository (i.e., homework, assignment, etc.) or the project was developed somewhere else and later put on GitHub for the purpose of storing it. Findings of the study by Kalliamvakou et al. [2016] show that 90% of projects on GitHub have less than 50 commits. According to the authors [Kalliamvakou et al., 2016],

*https://docs.github.com/en/github/creating-cloning-and-archiving-repositories/about-repository-languages

the median of the number of commits on GitHub is 06. We, therefore, excluded the repositories with less than six commits.

2. **Evidence of Collaboration: Remove repositories with only one or no contributor.** Real projects show signs of collaboration according to the studies [Kalliamvakou et al., 2016, Leotta et al., 2019, Vendome et al., 2017] on mining GitHub repositories. The projects having only one contributor are considered personal projects [Kalliamvakou et al., 2016]. Whereas, "*...development of a software system involving more than one developer can be considered as an instance of collaborative software engineering"* [Munaiah et al., 2017]. Since we were interested in real projects, we excluded the projects with zero or only one contributor.

3. **Evidence of interest from the community: Include repositories with at least one watcher OR at least one fork.** According to Leotta et al. [2019] 71.6% of the projects on Github are personal projects, dummy repositories, or assignments. One of the criteria for identifying a real project is the popularity of the project i.e., the interest of the people in the project from outside the development team [Kalliamvakou et al., 2016, Leotta et al., 2019, Vendome et al., 2017]. We considered two indicators of popularity for a repository mentioned in [Munaiah et al., 2017, Kalliamvakou et al., 2016, Borle et al., 2018, Leotta et al., 2019, Roehm et al., 2019, Allamanis and Sutton, 2013] i.e. *(i)* watchers and *(ii)* fork.

   A repository with at least one star or watcher shows there is at least one user, other than the developer, who has shown appreciation for the repository [Leotta et al., 2019, Vendome et al., 2017]. Whereas Fork is also considered an indicator of popularity and interest in a repository by Allamanis and Sutton [2013]. We consider evidence of interest from the community as inclusion criteria for the repositories.

4. **Fork Repositories:** As mentioned in some of the studies [Vendome et al., 2017, Kalliamvakou et al., 2016] we excluded forks to avoid over-representation of data. Fork repositories were replaced with their parents.

5. **Minimum Size of Code Base: greater than 100,000 bytes in the latest commit:** This criterion is used to exclude small repositories which might bias the results and are likely to be dummy repositories according to researchers [Roehm et al., 2019, Vendome et al., 2017, Leotta et al., 2019].

6. **Repositories with no Gherkin files:** After applying the above exclusion/inclusion criteria, we found that some of the resultant repositories did not have any Gherkin files. When we investigated such repositories, we found that Gherkin was incorporated in a branch and never in the master. We excluded such repositories.

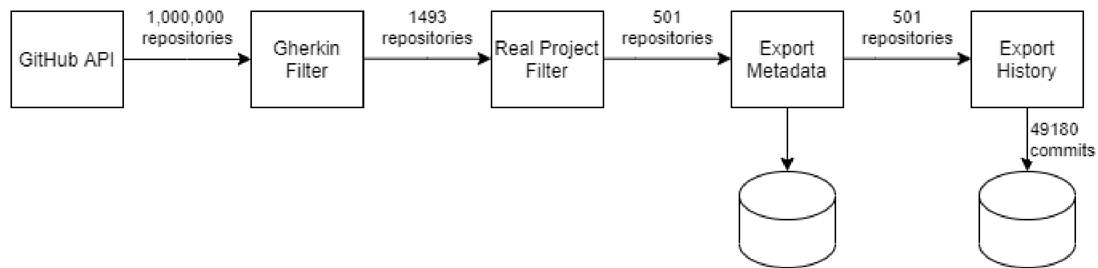


Figure 7.1: Pipeline diagram

7. **Repositories with no features:** Some of the resultant repositories did not have any features although there were few Gherkin files in them. The *.feature* files in these repositories contained random data and not features. Such repositories were excluded.

During the development of exclusion/ inclusion criteria for the GitHub repositories, we observed that the application of the criteria was not a straightforward process. For example, there were large repositories with no commit history and no watchers but the repository itself was large enough to be noticeable. Therefore, we developed a data-gathering workflow by using the devised exclusion/ inclusion criteria. We implemented the exclusion/ inclusion criteria in Python and developed a tool that gathered the meta-data through GitHub API.

For collection of results, a tool was developed in Python which implemented the primary selection criteria along with the exclusion/ inclusion criteria and collected the meta-data automatically. The tool used a five stage process. Figure 7.1 illustrates the five stages process of the tool. At first, the tool generated 1 million random GitHub repository identifiers. The repositories which matched the identifiers were marked as a match. At Stage 2, the tool filtered out the non-Gherkin projects by selecting the projects which listed Gherkin as one of the development languages. Stage 3 implemented the exclusion/ inclusion criteria for removal on the dummy projects. In Stage 4, the tool exported the meta-data from the last commit of the finally selected projects. In Stage 5, the tool generated the meta-data for each commit of all the finally selected projects to study the evolution of the project artefacts.

The execution of the five-stage process was completed in the first week of September 2020 and the extracted meta-data was stored in a *.csv* file format. The analysis was performed using a Python program that retrieved the data from the *.csv* files using SQLite[†] (i.e., a database management system) and generated graphs using Matplotlib[‡]. The graphs gave us an understanding of the growth and relationships between different project artefacts. The following sub-sections describe our experiment design in detail.

[†]https://sqlitebrowser.org
[‡]https://matplotlib.org/

7.2.2 Repository Data Set Preparation

A GitHub repository sampling and analysis tool was developed in order to retrieve a representative sample of projects containing Gherkin. For sampling, the pipeline tool was designed to use the GitHub-Python API client. A wrapper was also developed for the API to implement automated rate limiting to ensure that the tool complied with the GitHub API's terms of service.

The sampling and analysis pipeline is illustrated in Figure 7.1. In Stage 1, by applying simple random sampling technique [Thompson, 2012], a random sample of 1,000,000 GitHub repository identifiers (IDs) was generated in the range 0 to 251,108,600 (the approximate maximum number of repositories hosted on GitHub at the time of retrieval). Each sampled repository ID was then assessed using information retrieved from the GitHub API. First, the API was used to check whether a repository with the respective ID existed. If a repository was found, the API was queried for the set of languages that had been detected in the repository. Repositories found to contain Gherkin were initially marked as a match. The repository IDs that we could not access could have been deleted or private.

In Stage 2, all repositories found in Stage 1 were filtered, following the rationale proposed in Section 7.2.1. First, we replaced any forked repositories with their ultimate parents. Repositories were then filtered to ensure that they demonstrated evidence of sustained evolution and collaboration (by having at least six commits and more than one contributor). Then the repositories were filtered to ensure that they demonstrated evidence of interest from the community (by having at least one fork or at least one watcher). The repositories considered for selection either demonstrated evidence of: sustained evolution, collaboration, and interest from the community *or* by being a sizeable deposition of code (more than 100,000 bytes in the latest commit).

As an additional step, where the parent repository was found to not contain Gherkin, the nearest child to the parent (breadth-first search) that contained Gherkin was taken. These forked repositories were analysed separately and their parents were removed from the main data set. The reason for doing this was that we wanted to study *"primary"* repositories, not forks. However, the removal of forks would result in a small dataset. Searching for the primary repository allowed us to place a substitute for the fork. In some cases, we found parents of forks that did not contain gherkin. We used a breadth-first search of children to find the repository nearest to the parent that introduced Gherkin. We retained the parents as it gave us an opportunity to study gherkin getting introduced. To compare our sample of Gherkin repositories with the repositories on GitHub generally, a second sample of 10000 identifiers was generated and used to search the GitHub API as before, except that the repositories were required to not contain Gherkin files.

Meta-data was gathered for all repositories retrieved, based on the latest commit found in the main branch of the repository as reported by the GitHub API. The fields recorded captured information about the code base, contributors, issue tracker, pull requests, and Gherkin files, etc. In addition, each commit on the main branch of the repository was analysed to extract information about the practice of behaviour driven development in the project over time. In

particular, the number of feature files, features, scenarios, backgrounds, steps, and step types were recorded for each commit. The main branch was identified from the GitHub API or, if this failed, a number of common names (e.g., master) were tested. As well as providing an evolutionary view of feature changes in the projects, the data from the first and latest commits in the repository were added to the summary information for each project. The data gathered for the study was collated into a data set and is available for inspection and reuse `https://bit.ly/36wIQ5I`.

7.3 Results

This section is based upon the results of the analysis on a random sample of Gherkin and non-Gherkin projects. The section presents an overview of BDD in practice in open-source projects on GitHub, and provides an understanding of the size, duration, growth, and evolution of Gherkin projects and their artefacts. This section also presents a comparison of open-source Gherkin and non-Gherkin projects to show the similarities and differences between them.

The analysis in this section is performed on the 493 Gherkin projects which were randomly selected as a result of the experiment described in Section 7.2. However, some of the graphs in this section are created from various subsets of the 493 projects because not all projects fit the characteristics of all the graphs we were looking to create. For example, Figure 7.3b presents the average step count in the projects. One project was excluded from the analysis because the excluded project had no scenarios and division by zero is undefined. Similarly, Figure 7.10 illustrates the change in the practice of introducing the first Gherkin feature in a project. In order to plot this graph, it was important that we considered projects with reasonable age and commit history.

7.3.1 Prevalence of BDD on Github

Our initial step was to generate 1000,000 random identifiers. Testing these against the API resulted in a set of 438,975 repositories, i.e. 44% of our randomly generated identifiers matched repositories accessible on GitHub. Of these, 1493 (0.34%) were found to contain Gherkin. Applying Stage 2 *real project* filtering to this data set yielded 596 parent repositories, a reduction of 60%. Of these, 51 were discovered to not contain feature files and a fork was collected and curated in a separate data set. A further 44 projects were discovered to not in fact contain any parseable Gherkin feature files in their latest commits. In most cases, these projects contained file paths to text files with containing the sequence *feature* that appears to have caused the GitHub language identification heuristic to mis-classify them. Overall, 501 projects were found to contain at least one Gherkin feature file or 0.11% of the raw sample of valid repositories. While we were running the analysis, eight repositories became inaccessible making the final

number of repositories 493. Those eight repositories were either removed from GitHub or were made private by their owner(s).

Considering the non-Gherkin sample, 4903 randomly generated IDs matched repositories. Applying the same Stage 2 filter to our sample of 10000 non-Gherkin repositories yielded 2069 results, a comparable reduction of 58%. This suggests that repositories that contain Gherkin are as likely to pass the conditions of the filter.

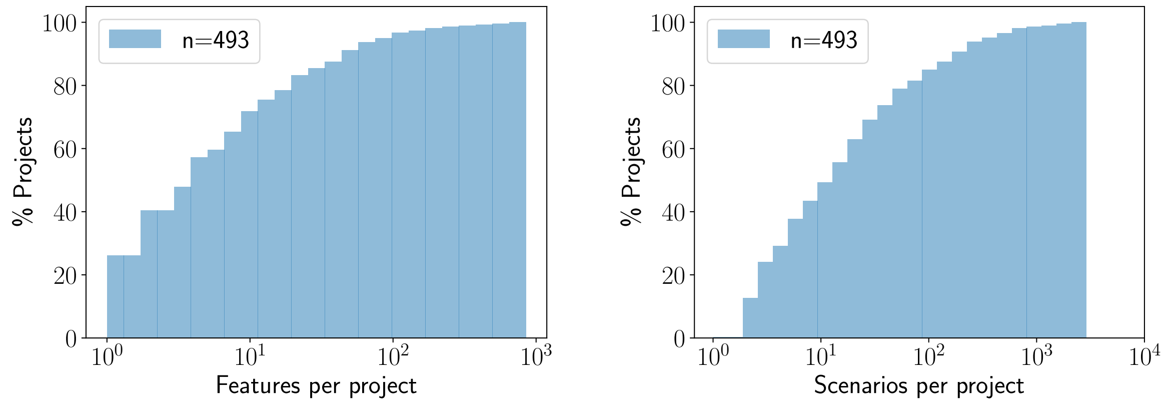
The sampling process allows us to estimate the popularity of the practice of Behaviour Driven Development in publicly accessible repositories through the maintenance of feature files in open-source software projects hosted on GitHub. As of January 2020, there are reported to be approximately 200 million repositories hosted on GitHub by June 2021[§]. Of these, our analysis suggests that approximately 87.79 million are accessible repositories. The rest of the repositories are either private or removed from GitHub. Out of the 87.79 million accessible repositories, 40% (i.e., 35.11 milion) exhibit the properties of a *real* projects according to our filter conditions. Our sampling process suggests that the percentage of the repositories containing Gherkin is 0.34%. Extrapolating from our sample, we estimate that there are approximately 119401 real projects employing Gherkin files on GitHub.

The initial sampling suggests that projects that contain Gherkin (and thus practice behaviour driven development) are rare on GitHub. This contrasts sharply with the findings of the study by Zampetti et al. [2020], who concluded that 27% of repositories in their study use BDD frameworks. Zampetti et al. [2020] took a sample of the top 50,000 open-source projects - ranked in terms of number of stars - written in the five most popular programming languages on GitHub i.e., Java, Javascript, PHP, Python, and Ruby. Also, according to the documentation for Cucumber [Aurlane, 2019], the most popular BDD frameworks are the ones developed for Java, Javascript, PHP, Python, and Ruby. Therefore, we can say that the sample in the study [Zampetti et al., 2020] was artificially boosted by selecting projects implemented in the languages believed to be already associated with popular BDD frameworks and the search for the projects was limited to popular (and thus more mature) projects. A relatively higher percentage of projects showing the use of BDD in the study [Zampetti et al., 2020] also implies that BDD is more popular in mature open-source projects. However, further investigation must be conducted to confirm this inference. The next sections characterise the data set of Gherkin projects in more detail.

7.3.2 Characterisation of Gherkin Projects

This section presents an overview of the Gherkin projects selected for this study. It draws on the analysis of the meta-data from the selected projects, and the graphs shown in this section describe the relationship between different elements of Gherkin specifications.

[§] `https://github.com/about`



(a) Project feature count distribution

(b) Project scenario count distribution

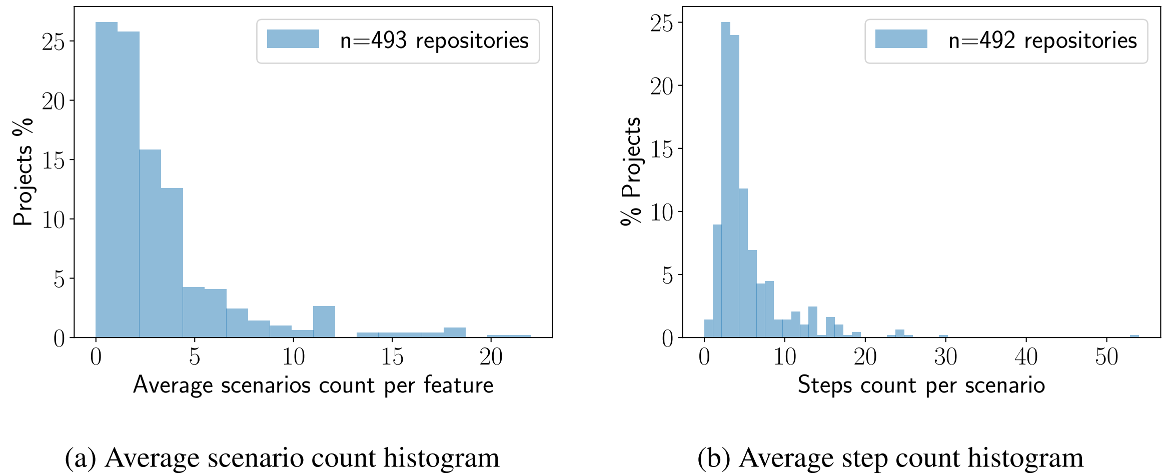
Figure 7.2: Cumulative histograms of feature and scenario counts for 493 repositories

Features and Scenarios

Figures 7.2a and 7.2b are cumulative histograms that illustrate the number of features and scenarios in the projects in the Gherkin data set. - The figures show that 90% of projects comprise less than 100 features and 80% comprise less than 100 scenarios, respectively. The data set does contain a small number of outliers, with projects at the extreme containing approximately more than 800 features and several thousand scenarios. These results may be contrasted with the survey completed by [Binamungu et al., 2018b]. The majority of practitioners (59%) in that study reported that the projects they worked on comprised more than 100 scenarios. The reason for this contradiction could be the difference in the research method. Binamungu et al. [2018b] surveyed the advocates of BDD which could be estimated as a potentially biased sample, whereas we analysed a random sample of open-source BDD projects. To investigate this further, we plotted histograms of average number of scenarios per feature and the average number of step count per scenario.

Figure 7.3a and 7.3b are histograms that illustrate the average number of scenarios per feature and average number of steps per scenario in the Gherkin projects. The projects in the data set were found to have a median average number of scenarios per feature of just 2 (StDev=3.27). In addition, Figure 7.3a shows that for more than 60% of projects, the average number of scenarios in the feature was less than 5. Further, the median average steps for a scenario was 4 (StDev=4.54) and the average scenario length for more than 90% of projects was 10 or fewer steps. Thus we can characterise a median Gherkin feature suite on GitHub as comprising 5 features, each of 2 scenarios, with each scenario comprising 5 steps. Several outliers were noticeable in the data set, comprising of very large feature suites of approximately 800 features and several thousand scenarios. However, these projects appear to be exceptional, with the majority of the projects comprising small feature suites.

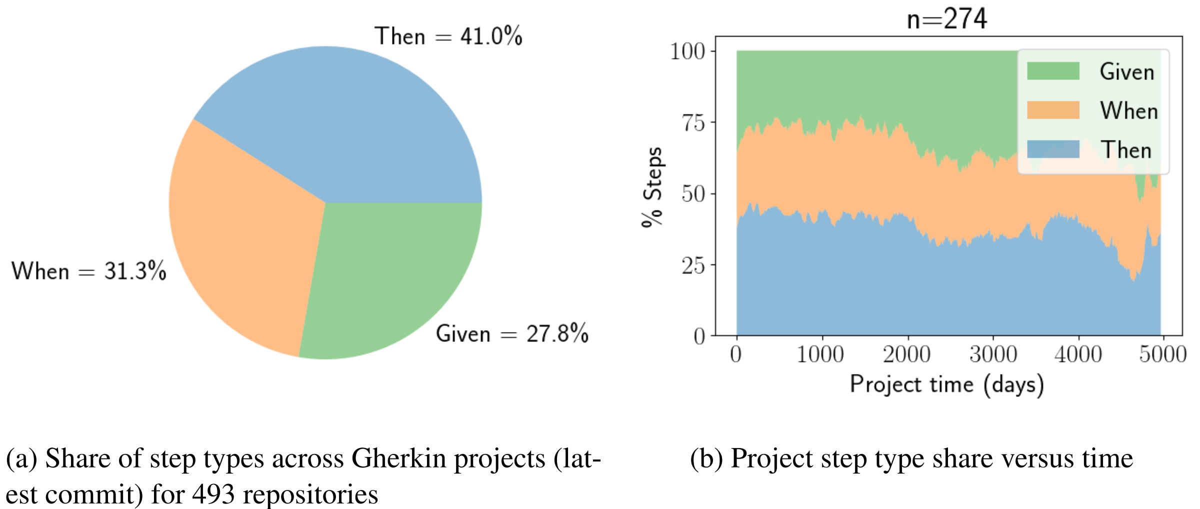
Next, we characterise the composition of scenarios. The analysis showed that major portion



(a) Average scenario count histogram

(b) Average step count histogram

Figure 7.3: Average scenario and step count

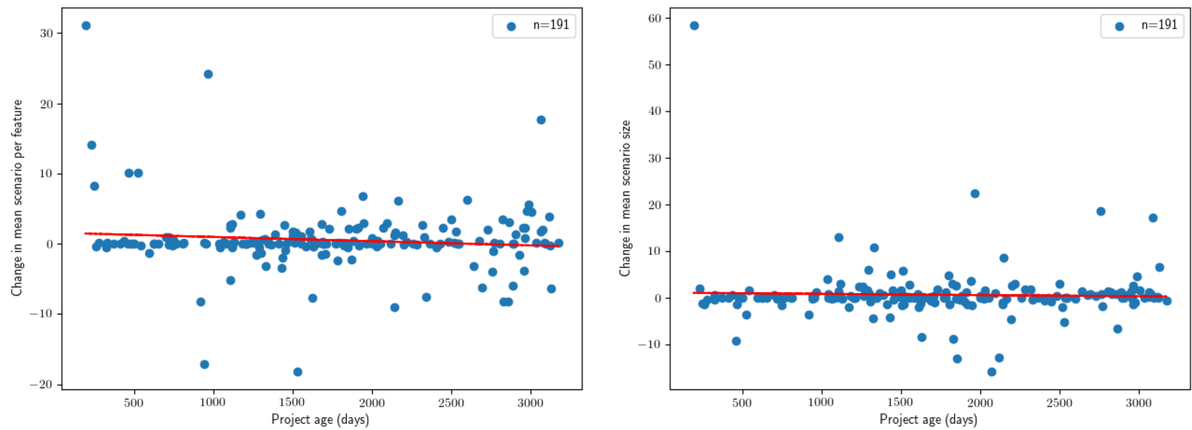


(a) Share of step types across Gherkin projects (latest commit) for 493 repositories

(b) Project step type share versus time

Figure 7.4: Project Given-When-Then share

of the total number of Gherkin steps consisted of assertions. To understand this, we plotted a pie chart of scenario elements and a stack plot that illustrates the proportion of Given-When-Then over time. Figure 7.4a is a pie chart of the scenario elements (i.e. Given, When, Then) using. The total number of Gherkin steps in the selected Gherkin projects was 350774. The figure shows that 40.99% of the total step count of the selected projects consisted of *Then* steps, whereas Given and When steps were 27.8% and 31.3% respectively. This proportion of steps type appears to be roughly consistent over time. Figure 7.4b is a stack-plot of scenario step types over time. The figure shows a smoothed share of step types over time for all projects in the data set. The figure shows that the proportion of Given-When-Then steps in Gherkin scenarios stay consistent over the life of a Gherkin project. To plot this figure, first, timestamps from all commits in all projects were collected. Then, step share in each commit was calculated as a proportion and then to allow for varying frequency of commits in projects and project duration,



(a) Change in number of scenarios per feature versus project age

(b) Change in scenario size versus project age

Figure 7.5: Change in number of scenarios and their size versus project age

interpolation was applied per project across timestamps. Then a mean average share of step type was calculated for each timestamp, before finally a smoothed moving average was calculated for each step type.

Growth of Features and Scenarios in a Project

Figures 7.5a and 7.5b are scatter plots of the rate of change in average number of scenarios and their sizes in projects versus the age of Gherkin projects in our data set. Figure 7.5a shows the change in average number of scenarios per feature per day and Figure 7.5b shows the change in average number of steps per scenario per day. Both scatter plots show no change in the average number of scenarios and their size over the life of a project. This analysis was performed on the projects with *known project history* (i.e., at least 100 commits and minimum age of 180 days). The figures show that average number of scenarios per feature and their size stay roughly consistent through out the life time of a typical Gherkin project. However, the figures do not show if the number of features and scenarios grow over the life time of a typical Gherkin project.

To understand this we plotted a time series of project feature count over the lifetime of the projects using Figure 7.6. The figure shows a growth in number of features over the lifetime of the projects. The figure shows that the features decrease in number towards the end of projects which shows that some of the features are either refactored or commented. The Figures 7.7a and 7.7b are scatter plots of average number of scenarios and average number of steps against the number of features and number of scenarios per project, respectively. The figures show that the projects with a greater number of features tend to have slightly longer scenarios. The increasing trend lines in the figures show that larger projects tend to have longer scenarios. However, more research is needed to confirm this observation. Also, the difference in number of scenarios and their sizes between projects with large and small number of features is not a lot i.e., the

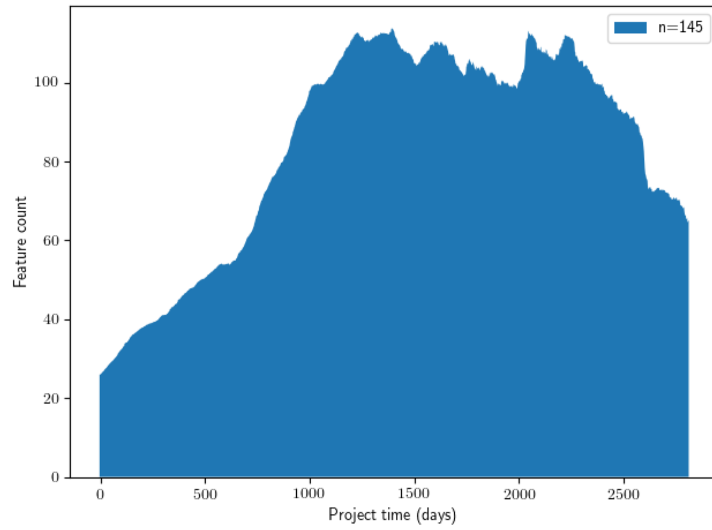
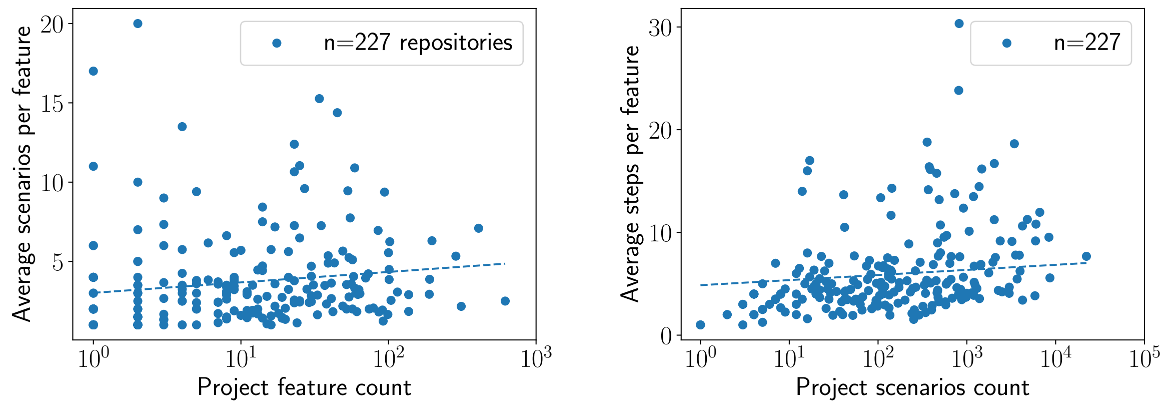


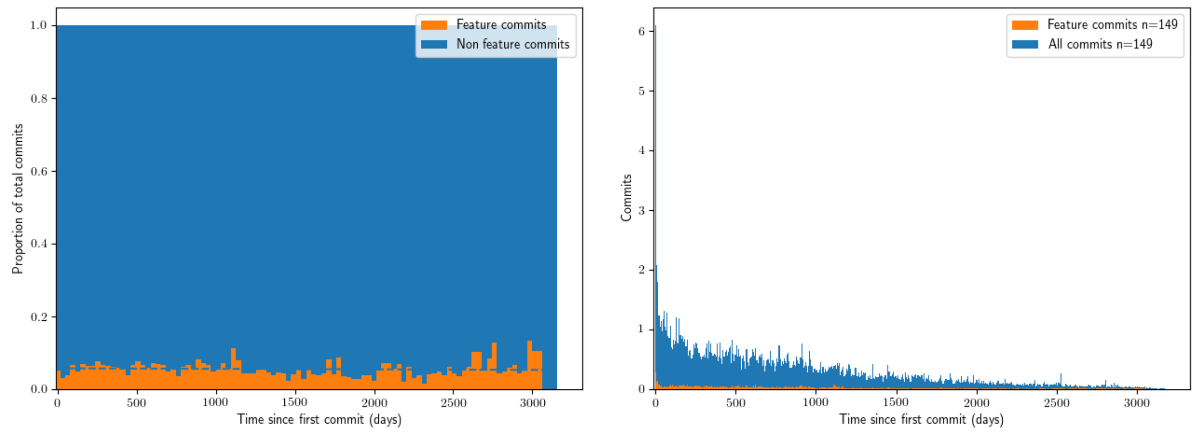
Figure 7.6: Feature count growth versus project age histogram



(a) Average number of scenarios per feature versus project feature count

(b) Average number of steps per scenario versus project scenario count

Figure 7.7: Average number of scenarios and steps versus the project feature and scenario count respectively



(a) Histogram of share of Gherkin commits versus Non-Gherkin commits over time

(b) Commit frequency over time

Figure 7.8: Share of Gherkin and Non-Gherkin commits and their frequency over time

difference is just a couple of scenarios and steps.

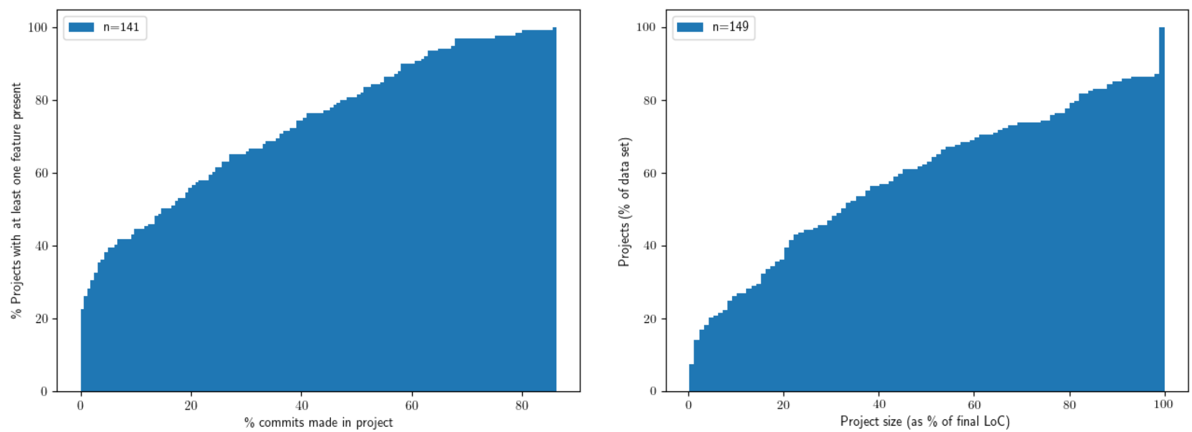
Incorporation of BDD in a typical Project

Various graphs were plotted to understand the practice of incorporation of Behaviour Driven Development (BDD). For example, Figure 7.8a shows a smoothed share of Gherkin and non-Gherkin commits over time for all projects in the data set. To plot this figure, first, timestamps from all commits in all projects were collected. Then, commit share for Gherkin and non-Gherkin commits was calculated as a proportion, and then to allow for varying frequency of commits in projects and project duration, interpolation was applied per project across timestamps. Then a mean average share of commit (i.e., Gherkin or non-Gherkin) was calculated for each timestamp, before finally a smoothed moving average was calculated for each commit.

The figure shows a very small number of Gherkin commits over time. The Gherkin commits are 5% of the total number of commits. To investigate further we plotted a commit frequency graph. Figure 7.8b shows the frequency of the commits over the lifetimes of the projects. The figure shows a decrease in the frequency of commits towards the end of the projects.

To understand when typically is Gherkin introduced in a Gherkin project we plotted a cumulative histogram of the percentage of commits made before first Gherkin feature. Figure 7.9a shows the percentage of commits completed before a feature was introduced. We can see that approximately 20% of projects introduced a feature in the first 1% commits. Moreover, all studied projects had at least one feature introduced before the final 10% commits. Figure 7.9a shows that approximately 75% of projects had introduced a feature within the first 33% of the total commits. It suggest that Gherkin introduced reasonably early for many projects.

In order to see how much development was completed before first Gherkin feature was introduced, we plotted a graph of percentage of lines of code completed before first Gherkin feature.



(a) Percentage of commits made before the first feature (b) Line of Code before introduction of first feature

Figure 7.9: Percentage of commits and LoC before first feature

Figure 7.9b is a cumulative histogram of percentage of project lines of code completed before the introduction of first Gherkin feature. The figure shows that less than 80% of the code was written in less than 80% of the projects, and less than approximately 22% lines of code were completed in less than 40% of projects. Although, the ascending shape of the graph suggests no average and shows that the projects introduced their first feature at varying percentages of lines of code completion, the median percentage of lines of code before the first feature was found to be 32. It means that approximately 32% of the total lines of code were written before the first feature was introduced in a project. This observation coincides with one of the observations in Chapter 6. In Section 6.5.1, we discussed that it was difficult for the developer to test before development because it was difficult for the developer to understand what the functionality would look like in advance. We can speculate by looking at Figure 7.9b that this problem exists in BDD projects in general.

In order to see if this practice of introducing first feature has changed over the years, we drew a scatter chart of days after which the first feature was introduced against the start date of a project. Figure 7.10 shows the days to the first feature in the Gherkin projects. The figure includes the projects started between the years 2012 and 2016. The mean line in the figure shows that there has been no change in the practice of introducing the first feature, over the years. The figure shows that there is no difference in the number of days lapsed after which the first feature was introduced in the Gherkin projects over the years. However, the projects are widely dispersed. A subset of projects introduce their feature immediately at the start of the projects whilst other projects are widely dispersed ranging up to over a thousand days.

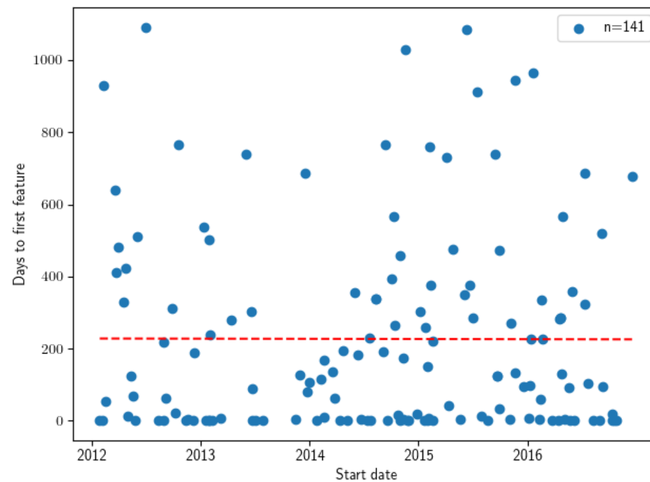


Figure 7.10: Change in practice of introduction of first feature in a project

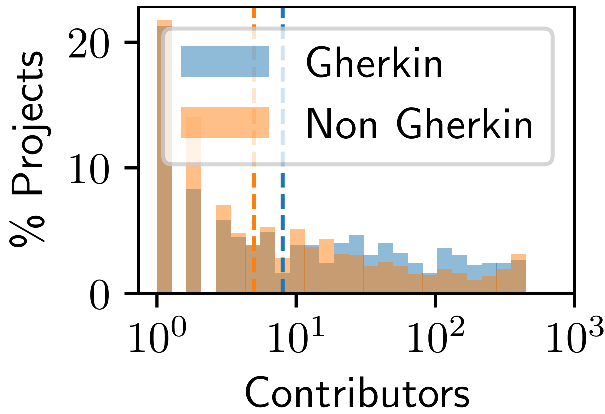
7.3.3 Gherkin versus Non-Gherkin Projects

The previous section gives an overview of BDD in open-source projects on GitHub. In this section, we compare the open-source Gherkin projects with a sample of non-Gherkin projects in order to understand the difference between them. This section presents a comparison between Gherkin and non-Gherkin projects by comparing the projects and their meta-data.

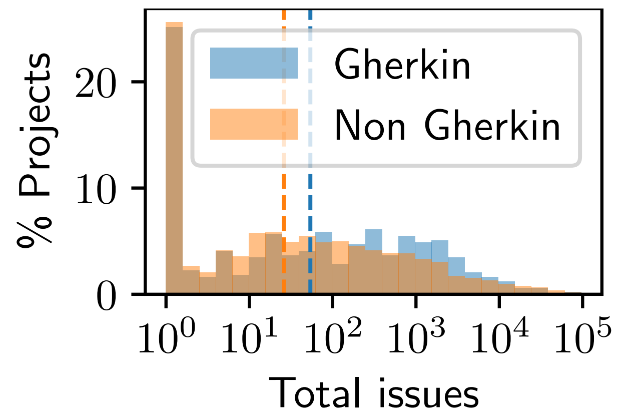
Analysis shows that a typical Gherkin open-source project on GitHub is larger than a typical non-Gherkin open-source project on GitHub. Figure 7.11 includes a graphical comparison of the characteristics of the selected projects. The figure includes histograms that compare number of contributors, total issues closed and open issues, total pull requests, open and closed pull requests, and (kilo) lines of code of Gherkin and non-Gherkin projects.

Visually, Gherkin projects seemed to have a greater number of contributors, total issues, closed and open issues, total pull requests, open and closed pull requests, and (kilo) lines of code; than non-Gherkin projects. This observation was confirmed through the values of the median for each graph. All the median values for Gherkin project characteristics except for the *open pull requests*, were greater than the median values of the respective characteristics in non-Gherkin projects. This shows that a typical Gherkin open-source project is larger than a typical non-Gherkin open-source project on GitHub. This observation is further confirmed through Figure 7.11h. The figure is a comparison of project size with respect to lines of code. The figure shows that a typical Gherkin open-source project is roughly 3.5 times larger than a typical non-Gherkin open-source project on GitHub.

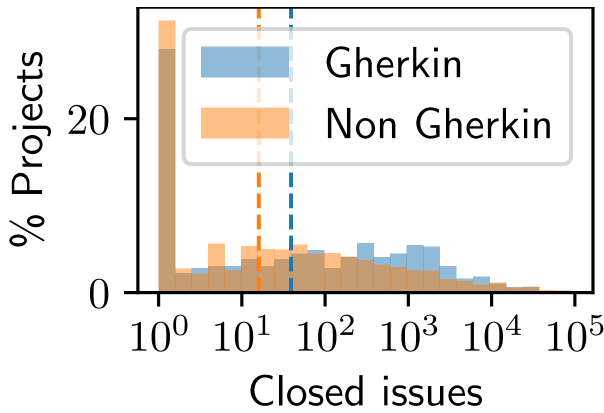
Figure 7.11g compares the number of open pull requests in Gherkin and non-Gherkin projects. The figure shows a low and equal number of median pull requests in both Gherkin and non-Gherkin projects. This implies that the majority of the projects in both samples (i.e., Gherkin and non-Gherkin projects) have a similar level of contribution or active moderation activity in them.



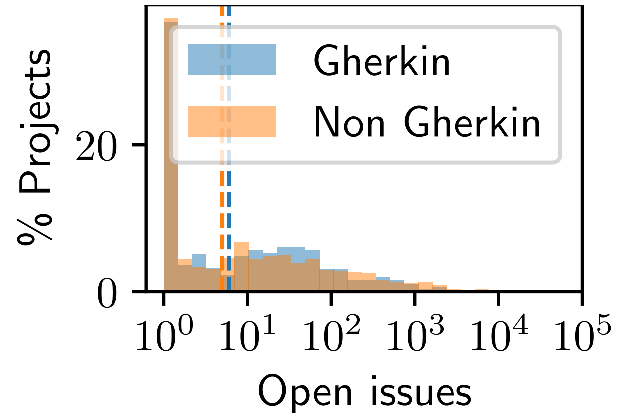
(a) Non Gherkin median=5 and Gherkin median=8



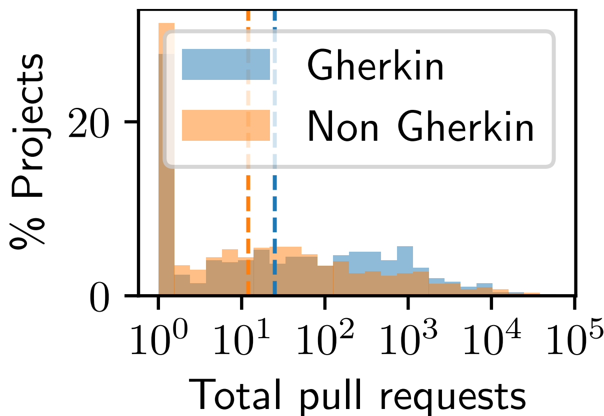
(b) Non Gherkin median=26 and Gherkin median=54



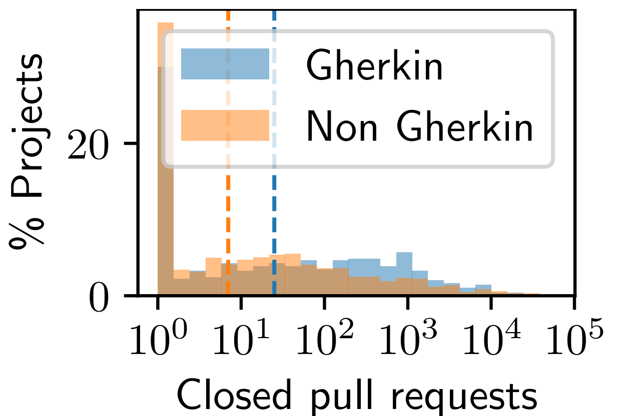
(c) Non Gherkin median=16 and Gherkin median=39



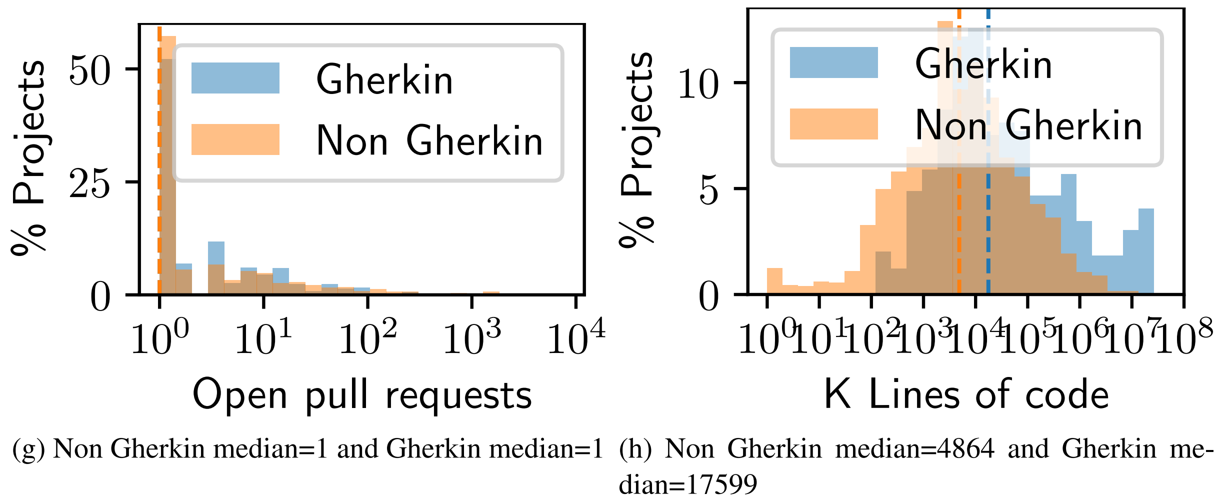
(d) Non Gherkin median=5 and Gherkin median=6



(e) Non Gherkin median=12 and Gherkin median=25



(f) Non Gherkin median=7 and Gherkin median=25



(g) Non Gherkin median=1 and Gherkin median=1 (h) Non Gherkin median=4864 and Gherkin median=17599

Figure 7.11: Comparison of characteristics of Gherkin and non-Gherkin projects

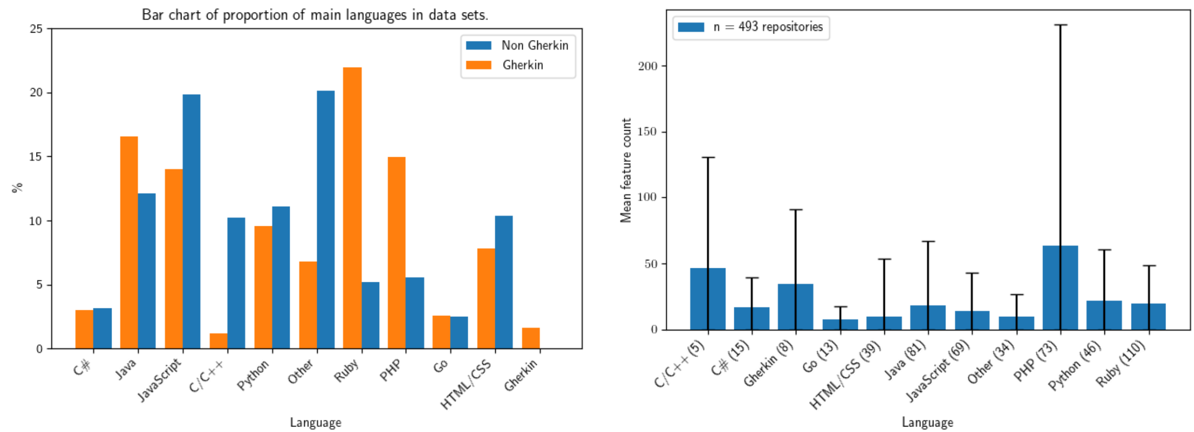
If we look at the number of pull requests in a few of the largest open-source projects[¶,‖,**] on GitHub, we see a small number of pull requests. For example, an open-source learning platform moodle has 599 contributors but only five open pull requests (until 18 March 2022). This shows that the project is active and regularly moderated. A high number of open pull requests in a project could mean that either moderators cannot handle the number of pull requests or the project has been abandoned by the moderators of the project.

Application Language

Figure 7.12a is a bar chart that illustrates the main languages employed in non-Gherkin and Gherkin data sets of projects. The figure shows that programming languages associated with web-application programming (Ruby, PHP) are noticeably more popular in the Gherkin than non-Gherkin data set(23% versus 5.5% and 15% versus 6% respectively). This suggests that Gherkin and Behaviour Driven Development may have received greater adoption in web application projects than in other technology domains.

Figure 7.12a shows that the most number of Gherkin projects were developed in Ruby making Ruby the most popular language for the development of the BDD projects, according to our data. Interestingly this finding coincides with the study by Zampetti et al. [2020]. Their analysis of 50,000 popular open-source projects written in five programming languages shows that "*...BDD is more adopted in Ruby than in other languages, and that RSpec is by large the most adopted BDD tool for Ruby*" [Zampetti et al., 2020]. However, the reason behind this popularity is not mentioned in the study [Zampetti et al., 2020]. One of the reasons behind the popularity of Ruby in BDD projects could be that Rbehave was one of the first user story-based BDD framework which was later integrated into RSpec [Chelimsky et al., 2010].

[¶] `https://github.com/996icu/996.ICU`
[‖] `https://github.com/moodle/moodle`
[**] `https://github.com/EbookFoundation/free-programming-books`



(a) Application language comparison: Gherkin (n=493) vs Non-Gherkin n=2009 Data Sets

(b) Average number of feature files in the popular languages

Figure 7.12: Popular application languages in Gherkin and non-Gherkin projects

The projects in PHP and Java language were found to be second and third in the relative majority of the Gherkin projects in Figure 7.12a. Significantly, projects associated with the C/C++ programming languages are a far smaller proportion of the Gherkin than non-Gherkin data set of projects (2% versus 11%), suggesting that Gherkin and Behaviour Driven development is not widely used in the projects developed in C/C++.

Figure 7.12b is a bar chart that shows the average number of gherkin files in the BDD projects developed in the popular languages. The figure shows that the Gherkin projects in PHP were found to have the majority of the total Gherkin files. This finding contradicts with one of the findings of the study by Zampetti et al. [2020] which says that "*...BDD frameworks are rarely used in Java and PHP projects*". According to the authors [Zampetti et al., 2020], BDD frameworks are used only in 2.26% of the overall PHP projects. Our data shows that although Gherkin projects are a very small portion of the open-source projects, the use of BDD is considerably higher in PHP and Java as compared to other technologies.

One of the reasons behind this contradiction could be the difference in the research method. Zampetti et al. [2020] took a sample of 10,000 repositories for each of the five popular languages on GitHub, including PHP. Whereas, we scanned 1 million random repositories irrespective of the development language and our results suggest that the popular languages on GitHub are popular in BDD projects as well. It is very much possible that BDD is not adopted widely in overall open-source PHP and Java projects but in the open-source projects that use BDD, PHP and Java are among the popular languages.

Project Duration

Figure 7.13 is a cumulative histogram that compares the distribution of project durations of Gherkin and non-Gherkin projects. Non-Gherkin projects range in duration from less than a

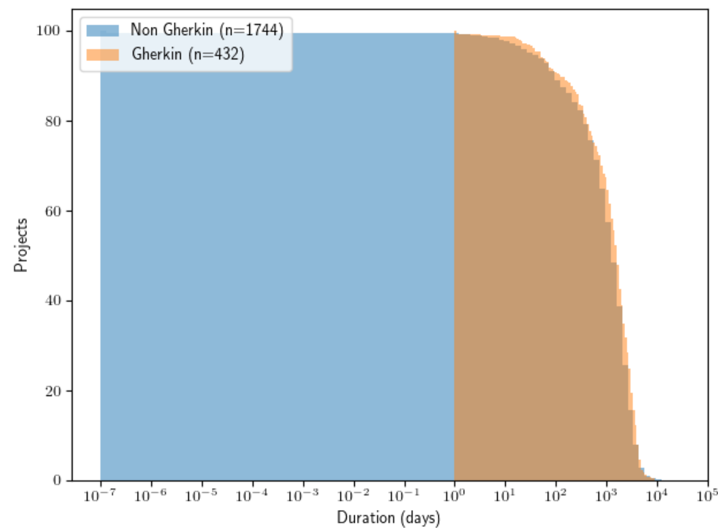


Figure 7.13: Project durations for projects with at least 10 commits

day in our sample, through to more than 10000 days. Conversely, project durations for Gherkin projects are more narrowly spread, with no project having a duration of less than a day.

Project Size

To compare Gherkin and non-Gherkin project size, we need to recall Figure 7.11h. The figure is a histogram of project lines of code counts for both Gherkin and non-Gherkin project data sets, for all projects with at least 1 line of code. The data graph shows that non-Gherkin projects range from less than 10 to more than 10 million lines of code. Gherkin project LoCs are distributed over a narrower range of more than 100 to approximately 10 million lines of code. In addition, Gherkin projects appear to be skewed towards large projects compared with non-Gherkin projects.

We investigated the causal relationship between the adoption of Gherkin and project size, as it is unclear from Figure 7.11h whether projects adopt Gherkin because they have reached a certain size, or whether projects that adopt Gherkin are more likely to *become* large. Figure 7.14 is a histogram of the number of lines of code completed before the introduction of the first feature in the Gherkin projects. The mean number of lines of code written before the first feature was approximately 8000. This shows that Gherkin projects tend to already be sizeable when first feature is introduced. If we compare Figure 7.14 to Figure 7.9b, we can say that 32% of the line of code consists of 8000 lines of code making an average open-source BDD project consisting of approximately 25000 lines of code in total.

Contributors

Figure 7.15a is a histogram that illustrates the distribution of contributors amongst non-Gherkin and Gherkin projects. As can be seen, both distributions range from just a single contributor

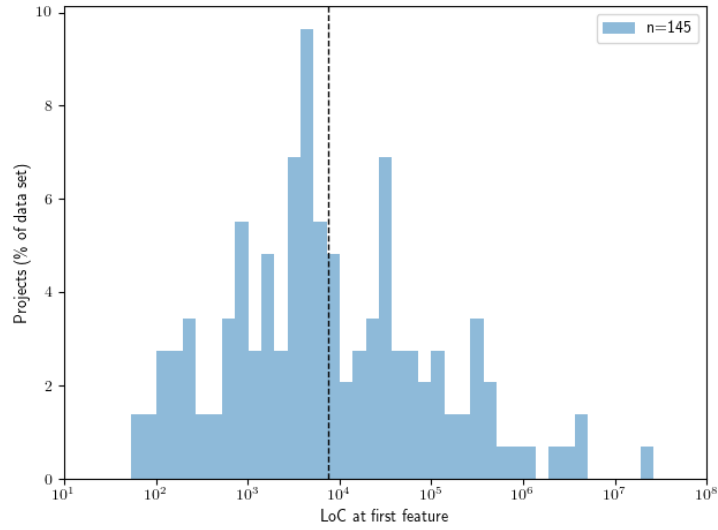
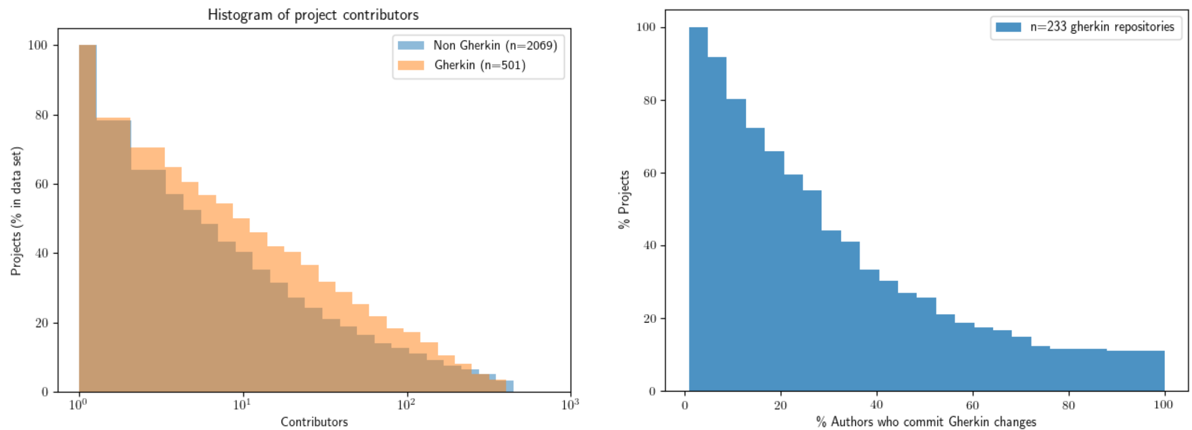


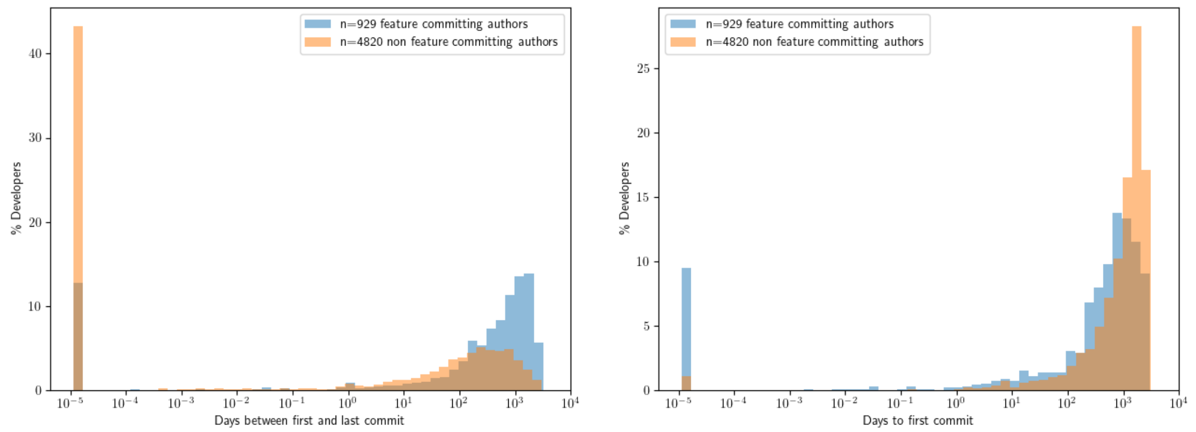
Figure 7.14: Lines of code at first feature in Gherkin projects histogram



(a) Histogram of project contributors in Gherkin and non-Gherkin projects

(b) Proportion of Gherkin authors in project

Figure 7.15: Histogram of contributors in Gherkin and non-Gherkin projects



(a) Developers' lifetime in Gherkin projects (b) Days to first commit in Gherkin projects

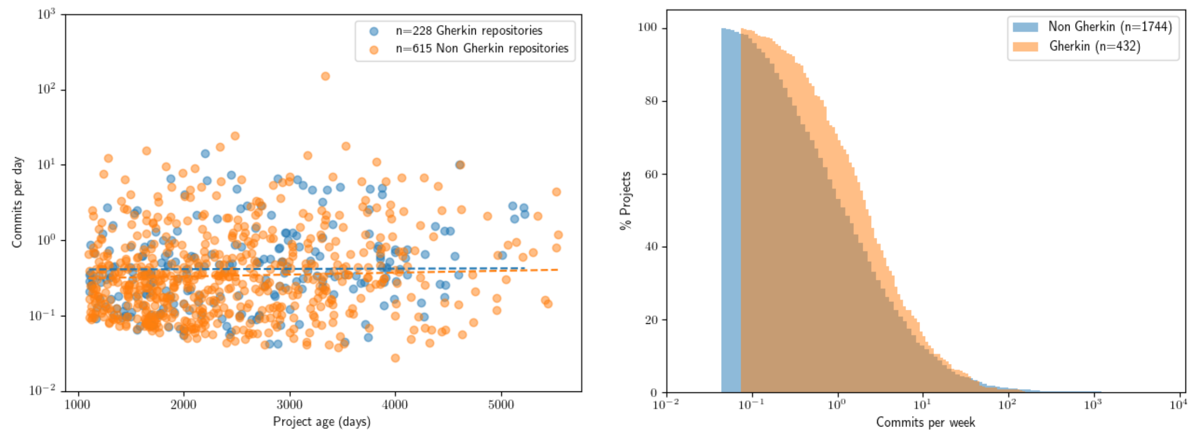
Figure 7.16: Developer's lifetime and earliest involvement in Gherkin projects

through to more than 400. If we look at Figure 7.15a, we see that both distributions are skewed towards small contributor counts. Also, we can see that the first decline in the graph from the top takes place at around a little less than 80% of the projects. This means that projects with just one or few contributors account for at least around 20% of projects in both cases. However, the histogram does suggest that the Gherkin projects attract more contributors than non-Gherkin projects. The median contributor count for Gherkin projects is 8, compared with 5 for non-Gherkin projects. This difference in contributors may be accounted for by the larger project sizes as described previously.

Although the Gherkin projects tend to attract more contributors, only a minority commit to Gherkin artefacts. Figure 7.15b is a cumulative histogram of percentage of developers who commit Gherkin changes versus percentage of projects. The figure illustrates the share of contribution between Gherkin and non-Gherkin developers in Gherkin projects. The X-axis shows the percentage of developers who commit Gherkin changes and the Y-axis shows the percentage of projects. If we look at Figure 7.15b, we see that 50% or more contributors make changes to 25% of the projects. The figure shows that on average, around third of contributors to projects containing Gherkin make changes to feature files i.e., about 60% of projects.

To understand how long Gherkin developers stay involved in Gherkin projects, we plotted a histogram of contributors in Gherkin projects against the number of days between first and last commit. Figure 7.16a shows the lifetime of Gherkin and non-Gherkin contributors in Gherkin projects. The X-axis in the figure shows the number of days between first and last commit. Y-axis shows the percentage of developers involved in the projects. The figure shows that Gherkin developers tend to have longer involvement with the projects.

In order to understand the earliest involvement of Gherkin and non-Gherkin developers in the project, we plotted a histogram of days to first commit with respect to Gherkin and non-Gherkin developers. Figure 7.16b shows the number of days between first commit made by Gherkin and



(a) Age of Gherkin and non-Gherkin projects versus change frequency

(b) Gherkin and non-Gherkin projects commit history histogram

Figure 7.17: Change frequency and commit history of Gherkin and non-Gherkin projects

non-Gherkin developers in Gherkin projects. The X-axis in the figure shows the number of days to the first commit. The Y-axis shows the percentage of developers who made the first commit. The figure shows that the Gherkin developers join projects earlier than non-Gherkin developers.

Project Commit Frequency

Figure 7.17a is a scatter chart, and it compares the frequency of commits per day in Gherkin and non-Gherkin projects. The figure shows no significant difference between the commit frequency of Gherkin and non-Gherkin projects. Figure 7.17b is a cumulative histogram that shows the cumulative commit history of the Gherkin and non-Gherkin projects. The figure shows that 60% of the non-Gherkin project had at least 1 commit per week. Whereas, over 80% of the Gherkin projects had at least one commit per week. The comparison between the commits of Gherkin and non-Gherkin projects show that the Gherkin projects have more number of per-week commits.

7.4 Discussion of the Results

Perhaps the most striking aspect of the results is the relative scarcity of open-source projects that have adopted Behaviour Driven Development. There is some variation in this respect with regard to the previous literature. CollabNet VersionOne [2019]'s survey of industrial practice suggests considerable uptake of BDD and Binamungu et al. [2018b]'s survey of practitioners also suggests that BDD has achieved widespread adoption. However, both surveys rely on self-reporting by practitioners which may be biased by participation enthusiasm. In Binamungu et al. [2018b] case, BDD communities of practice were specifically targeted in the participant recruitment strategy. To a lesser extent, the CollabNet VersionOne [2019] studies also rely on

self-reporting, and practitioners who consider that they practice agile may be more inclined to participate.

The empirical study by Zampetti et al. [2020] also concluded relatively high adoption of BDD in open-source communities. However, the sampling strategy adopted may also have influenced the conclusion, since a decision was taken to focus on popular software repositories that were identified as containing code in languages known to have good support for BDD frameworks. Our own strategy sought to obtain a more representative sample of software projects through random sampling of repository IDs rather than applying selection criteria, although both studies use similar post-sampling filtering strategies to identify 'genuine' software projects.

In addition, as Zampetti et al. [2020] themselves note, the observed use of SpecFlow style frameworks was in many cases more for describing unit tests rather than end-user features, which is more commonly associated with the practice BDD. A conclusion that could be drawn from this comparison is that neither SpecFlow nor Cucumber frameworks are commonly used for practicing BDD, but that due to the closer association between documented requirements and executable test code, SpecFlow frameworks are found to be convenient for maintaining unit tests. Further research is required to understand this difference.

As well as finding the adoption of BDD to be rare, within our sample, we found that repository feature suites are quite small. Several outliers were noticeable in the data set, comprising of very large feature suites of approximately 800 features and several thousand scenarios. However, these projects are exceptional. Therefore, we can characterise a median Gherkin feature suite on GitHub as comprising 5 features, each of 2 scenarios, with each scenario comprising 5 steps.

These figures contrast with the earlier work, specifically the exploratory study of open-source projects by Chandorkar et al. [2022], who suggest in their review of 23 repositories a median of 15 features per project. Again, this difference may well be accounted for by initial sampling policy, with Chandorkar et al. [2022] adopting a selection strategy of a relatively small number of repositories, based on popularity rather than random sampling by repository ID. Similarly, the respondents to Binamungu et al. [2018b] reported the maintenance of much larger feature suites than we encountered in our study. Again, this result may be due to sampling differences, with Binamungu et al. [2018b] sampling from self-reporting BDD practitioners, predominantly in the commercial software industry.

Our findings suggest that although feature suites are maintained throughout a software project and are proportionate to the size of the project code base as it grows, the number of features and scenarios is kept small. This may be evidence of a deliberate decision to use BDD frameworks for documenting and managing high-level acceptance tests that are useful for preventing the introduction of regressions. In addition, the small suite size may reflect the cost of maintaining large collections of scenarios and code functions, a maintenance challenge also reported by Binamungu et al. [2018b]'s respondents. Although this finding requires further investiga-

tion, it suggests that scaling and extending the coverage of BDD scenarios using the existing frameworks and tools incurs too great a maintenance cost.

Separately, the results of our own study point to a correlation between the maturity of an open-source project and the adoption of BDD (Figure 7.11). Comparing our two samples of gherkin and non-gherkin repositories, we found that Gherkin repositories on average have larger code bases, more issues, pull requests, and contributors and are older and experience more frequent commits. All these indicators suggest that projects that incorporate Cucumber-style BDD frameworks are more established than those without. However, it is unclear from this analysis whether the early adoption of BDD within a software project is a contributing factor in attracting contributors and activity to a project, or whether BDD is adopted once projects have established a certain level of maturity.

Within projects that adopt Cucumber-style BDD we observed a tendency to continue maintaining features throughout the project life-cycle, with a median of just 6 commits and 0 days since the last feature modifying commit in the repositories in our sample. However, this aspect of the project work is only undertaken by a minority of contributors (median 25% of authors). Our analysis also suggests these represent a 'core' who join the project early in its creation, are more intensively involved in project work, and remain with the project for longer.

This characterisation of feature-changing and non-feature-changing authors is striking and may have implications for owners of open-source projects seeking to recruit new participants. As reported by Lee et al. [2017], new recruits to open source projects may encounter obstacles to participation and be deterred from making a sustained contribution. Our study suggests that new contributors to projects that contain Cucumber-style BDD may be initially able to make small changes to the project that do not noticeably impact the feature files in the repository. The relatively small size of feature suites observed in our study, indicating limited test coverage to key 'smoke tests' supports this argument. If new contributors choose to become more familiar with the project and make a more sustained contribution, they begin to take on more responsibility and specify requirements through the creation or maintenance of Gherkin features.

However, new contributors may also be unable to identify situations where simultaneous modification of code and feature file is necessary unless the code change actively cause a test defined in the feature file to fail since there is no syntactic link between the name of a step and the step definition function that realises it. As a consequence, required changes to feature files will need to be detected in code review or become the responsibility of the core contributors to the project as a separate task. The recent work by Irshad et al. [2022] demonstrates the potential for automated detection, although their model is dependent on a dataset of commit-contemporaneous changes to code and features and does not necessarily account for situations where feature files *should* have been updated. Further, the lack of a strong semantics for Gherkin means that contributors may be uncertain as to *what* changes should be made to a feature file. These obstacles could potentially act as a limiting factor on the attraction of new contributors

once a project containing Cucumber-style BDD features becomes established.

Further work is required to understand these dynamics and the implications for tool development to assist contributors to BDD-cucumber projects. In particular, given the evidence of continued maintenance of Gherkin artefacts throughout a project lifetime, tooling may be required to better support concurrent alteration, maintenance, and refactoring of Gherkin features alongside other project artifacts. Irshad et al. [2022] and Binamungu et al. [2018a] have begun to develop and evaluate tooling for detecting (and consolidating) duplicate scenarios within Gherkin feature suites; and Storer and Bob [2019] investigated the potential to use natural language processing to mitigate the need for manually maintained step definition functions. However, much of this work is preliminary and awaits trial in practice. Further, a broader range of refactoring techniques may be desirable. For example, tooling to apply refactoring within Gherkin feature suites, such as extracting or inlining background sections, and splitting or merging features may be useful in a similar way that application code can be automatically refactored. However, further work beyond the present study is required to understand the details of current BDD workflows in actual practice and how such tooling might augment, or enable new approaches.

The previous study from Chapter 6 shows that the developer faced difficulty in maintaining the BDD suite since it was taking too long. Also, it was difficult for the developer to imagine the functionality in advance and write tests before the actual development. We see a similar pattern in the data collected during this study. The analysis shows that the average size of Gherkin test suite in a typical Gherkin open-source project on GitHub is small even though a typical Gherkin project appears to be three times larger than a non-Gherkin open-source project in terms of lines of code. It seems that the size of Gherkin test suite is deliberately kept small may be due to the effort required for its maintenance. However, further investigation would be required to see if tests were written after the development of the Gherkin open-source projects as well.

7.5 Threats to Validity

This section discusses the threats to the validity of this study. In this section, we have discussed three types of validity threats i.e., construct, reliability, and external validity.

Construct validity threat

If the Python tool does not accurately represent or capture the desired concepts, it may introduce a threat to the construct validity of this study. This could result in misleading or inaccurate graphs, which could undermine the validity of the findings of the study. For example, the tool lacked certain features or functionalities that could help in investigating further aspects of the application of BDD such as a natural language processing capability to study the evolution of BDD scenarios.

Reliability Validity Threat

A potential reliability validity threat for this study could be the potential inconsistencies in results produced by the Python tool over time. If the tool produces different results from the same data across different instances, it may introduce a threat to the reliability of this study. It is difficult to draw accurate and meaningful conclusions if there are inconsistencies in the results.

External Validity Threat:

A potential external validity threat to this study could be the limited generalisability of the findings to other contexts such as another sample of open-source projects. The study was conducted using a specific sample of open-source projects and a set of statistical techniques. The findings may not apply to a broader population of open-source projects.

Addressing Threats to Validity

At the start of this study, random open-source projects containing Gherkin were studied manually which helped us in devising criteria for selecting only the real projects and filtering out dummy projects. The validity of the exclusion/inclusion criteria for selecting "*real projects*" was tested on a random sample of non-Gherkin open-source projects. While doing the manual inspection, we created a few graphs manually and understood the construct we wanted to measure. The tool was implemented at the end of this *inspection process* to automate the measurement process. Additionally, long discussions between the author of this thesis and his supervisor took place to optimise the algorithms used for the selection of open projects and the creation of various graphs.

Every graph was manually inspected after its creation. The tool was tested on various sets of data extracted from various sub-sets of open-source projects containing Gherkin. This process evolved the tool and helped us in making the tool more reliable.

This study has a very specific context i.e., analysis of open-source projects on GitHub which contained Gherkin language. Thus, the findings of this study must be viewed in its context. However, we acknowledge that we could not test and compare our findings using another platform such as GitLab due to the time limitations of this Ph.D research.

7.6 Summary

This chapter provides an overview of BDD on GitHub. The aim was to analyse open-source BDD projects on GitHub to elaborate on *BDD in practice*. The use of BDD in a project was indicated by the use of Gherkin in a project because Gherkin is the language used for writing BDD features and scenarios. This is why a project in which BDD is incorporated is also known as *Gherkin project*.

Since our target was to analyse the meta-data and project contents from real projects, a tool was developed in Python which implemented a five-stage process including an exclusion/inclusion criteria to filter out the dummy projects. The validity of the filtration process was cross-checked with the help of a random sample of non-Gherkin projects on GitHub i.e., real projects filter reduced Gherkin samples by 60% and non-Gherkin samples by 58%. The tool not only extracted meta-data from the most recent commits in the projects but also extracted the meta-data at each commit of each project to present an understanding of the evolution of Gherkin artefacts. Our analysis suggests that around 40% of the accessible open-source repositories on GitHub consist of real projects out of which only 0.34% contain Gherkin.

Various graphs from the extracted meta-data were plotted to understand the growth and relationship between the features and scenarios. Our analysis suggested that Gherkin projects tend to be larger than non-Gherkin projects. We observed that the majority of Gherkin projects have a small feature set size i.e., approximately 70% of projects have ten or fewer features whereas, a handful have less than 100 features. With the exception of a few outliers, the median number of features in a Gherkin project on GitHub was 5, each having 2 scenarios, with each scenario comprising of 5 steps. We also found out that the large projects (i.e., the projects with a greater number of features) tend to have slightly longer scenarios.

Our analysis showed that the introduction of Gherkin features into a project does not significantly increase the size of the project. It means incorporation of BDD does not have any impact on the growth of a project. We also learned that the number of features and scenarios in the projects grows over the project life cycle however, the average number of scenarios in a features in a project and their size does not change during the life cycle of the project. It means that the details in the Gherkin requirements specifications do not evolve to an extent that can impact the average size of the scenarios.

The practice of incorporating of Gherkin language in the projects was understood with the help of various graphs. On average, 10,000 lines of code are written within the first 20% of the commits before the first feature was introduced in the project. This observation coincides with a finding from Chapter 6. According to one of our observations from the previous chapter, it is difficult for a developer to write test code before development since it is often hard to imagine the functionality in advance. We can speculate that this could be one of the problems faced by projects in general.

Our analysis shows that the practice of introduction of the first feature has not changed over the years. On average, all the projects introduce their first feature within the first eight months of the project age. Although the Gherkin projects duration vary between a day and 10,000 days, the percentage of Gherkin commits out of the total project's commits was very low i.e., 5%. From Figure 7.3 we can see that average size of Gherkin test suite in an open-source BDD project on GitHub is quite small i.e., 5 features and 2 scenarios per feature. However, the Figure 7.11h shows that an average open-source BDD project on GitHub is sizeable. This could mean that

Gherkin is maintained only for a small amount of functionality in a project; which could be the reason for a low Gherkin commit rate.

Our analysis showed that the developers who make Gherkin commits in a project are usually the ones who get involved in the project at very early stages. Also, the people who make Gherkin commits have the longest involvement in the project. This could also mean that the people who incorporate Gherkin are usually the primary contributors in the project or vice versa.

Comparison between Gherkin and non-Gherkin projects showed similarities between them. The three differences between Gherkin and non-Gherkin projects were: (i) the technology domains where BDD is receiving more adoption than the other technology domains i.e., Gherkin is significantly popular in the web development projects. (ii) Gherkin projects have slightly more number of commits than the non-Gherkin projects, and (iii) Gherkin projects attract more number of contributors than the non-Gherkin projects.

Chapter 8

An Analysis of Bad Smells in Gherkin Specification

This chapter is a continuation of the Chapters 6 and 7. In Chapter 7, we presented an overview of the open-source Gherkin projects and their comparison with non-Gherkin open-source projects. The previous chapter presented a statistical analysis of the contents and the metadata associated with Gherkin projects. This analysis gave an overview of the relationships between various artefacts of Gherkin projects. Specifically, the focus of the chapter was to characterise the practice of Behaviour Driven Development (BDD) in open-source projects on GitHub.

In Chapter 6, we discovered a potential for bad smells in Gherkin specifications. In this chapter, we further investigate this observation and see if the potential for bad smells observed in the Gherkin language in Chapter 6 is manifest in open-source projects. In order to do that, requirements specifications written in Gherkin were analysed to detect bad smells which could potentially lead to maintenance issues in Gherkin specifications. This chapter builds an understanding of the extent of these practices and patterns.

Gherkin specifications from the open-source Gherkin projects recovered from GitHub (as described in Chapter 7) were used to perform this analysis. These projects were curated from a random sample of open-source projects on GitHub. The next section describes the concept of *bad smells* and objectives of this chapter in more detail. Section 8.2 explains which bad smells were selected for this study for the analysis. The selection of bad smells was based upon: (i) applicability of the bad smell in the context of Gherkin specifications and (ii) the feasibility of doing analysis within the duration of this Ph.D research. Section 8.3 discusses each of the selected bad smells in detail. The section also documents the extent of the existence of each bad smell in the requirements specifications of open-source Gherkin projects on GitHub. Section 8.4 tests whether bad smells are correlated with other characteristics of Gherkin specifications, such as scenario size. Section 8.5 discusses the threats to the validity of this study. Section 8.6 presents the lessons learned from our analysis of the selected bad smells and the summary of the chapter.

8.1 Objectives of the Experiment

Bad smells in software engineering are mostly associated with bad design or structural inflexibility in a software artifact [Mäntylä et al., 2003, Garcia et al., 2009]. The term *bad smells* was first used by Fowler [1999] to refer to sub-optimal code structures which may cause harmful effects. Initial focus of the research on bad smells was on software code [Balazinska et al., 2000, Mantyla, 2003]. Later, other researchers [Van Rompaey et al., 2007] began to consider the nature of bad smells in software tests. Over the period of time, the discussion around bad smells spread to other areas of software engineering such as architecture [de Andrade et al., 2014], databases [Sharma et al., 2018] and web usability [Grigera et al., 2014] etc.

Bad smells in software system artefacts do not usually stop a software system from executing [Farcic and Garcia, 2018] but hinder the evolution of a software system [Yamashita and Moonen, 2013]. Their existence in software artefacts could lead to maintenance issues in a software project [Farcic and Garcia, 2018, Yamashita and Moonen, 2013]. Bad smells can exist in various software artefacts such as code, test suite, and requirements specification in different forms including bad design decisions, duplication, and ambiguities.

As mentioned, this chapter is a continuation of Chapters 6 and 7. One aspect of results from the case study discussed in Chapter 6 was the discovery of *violation of AAA pattern* and *assertion roulette* bad smells in Gherkin specification. We also learned that, due to its natural language structure, Gherkin provides a lot of freedom for writing requirements specifications. Therefore, there is potential for developers to write requirements specifications that contain bad smells which would increase the cost of evolution of a software system even if the requirements express the desired functionality correctly. It is reasonable to speculate that bad smells known from source code quality practice could also exist in Gherkin specifications. We, therefore, decided to investigate whether the potential for bad smells observed in the Gherkin language in Chapter 6 were manifest in open-source projects. Therefore, in this chapter, we discuss bad smells and maintenance of Behaviour Driven Development (BDD) requirements specifications.

Since the literature, particularly empirical research on BDD is scarce [Egbreghts, 2017, Solis and Wang, 2011], it is pertinent to know what practices could lead to bad smells in BDD specifications so that this study could serve as a guide for people planning to use BDD. Specifically, the literature on BDD test smell is almost non-existent. This is also pointed out as a research opportunity by Binamungu et al. [2018b]. Therefore, the research objectives within the context of this chapter are:

- Map the bad smells in unit test to Gherkin specifications in open-source Gherkin projects on GitHub on the basis of apparent applicability.
- Explore the prevalence of the applicable bad smells in Gherkin specifications in open-source Gherkin projects on GitHub.

- Examine if the applicable bad smells in Gherkin specifications in open-source Gherkin projects on GitHub are correlated with other Gherkin artefacts.

Several factors including knowledge gained from the literature, our experience with BDD in Chapters 6 and 7, manual inspection of Gherkin specifications, and the discussions between the author and his supervisor played a key role in identification and selection of bad smells in Gherkin specifications. Knowledge gained from the literature provided a theoretical background on BDD. The literature also provided a background on the use of Gherkin and a brief overview of the potential for bad smells in Gherkin specifications. The literature on *bad smells in unit tests* helped us in identifying the bad smells that could be applicable to Gherkin specifications. Manual inspection of randomly selected repositories allowed us to validate the mapping of bad smells within unit test sites to Gherkin specifications.

8.2 Review of Bad Smells in Gherkin

The issue of *bad smells* in Gherkin specifications is acknowledged by several researchers [Binamungu et al., 2018b, Suan, 2015]. Binamungu et al. [2018b] used an online survey to gather responses from 75 BDD practitioners from 26 countries in order to understand the extent of use, the benefits, and the challenges of BDD. Along with the discussion on the benefits of BDD, the study highlighted the challenges specific to the maintenance of BDD specifications. The challenges highlighted in the study included the slow speed of a test suite, the need to maintain BDD tests in addition to unit tests, and duplication detection. According to the authors, large BDD test suites are not only difficult to manage and maintain but fault correction is also difficult. The authors listed investigating BDD test smells as one of the future research opportunities.

The focus of the study by Suan [2015] was on duplicate detection in BDD specifications i.e., cloning. The author used the text matching and dice coefficient algorithm to identify likely duplicates. The intersection process collects a pair of consecutive written units that exist in both pieces of text and then the formula for calculating dice coefficient is applied. The results were not completely accurate but the concept and the problem itself are very important. Duplication in BDD scenarios could create maintenance issues in case of refactoring of the specifications i.e., the same change will have to be made at multiple places because of duplication.

Existing research [Binamungu et al., 2018b, Suan, 2015] demonstrates the concern in the industry for the potential for bad smells to hinder maintenance in BDD specifications. However, the literature lacks strong evidence for either the presence of bad smells in BDD specifications, or indeed, the nature of what constitutes a bad smell in Gherkin.

(((Bad smells) AND Behaviour Driven Development) AND software)
(bad smells in BDD) AND gherkin))
(bad smells in gherkin specifications)

Figure 8.1: Search Strings

8.2.1 Gherkin Bad Smells identified in Peer-Reviewed and Grey Literature

An online search was performed using Google scholar [LLC, 2004] employing the search strings specified in Figure 8.1. The search revealed a lack of literature on bad smells in Gherkin specifications. So, we adopted the practice described by Garousi and Küçük [2018] and extended the scope of the search to grey literature (i.e., blogs and webpages etc.) [Kennedy, 2012, Knight, 2017, CucumberStudio, 2016, Stenberg, 2016].

These webpages also included a part of the documentation on Cucumber [CucumberStudio, 2016] which discussed *Anti-patterns* in Gherkin. The first anti-pattern in the documentation was: *having scenarios steps that cannot be reused across scenarios or features*. The documentation states that the existence of this anti-pattern in the test suite "*...may lead to explosion of step definitions, code duplication, and high maintenance costs"*. This implies that scenario steps that cannot be reused across features or scenarios in a test suite must be avoided. However, the documentation does not acknowledge the instances where this anti-pattern cannot be avoided, thus emphasising only the reuse of steps.

The second anti-pattern in the documentation [CucumberStudio, 2016] was: *conjunction steps*. This problem arises when two or more phrases, each of which describes an independent behaviour, are combined in a single step. According to the documentation, combining two or more independent phrases in a single step makes the step too specialised and hard to reuse.

Instead, the documentation recommends combining two or more steps using *abstract helper method*. This capability is specific to Java and used when a specific repetitive task is shared between multiple methods or classes. This capability helps in reducing the number of redundant Gherkin steps. According to the documentation, combining several steps into one (in this way) "*makes your scenarios easier to read"*. However, the wording presently used to explain this concept in the documentation [CucumberStudio, 2016] encourages *combining the steps* rather than *reducing the number of steps*. Other examples of Gherkin Bad smells in the Grey literature include violation of AAA pattern [Sundberg, 2016], Lazy Scenario Outlines [Sundberg, 2016, Stenberg, 2016] and Multiple Assertions [Sundberg, 2016] etc.

We observed that the information we found on the topic in grey literature was more than what was available in peer-reviewed literature. However, the information lacked details and the arguments were under-developed. For instance, the documentation [CucumberStudio, 2016] on Cucumber (BDD test management Tool) discusses the *Anti Patterns* but the description does not include examples explaining the anti-patterns in detail. Also, the documentation does not state

if these are the only potential bad smells in Gherkin specifications.

8.2.2 Mapping Bad Smells

In the literature, bad smells in tests are predominantly discussed in the context of unit testing. In BDD, scenarios are implemented as one or more functional tests i.e., different units integrated to perform the tests. Due to the lack of literature and guidance on the bad smells in BDD specification and the close relationship and structural similarities between BDD scenarios and unit tests, we decided to expand the scope of this study. We used the existing knowledge of the bad smells in unit tests to see which bad smells in unit tests were applicable to the BDD specifications.

We found a number of studies that discuss and group the bad smells in the unit tests into various categories. For example, Meszaros [2007] categorised the test smells into three kinds i.e., *(i) Code Smells*: normally observed while reading the code e.g., Obscure Test; *(ii) Behaviour Smells*: encountered when the tests are compiled or run e.g. Fragile Tests; and *(iii) Project Smells*: are the defects found during formal testing by the users or the customers e.g., Buggy Tests.

Reichhart et al. [2007] presented an approach and implemented it as an experimental tool for the qualifying tests. The authors also defined criteria for determining the test quality. The authors evaluated their approach on a large sample of unit tests from open-source projects. They identified 27 bad smells in unit tests during their study.

Bavota et al. [2015] reported the results from two empirical studies they conducted to find out the extent of bad test code smells and their impact on program comprehension during maintenance activities. Their first study was an exploratory study of 27 software systems. The second study consisted of a controlled experiment involving four groups of participants with varying levels of professional experience. The findings showed a frequent occurrence of the test smells in software systems. According to the authors, the test smells have a negative impact on programmers' comprehension during maintenance activities.

The focus of a systematic literature review [de Paulo Sobrinho et al., 2021] of the studies published on bad smells between 1990 and 2017 was on examining various kinds of bad smells in the unit tests and the evolution of researchers' interest in them. The authors also investigated the co-occurrence of the discussion of different bad smells in the study. The study provided a list of 104 bad smells from 351 papers. According to their findings, the bad smells studied the most are: (i) Duplicate code, (ii) Large Class, (iii) Feature Envy, (iv) Long Method, and (v) Data Class.

We came across several studies [de Paulo Sobrinho et al., 2021, Reichhart et al., 2007, Meszaros, 2007, Van Rompaey et al., 2007, Yamashita and Moonen, 2013, Bavota et al., 2012, Garousi et al., 2018] which discussed a number of bad smells. The biggest problems we faced were: *(i)* the unavailability of an exhaustive list of bad smells, *(ii)* the total number of bad smells

discussed in each of the studies was different, *(iii)* the focus of the formally published literature on the bad smells in code, whereas, the literature discussing an exhaustive list of bad smells in structure of a unit test appeared to be scarce.

We also observed that the information available on the bad smell on online blogs, white papers and internet articles was more detailed and covered more number of bad smells than the formally published research. Also, we did not want to limit ourselves to the bad smells in the code. Since we were investigating BDD specifications, we also wanted to look at the structural bad smells like Arrange-Act-Assert (AAA) pattern violation. Therefore, it was important to take the information available in grey literature into account.

We were able to find a study by Garousi and Küçük [2018] which used both, formally published sources and the grey literature for their systematic literature review on bad smells in software tests. They compiled a list of 182 bad smells from 166 sources including 120 (72.2%) from the grey literature (e.g., internet articles and white papers). The authors prepared a spreadsheet* of the bad smells in which they grouped the bad smells into six themes based upon their natures.

We decided to analyse bad smells in unit tests and select the bad smells which appeared applicable to BDD specifications. We copied the spreadsheet and performed deductive reasoning by: *(i)* searching and reading the definitions and online examples for each of the 182 bad smells; and *(ii)* having a discussion between the author of this thesis and his supervisor about the bad smells which potentially seemed applicable to the BDD specifications. The criteria for the selection of the bad smells was their applicability to the BDD specifications. Bad smells which had no apparent effect on the BDD specification or were not directly applicable to the BDD specifications, were not selected during the process.

As a result of the above activities, we selected eight bad smells out of the 182, *(1)* Slow Test; *(2)* Eager Test; *(3)* Over-specification; *(4)* Testing Happy Path only; *(5)* General fixture; *(6)* Duplication; *(7)* Assertion roulette; and *(8)* Obscure Test. During the selection process, we also observed that some of the 182 bad smells were duplicates i.e., a synonym for another bad smell in the same list. For example, *Long Test* and *Complex Test*, the two bad smells in the list, were the synonyms for Obscure Test which was also listed among the 182 bad smells.

In addition to the deductive reasoning, we *(i)* used our background knowledge of BDD and *(ii)* manually inspected the BDD feature files from a random sample of repositories which led to the discovery of three more bad smells i.e., AAA pattern violation, Lazy Steps, and Lazy Scenario Outline. AAA pattern violation was discovered during the action research (Section 6.3) in Chapter 6. Whereas, the *Lazy Steps* and *Lazy Scenario Outline* bad smells were observed during the manual inspection of the feature files. Figure 8.2 lists the bad smells apparently applicable to the BDD specifications. The third column of Figure 8.2 provides a brief explanation of the relevance of the bad smell in the context. We do not claim that this is an exhaustive list of the

*`https://goo.gl/1ZrL65`

	Bad Smell	**Applicability to BDD specifications**
1	Slow Test	The execution of BDD Test Suite could become slow due to various reasons such as a large test suite with heavy coupling or unnecessary use of Scenario Outlines etc.
2	Eager Test	Testing multiple behaviours in a single Gherkin step.
3	Over-specification	Imperative scenarios.
4	Testing Happy Path only	Not having alternate scenarios for handling unexpected inputs and outputs.
5	General fixture	Having *Given* steps that are not used by the scenario.
6	Duplication	Duplication of Gherkin steps.
7	AAA pattern violation	Interleaved Given-When-Then statements
8	Assertion roulette	Multiple *Then* statements in a scenario
9	Obscure Test	Long and complex scenarios
10	Lazy Steps	Single row table(s) attached to scenario step(s)
11	Lazy Scenario Outline	Scenarios outlines with a single row

Figure 8.2: Applicable bad smells in the context of BDD specification

applicable bad smells in BDD specification. It is possible that we, unintentionally, missed *other* bad smells that are also relevant to the BDD specifications.

After the selection of the bad smells, we performed a feasibility assessment, considering the time required to collect evidence for each of the bad smells in the Figure 8.2. As a result of our feasibility assessment, we selected the following five out of 11 applicable bad smells.

- AAA pattern violation
- Assertion roulette
- Duplication
- Lazy Steps
- Lazy Scenario Outline

The scope of the rest of the six bad smells was either too broad or collecting evidence for them required an amount of work that could not be covered during the course of this Ph.D. For example, investigating *Slow Test* in BDD specifications would require automating the execution of the test suite of 493 projects and then finding out which of them are *slow* and why. It is possible that the slow execution of a test suite could be an effect of *other* bad smells in a test suite, investigating each of which requires a considerable amount of time. Similarly, the detection of multiple behaviours in Gherkin steps (i.e., Eager Test) would require using natural language processing techniques to understand and detect independent phrases in Gherkin steps. As much as we wanted to cover the six *unfeasible* bad smells, the amount of work required to gather evidence for such bad smells was not possible within the limited duration of this Ph.D.

8.2.3 Experiment Design

This section describes the bad smells that were selected for the analysis and explains the process of calculating the existence of those bad smells in open-source BDD projects' specifications on GitHub.

In order to extract the data electronically from the projects selected earlier (in Chapter 7), the Python tool discussed in Section 7.2 was extended. This tool was initially developed to extract project-related metadata from open-source BDD projects on GitHub. To explore the potential for bad smells in Gherkin specifications, the Python tool was extended to perform statistical calculations on the smells selected for analysis in this chapter. The definition of each bad smell calculated by the tool is as follows:

- **AAA pattern violation**: Arrange-Act-Assert also known as *AAA pattern*, is considered to be one of the best practices for structuring unit tests [Chaczko et al., 2014, Axelrod, 2018, Ma'ayan, 2018]. The (AAA) pattern was observed and named by Bill Wake[†] in 2001, and the purpose was to make the structure of unit tests readable and maintainable by organising a unit test into three clear and distinct steps i.e., Arrange, Act, and Assert. The AAA pattern is equivalent to Given-When-Then scenario structure in BDD, and it arranges the Gherkin scenarios into the Arrange, Act, and Assert format.

  Each scenario was checked for the presence or absence of interleaved Given-When-Then steps. Every instance of recurring *Given* step(s) after *When* or *Then* step(s), or recurring *When* step(s) after *Then* step(s) was counted as a violation of AAA pattern. This means every single instance of interleaved Given-When-Then in a scenario was counted as one violation of the AAA pattern.

- **Assertion roulette**: It is commonly agreed that having multiple asserts could potentially mask bugs because, in case of failure of an assert, it is not possible to know the pass or fail status of the subsequent assert [Tufano et al., 2016, Ma'ayan, 2018].

  In order to verify if multiple assertions in a BDD scenario could also mask bugs as they do in the unit tests, we created a pilot project in Python. For this, A feature file with a single scenario, having four steps (each starting with Given, When, Then, and And respectively) was created. We generated code stubs for all the four steps with a failing third step. When we executed the specification we observed that the fourth step was skipped because of the failure of the third step. The skipped steps do not execute therefore, the pass or fail status of the fourth step was unknown which could potentially mask bugs in the fourth step. This pilot project demonstrated that if an assertion in a Gherkin test fails, the failure or successful execution status of a subsequent assertion can not be determined.

[†] `http://xp123.com/articles/3a-arrange-act-assert/`

The number of Given, When, Then statements were calculated for each repository. To know how many repositories have scenarios with more than one assertion, we subtracted the total number of scenarios in a project from the total number of *Then* in a project. If the number of total assertions was more than the number of total scenarios in a project, it was an indication of assertion roulette bad smell.

- **Duplication**: By *duplication of Gherkin steps or clones* we mean repetition of two or more consecutive steps that already exist in another scenario. The tool implemented two algorithms for searching for clones in Gherkin specifications: (i) suffix trees for searching for clones within a single scenario, (ii) the longest common subsequence table for searching between two scenarios.

  These algorithms were used to detect clones. At first, all scenarios in a project, and all pairs of scenarios were checked for potential clones. In both cases, a clone in Gherkin was defined as a sequence of two or more steps that recur elsewhere, either in the same scenario or within another scenario in the project. When a clone was found, it was added to a clone tree. Once all scenarios and pairs were evaluated for possible clone candidates, the set of clone trees was sorted by maximum depth. Clones were then extracted in size order from the project. Each instance of repetition of two or more consecutive Gherkin steps within or across the feature files within a project was counted as one clone.

- **Lazy Steps Data Table:** A Gherkin language terminology called *Step Argument*[‡,§] refers to a capability of attaching data to a Gherkin step. This data could be a block of text or a table of data. In Python, this data is passed as an attribute to the *context* variable which is then passed into a step function. A single row in a step data table is an overuse of this capability and adds unnecessary complexity to the specifications and implementation code. Data tables are often misunderstood with scenario outlines. It is important to understand that in a *Scenario Outline*, each row in the example table executes for the whole scenario. Whereas, a *Data Table* only executes for the single step under which the data is defined.

  To find the number of lazy steps, all feature files in each project were parsed programmatically with the help of the Python tool. Each feature file was checked for scenarios with steps having tables containing a single row. Each of such instances was counted as one lazy step.

- **Lazy Scenario Outline:** Keeping identical scenarios in a feature file creates duplicates and rework in case of an update. Identical scenarios having the same wording but different data values in one feature file could be collapsed into a single *Scenario Outline*. Refactoring identical scenarios into *Scenario Outline* and organising the data into tables not only

[‡] https://cucumber.io/docs/gherkin/reference/
[§] https://behave.readthedocs.io/en/stable/gherkin.html#step-data

improves readability but saves rework in case of an update. This capability provides ease of maintenance and helps in reducing text without losing the descriptive value of text.

To find the number of lazy outlines, all feature files in each project were parsed programmatically with the help of the Python tool. Identical scenarios with different data values can be combined using *scenario outline* whereas, a scenario outline with a single row can be written more concisely as an ordinary scenario. Therefore scenario outlines with only a single row in the example table were considered a bad smell. Each *Scenario Outline* was programmatically checked for the presence of a single row in a table. Every *Scenario Outline* with less than two rows was counted as one lazy Scenario Outline.

While manually inspecting the Gherkin specifications from the selected open-source projects, we noticed the existence of another bad smell which is inverse of *Lazy Scenario Outline* discussed in this chapter. This bad smell, which we noticed, consisted of identical scenarios with different data. Such scenarios should have been combined into a scenario outline. However, we did not evaluate the existence of this bad smell because the work required to detect this bad smell was complicated and could not be implemented within the course of this Ph.D.

8.3 Results

This section describes the extent of the existence of the (selected) bad smells in open-source BDD projects' specifications and the potential impact of the bad smells on maintenance of BDD specifications.

An overview of the existence of bad smells discussed in this chapter is presented using Figure 8.3. The figure is a bar chart in which each bar represents the percentage of projects with each bad smell in the 493 Gherkin projects. There are six bars in the figure, and each bar represents percentage of Gherkin projects with a particular bad smell. The figure shows that Gherkin specifications of approximately 36% projects contain at least one violation of Given-When-Then order (i.e., AAA pattern violation), 70% Gherkin projects have at least one scenario with more than one assertion, and 68% Gherkin projects have at least one instance of duplication of two or more consecutive steps (i.e., clones). The figure shows that 3% of the Gherkin projects have at least one unnecessary *Background* i.e., 3% projects had at least one feature file with one scenario and a *Background*. Because the percentage of projects with the unnecessary *Background* bad smell was negligible, we did not investigate them any further. Approximately 30% projects have at least one lazy step i.e., at least one instance where a scenario step has a table with only one row. These scenarios could be more concise such that the data in the table is moved to *within* the relevant step(s). Approximately 12% of the projects have at least one lazy *Outline* table where the table had just one row i.e., a scenario should have been used instead of an *Outline* table.

Figure 8.4 is a bar chart for the bad smells. The figure contains pairs of bars where each pair

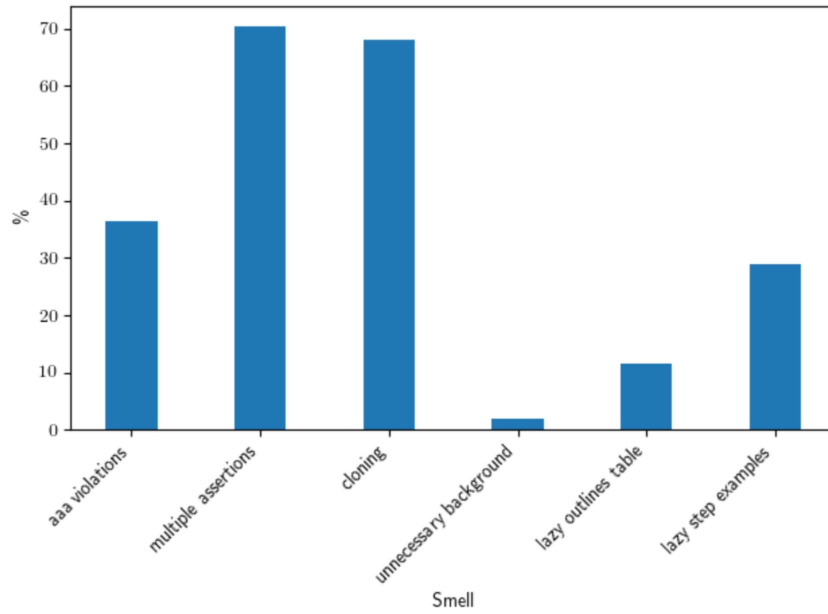


Figure 8.3: Projects' bad smells bar chart

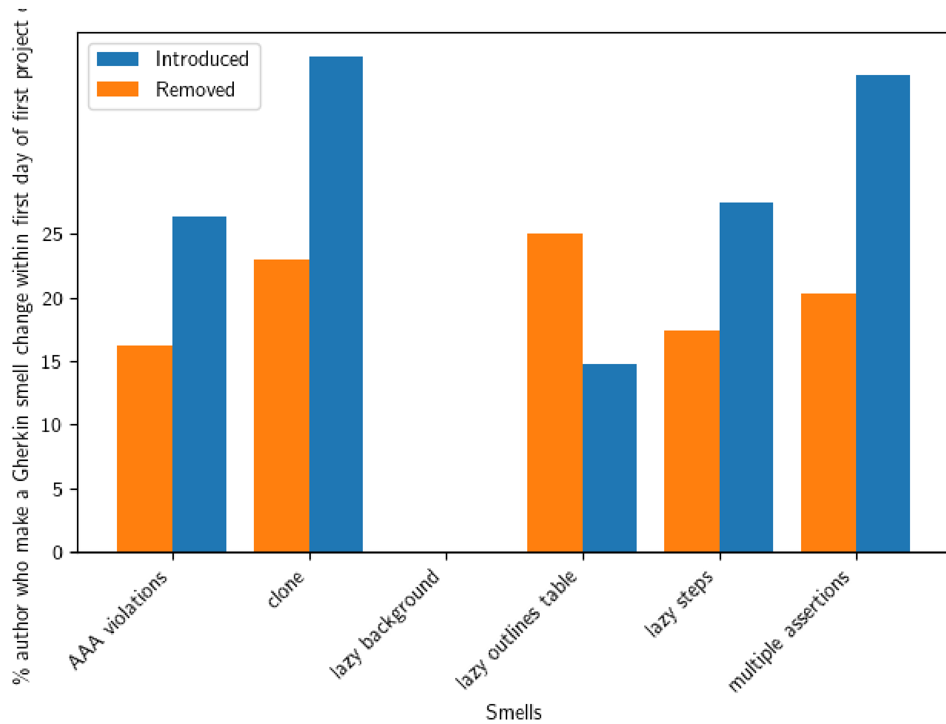


Figure 8.4: Contributor time to bad smell introduction

Arrange-Act-Assert (Test Driven Development)		**Given-When-Then (Behaviour Driven Development)**	
Arrange	setup and initializations required for the test	**Given**	describes the preconditions for the scenario and prepares the test environment
Act	actions required for the test	**When**	describes the action under test
Assert	verification of outcome of the test	**Then**	describes the expected outcomes

Figure 8.5: Arrange-Act-Assert vs Given-When-Then

represents the proportion of Gherkin contributors who removed or introduced the bad smells into the project within the first day of their Gherkin commit. This graph shows the default writing style of the contributors who write Gherkin scenarios. This shows us the proportion of the contributors who are unaware of the concept of bad smells in Gherkin scenarios. We can see from the Figure that this value is between 15% and 45%. The Figure shows that the majority of the bad smells in Gherkin scenarios are introduced by 15% to 45% Gherkin contributors within the first day of their project commit. The following sub-sections discuss these bad smells in detail.

8.3.1 Arrange-Act-Assert vs Given-When-Then

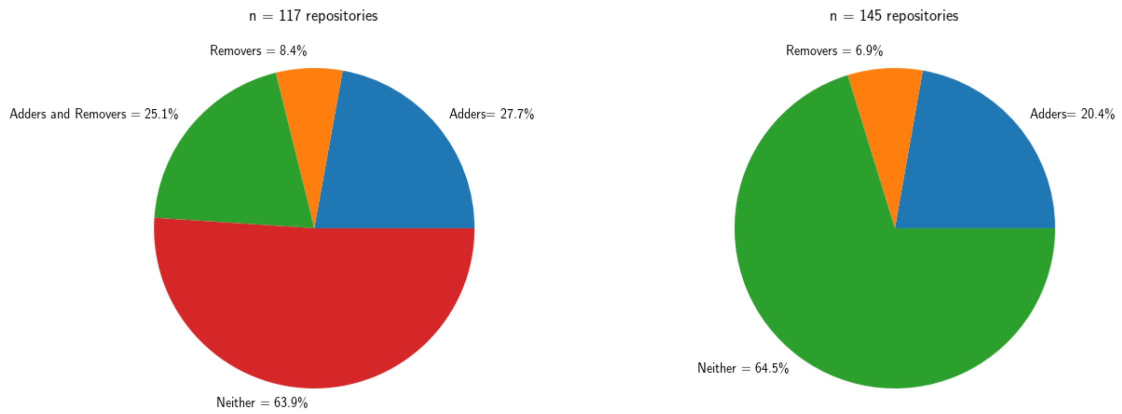
Although the AAA pattern is a widely accepted practice for structuring unit tests [Axelrod, 2018, Chaczko et al., 2014, Ma'ayan, 2018], we did not find a single study that cross-examines the pattern in detail. However, we came across several online blogs and online discussion threads[¶,‖,**] which discuss the pattern in more detail than the published research.

If we compare the concept of AAA pattern (in unit tests) with Given-When-Then (in BDD) and vice versa, we see similarities. The notion and structure of the BDD Given-When-Then specification template is a reflection of the AAA pattern [Khorikov, 2020a, Northwood, 2018, Ritchie, 2016]. Figure 8.5 describes the AAA pattern and its equivalent in BDD specifications in the form of Given-When-Then. The difference between them is the context i.e., the AAA pattern is seen in the context of code and unit tests, whereas Given-When-Then in BDD is associated with Gherkin specifications. According to Khorikov [2020b], the only difference between AAA and Given-When-Then is that the latter is more readable to non-programmers and non-technical people. In a recent study, Oliveira and Marczak [2018] have listed quality attributes of BDD scenarios. According to the authors, mixing the step order makes the scenarios unreadable. Mixing the Given-When-Then step order is equivalent to the violation of Arrange-Act-Assert pattern (in unit testing) which creates a disarrangement and makes the code unreadable [Axelrod, 2018, Chaczko et al., 2014, Ma'ayan, 2018]. Figure 8.4 shows that approximately 27% of the

[¶] https://developers.mews.com/aaa-pattern-a-functional-approach/

[‖] https://freecontent.manning.com/making-better-unit-tests-part-1-the-aaa-pattern/

[**] https://java-design-patterns.com/patterns/arrange-act-assert/



(a) Contributors who add or remove AAA pattern violations

(b) Gherkin commits which added or removed AAA pattern violations

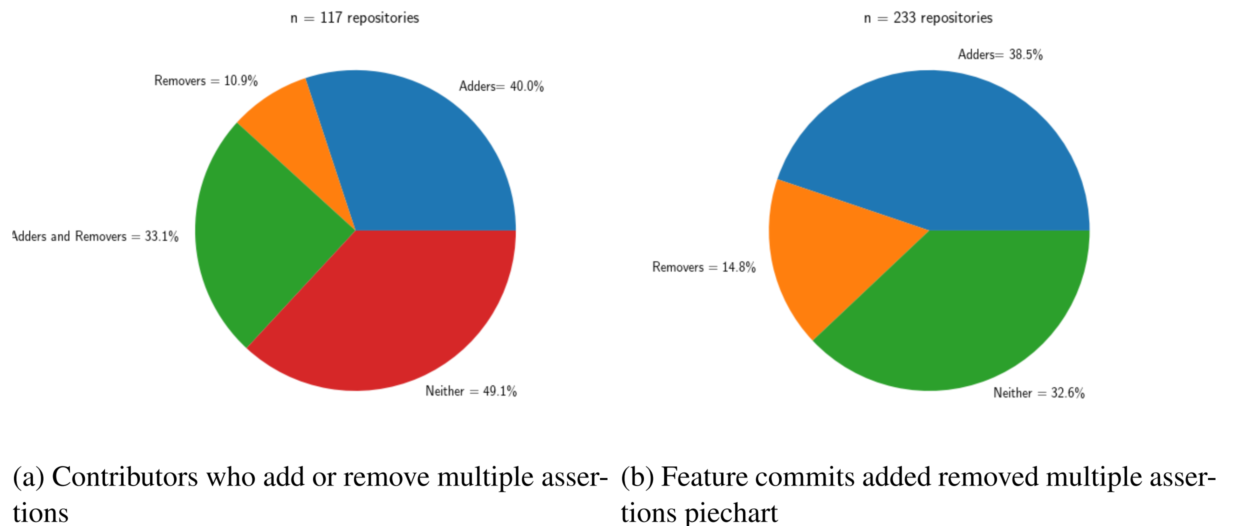
Figure 8.6: Pie charts showing addition and removal of AAA pattern violations

total contributors who introduced AAA pattern violations, introduced them on the first day, whereas approximately 16% of the contributors removed the AAA pattern violations within their first day of Gherkin commit.

Next, we calculated the proportion of the Gherkin contributors who added or removed AAA pattern violation bad smell, and the proportion of Gherkin commits in which this bad smell was added or removed. Figure 8.6a is a pie chart showing the distribution of Gherkin contributors on the basis of their role in adding or removing AAA pattern violations in Gherkin specifications. The figure represents the project contributors who made Gherkin commits to the projects. The figure shows that out of the total number of contributors who made Gherkin commits, 27.7% contributors added AAA pattern violations, 8.4% removed AAA pattern violations, while 63.9% neither removed nor added AAA pattern violations. Out of the total 36.1% (i.e., 27.7% adders and 8.4% removers), 25.1% were the contributors who were both adders and removers. This means that the remaining 11% is the sum of the contributors who either added AAA pattern violations or removed them.

There was no wide distinction between the adders and removers due to the overlapping between them. The majority of the developers who added AAA pattern violations also removed AAA pattern violations. Even though there was a major overlap between adders and the removers, we still wanted to know the proportion of adders and removers separately. Therefore, we plotted them as separate slices in the pie chart.

Figure 8.6b is a pie chart that illustrates the proportion of commits during which the AAA pattern violations were added or removed. The figure shows that 6.9% commits were made during which the AAA pattern violations were removed, AAA pattern violations were added during 20.4% commits, while during the 64.5% of the total Gherkin commits neither the AAA pattern violations were added nor removed.



(a) Contributors who add or remove multiple assertions

(b) Feature commits added removed multiple assertions piechart

Figure 8.7: Pie charts showing addition and removal of multiple assertions

8.3.2 Multiple Assertions

Apart from the benefits of AAA pattern, the discussion on the online blogs[††‡‡] revolves around the *use of single vs multiple asserts - per test* Results from an empirical study by Palomba et al. [2016] show that test smells frequently occur because of multiple assertions in a unit test. The authors also found that test smells have a strong correlation with the size of the system. Garousi and Küçük [2018] also list multiple assertions in a unit test as one of the main sources of bad smells. According to Knight[§§] multiple assertions within a Gherkin scenario often implies violation of *Cardinal Rule of BDD* i.e. more than one thing is being done in a single scenario.

We calculated the repositories with scenarios with more than one assertion by subtracting the total number of scenarios in a project from the total number of *Then* in a project. Figure 8.3 shows that 70% repositories had scenarios having more than one assertion, while 25% repositories did not have any scenarios with multiple assertions. There were 5% (i.e., 24) projects with scenarios greater than the number of total assertions which means *scenarios having no assertion at all* (i.e., incomplete scenarios). Figure 8.4 shows that 38% of the contributors who contributed to multiple assertions bad smell, introduced multiple assertions within the first day of their Gherkin commit. The figure shows that approximately 20% contributors removed multiple assertions within the first day of their Gherkin commit.

Next, we calculated the proportion of the Gherkin contributors who added or removed *multiple assertions* bad smell, and the proportion of Gherkin commits in which this bad smell was added or removed. Figure 8.7a is a pie chart of contributors who add or remove *multiple assertions* bad smell. The figure shows that 40% contributors added multiple assertions to the

[††]for example: `https://samueleresca.net/2017/08/maintainable-unit-tests/`

[‡‡]`https://jamescooke.info/arrange-act-assert-pattern-for-python-developers.html`

[§§]https://automationpanda.com/2018/01/31/good-Gherkin-scenario-titles/

```
Feature: Check Email

        Scenario: Read an email from inbox
                Given The user has an account
                And the user is logged in
                When user selects an email to read
                Then the email is displayed

Feature: Reset password

        Scenario: User requests a a password reset
                Given The user has an account
                And the user is logged in
                When the user submits a password reset request
                Then the password is successfully reset
```

Figure 8.8: Example of Duplication

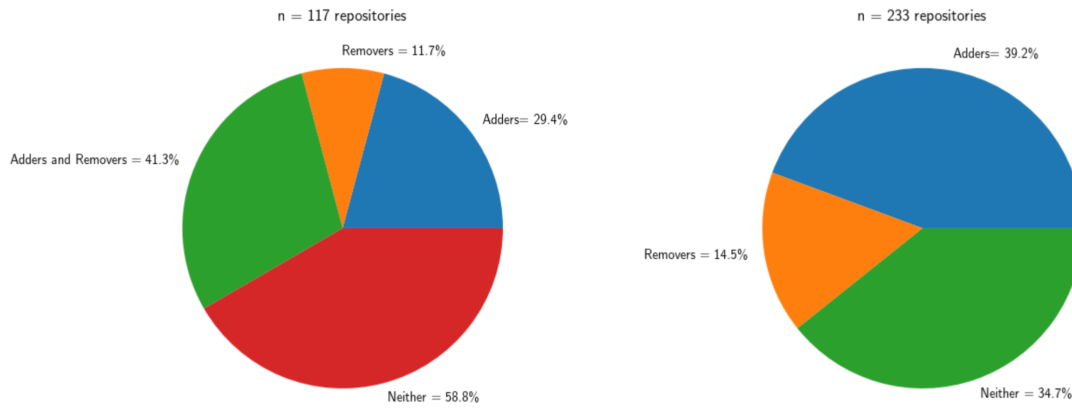
projects, 10.09% contributors removed multiple assertions and 49.1% Gherkin contributors neither added nor removed multiple assertions. Out of the total 50.09% (i.e., 40% adders and 10.09% removers), there were 33.1% overlapping contributors who performed both functions i.e., addition and removal of multiple assertions to the scenarios.

Figure 8.7b is a pie chart showing the proportion of commits in which multiple assertion bad smell was removed or added in the projects with multiple assertions bad smell. The figure shows that multiple assertions were added in 38.5% of the Gherkin commits made in 233 projects. During 14.8% commits, multiple assertions were removed, whereas no *multiple assertions* were added or removed in the rest of the 32.6% Gherkin commits.

8.3.3 Duplication of Gherkin steps

Figure 8.8 illustrates an example for understanding step duplication. There are two consecutive and identical *Given* steps across two scenarios in the figure. When two or more identical steps start appearing together at various places in a Gherkin test suite, it indicates a potential for implicit dependency between those steps. This dependency between the steps is similar to the *routine call coupling* in programming where one function calls another function. The problem, in this context, appears when a change in a clone is required. In case a change is required in one place, all the other instances of the clones will have to be checked manually for potential changes. In case of a large test suite manually checking every scenario in every feature file could be nearly impossible.

In a study, Suan [2015] discussed the issue of duplicates in BDD scenarios. This research work was focused on the detection and removal of textual duplication. The author discussed the *good* and *bad* duplication in the BDD scenarios. The study was focused on the duplication within a feature file whereas, the duplicates across the feature files were out of the scope of the study. According to the author, the duplicates are "*good*" when the scenarios, in which



(a) Contributors who add or remove clones (b) Feature commits added removed clones piechart

Figure 8.9: Pie charts showing addition and removal of clones

the duplicates exist, have different pre-condition(s) (i.e., *Given* step(s)) and/or outcome(s) (i.e., *Then* step(s)). We, however, know that the steps that are common between all the scenarios in a feature file should be put in the *Background*. Any duplicates other than the ones merged in *Background* within a feature file are unavoidable i.e., when the duplicate steps are not common to "*all*" the scenarios in a feature file. Only because something is unavoidable does not make it "*good*" or acceptable. Moreover, the argument provided by the author becomes invalid when considering the duplicates across feature files. In fact, there are no good or bad duplicates. All duplication is bad when it comes to making a change to duplicate steps manually. The issue of duplication in Gherkin specifications is also recognised by Binamungu [2020]. This research work was focused on the detection of semantic duplicates instead of textual matching.

Each occurrence of two more consecutive and identical steps was counted as one *duplicate* or one *clone* during our calculation. Figure 8.3 shows that 69% of the open-source Gherkin projects on GitHub have at least one clone in their test suite. Figure 8.4 shows that 40% of the Gherkin contributors introduced the duplicate steps within the first day of their Gherkin commit, whereas 23% removed the duplicate steps on the first day.

Next, we calculated the proportion of the Gherkin contributors who added or removed *duplicate steps* bad smell, and the proportion of Gherkin commits in which this bad smell was added or removed. Figure 8.9a shows that 29.4% of the contributors who made Gherkin commits in the projects with duplicates (i.e., clones), added duplicate steps, whereas 11.7% removed duplicates. Out of the total adders and removers i.e., approximately 41%, all of the contributors were adders and removers. This means there were no contributors who only added duplicate steps or only removed duplicate steps.

We plotted a pie chart to differentiate between the commits that added the duplicate steps bad smells and the commits that removed the bad smells. Figure 8.9b shows that 39.2% Gherkin commits contributed to the duplicate steps bad smell, whereas 14.5% commits were made in

```
Scenario: Number of books by author
        Given I have a book in the store called "name"
        When I search for books by "author"
        Then I find "number" books
```

Figure 8.10: Example of step parameters

```
Scenario: translate text
        Given a sample text loaded into the frobulator
                 """
                Hallo, das ist ein Satz auf Deutsch.
                """
        When Bob presses translate
        Then the text in the frobulator is translated into English as
                 """
                Hello this is a phrase in German.
                """
```

Figure 8.11: Example of a step data (text)

which duplicate steps were removed. During the rest of the 34.7% Gherkin commits no duplicate steps were added or removed.

8.3.4 Lazy Steps Data Table:

The basis of the bad smell discussed in this section is the overuse of a capability in Gherkin called "*paramterised BDD steps*". When using this capability, parameters are specified in Gherkin steps using double (") or single (') quotes. For example, Figure 8.10 shows an example of a scenario through which a person can search the number of available copies in a store of a book written by an author. The above scenario can become cluttered or suffer duplication when multiple data values are added to a step. For this purpose, Gherkin provides *Doc strings* and *Data Tables*. *Doc strings* or *Data Tables* are used when we want to pass more data to a step than what can fit on a single line.

Figure 8.11 is an example of *Doc strings* which describes a scenario with step argument where a block of text is attached to a *Given* step. The *Given* step in the figure takes a block of text as an input and passes it as an attribute to a step function. Figure 8.12 describes a step function for the scenario described in Figure 8.11. The block of text is passed as an attribute into the context variable in the step function in the figure for manipulation.

Data tables are used for passing a list of values to a step definition (i.e., step implementation code). A table of data with some values in rows and columns is passed as a variable to a step function for manipulation. Figure 8.13 is an example of a scenario with a data table attached to one of its steps. The scenario in the figure counts the number of people in various departments. A data table is attached to the Given step. This data table will be passed to a step function exactly like the example in Figure 8.12.

```
@given('a sample text loaded into the frobulator')
def step_impl(context):
        .........
        .........
        .........
```

Figure 8.12: Step function for scenario in Figure 8.11

```
Scenario: Count people in departments
        Given a set of specific users
                | name                   | department    |
                | Barry                  | Sales         |
                | Bob                    | New Items     |
                | Peter                  | New Items     |
        And Sara is an admin
        When Sara counts the number of people in each department
        Then Sara will find two people in "New Items"
        And Sara will find one person in "Sales"
```

Figure 8.13: Example of a step data table

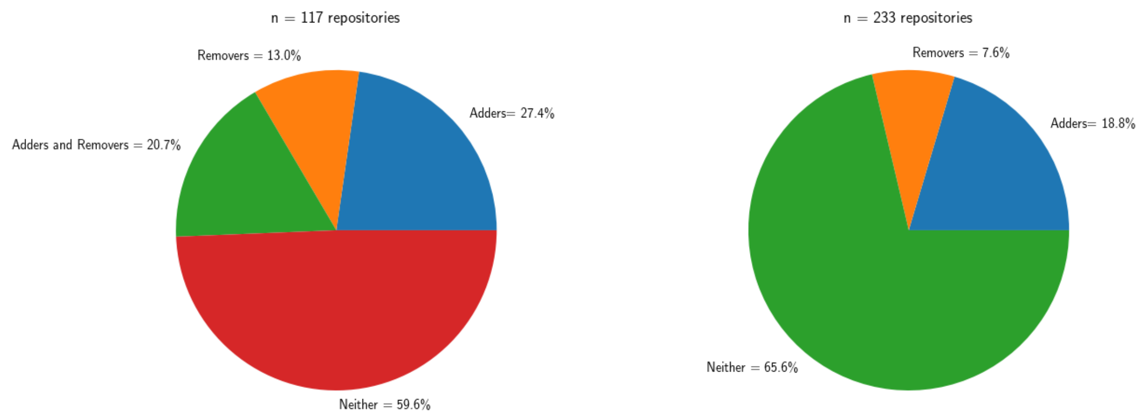
Implementation for passing a data table is more complicated than passing a single value as a parameter in BDD. Therefore, the use of a step argument is inappropriate if the data table consists of a single line. In case of a single line in a step argument table, the table must be removed and data should be written as part of the relevant step phrase i.e., inline within a sentence. Since we did not find any literature on lazy steps bad smell in Gherkin, this phenomenon could be better understood by borrowing the concept behind *Lazy Class* bad smell.

According to Fowler [2018], a *Lazy Class* represents a class that is under-used. Such classes could impede development by adding unnecessary complexity to the code. Lazy class is one of the code smells which cause clutter in the code. A lazy class must be removed and merged with another class. The same concept applies to lazy steps in Gherkin specifications. Lazy steps in Gherkin mean that data which could be written as part of a step is being attached to the step as an argument instead. Such steps must be refactored and rewritten.

Refactoring the scenarios is the same as refactoring the code. Refactoring improves maintainability [Kim et al., 2012]. In a recently published study, Irshad et al. [2022] emphasise on importance of refactoring of BDD specifications. According to Borg and Kropp [2011], changes in specifications are inevitable because of the accommodation of frequent changes in agile development.

Each occurrence of a single row in a data table attached to a step as a step argument is counted as one lazy step. Figure 8.3 shows that 29% of the open-source Gherkin projects have at least one occurrence where a scenario step argument should be refactored into a scenario step. Figure 8.4 shows that 29% of the contributors who contributed to the *lazy steps* bad smell, added lazy steps within the first day, whereas 17% removed the *lazy steps* bad smell.

Next, we calculated the proportion of the Gherkin contributors who added or removed *lazy*



(a) Contributors who add or remove lazy step examples piechart

(b) Feature commits added removed lazy step examples piechart

Figure 8.14: Pie charts showing addition and removal of lazy steps

steps bad smell, and the proportion of Gherkin commits in which this bad smell was added or removed. Figure 8.14a is a pie chart that shows that 27.4% contributors added lazy steps to the scenarios in open-source Gherkin projects, whereas 13% contributors removed lazy steps. There were 20.7% contributors who performed both additions and removals of lazy steps. Majority of the contributors i.e., 59.% neither added nor removed the lazy steps. Figure 8.14b is also a pie chart which shows that 18.8% Gherkin commits contributed to the addition of lazy steps, whereas during 7.6% Gherkin commits, lazy steps were removed or refactored. During a significant percentage of Gherkin commits (i.e., 65%), the lazy steps bad smell was neither added nor removed.

8.3.5 Lazy Scenario Outline

According to the documentation provided for Cucumber[¶¶], the *Scenario Outline* keyword can be used for running the "*same scenario multiple times with different combinations of values.*" Figure 8.15 is a simple example of a *Scenario Outline* where two scenarios are collapsed into one *Scenario Outline*. Each row in the table refers to a single scenario which means that the *Scenario Outline* will run two times. Every time the *Scenario Outline* executes, the references are replaced with the data from the row that is being run.

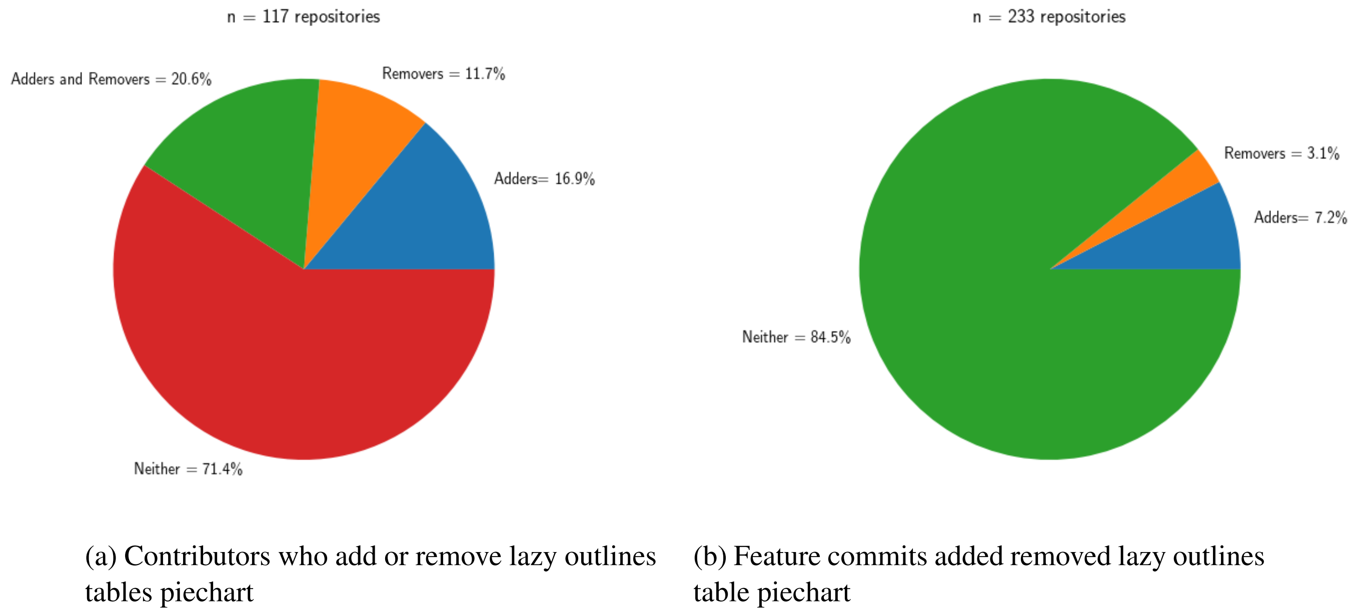
Knowing when to use *Scenario Outline* is as important as to know *when not to use it*. Scenario Outlines are used when there are two or more identical scenarios (with different data) in a feature file but it should not be used in case of a single scenario. A single row in a scenario outline is an indication that the scenario outline is under-used. This problem is identical to the problem behind Lazy Class bad smell explained in the previous section. Any *Scenario Outline* with a single row is an indication of a need for refactoring. Such *Scenario Outline* must be

[¶¶] https://cucumber.io/docs/gherkin/reference/

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

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

Figure 8.15: Example of a scenario outline



(a) Contributors who add or remove lazy outlines tables piechart

(b) Feature commits added removed lazy outlines table piechart

Figure 8.16: Pie charts showing addition and removal of lazy outline tables

refactored and replaced by a scenario with inline data. A *Scenario Outline* with a single row means that it is a single scenario therefore it should be written as such.

Each occurrence of a scenario outline with a single row in the table was counted as one *lazy scenario outline* bad smell. Figure 8.3 shows that 12% of the open-source Gherkin projects have at least one occurrence of lazy outline. Figure 8.4 shows that from the contributors who introduced lazy *Scenario Outline* tables into the Gherkin projects, 15% introduced lazy *Scenario Outline* tables bad smell within the first day, whereas 25% of the contributors removed lazy *Scenario Outline* tables within the first day of their Gherkin commit.

Next, we calculated the proportion of the Gherkin contributors who added or removed *lazy scenario outline* bad smell, and the proportion of Gherkin commits in which this bad smell was added or removed. Figure 8.16a is a pie chart showing the proportion of Gherkin contributors who added or removed lazy scenario outlines. The figure shows that 16.9% of the total Gherkin contributors added lazy *Scenario Outline* tables to open-source projects' Gherkin specifications, whereas 11.7% contributors removed lazy *Scenario Outline* tables. Out of the 28.6%

(i.e., 16.9% adders and 11.7% removers), there were 20.6% contributors who performed both i.e., additions and removals of *Scenario Outline* tables. A significant percentage i.e., 71.4% of the total Gherkin contributors neither added nor removed lazy *Scenario Outline* tables. Figure 8.16b is a pie chart showing the proportion of Gherkin commits in which lazy outlines were added or removed. The figure shows that 7.2% commits out of the total Gherkin commits in the open-source projects contributed to the addition of lazy *Scenario Outline* bad smell, whereas 3.1%. commits were made during which this bad smell was removed. However, an overwhelming majority (84.5%) consisted of commits during which neither the *lazy scenario outline* bad smell was added nor removed.

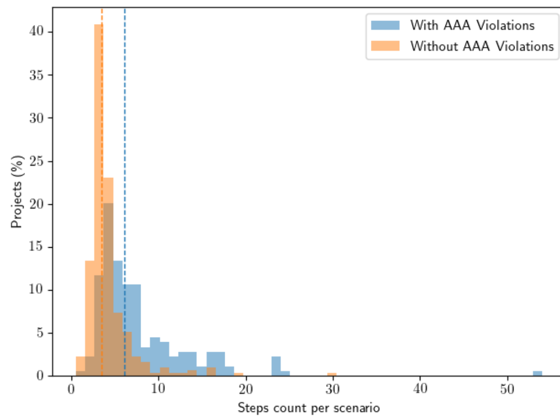
8.4 Relationship between Gherkin Specifications Bad Smells and Other Gherkin Artefacts

In order to understand the potential relationship between the selected bad smells in Gherkin specifications in our dataset and other Gherkin artefacts in our dataset (e.g., size of scenarios), we plotted several sets of graphs. These graphs include: comparison histogram of scenario sizes in the projects with and without a particular bad smell, time series of bad smell density, comparison histogram of *life in the project* of Gherkin contributors who contributed or did not contribute towards a bad smell, and scatter chart of bad smell density versus the number of contributors.

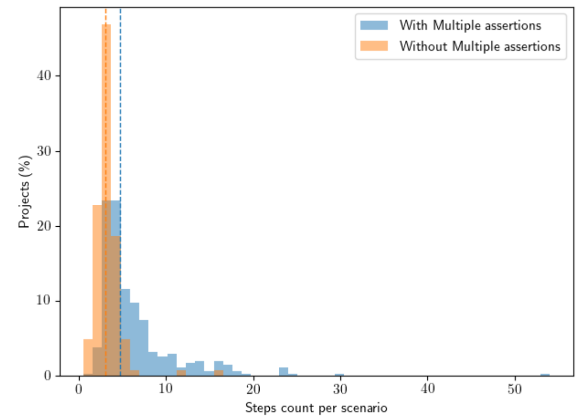
8.4.1 Relationship with the size of scenarios

Figure 8.17 represents a comparison between the length of scenarios in the projects with and without bad smells. The figure consists of five sub-figures. Each sub-figure contains two overlapping histograms. The X-axis shows the average size of scenarios and Y-axis shows the percentage of projects. One histogram represents scenario sizes in projects with one of the five bad smells discussed in this chapter. The second histogram represents average scenario sizes in projects which do not contain the respective bad smell. These two histograms in each sub-figure compare the average sizes of scenarios in projects with and without each bad smell discussed in this chapter. Visually, scenarios in the projects with bad smells seem to be slightly longer than scenarios in the projects without the respective bad smells.

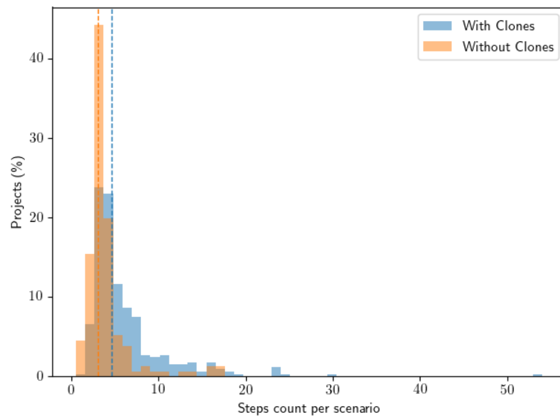
To confirm this observation we calculated the median scenario sizes for both histograms in each sub-figure. Figure 8.17a shows that the median number of steps in a typical scenario in projects with AAA pattern violations in our dataset was 6, whereas the median number of steps in the projects with no AAA violations was 4 (rounded to the closest integer). Similarly, the number of steps in a typical scenario in projects with multiple assertions in Figure 8.17b was 5, whereas in the projects with no multiple assertions, the median number of scenario steps



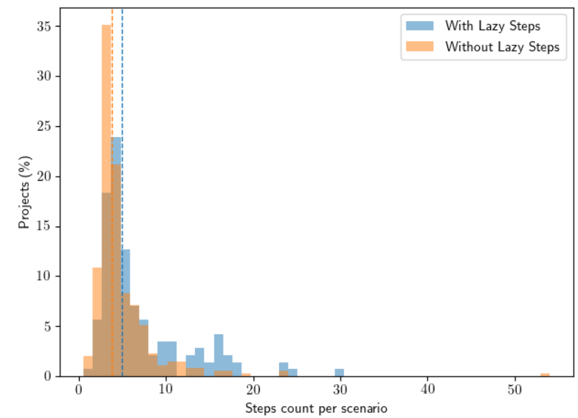
(a) With (n=179, median=6.1, blue) and without (n=313, median=3.5, orange) AAA Violations.



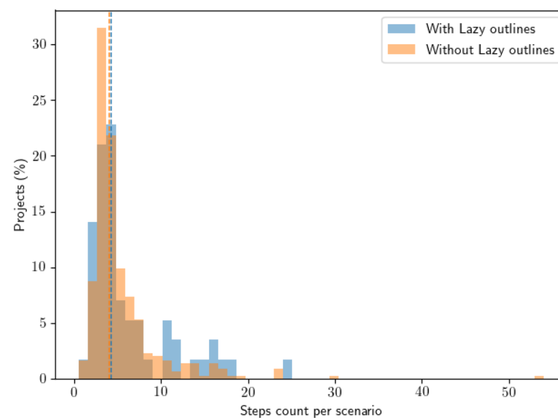
(b) With (n=347, median=4.8, blue) and without (n=145, median=3.0, orange) Multiple assertions.



(c) With (n=336, median=4.6, blue) and without (n=156, median=3.0, orange) Clones.



(d) With (n=142, median=4.9, blue) and without (n=350, median=3.8, orange) Lazy Steps.



(e) With (n=57, median=4.2, blue) and without (n=435, median=4.0, orange) Lazy outlines.

Figure 8.17: Histograms of scenario sizes for repositories with and without selected smells

was 3. Figure 8.17c shows that the median number of steps in a typical scenario in projects with duplicates is 5, whereas in the projects without duplicates, this number is 3. Figure 8.17d shows that the median number of steps in a typical scenario in projects with *lazy steps* bad smell is 5, whereas in the projects with no lazy steps bad smell, the median number of scenario steps is 4. Figure 8.17e shows that in the projects with and without *lazy outline* bad smell, the number of median steps in a typical scenario is roughly equal i.e., 4. These observations imply that projects with bad smells have slightly longer scenarios than projects with no bad smells. Unfortunately, using the data we have, we are unable to draw any conclusion on *whether or not a longer scenario size causes these bad smells*.

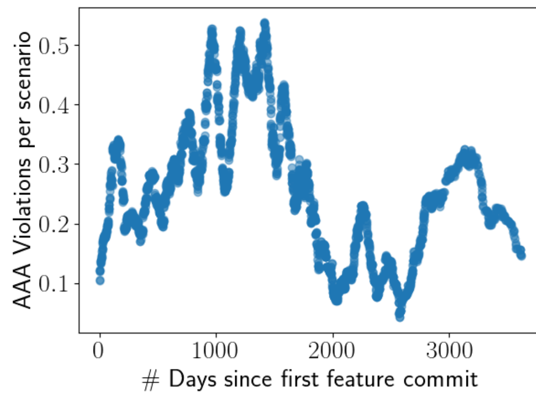
Next, we wanted to know if longer projects have a greater number of bad smells. So, a set of scatter charts was plotted to understand the density of bad smells (i.e., average bad smell count per scenario) over the lifetime of a typical Gherkin project. Figure 8.18 is a set of sub-figures that show a time series of an average number of bad smells. Each of these sub-figures was plotted using the projects with a bad smell and a reasonable commit history. By *reasonable commit history* we mean the repositories which had a minimum of 100 commits, minimum life duration of 100 days, and project starting in or after January 2005 (i.e., after the conception of BDD).

Figure 8.18a was plotted using 116 projects with AAA pattern violation bad smell that also had a reasonable commit history. Similarly, Figures 8.18b, 8.18c, and 8.18d were plotted using 174 projects for multiple assertions, 172 projects for duplicates (or clones), and 82 projects for lazy steps respectively. Figure 8.18e was plotted using 38 projects with *lazy outlines* that also had a reasonable commit history. The shapes of the sub-figures do not show any correlation between the average count of occurrence of bad smell in a typical scenario and the age of the projects. We see a periodic cycle of increase and decrease in the graphs but if we look at the Y-axis the increase and decrease seem to be negligible (i.e., in fractions).

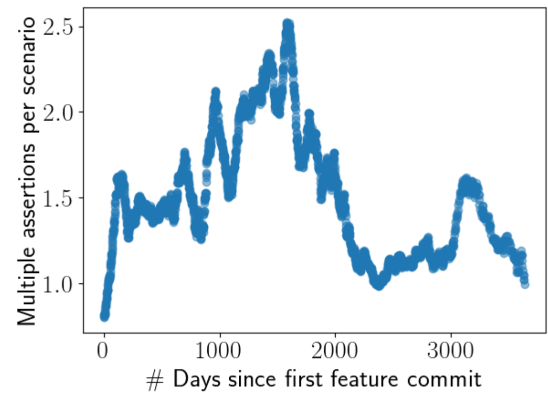
8.4.2 Relationship with contributors

It was observed during the study documented in Chapter 6 that bad smells are associated with factors associated with project contributors. These factors could include: prior experience in BDD, specification writing styles, background knowledge of BDD etc. First, we wanted to understand if the number of contributors involved in a project has an impact on the bad smells discussed in this chapter. To understand this, we plotted scatter charts of the total number of bad smells in a project against the number of contributors in a project using Figure 8.19.

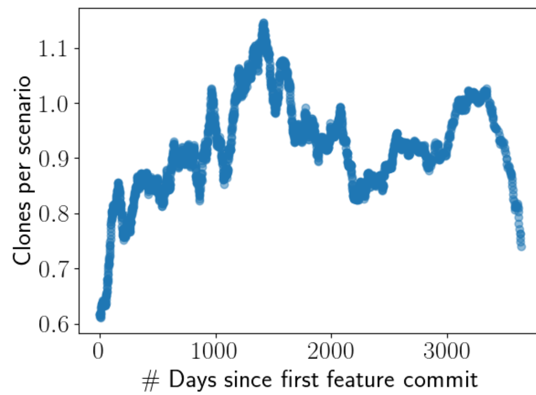
Figure 8.19 is a set of sub-figures each of which is a scatter chart. The X-axis shows the number of contributors and the Y-axis shows the number of bad smells. Each dot on these scatter charts represents the total count of a bad smell in a project versus the total number of contributors in a project. However, the data in the graphs is widely dispersed and the shape of the graphs shows no relationship between the number of bad smells and the number of contributors



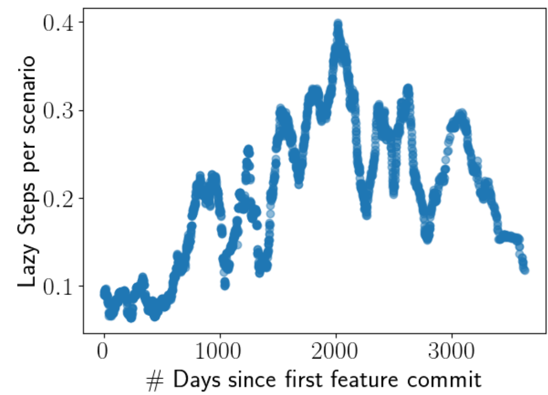
(a) AAA Violations, 116 repositories



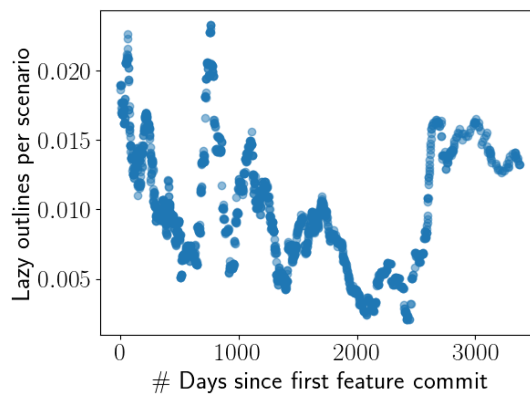
(b) Multiple assertions, 174 repositories



(c) Clones, 172 repositories

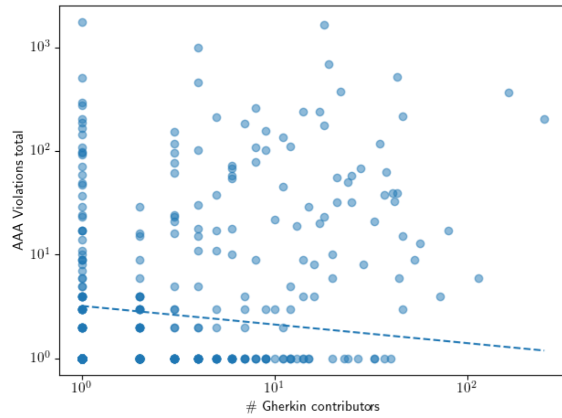


(d) Lazy Steps, 82 repositories

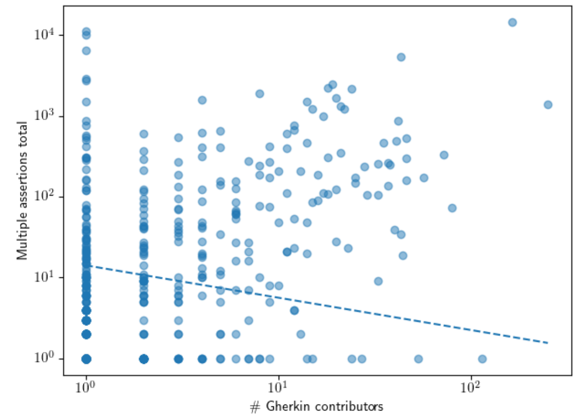


(e) Lazy outlines, 38 repositories

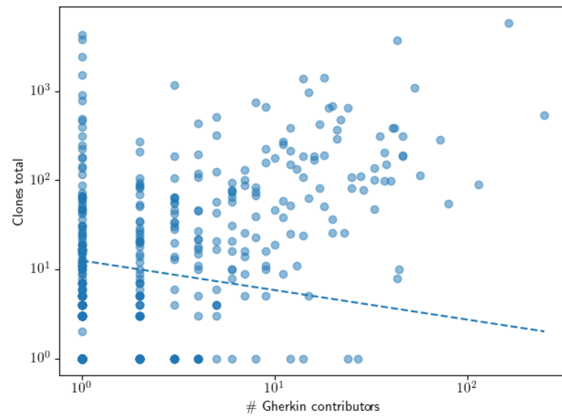
Figure 8.18: Time series of project smell density



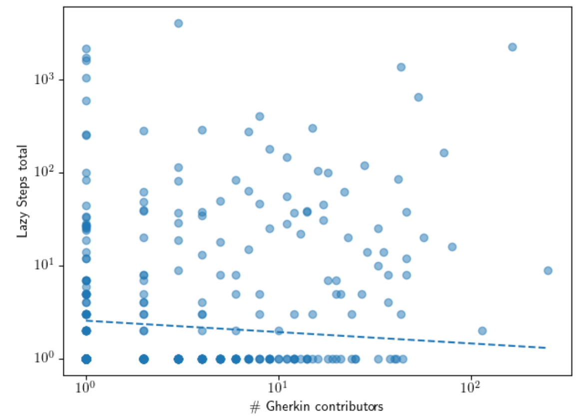
(a) AAA Violations, 493 repositories



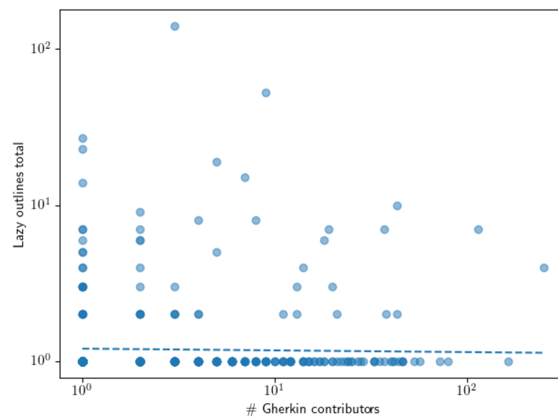
(b) Multiple assertions, 493 repositories



(c) Clones, 493 repositories



(d) Lazy Steps, 493 repositories



(e) Lazy outlines, 493 repositories

Figure 8.19: Scatter plots of project total smell counts against project gherkin contributor counts

in a project. Although the mean line shows some minor reduction, this is an arithmetic mean and the overall data is scattered so it does not show any correlation. It means that the number of contributors in a project does not impact the bad smells such as AAA pattern violations, Multiple assertions, Clones, Lazy steps, and Lazy outlines.

Second, we wanted to see if there is a difference between the life (i.e., time in the project) of the contributors who introduce bad smells and the life (i.e., time in the project) of the contributors who do not introduce bad smells. Figure 8.20 is a histogram of responsibilities of the contributors who introduce bad smells versus the contributors who do not introduce bad smells. The sub-figures represent the percentage of the contributors involved over the lifetime of the project(s). The only noticeable difference in the average is in the Figure 8.20d. To justify this, we can only speculate that it is possible that the contributors learned step argument capability in Gherkin sometime later during the project and started using it which also led to the introduction of lazy step data tables bad smell at a relatively later stage in the project. The other figures show no difference between the involvement of the contributors who introduce bad smells and the involvement of the contributors who do not introduce smells.

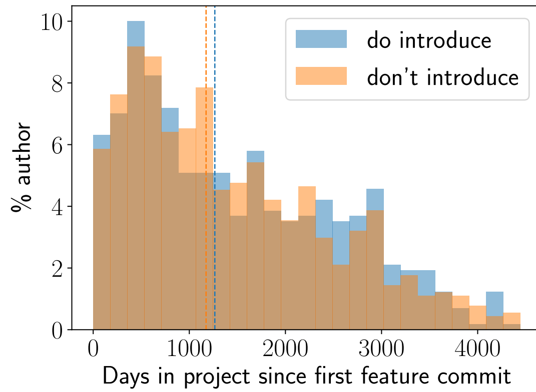
Third, to see if the duration of contributors' involvement has any impact on the introduction of bad smells, we drew a scatter chart of the bad smells introduced by contributors versus the age of contributors in the project(s) using Figure 8.21. The Y-axis in the figures shows the total count of bad smells and the X-axis shows the number of days passed since the first feature was introduced in a project. Each dot in each scatter chart represents the total number of bad smells introduced by a contributor and the number of days that the contributor was active in a project. The data appears to be widely dispersed which means that the bad smells have no relationship with the duration of involvement of the contributors in the project(s). The concentration in the scatter plots only shows that more authors are involved at the start of a project.

8.5 Threats to Validity

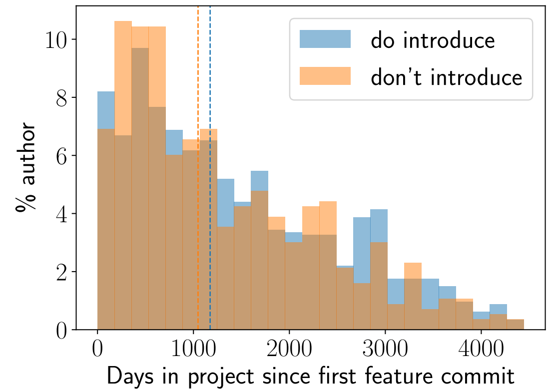
This section discusses the threats to the validity of this study. In this section, we have discussed three types of validity threats i.e., construct, reliability, and external validity.

Construct validity threat

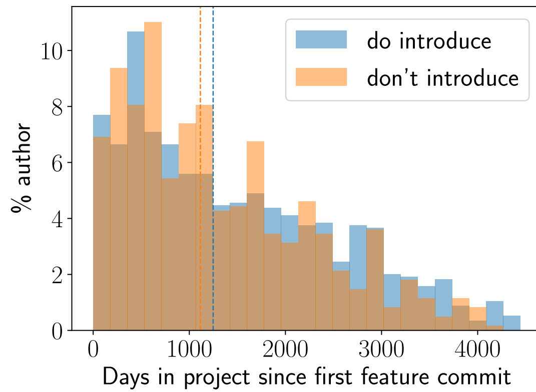
A potential construct validity threat for this study could be an incorrect definition of a *bad smell* and the context of this study. If "*what constitutes a bad smell*" is not described appropriately, many of the *bad smells* that do not fit in the description may be ignored. Also, since the literature on bad smells in Gherkin scenarios was unavailable and a list of potential candidates for bad smells in Gherkin scenarios was adopted from the literature on bad smells in unit tests, it is possible that some of the potential bad smells specific to Gherkin scenarios were not discovered.



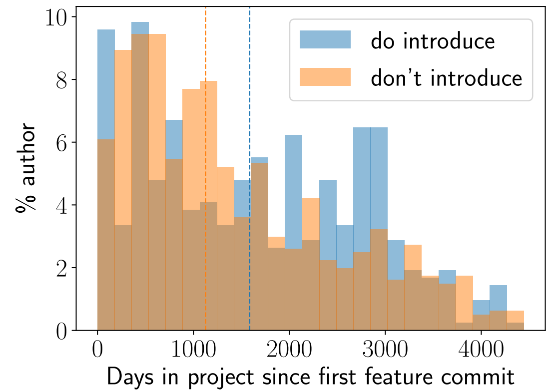
(a) Authors who do (blue, n=570) and don't (orange, n=904) make AAA violations.



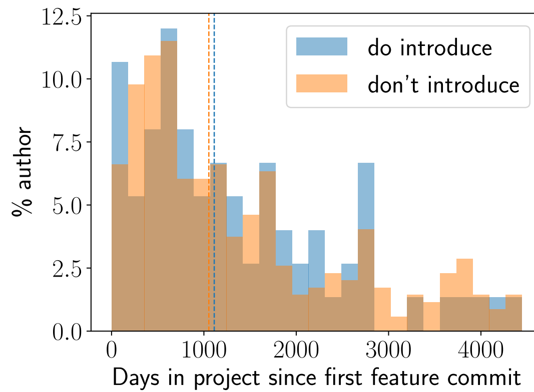
(b) Authors who do (blue, n=1135) and don't (orange, n=565) make Multiple assertions.



(c) Authors who do (blue, n=1143) and don't (orange, n=608) make Clones.



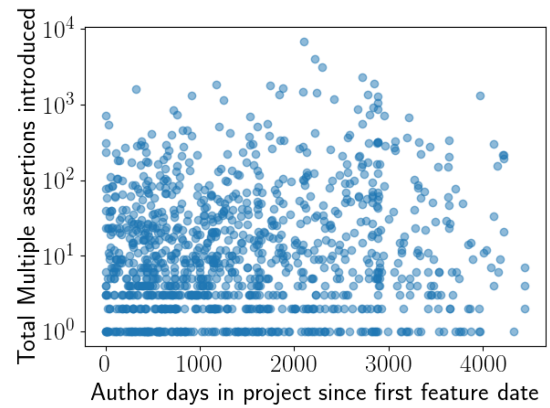
(d) Authors who do (blue, n=417) and don't (orange, n=805) introduce lazy step tables.



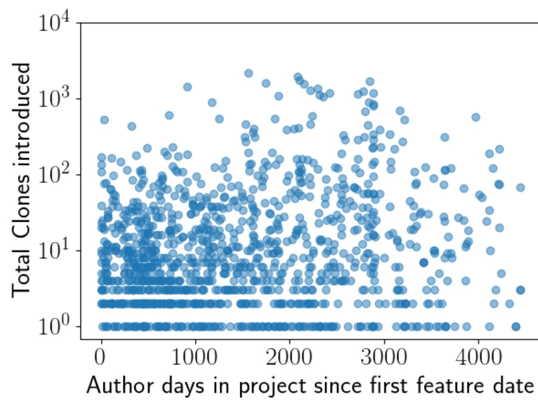
(e) Authors who do (blue, n=75) and don't (orange, n=348) introduce lazy outlines table.

Figure 8.20: Histogram of time working on a repository for commit authors who do and do not make changes to selected smells for 274 repositories with known history

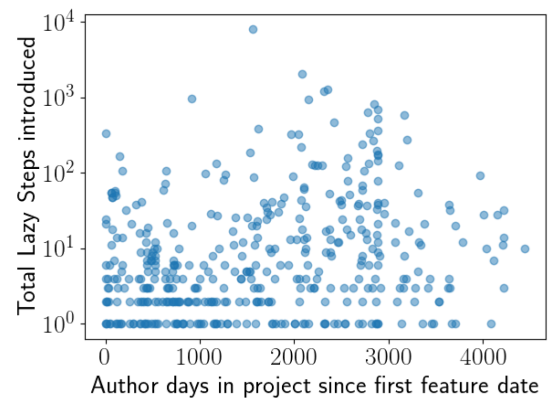
(a) Scatter plot of total AAA Violations introduced versus author days in project.



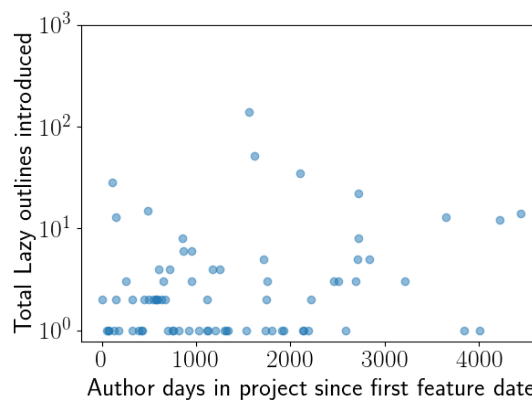
(b) Scatter plot of total Multiple assertions introduced versus author days in project.



(c) Scatter plot of total Clones introduced versus author days in project.



(d) Scatter plot of total Lazy Steps introduced versus author days in project.



(e) Scatter plot of total Lazy outlines introduced versus author days in project.

Figure 8.21: Scatter plots of total smells introduced versus author days in a project for 201 projects

At the start of this study, the term *bad smell* was defined in the context of this study clearly. Then after not being able to find literature on bad smells in Gherkin, bad smells in unit tests were analysed for their applicability in the context of Gherkin. Finding an exhaustive list of bad smells and then deciding which bad smell to investigate may also be a construct validity threats for this study. Various research studies were found which discussed some bad smells in the context of unit tests. The number and nature of bad smells discussed in each study was different from each other. It was important to have an exhaustive list of bad smells in unit tests. The next step was to decide which bad smell to investigate. It was important to understand the context and applicability of each bad smell in the given context.

Reliability Validity Threat

The errors or inconsistencies in collecting data for constructing the graphs can threaten the reliability of this study. The ambiguity in the criteria for identifying bad smells can lead to inconsistent application of the criteria. The errors in the collection and analysis of Gherkin files can lead to the loss of important data. The identification and interpretation of bad smells can also introduce errors in the results and compromise the reliability of the study.

External Validity Threat:

The findings of the study may not be applicable to other programming languages, as Gherkin is specific to Behaviour Driven Development (BDD) and may not represent the broader software development community. Also, we conducted this study using only one platform i.e., GitHub. We did not conduct further investigation on another platform such as GitLab.

Addressing Threats to Validity

We did not find an exhaustive list of bad smells in unit tests in peer-reviewed literature, so we searched grey literature. We found an exhaustive list of bad smells in unit tests in grey literature. Each of the bad smells in the list was carefully analysed for its applicability in the context. A total of 11 bad smells were selected from a list of 182 bad smells. Only five of the 11 bad smells were selected for further investigation because the scope of six of the bad smells was either too broad or would require work which was not possible within the limited time for this Ph.D research. A robust data collection process that minimises the chances of data loss or missing data was implemented through the Python tool. This shows that we followed a systematic process for defining *bad smells* and then deciding which bad smells to investigate.

8.6 Summary

In computer science, the structural inflexibilities in software artefacts are referred to as *bad smells*. They can emerge in various software artefacts such as code, design, architecture, and in this case, specifications. The bad smells do not stop the software system from executing but make the evolution of the software system very difficult. Bad smells are one of the indications of the need for refactoring.

This chapter discusses bad smells in Gherkin specifications. Five bad smells were selected for the analysis on the basis of applicability and feasibility (as explained in Section 8.2.2). These bad smells were:

(i). **violation of AAA pattern**: (in this context) means mixing of Given-When-Then step order. According to Oliveira and Marczak [2018] means Given-When-Then step order makes the scenarios unreadable. Violation of the AAA pattern could potentially impact the readability and maintainability of Gherkin specifications.

(ii). **multiple assertions**: or assertion roulette (in this context) means having more than one than one assertion in a Gherkin scenario. Multiple assertions in a scenario could mask bugs because in case of a failure of an assert, it is impossible to know the pass or fail status of the subsequent asserts [Tufano et al., 2016].

(iii). **duplication of steps**: (in this context) means the repetition of two or more consecutive steps across scenarios. Such steps are difficult to maintain. If a change occurs in such steps, it has to be reflected in its *clones* as well. Doing this task manually involves the risk of missing a few duplicates where a particular change was supposed to be reflected as well. Duplicate steps in Gherkin impact the maintainability of Gherkin files.

Dealing with duplicates is more difficult when the duplicates occur across feature files. One potential way of addressing this issue is to include a "*referencing"* feature (similar to the concept of *referencing* in programming) in Gherkin tools. If implemented, through this functionality, each set of duplicated steps may be written in a separate file and referenced where required. This could minimise the need for making changes at multiple places when such a need arises.

(iv). **lazy steps**: (in this context) means overuse of *step data*. Single line data tables in Gherkin steps should be refactored and the data should be put *within* the respective steps. Having unnecessary tables amounts to overuse of the *step data* capability and is similar to the lazy class bad smell.

(v). **lazy outlines**: means overuse of *scenario outlines*. This refers to scenario outlines with a single row. According to the documentation on Cucumber, scenario outline "*...steps are*

interpreted as a template which is never directly run" [CucumberStudio, 2019a]. Overuse of scenario outlines leads to slow Cucumber tests.

Figure 8.3 shows that a significant proportion of Gherkin projects contain these bad smells. Further analysis showed that the bad smells are not correlated with the size or duration of the project, the size of scenarios, the number of contributors in the project, or the duration of contributors' involvement in the project. Also, there is no difference between the duration of involvement of the contributors who introduce bad smells in the project and the duration of involvement of the contributors who do not introduce bad smells in the project.

We made two observations during this study. First, we learned that there is a lack of research on bad smells in Gherkin specifications. This observation was made while searching for literature to back our arguments in the sections explaining each bad smell. Second, we learned that Gherkin provides a general Given-When-Then format for expressing the steps but it does not put any constraints on how these steps should be used. In other words, Gherkin provides a lot of freedom as a result of which it is easy for the developers to write specifications that are functionally correct but structurally inflexible i.e., contain bad smells. Also, the research community has not established quality criteria for writing Gherkin specifications. Oliveira et al. [2019] proposed a question-based checklist to assess the quality of BDD specifications. However, the quality criteria described by Oliveira et al. [2019] is subjective and requires experience and understanding of the person writing Gherkin. For example, *Not too many details* is hard to determine and requires judgement of the person writing Gherkin scenarios.

We believe the existence of bad smells on Gherkin specifications is dependent upon the contributors' way of writing Gherkin specifications. Out of the contributors who added bad smells to the projects, a significant percentage of contributors added the bad smells within the first day of their Gherkin commit.

Chapter 9

Conclusions

This thesis proposed that the practice of Behaviour Driven Development, its associated artefacts, and the intrinsic nature of the associated language, Gherkin may incur significant additional overhead for software engineers in terms of technical debt and ongoing maintenance in the real-world software engineering environment. On the basis of the analysis of results from an exploratory case study on the use of agile methods in large-scale safety-critical projects, BDD was explored for its potential for application in requirements management. An exploratory case study involving action research and semi-structured interviews, and an experiment, presented in Chapters 6 - 8, was conducted to understand and evaluate Behaviour Driven Development (BDD). The purpose of the investigation was to explore the challenges associated with BDD so that this research could serve as a guide for people planning to employ BDD.

The next three sections of this chapter address the research questions presented in Chapter 1. Section 9.4 discusses the research contributions made during this Ph.D. Section 9.5 explains the interconnection of the case studies and the experiment presented in this thesis. The section also explains how the challenges associated with BDD were discovered which could provide guidance to the people intending to employ BDD. Section 9.6 examines the scope and the validity of this research followed by a discussion on the limitations of this research in Section 9.7. Section 9.8 discusses the research implications, Section 9.9 presents areas for future work, and Section 9.10 concludes the chapter.

9.1 Thesis Research Question 1

This Ph.D research started with studying application of agile methods for development of large-scale safety-critical systems. Studying relevant literature revealed the perception that agile, in its traditional form, is unsuitable for development of safety critical systems. We also learned that there is a lack of empirical research in the said field which made us want to conduct an empirical study. The focus of the empirical study was on exploring the challenges of using an agile method in the context of safety-critical systems development. The answer to the first research question,

"*What are the challenges of adopting agile methods in large-scale software development activities, particularly in regulated environments?*"

can be derived from the exploratory case study presented in Chapter 4. This study reports the results of a series of interviews and workshops in a large avionics company who were experimenting with the incorporation of agile software development into their software development process. The research yielded 13 challenges faced by different software teams interviewed during the study concerning the application of agile software development for safety-critical systems. The challenges can be grouped into three categories:

- the influence of wider Waterfall like systems engineering processes on the practice of agile software development within a single team;
- the necessarily complex interactions with external stakeholders, including multiple customer roles; and
- the demand for documentation to meet required regulatory standards.

We also found that cultural resistance within the company was a cross-cutting concern, limiting the use of elements of agile software development.

The semi-structured interviews undertaken within the exploratory study revealed that agile in its traditional form is unsuitable for use in large-scale safety-critical development context. The requirements (such as quality assurance requirements) mandated by the regulatory standards and the complex dynamics of large-scale systems limit the freedom offered by agile. An agile process needs to be tailored according to the organisational and project-related needs.

A number of case studies and experience reports in the academic literature have reported on the adoption of agile in diverse domains, including railways [Jonsson et al., 2012], medical science [McHugh et al., 2013] and most relevant to the present research, avionics [Wils et al., 2006, Chenu, 2012]. Many of these studies conclude that agile software development requires adaptation for application to safety-critical systems. For example, Notander et al. [2013] conclude that agile software development, while not incompatible with typical safety-critical standards, needs to be modified for use on safety-critical system projects. The practice of adapting and customising methods and practices to suit local needs has been reported for other software domains [Fitzgerald et al., 2006, Wang and Wagner, 2016b, Conboy, 2009]. However, there has been very little reported in the literature on the experience of practitioners who have applied necessary adaptations to agile methods or practices in the context of safety-critical system development. Therefore there are many open questions about the selection of particular adaptations and their efficacy in different contexts.

During the study in Chapter 4, we learned that requirements management in large-scale safety-critical systems is a challenge. Several researchers agree with this conclusion and identify

requirements management as one of the challenges of agile requirements engineering in large-scale development environment [Inayat et al., 2015a, Vilela et al., 2017, Steghöfer et al., 2019, Uludag et al., 2018, Dikert et al., 2016, Kalenda et al., 2018, Kasauli et al., 2018b]. The level of coordination and communication required for an agile process to be successful is limited in a large-scale environment involving multiple teams, concurrent development of hardware and software, interdependence of hardware and software components, unsynchronised pace of development of different project components etc. The overall complex dynamics of large-scale systems have an adverse effect on agile management of a project's requirements.

Three out of the six themes in the empirical study on the challenges of agile requirements engineering in large-scale systems by Kasauli et al. [2021] focus on the challenges related to requirements management, shared understanding, and communication. Requirements management also appears as one of the challenges of agile requirements engineering in the large-scale context in two different Systematic Literature Reviews [Dikert et al., 2016, Inayat et al., 2015b]. A case study [Paasivaara et al., 2018] conducted at Ericsson found the challenges associated with requirements management among the top most challenges in implementing the use of agile in large-scale software development. While recognising the increasing popularity of agile, Venkatesh and Rakhra [2020] state that "*the problems faced by large-scale organisations when they implement agile methods are yet to be solved*".

We believe that the Agile Manifesto needs to be revisited for its incorporation in the context of safety-critical systems and for adding clarity to the agile principles. Agile principles are often misinterpreted [Hohl et al., 2018] making agile look like an ad hoc process that discourages documentation and detailed analysis. Several researchers have argued that the agile principles conflict with the philosophy behind regulatory standards [Coe and Kulick, 2013, Chapman, 2016, Glas and Ziemer, 2009, Ramesh et al., 2010, Turk et al., 2005]. For example: the way people interpret agile principle *working software over comprehensive documentation* seems like documentation is discouraged in agile [Ramesh et al., 2010, Turk et al., 2005, Coe and Kulick, 2013].

In reality, agile principles promote quick development and accommodation of change. Agile is not against documentation [Ozkan, 2019]. The focus of agile principles is on avoiding waste which means removing the production of artefacts without which the system is still usable. If the system is unusable without performing a certain activity or production of a certain artefact, the activity or the artefact is of value and has to be made part of the project.

Ozkan [2019] argue that the need of possible updates to the agile manifesto in order to support the agile development of modern-day software should be investigated. Wils et al. [2006] report the findings of their study conducted at Barco (a major Belgian avionics equipment supplier). The authors of the study performed an analysis of the agile principles and did not find conflicts between agile principles and the avionics regulatory standard DO178B. However, they [Wils et al., 2006] re-interpreted three of the agile principles. (i) Value customer satisfaction: In

avionics, value lies in the suitability of software for flight operations. Correctness in behaviour and functionality is of the highest value. (ii) face-to-face communication: A lot of information is lost during informal face-to-face communication. Since the communication is a part of the project, it should be logged and documented. (iii) Working software is a primary measure of success: In regulated systems, certification is a part of the project and without it, a project is not considered complete. We believe that agile manifesto is widely misunderstood. For example, according to many researchers [Jyothi and Rao, 2011, Sharma et al., 2012, DŽANIĆ et al., 2022], agile promotes minimum documentation. Stettina and Heijstek [2011] conducted a survey and obtained responses from 79 agile practitioners. According to one of their findings, "... *documentation is rather seen as a burden"*. Therefore it is imperative that agile manifesto is revisited to add clarity that agile is against waste and unnecessary work and not documentation.

9.2 Thesis Research Question 2

Our first empirical study with a large avionics company (discussed in Chapter 4) revealed that requirements management in large-scale safety-critical systems is a concern. To further our investigation we decided to explore the use of an agile method called Behaviour Driven Development (BDD) which is primarily based upon the idea of requirements communication, understanding, and management. During the literature review on BDD (in Chapter 5) we learned that there is a lack of empirical research on BDD. Especially, literature on the challenges of BDD is almost non-existent. This motivated us to conduct an empirical study to explore challenges of BDD. The answer to the second research question

"*What are the challenges of adopting Behaviour Driven Development (BDD) for the purposes of requirements engineering and acceptance testing in the early phases of a software project?"*

is derived from Chapter 6. The chapter reports the findings of action research and post hoc semi-structured interviews in a large avionics company where a software development team experimented with the use of Behaviour Driven Development in the development of a project. The duration of the study was spread over a period of 16 months.

The project discussed in the study was a sub-part of a larger project. The larger project's team consisted of six members out of which three members worked on the sub-part. The sub-part (referred to as *the project*) was developed using BDD. The three team members working on the project included a product owner, a scrum master, and a developer. The team members had varying levels of experience from the developer being a fresh graduate to the product owner with over seven years of development experience.

The initial project requirements were a set of tabular use cases documented by the product owner. With the incorporation of BDD into the project, the use cases were converted into

Gherkin features. During the conversion of the use cases into Gherkin features and transcription of the respective scenarios for them, it was discovered that those use cases were wrong and did not express the users' intention. This discovery was made when scenarios were simulated with the actual data.

This shows that use of BDD could facilitate the validation and evaluation of completeness of requirements. Scenario writing involves writing an end to end example of the execution of a feature. We observed that simulation of scenarios with actual data helps in looking at a feature in a real life context which also helps in evaluating the completeness of a feature.

Since the initial set of requirements did not express the users' intention, they were discarded and a new set of requirements were elicited in the form of BDD/Gherkin features in a user story workshop. The participants of the user story workshop were product owner, scrum master and two members from the larger project team. It should be noted that the developer was not part of this user story workshop because he joined the organisation after the elicitation of the set of features in the user story workshop.

The action research and interviews undertaken during the study revealed that the use of BDD encourages communication between team members. As mentioned earlier, scenario writing involves simulating the execution of features with actual data. The team members acknowledged that looking at the scenarios in real life context initiated discussions between them.

The research study (discussed in Chapter 6) yielded eight challenges of applying BDD. The challenges discussed in the study were discovered during the transcription of requirements as Gherkin features and the overall application of BDD during a single commercial project. These challenges which resulted from observations during a real-life industrial project can be grouped into three categories:

- lack of tool support for detection of bad smells in Gherkin specifications;
- limitations of Gherkin; and
- difficulties experienced by the project team in using BDD as a method.

We have discussed in Chapter 6 that the Gherkin language provides a lot of freedom due to its natural language structure. Therefore, it is very easy for the developers to write specifications that are structurally inflexible. We refer to such structural inflexibilities in Gherkin specifications as bad smells. The existence of bad smells in Gherkin specifications could lead to maintenance issues in specifications.

During the study in Chapter 6, we detected a violation of Given-When-Then order of steps and multiple assertions. These bad smells are referred to as AAA pattern violation and assertion roulette respectively, in the context of unit tests. Oliveira et al. [2019] considered preserving the order of Given-When-Then as one of the necessary criteria for the quality of BDD scenarios. According to the authors, violation of Given-When-Then step order makes the scenarios

unreadable. Whereas according to the documentation* of a Gherkin unified functional testing tool, assertion roulette is a violation of *Cardinal Rule* of BDD i.e., one scenario should cover exactly one single, independent behaviour.

To the best of our knowledge, there are no readily available tools that could detect bad smells in Gherkin specifications. The closest work we found is a tool called GherkinLint† on GitHub. The tool is implemented in JavaScript, and performs an automated analysis on Gherkin files using a number of default rules. However, all of these rules target stylistic errors in Gherkin. For example, the tool flags an error if a Gherkin file has no scenarios or if a feature in a Gherkin file has no name. The tool does not detect bad smells. The lack of readily available tools for detecting bad smells such as AAA pattern violation and assertion roulette makes it difficult to automatically detect bad smells in Gherkin. This means that the detection of bad smells requires manual intervention at the moment. The tool we developed helped us in extracting the meta-data and determine the presence of bad smells discussed in Section 8.2.2.

The limitations of Gherkin highlighted the capabilities that are presently missing in Gherkin and potentially needed in the future. To the best of our knowledge, there is no study that primarily focuses on the technical limitations of Gherkin. However, the study by Irshad et al. [2021] discusses a limitation of Gherkin among other challenges of BDD identified in their study. The authors conducted six workshop sessions with BDD practitioners to understand the benefits and challenges of BDD. The authors identified the lack of “*versioning control of behaviours”* as one of the challenges of using BDD. According to one of the findings of the study, it is difficult to keep track of the changes to the behaviours when multiple stakeholders are involved in the project. However, the study does not discuss this issue in detail and the authors [Irshad et al., 2021] suggest the use of versioning control softwares to handle this issue.

This shows that the use of BDD involves requirements traceability issues which also coincides with one of our findings in Section 6.5. Our findings focus specifically on hierarchical and horizontal relationships between BDD requirements, whereas the finding by Irshad et al. [2021] discusses the version control and ownership of the requirements. However, both findings target different aspects of the same issue i.e., traceability.

In the finding, we discuss how Gherkin does not demonstrate the process of evolution of features. In other words, Gherkin does not have a requirements traceability mechanism. By traceability, we mean both vertical and horizontal traceability. Vertical traceability, in this context, refers to the links and sources an item evolved from, whereas horizontal traceability refers to connections an item has with its peers or other artefacts at the same level of hierarchy. Silva and Fitzgerald [2021] has proposed automatic parsing of BDD stories to determine (horizontal) consistency between BDD scenarios and other artefacts such as classes. The issue of horizontal consistency is also addressed in some other studies [Silva and Winckler, 2017, Silva et al.,

*https://www.gherkinuft.com/gherkin

†https://github.com/vsiakka/gherkin-lint

2019c,b, 2020b] as well. The problem of vertical traceability in BDD artefacts is also discovered as a challenge during the study by Silva [2016].

While the literature on the limitations of Gherkin language is scarce to non-existent, we found some more limitations of Gherkin (as discussed in Section 6.5). For example, Gherkin looks at a feature from the point of view of a single actor which is an issue when multiple actors are involved in the completion of a single task. Furthermore, Gherkin does not support concurrent execution of two or more steps.

Lack of guidance on BDD greatly influences the success of its use in the industry [Zampetti et al., 2020]. The theoretical description of BDD activities does not take real-world factors (like the experience of the people using BDD or the size of a project) into account; hence it is very difficult to know what to do when a team faces a procedural issue. Attempting a new method without guidance could raise the overall development cost significantly and cause delays which, consequently, alleviates the benefits of using an agile process.

During the study, we observed that the team's experience of using BDD was different than what is described theoretically by Smart [2014] and Wynne et al. [2017]. The BDD process consists of a number of activities that could be summarised in five iterative steps (as described in Section 5.3). These iterative steps include *(i)* determination of the business goal, *(ii)* definition and documentation of a set of major features and determination of the relative value of each feature, *(iii)* illustration of features with examples, *(iv)* description of examples as BDD scenarios, *(v)* writing test code for each scenario. Although the theoretical description of BDD activities looks simple, the description lacks the consideration of real-world factors such as size and nature of the project, organisational structure, availability and level of experience of the team members, etc.

The difference in BDD process and applying BDD in practice started appearing from the start of the incorporation of BDD. For example, BDD process recommends three amigos meeting for elicitation of features in which the product owner, developer, and the tester participate. The purpose of the three amigos meeting is to analyse each feature from three different perspectives. As discussed in Chapter 4, the organisation did not encourage the assumption of specific roles by the employees. Each member of every team in the organisation was expected to be an all-rounder in all the fields of software development. Specifically, there was no separate tester role in the project team. Testing was performed by the developer himself. We believe that the team did not see the relevance of the three amigos meeting because of not having a separate tester role in the team, hence did not adopt it. However, in the case of regulated systems such as safety-critical systems, regulatory standards such as DO-178C [RTCA] mandate a separate and independent testing role.

As discussed earlier, the testing was performed by the developer. It means two out of three roles of the *three amigos meeting* (i.e., the tester and the developer) were already covered by the developer. The third role in the three amigos meeting is the product owner. The developer

and the product owner already communicated frequently. However, this specific *three amigos* meeting could not take place simply because of the missing third amigo i.e., the tester. This shows that the theory behind the three amigos meeting is based upon the assumption of the existence of these three roles in a team and does not take other possibilities into account. This also shows that the theoretical description of the activities does not always match the reality. This phenomenon is similar to the findings of the study by Stray et al. [2020b]. The study shows that the traditional way of conducting daily stand-ups as described in the theory is counterproductive. Instead, the teams should adapt the meetings according to what they find beneficial. For the sake of argument, if we assume that three amigos meeting is not a meeting between three individuals and a single person can assume these three roles, our assumption is contradictory to the statement by Smart [2014]. The author explicitly states "*...the "Three Amigos." Three team-members - a developer, a tester, and a business analyst or product owner - get together to discuss a feature and draw up the examples"* implying that this meeting is between three individuals. At another place, Smart [2014] states that "*...in this approach, the three will sit around a computer and write up an initial draft of the automated scenarios together"*. The context in which the "*three amigos"* meeting is discussed in various research studies [Wang et al., 2018, Wang and Wagner, 2018, Elshandidy et al., 2021] also suggests that this is a meeting between three individuals and the purpose is to have three different perspectives on the matter.

We also learned that it is difficult to imagine the functionality before implementing it. As mentioned earlier, the developer joined the organisation after the elicitation of the project requirements. He was not familiar with the project. Although he acknowledged that using BDD as a tool encouraged communication within the team in the form of discussion about the data in the scenarios, it was difficult for him to completely understand and visualise the scenarios without developing them. This led to the development of functionality before writing the tests for them. Our observation concurs with the observation made by Zampetti et al. [2020]. The authors employ various research methods to elaborate on the adoption of BDD in open-source projects. According to the authors, the developers often avoid practicing the test-first principle of BDD. However, the reason for this is not discussed in the study.

During the study (discussed in Chapter 6), we observed that two interconnected factors played a key role in *not practicing the test-first principle of BDD*. The first factor was the pressure to meet the schedule. Software companies around the world cannot dedicate unlimited resources for an unlimited amount of time to any project. The software projects around the world follow certain schedules and deadlines. We observed that it is easy to get carried away with investing more time than required while refining the BDD scenarios. Unless there are criteria to know when the scenarios are good enough, developers do not know when to stop refining the scenarios and move to development. However, this issue is not specific to BDD. The pressure to write production code instead of BDD scenarios to meet the schedule is a broader issue with testing as a whole. It is widely recognised that many companies will be pressured

into coding over testing, especially with tight schedules.

Oliveira et al. [2019] proposed a checklist to assess the quality of BDD scenarios. This checklist is a set of twelve questions each of which focuses on one of the three components of Gherkin specifications (i.e., feature, scenario, and step). However, the questions in the checklist are more or less a set of instructions on *what to look out for* than a clear criteria to evaluate the scenarios against. Moreover, the wording of the questions is vague. For example, question number five in the checklist says *How different each scenario is from the others?*. Unless we have a description of what qualifies as a *difference*, we cannot know if a scenario is different. If we look at the rationale behind question number five of the checklist, it says that every scenario of a feature should represent a separate variation of events. This could mean that every scenario for a feature should have a different outcome even though most of the steps in most of the scenarios are common i.e., the *Then* statements should be different even if the *Given* and *When* statements are common between the scenarios are common.

During the study, we observed that investing too much time in refining the BDD scenarios can cause delay and concerns among the management who are more interested in the outcome of the project which ideally would be a working software. We observed a similar situation where the developer started feeling that it was taking too long to refine the scenarios. He did not know if the scenarios were good enough to proceed with writing the tests for them. The expectations within the company to deliver working software on time made him pause the refinement of the scenarios and write the production code.

The second factor that played a key role in *not practicing the test-first principle of BDD* was not knowing how much work is needed to implement a certain feature. BDD emphasises the importance of abstraction in requirements. According to the documentation on Cucumber [CucumberStudio, 2019b], imperative scenarios include implementation details that are so closely tied to the mechanics of the UI that the tests become brittle i.e., need updating more often. Instead, the documentation [CucumberStudio, 2019b] recommends writing declarative scenarios that focus on the behaviour instead of how a user interact with a system. The issue with abstraction in BDD is that the requirements become open to interpretation and also, it hides complexity embedded in the code which makes it harder to assess the work that needs to be done to complete a requirement. The project team in our case also felt that often the complexity of a feature is not visible from the scenarios. The developer experienced situations where the implementation was too long and complex. Sometimes it was possible to go back to the scenarios and break them down and sometimes he continued with the implementation.

As it was not until the development started that the developer realised that some of the scenarios were too complex and needed to be broken down. Again, going back and forth between the development and the refactoring of the scenarios was taking too long. Therefore, the developer decided to continue the development and write the scenarios and the respective tests at the end of the development. Our observation complements one of the findings of the study

by Zampetti et al. [2020]. They also observed that often the changes to BDD test cases are driven by the change in production code instead of happening otherwise, ideally. Yang et al. [2019] demonstrate a method for correlating feature changes and subsequent code changes, but this would need to be generalised for code changes that also preceded feature changes. Also, the short time scale for change assumed by Yang et al. [2019] may not be valid i.e., one work week. Stark et al. [1999] collected data from an organisation on 44 software releases spanning seven products. During the study, the rate of change was measured as 1.4 changes/month. This supports our argument that a one-week time scale for measuring change by Yang et al. [2019] may not be an appropriate time period.

As discussed before, the developer wrote the Gherkin features after the implementation which shows that Gherkin was used for documentation instead of testing. The developer based the Gherkin files and the tests on the implementation which means that the tests and the Gherkin files were an *elaboration* of what was already implemented. In future work, it would be valuable to investigate whether Gherkin is more commonly used for documentation or design in an empirical study.

We believe that the recommended ways of BDD need to better reflect the observed complexities in practice. Also, the theoretical framework described in the literature is based upon assumptions. For example, BDD process does not describe the professional profiles of the people involved in the project. We have observed that the use of BDD requires a considerable amount of prior experience in BDD. According to the Österholm [2021], one of the drawbacks of BDD is that its adoption requires prior experience with TDD (Test Driven Development). Irshad et al. [2021] conducted six workshop sessions with BDD practitioners and found that in order to make improvements in a process based upon BDD, the practitioners require prior experience with BDD.

Moreover, the BDD process does not take the amount of effort required to write the BDD specifications and tests into account. At the moment, BDD process lacks the consideration for real life factors which could adversely impact a project e.g., duration and complexity. Hence more understanding of actual behaviour and empirical research is needed in order to learn from people's experience of applying BDD in their context.

9.3 Thesis Research Question 3

The scope of the findings from the empirical study (discussed in Chapter 6) was limited to a single project. We extended the scope of the investigation on the challenges of BDD to the open-source projects on GitHub. By doing this, we not only wanted to present an overall picture of BDD in practice but also investigate the practices that could potentially create maintenance issues in Gherkin specifications. The answer to the third research question

"What are the specification writing practices in the existing open-source BDD projects that could result in maintenance challenges in behaviour driven development software projects?"

is derived from Chapter 7 and 8.

Chapter 7 presents an overall picture of BDD in practice. To do this, we implemented a tool in Python that selected a sample of open-source projects through a five-stage process. In the first stage, 1 million random GitHub repository identifiers were generated. These identifiers were then matched with the repositories on GitHub. The repositories which matched the identifiers were marked for selection. In the second stage, the non-BDD projects were filtered out from the selected projects by removing the projects that did not contain Gherkin language. Please note that we were interested in real projects. Therefore, in the third stage, an exclusion/ inclusion criteria (as discussed in Section 7.2.1) was applied to filter out the dummy projects.

After removing the projects which did not contain any feature the final number of projects was 501. While running the analysis, eight repositories became inaccessible making the final number of repositories 493. In the fourth stage, the tool generated meta-data from the contents of the projects from the most recent commit. In the last stage, the meta-data from the contents of the projects was generated from all the commits of each project.

The analysis shows that only 0.34% of open-source repositories on GitHub contain Gherkin. This indicates a low adoption of BDD in open-source projects. The reason behind this low rate of adoption could be explained using the lessons learned during the study discussed in Chapter 6. In Chapter 6, we learned that the adoption of BDD into a project requires an investment of additional time and resources. Zampetti et al. [2020] and Irshad et al. [2021] seem to agree with our opinion. According to Zampetti et al. [2020], BDD as a method is quite effort-prone. The authors argue that BDD is a way of working, and not just the adoption of a framework. Irshad et al. [2021], based upon the findings of their industrial evaluation of the application of BDD, argue that BDD is a time-consuming process, and adoption of BDD requires training, long-term commitment, and additional resources. Another reason for this low adoption of BDD in open-source projects on GitHub may be the nature of open-source projects. Usually, the participants and contributors in an open-source project are volunteers and there is no formal customer role. This means there is very little "*customer involvement*" in an open-source project which could make the benefit of communication offered by BDD less apparent.

We selected a random sample of non-Gherkin projects to see the difference between Gherkin and non-Gherkin projects. Our analysis showed no significant differences between Gherkin and non-Gherkin projects. However, the data showed that the adoption of BDD is more popular in web development languages, such as PHP. The analysis showed that the test suites (i.e., the number of Gherkin feature files) in the projects that typically adopt BDD were small. When we compared the lines of code, the projects that adopted Gherkin were found to be larger than non-Gherkin projects i.e., approximately three times larger. The median number of features in

the dataset was found to be 5, each having 2 scenarios with 5 steps each.

We wanted to see when people typically start incorporating BDD into the projects. Although the data was too scattered to make a conclusive judgement but we learned that, on average, people start incorporating BDD during the first six months of the project. We confirmed this assumption by looking at the percentage of total commits made before the first feature was introduced in a project. We learned that, on average, first feature is introduced into the project within the first 15% of the commits. To see if there has been any change in the practice of the introduction of first feature, we extended this analysis to a number of years.

We learned that the practice has not changed over the years. There has been no change in the average number of days after the first feature is introduced over the years. BDD is not practiced to a great extent, it's only used at a small scale, but projects that adopt BDD tend to be bigger and those contributors who make BDD changes tend to be much more committed. We learned that the people who incorporate BDD into the project have the longest and earliest involvement in the projects. This means that the decision to incorporate BDD and maintenance of BDD specifications are performed by the primary contributors in the projects.

The focus of Chapter 8 was on bad smells in Gherkin specifications. The selection of bad smells was made on the basis of applicability and feasibility (as explained in Section 8.2.2). First, we decided to use the knowledge on bad smells in unit test because of the scarcity of literature on bad smells in BDD and the similarities between the structures of a Gherkin scenario and unit test. We used the list of unit test bad smells compiled by Garousi and Küçük [2018], and theoretically mapped each smell to Gherkin scenarios. Out of 182 unit test bad smells in the list, eight appeared theoretically applicable to Gherkin scenarios. Later, three more bad smells were added to the list of applicable bad smells making the total number of applicable bad smells eleven. These three bad smells were discovered during the manual inspection of Gherkin specifications of randomly selected projects from our sample of 493 Gherkin projects. Next, we performed a feasibility analysis by discussing the effort and time required to calculate each of the 11 selected bad smells. Finally, five bad smells were selected for analysis. Effort and time required for the rest of the six bad smells appeared greater than what could be achieved within the course of this Ph.D.

The analysis showed that a significant number of open-source Gherkin projects' specifications contain these bad smells. For example, approximately 36% Gherkin projects contain AAA pattern violation, 70% projects have multiple assertions, 68% projects contain duplications(i.e., clones), 12% of the projects have lazy outline, and 30% projects have lazy steps. A large majority of project contributors introduce these bad smells within the first day of their Gherkin commit. For example, approximately 45% of the contributors who make Gherkin commits, introduced clones within the first day of their Gherkin commit.

We investigated the relationship between bad smells and *other* Gherkin artefacts e.g., size of scenarios, number of contributors in a project, or the duration of involvement of contributors who

introduce bad smells in projects. These graphs included histograms of scenario sizes, time series of project smell density, scatter plot of the number of bad smells against Gherkin contributors in a project, histogram of the time for which contributors were involved in projects, etc.

The graphs showed no relationship between the bad smells and the number of contributors involved or the duration of their involvement in a project. However, the projects with bad smells tend to have slightly longer scenarios than the scenarios of the projects with no bad smells. We were unable to determine if longer scenarios cause bad smells or vice versa. Since a large percentage of these bad smells are added within the first day of a Gherkin commit in a project, we could speculate that the reasons for these bad smells could be the lack of experience of contributors in BDD or simply the way they write specifications.

We believe we have identified evidence of technical debt (i.e., bad smells) in the Gherkin code. Technical debt in software engineering is known as the financial consequences of trade-offs between minimising the product time to market and poor specification and implementation [Ampatzoglou et al., 2015]. Technical debt is recognised as a multiplier to the cost of new features. It is a function of project size (i.e., same rate of technical debt is more costly for bigger than smaller projects)[Guo et al., 2016]. The Gherkin suites we observed were small but the open-source Gherkin projects on GitHub appear to have a greater average number of lines of code than non-Gherkin projects i.e., three times. This suggests that development teams deliberately keep BDD suites small to avoid incurring technical debt, but this potentially comes at the cost of limiting test suite coverage and therefore effectiveness. This means it is hard to use BDD in large projects because of the cost of maintenance of additional (Gherkin) artefacts.

We do not claim that these bad smells are the only type of bad smell in Gherkin specifications. There could be various other bad smells in Gherkin specifications but the literature on bad smells in Gherkin is scarce at the moment. To the best of our knowledge, there are only two studies [Suan, 2015, Binamungu, 2020] that investigated one of the bad smells in Gherkin specifications i.e., duplication or clones. We did not find any other studies on any other bad smells in Gherkin. Lack of guidance on BDD could be one of the reasons for these bad smells in Gherkin specifications.

9.4 Contributions

Chapter 3: This chapter significantly extends the existing knowledge on the application of agile software development within safety-critical systems engineering by reviewing the relevant literature on the topic. We conducted a Systematic Literature Review (SLR) on challenges of the application of agile methods in safety-critical systems development. The literature review included an exclusion/inclusion criteria which filtered down the studies that were irrelevant. A total of 56 studies were selected for the review. The information gathered during the SLR was the result of the synthesis of the selected studies. The extent of the information generated from

the literature review allowed us to gain significant insight into the state of the field. Specifically, we reported on what challenges of the application of agile for the development of safety-critical system development appear in the literature. We elaborated on these challenges by integrating the findings of the studies. This helped us in presenting an overall picture of the state of the field. We grouped the challenges of using agile methods in safety-critical systems into various themes such as organisational culture and training, project management, documentation, regulatory standards, design and architecture, as well as, statements that look like perceptions of the researchers for which no evidence was presented in the respective studies. The work, therefore, provided an overview of the available literature on the reported challenges of employing agile software development for safety-critical systems and provided a foundation for research in this Ph.D.

Chapter 4: This chapter significantly extends the existing evidence base for the application of agile software development within safety-critical systems engineering by investigating the challenges from the perspective of practitioners. We conducted four semi-structured interviews with employees of the company in a variety of roles in different software projects and with diverse experiences. The interview structure was based upon the information gathered during an initial exploratory conversation with two senior employees. The findings of the study were validated in a workshop with a wider number of participants drawn from across the company's software development function. The extent of the material generated from these interviews allowed us to gain significant insight. Specifically, we reported on how some teams within the company have employed an agile software process (Scrum) within a Waterfall process for the wider systems engineering project. We elaborated on this integration by describing how the teams have made necessary customisations to Scrum to fit within this process. We described the successes that the teams had experienced in employing and adapting individual agile practices, such as planning poker, continuous integration, automated static analysis, and code reviews, as well as, discussing where the use of agile software development had led to drawbacks. We also investigated practices that the teams had not employed, such as pair programming and user stories, and discussed the rationale for this from the teams' perspective. Where appropriate, we related these insights to the available literature. The work, therefore, provides a substantial case study based on evidence from industry of the real-world challenges of employing agile software development for safety-critical systems and provide a foundation for future research in addressing these challenges.

Chapter 6: This chapter significantly extends the evidence base for the challenges of using Behaviour Driven Development (BDD) by investigating the challenges of BDD from the perspective of practitioners. In addition to the four iterations of action research, we conducted four semi-structured interviews with three employees of the company in a variety of roles in a

software project and with diverse experiences. The action research was conducted to align the project development with the BDD process.

The interview structure was based upon the information gathered during the action research. We also compared BDD in theory with BDD in practice. This comparison was based upon the lessons learned during the study. The extent of the material generated from these interviews allowed us to gain significant insight into the difficulties of applying BDD. Specifically, we reported on how a team within the company employed BDD and the difficulties they faced in using BDD as a process. We elaborated on this experience by describing how the teams made necessary customisations to the process to match their needs.

We described the successes that the teams experienced in employing and adapting BDD such as communication within the team, as well as, discussing where the use of BDD led to difficulties. We investigated the technical limitations of Gherkin and the BDD practices that the teams did not employ (e.g., *test first* concept in BDD), and discussed the rationale for this from the teams' perspective. Where appropriate, we related these insights to the available literature. The work, therefore, provides a substantial case study based on evidence from industry of the real-world challenges of employing BDD and provide a foundation for future research in addressing these challenges.

However, this study was focused on a small project developed by a small team using BDD for the first time. The team used BDD in the early phase of the project for the short life span of the project. Therefore, this study reports the challenges experienced by a team who practiced BDD for a limited duration and did not have any prior experience of using BDD. We are unable to predict if we would discover more challenges or find solutions to the reported challenges, had the team practiced BDD for a long term. A further investigation would be required to study the challenges of BDD discovered during a long-term use.

Chapters 7 and 8: These chapters significantly extend the evidence base for the use of Behaviour Driven Development (BDD) by investigating the open-source BDD projects on GitHub. We conducted an online experiment on GitHub to understand the state of BDD by studying the project-related artefacts in the open-source projects which incorporated BDD. To the best of our knowledge, this is the first study in the field of Behaviour Driven Development which involves studying bad smells in open-source BDD projects available on GitHub and drawing statistical inferences from them.

The experiment structure was based upon the knowledge gained from the literature and the exploratory study in Chapter 6. The experiment consisted of five steps through which a random sample of open-source BDD projects was selected for analysis. The analysis involved generating graphs. The extent of the information generated from these graphs allowed us to gain significant insight into the overall state of BDD on GitHub e.g., typical size of a project, typical number of features and scenarios, etc. Specifically, we reported on how and when BDD is incor-

porated in open-source projects. We plotted graphs showing that the people who make Gherkin commits have the longest involvement in the projects. The analysis also shows that there is no considerable difference between the open-source BDD and non-BDD projects on GitHub

We also investigated Gherkin specification writing styles and practices that could be regarded as *bad smells*, such as AAA pattern violations, and lazy scenarios. We discussed the consequences of these bad smells from the perspective of the evolution of the project. Where appropriate, we related these insights to the available literature. The work, therefore, provides a substantial empirical study based on evidence from real open-source projects on GitHub and a foundation for future research on the bad smells in Gherkin specifications. Moreover, we generated a dataset of Gherkin projects that can be used in future research.

9.5 Interconnection of the Studies

In addition to the literature reviews (documented in Chapters 3 and 5), two sets of interviews, four iterations of action research, and an online experiment was conducted over the course of this Ph.D research (presented in Chapters 4, 6, 7 and 8). This Ph.D was conducted in progressive steps. The first study was conducted with an avionics company to learn from their experience of using an agile method. Our focus during the study was on the challenges faced during the application of the agile method in large-scale safety-critical systems development. As a result of the study, a number of challenges associated with the application of agile development method were discovered. Requirements management appeared to be one of the major concerns during the application of agile methods in large-scale safety-critical systems development context.

To extend the study and investigate this further, we conducted another study to try Behaviour Driven Development (BDD) for the development of a project at the (same) avionics company. BDD is an agile method which is based upon the idea of shared understanding through requirements. Literature on the challenges of BDD appeared to be scarce. Therefore, the primary focus of the study was on the challenges related to BDD.

During the study, we found that BDD works well for the decomposition of requirements and encourages communication between team members. The study yielded eight challenges for the application of BDD in practice by the software teams adopting BDD for the first time which can be grouped into (i) lack of tools for detecting bad smells in Gherkin specifications, particularly during the translation of requirements from other formats, (ii) technical limitations of Gherkin such as lack of support for multiple actors, and (iii) hurdles in using BDD as a process. We learned that Gherkin provides a lot of freedom for writing the requirements specifications. One of the drawbacks of this freedom is inflexible requirements. In BDD, the developers sometimes transcribe functionally correct but structurally inflexible requirements. Such inflexibilities (i.e., bad smells) in the requirements impede the evolution of the overall system. It is possible that we were not able to discover all the challenges of using BDD because the scope of this study was

limited to a single project.

In order to extend the scope of our investigation and get an overview of the state of BDD we analysed the open-source projects on GitHub. This third study was divided into two chapters i.e., Chapter 7 and 8. Chapter 7 presents a birds-eye view of BDD in open-source projects on GitHub. The chapter presents a comparison of BDD and non-BDD projects and analyses the open-source projects to learn about the evolution of BDD artefacts. Chapter 8 evaluates the projects for the existence of bad smells in Gherkin specifications.

The research helped us in understanding the impediments in handling the requirements specifications when using agile as a development method in large organisations. It also helped us in understanding the technical limitations and hurdles in using BDD as a process. We have learned that the mere incorporation of a process such as BDD for the management of requirements specifications does not solve the problem. The freedom provided by Gherkin as a language is its own enemy. People must be aware of the potential maintenance issues when using BDD as a process. However, one possible avenue to address this issue could be the use of Domain Specific Languages (DSLs). A Domain-Specific Language (DSL) is usually a computer language that is tailored to a specific application domain e.g., Railways, Healthcare, Robotics, Banking etc [Fowler, 2011, Kosar et al., 2016]. In this approach, solutions can be expressed at the level of abstraction of the problem domain, allowing substantial gains in expressiveness and ease of use [Kosar et al., 2016]. For example, Rocha Silva [2022] propose the use of a Gherkin style DSL to test web-based graphical interfaces. This approach preserves the abstraction in Gherkin scenarios keeping the necessary formalism to avoid misinterpretations.

9.6 Scope and Validity

We have acknowledged and discussed the scope and validity issues of this research in the present section. First is the issue of the ability to generalise the results on the basis of a single case study. This issue is acknowledged and discussed by Kennedy [1979] in detail. The author regarded single case studies as "*studies of single events, or disaggregated studies of multiple events*". According to the author, findings from single case studies should not be used for drawing inferences.

This opinion has been rejected by several researchers [Flyvbjerg, 2006, Hammersley et al., 2000]. For example, Flyvbjerg [2006] has referred to this opinion as a misunderstanding and argued that a single case study, because of its in-depth approach, can be considered as a representative example of a larger population. Hammersley et al. [2000] argued that if reasonable assumptions about the similarities in smaller units of a large population can be made then the findings from studying a single unit can be considered as representative of the findings from studying that large population.

Before we discuss the validity of the results from the two individual case studies (discussed

in Chapters 4 and 6), we address the validity of the case i.e., the company (we conducted our research with). In view of the arguments by Flyvbjerg [2006] and Hammersley et al. [2000], the company must have close similarities with other organisations in the same industrial sector in order to qualify as a case with findings that can be generalised. According to Todd and Humble [2019], most avionics companies around the world are large organisations. This means in order to be a representative of the *most avionics companies* (i.e., large population), the company we conducted our research with has to fall under the *large organisation* category.

According to various business blogs[‡,§], we learned that the company (where we conducted our case studies) is counted among the top twenty aerospace manufacturers in the world. Also, by reading and comparing the profiles of different avionics companies of the same level, we learned that the nature of the work and the size of the projects undertaken by the company were not very different from other large avionic companies. These similarities make the company a representative sample of its peer companies.

Walsham [1995] categorised generalisation from case studies into four types: (i) concepts, (ii) theory, (iii) implications, and (iv) rich insight. The results from both case studies (discussed in Chapters 4 and 6) can be generalised from the rich insight perspective. The first case study was an exploratory study which gave us an insight into a variety of topics including project visibility, compartmentalised departments in large organisations, the flow of information, and customer interaction in large organisations. The second case study was also an exploratory case study which helped in exploring the use of Behaviour Driven Development (BDD) for the development of a project. The case study gave us an insight into the challenges of BDD and helped us in discovering the limitations of the Gherkin language.

9.7 Limitations

This Ph.D research consists of a number of independent but interconnected studies discussed in various chapters of this thesis. There are several limiting aspects to our research which we have discussed as potential avenues for future work. First, we would like to discuss one of the common limitations of the research work discussed in Chapters 4 and 6. The studies discussed in Chapters 4 and 6 were conducted with the company, and the limiting factor was the nature of our engagement with the company. The sensitive nature of much of the work in the company necessarily limited our access to the details of the project(s). Our findings were primarily based on the perspectives given to us by our interview participants, and we were consequently unable to verify them through independent inspection of other sources of evidence, such as project software repositories and software process documentation.

In the study discussed in Chapter 4, the interview participants were selected by the com-

[‡] www.salesartillery.com/
[§] www.flightglobal.com

pany, based on their availability and different perspectives and experiences of agile software development. Considerable effort was made by the researchers to establish a relationship with the company to allow the interviews to be conducted in the described form. We believe the arrangements reflect the constraints imposed on much of the research conducted in safety-critical contexts, given the often sensitive nature of such work. However, this does create threats to the validity of the work, which we have sought to mitigate by relating the findings to those available in the literature.

Second, we note that one of our findings during the interview stage of the research in Chapter 4 was not validated during the review workshop, concerning the conflict between agile software development to software quality assurance and that demanded by regulatory standards. This topic was included in the interview instrument because of the prevalence of the challenge in the literature. Specifically, Notander et al. [2013] reported that independent testing of complex systems, in accordance with the regulatory standard, DO-178C [RTCA] was very difficult due to the need for significant specialist knowledge about the test subject. It was anticipated that this challenge would also be identified by the participants, particularly given that agile software development advocates that testing should be conducted by the software team as part of the design and implementation process.

However, the issue was rejected during the validation workshop. According to the interview participants, they had great difficulty getting their system tested by an independent quality assurance team. The independent quality assurance team did not have the inherent knowledge of the system needed to develop effective tests. To mitigate this, the software team conducted training and workshops with the independent test teams but found these insufficient. So the software team performed the testing themselves while the independent quality assurance team *acted as witnesses to the testing* and signed off the documentation at the end. This approach worked well for the software team and was perceived to satisfy the demands of the standard for independent testing whilst also enabling effective tests to be developed.

The rejection of this challenge was surprising to us because the standard DO-178C mandates an independent testing body. Later reviewing the interview material, we noted that during one of the interviews a lead software engineer agreed that the risk of bias in this approach was "*... a problem, it's an ongoing problem.*" In reviewing this, it is possible that the participants do not view the approach to testing as problematic with respect to the standard, but are still concerned about the risk of bias, regardless. The issue highlights the risk in our research method of misinterpretation of findings. However, the validation step is applied to mitigate this.

Third, the study discussed in Chapter 6 was limited to a single commercial project. It was difficult for us to generalise the findings on the basis of a single commercial project. As discussed before, we did not have access to any other sources of data gathering than appointment-based access to the team members, and our findings are based upon the observations made during the action research and the opinions of the team members during the interviews. We were un-

able to verify the findings with the help of a supplementary source of evidence such as code or documentation.

The study (discussed in Chapter 6) was conducted in an uncontrolled environment. All project-related decisions such as project schedule, requirements, team members, etc. were made by the organisation and the project team. The organisation and the project team was responsible for deciding what and when to develop, and our interaction with the team was limited to the meetings and the requirements specifications. It is possible that we missed some of the vital observations which we could only make if we were directly and closely involved in the development of the project and were based in the company physically. Being physically present at the company and closely involved in the project would let us observe behaviours in the team that can not be discovered otherwise.

Fourth, the team members had no prior experience of using BDD in a project. The product owner and the scrum master had some background knowledge of BDD as a method but the developer was completely unfamiliar with BDD. This lack of experience was one of the reasons that the project team members were unable to detect the technical limitations of Gherkin themselves. These limitations were observed and discovered by the author of this thesis and presented to the project team. The project team agreed to the limitations but was unable to discuss an alternative due to their lack of experience with the tool and the methodology.

Fifth, the tool implemented in Python (discussed in Section 7.2) was used to select a random sample of *real* projects in which BDD was incorporated. The tool extracted the meta-data from the projects and performed additional analysis on the contents of the projects to extract more data. We believe that the tool could be extended to incorporate more functionality. For example, the tool could be extended to analyse the state of the practice of *test first* in BDD in open-source projects. Also, the tool could be extended to detect *other* bad smells than the ones discussed in Section 8.2.2. Restricted functionality of the tool at the moment is one of the limitations of this research.

9.8 Research Implications

This PhD research began with investigating the use of agile methods in large-scale safety-critical systems, and later on narrowed down to investigating Behaviour Driven Development (BDD) in practice. This PhD research was conducted in progressive steps with a focus on three research questions. These research questions are described in Section 1.3. The study described in Chapter 4 was conducted to find the answer to the first research question. The study in Chapter 6 was conducted to answer the second research question, whereas the studies described in Chapters 7 and 8 were conducted to find the answer to the third research question. The conclusions of these studies are described in the form of answers to the three research questions in the first three sections (i.e., 9.1, 9.2, and 9.3) of this chapter.

Please note that the work in this thesis should be treated as exploratory. Hence, the insights, lessons learned and challenges identified during this research need to be investigated further in order to understand the underlying reasons behind the challenges identified during this research. However, this section provides implications for research based upon the results of this thesis. The implications are based upon the *company's* experience of using agile methods and what we learned during our study on GitHub.

In summary, the results from Chapter 4 and 6 showed that Agile software development is not a plug-and-play solution for any organisation. It is a common misunderstanding that adopting an agile method such as Scrum automatically makes a team or an organisation agile. Especially, in a large-scale project environment where multiple teams working on their own individual tasks are involved in a project, it is hard to practice agile at the overall project level.

On the contrary, agile philosophy revolves around a set of principles that promote shared vision, common understanding, and collaboration, whereas agile methods are the roadmaps for reaching these objectives. The risk is that the principles of agile will be lost if the focus becomes on myopic application of methods without consideration of context. Organisations or teams wanting to adopt agile must focus on the underlying philosophy of agile and strive to reach the objectives defined by agile principles instead of focusing on agile methods. The processes, tools, and methods that are adopted to reach these objectives could differ from project to project and organisation to organisation.

We focused on requirements engineering because it appeared as one of the major concerns in Chapter 4. So we decided to explore BDD because of its focus on requirements communication, understanding, and management. Chapters 6, 7 and 8 highlighted the difficulties of applying BDD in practice. The results show that the use of BDD could incur significant cost of Gherkin specifications maintenance. This cost could increase if BDD practitioners are unaware of the challenges and limitations the use of BDD entails. The results from Chapters 6, 7 and 8 imply that the use of BDD requires a significant amount of prior experience which should also include knowledge of *do's and don'ts* of BDD. Till now, the research community has not been able to provide an established guideline for avoiding maintenance issues in Gherkin specifications. Therefore it was pertinent to explore the challenges associated with BDD in practice. These challenges could serve as a guideline for further research.

Moreover, the activities of BDD described in theory need to be investigated in real-life situations so that their significance, context, and applicability in different circumstances are clear. At the moment, the literature available on BDD does not account for the real world circumstances which could impede the significance or relevance of various activities of BDD as described in theory.

Question	
1	Can lightweight gate reviews be used to achieve the same quality of the design?
2	How can requirements for complex systems be better structured and decomposed to enable agile development efforts?
3	How can continuous integration methods be extended to satisfy the heterogeneous nature of complex systems engineering projects in safety-critical environments?
4	How can agile customer management methods be adapted to the complex customer structure of safety critical systems?
5	To what extent can the maintenance of documentation be automated, or better integrated into the cost estimation process?
6	What is an appropriate level of abstraction needed in the Gherkin scenarios in order to balance the complexity between code and scenarios?
7	How can we determine the quality of Gherkin scenarios?
8	What is a reasonable amount of time for execution of a Gherkin test suite?
9	When following BDD, how can we know what requirement(s) or which feature(s) a particular feature evolved from, to ensure traceability?
10	How can we detect more than one independent behaviours in a single step of a Gherkin scenario?
11	How can we measure the readability of Gherkin scenarios?

Figure 9.1: Future work research questions

9.9 Future Work

Despite the limitations described above, the research has identified several key themes during the course of this Ph.D. This section discusses future work and provides a roadmap for addressing these challenges. Beyond these broad challenges, we have identified a set of immediate research questions to guide future efforts in this area, summarised in Figure 9.1. These questions are indicative of immediate research directions that can be undertaken in the short term within these broad themes.

Questions 1 and 2 address the theme of mitigating the pressure for Waterfall development processes for software engineering processes. Question 1 concerns the development of lightweight design review methods that accommodate more rapid changes in software design without compromising on design quality. We envisage leveraging existing agile methods and practices to

facilitate this, such as continuous inspection techniques. Question 2 concerns the need for alternative approaches to the structuring of requirements specifications to better support decomposition of requirements in complex systems such as BDD. In particular, there is a need for comparison between BDD and *other* requirements documentation techniques to see whether a feature driven approach to requirements engineering, embodying detailed specifications as user stories and scenarios provides better decomposability.

Question 3 and 4 address the theme of coordinating the stakeholder relationships (both internal and external) within complex systems engineering projects. In particular, software engineering has developed sophisticated techniques for achieving continuous integration of software products. We envisage that these techniques can be extended further across the technology stack of firmware and hardware through networked deployments of software on hardware under development, or the development of realistic hardware simulators concurrently with hardware development efforts. Similarly, recent advances in software process development that enable abstraction of hardware, such as virtualisation, DevOps and Infrastructure as Code may be adapted to provide solutions to this integration challenge.

Separately, Question 4 concerns the adaptation of agile customer management techniques, through the product owner to complex systems projects. By convention, agile software development assumes that all the interests of "*the customer*" can be represented to the software team via the product owner, shielding the development team from the conflicts, tensions, and negotiations that may occur between different stakeholders. However, the size and complexity of large-scale systems engineering projects, together with the typically complex interplay between stakeholders (recall Figure 4.4) makes the allocation of this role to a single person impractical. Several authors have described proposals or experiences of scaling agile methods and practices, particularly for scaling the role of the product owner. For example, Lowery and Evans [2007] reports on experiences of implementing a hierarchy of product owners in the BBC's iPlayer app. They found that a critical aspect of their approach was ensuring coordination between product owners and scrum masters in the different teams and placed significant emphasis on time in the product owners' schedules to accomplish this. The popular Scaled Agile Framework [Leffingwell, 2016] also advocates the use of a hierarchy within product ownership, between product managers who are responsible for the high-level direction and product owners who are embedded in particular teams focused on specific aspects of functionality. There is a need to explore how these hierarchical approaches to managing the relationship with customers through the product owner can be adapted to both the heterogeneous nature of systems engineering projects which combine a variety of software and hardware elements; and the consortium arrangement of customers in systems engineering projects.

Question 5 concerns the automated generation of supplemental documentation, addressing the need to reduce friction in Software Engineering projects. Traceability remains a critical component of standards and regulations for safety-critical environments. There will be an on-

going need to produce evidence that system artefacts remain consistent with their requirements and design, such that any associated safety evaluations are reliable. To adapt agile software development to fit with this context, there is a need to develop mechanisms for automatically re-generating artefacts as changes occur, or better support their continuous maintenance alongside mainstream development efforts. A factor here will be to integrate documentation maintenance efforts into ongoing software development task cost estimates, such that all necessary changes are continuously tracked. Similarly, there is a need to develop better methods for modeling and representing dependencies amongst software project artefacts, such that when changes occur the impact can be more efficiently assessed. For example, Silva and Winckler [2017] proposed automated verification of software artefacts by parsing sentences of BDD scenarios using a case study. Their proof of concept shows that they were able to identify even the fine-grained inconsistencies in BDD artefacts.

Crucially, the study discussed in Chapter 4 has demonstrated that there is a need to adapt agile software development to fit within the constraints of software development for safety-critical systems and investigated the specific challenges in detail. In particular, there is a need to understand how agile software development can be scaled to fit large-scale, complex systems engineering efforts comprising multiple development efforts that include both software and hardware components on projects that may last many decades. Ultimately, these questions reflect the need to better align the *tempo* of safety-critical system developments and that assumed by agile software development. The agile philosophy is to accommodate the constant, rapid, concurrent change of software development projects, due to inevitable external pressures. The complexity created by this change is then mitigated through the disciplined application of a combination of tools and methods. Conversely, the philosophy in software development for safety-critical systems is to deliberately constrain options for (and pace of) change in order to maintain the traceability of artefacts. Applying agile software development to safety-critical systems will, therefore, require the development of tools and methods that provide for the same standard of continuous traceability.

Question 6, 7 and 8 address the improvement in quality of the Gherkin test suite in BDD. Question 6 concerns the concept of abstraction in BDD. The available text on BDD such as online blogs[¶] and scientific studies [Oliveira et al., 2019, Silva and Fitzgerald, 2021] discuss the importance of writing the scenarios without emphasis on the implementation details. According to the advocates of BDD [Smart, 2014, Wynne et al., 2017], scenarios should be declarative (i.e., specifying what needs to be accomplished) and not imperative (i.e., specifying how something should be accomplished). The focus in the scenarios should be on *What* and not *How*.

Declarative scenarios hide implementation details, whereas, imperative scenarios contain implementation details that tightly couple the scenarios with the input and the UI. Such scenarios are not only brittle but also unreadable according to Wynne et al. [2017]. Despite the

[¶] `https://cucumber.io/docs/bdd/better-gherkin`

emphasis on writing declarative scenarios in the text related to BDD, we believe that declarative scenarios could be open to interpretations. When relying on *what to do* and no information on *how to do*, it is possible to write code that satisfies a (declarative) scenario but does not follow the workflow the user had intended. Moreover, hiding the implementation details obscures the estimation of time and effort required to accomplish a task. We believe that there needs to be a balance between being declarative and being imperative when it comes to writing scenarios. The developers should be able to anticipate the amount of time and effort required to complete a task. We envisage that multiple experiments need to be conducted to define how much information in a scenario is considered as *just enough* and that the complete rejection of imperative style for scenarios should be revisited.

Question 7 concerns the quality criteria of the BDD scenarios. Some work has already been done in this regard by Oliveira et al. [2019]. The study [Oliveira et al., 2019] provides a list of questions in the form of a checklist. The aim of the checklist is to provide a guideline for writing short, readable and optimal tests. Binamungu et al. [2018b] identify slow execution of BDD test suites as one of the challenges. According to the authors, the problem of slow BDD test suites is due to duplication and "*other concerns*". However, the authors do not elaborate on these "*other concerns*". We did not find a published study that discusses the reasons for slow BDD test suites in detail. Nevertheless, we found a blog [Tomas, 2022] which discusses the reasons for slow BDD test suites in detail. Tomas [2022] discusses 9 reasons for slow BDD test suites e.g., test using a database, test through UI, dependencies between tests, and presence of dead code, etc. According to Tomas [2022], the speed of BDD test suite can be improved by taking steps like: reduction of tests' size by breaking them up, optimisation of database queries, reduction of UI interaction, etc. Question 8 is a consequence of the answer to Question 7, and it concerns the execution speed of the Gherkin test suite. We envisage the establishment of the factors that slow down the execution of the Gherkin test suite. Doing so will provide a guideline for writing optimal Gherkin tests.

Question 9 is an extension of Question 2. At the moment, Gherkin does not have a requirements traceability mechanism in place. It is difficult to know which feature a particular feature evolved from. We envisage enhancing traceability of requirements in complex systems through the adaptation of behaviour driven development techniques. Cucumber (i.e., BDD tool for Gherkin) could be extended to incorporate a requirements tree.

Question 10 concerns one of the bad smells in Gherkin specifications. According to Smart [2014] and Wynne et al. [2017], each step in a Gherkin scenario must perform a single independent task. Describing more than one action in a single scenario step is referred to as a Gherkin *Anti Pattern* by the documentation on Cucumber[‖]. Such steps must be split into different smaller steps such that each step represents a single independent action. At the moment, one of the indications of multiple actions in a single step is the use of conjunction within a Gherkin step ac-

[‖] https://cucumber.io/docs/guides/anti-patterns/

cording to the documentation on Cucumber. We already debunked this myth in 8.2. We envisage using natural language processing techniques to detect more than one independent behaviours in a single step.

Question 11 is a further breakdown of Question 7. Question 11 concerns the quality of Gherkin scenarios. The foremost purpose of the Gherkin scenarios is to communicate the requirements to the non-technical stakeholders of a project [Smart, 2014, Wynne et al., 2017]. For this purpose, the readability of the scenarios is very important. At the moment, the factors that play a role in the readability of a Gherkin scenario are unknown. We envisage a clear definition of readability in the context of Gherkin specifications. In the next step, we envisage conducting multiple experiments to establish a boundary value for each of those factors e.g., the appropriate length of a Gherkin scenario. In order to solve the issue of communication between subject matter experts and software developers, Rocha Silva [2022] propose the use of Domain-Specific languages (DSL). Although the paper does not specifically discuss the use of DSL for readability purposes, the author points out that "*Requirements expressed through a textual high-level DSL are more precise and easier to read than the same information expressed in free natural language*" implying Gherkin. However, more research is needed to study the readability of BDD scenarios described in Gherkin and DSL.

9.10 Summary

This chapter concludes this PhD research and presents the results of this research. The research presented in this thesis investigated the use of agile method, specifically, BDD in practice. The use of BDD was investigated in a project in a large organisation and afterward the scope of the research was extended to the open-source projects on GitHub. The high-level findings from these investigations show that Behaviour Driven Development must be adopted with care because its use often incurs an overhead cost of maintaining Gherkin specifications. Because of the natural language structure of Gherkin and the lack of research and available guidance, it is easy to write specifications that could impede the evolution of a system in the future. Nonetheless, the *overhead* of maintaining Gherkin could also pay off later in a project in the form of bug reduction and high-quality code. However, more research is required to investigate this. The experience and knowledge gained throughout this research provide foundational work to further investigate BDD for challenges and limitations.

Bibliography

Adil A Abdelaziz, Yaseen El-Tahir, and Raheeg Osman. Adaptive software development for developing safety critical software. In *2015 International Conference on Computing, Control, Networking, Electronics and Embedded Systems Engineering (ICCNEEE)*, pages 41–46. IEEE, 2015.

Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, and Juhani Warsta. Agile software development methods: Review and analysis. *arXiv preprint arXiv:1709.08439*, 2017.

Muhammad Faisal Abrar, Muhammad Sohail Khan, Sikandar Ali, Umar Ali, Muhammad Faran Majeed, Amjad Ali, Bahrul Amin, and Nasir Rasheed. Motivators for large-scale agile adoption from management perspective: A systematic literature review. *IEEE Access*, 7:22660–22674, 2019. doi: 10.1109/ACCESS.2019.2896212. URL https://doi.org/10.1109/ACCESS.2019.2896212.

Miltiadis Allamanis and Charles Sutton. Mining source code repositories at massive scale using language modeling. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 207–216. IEEE, 2013.

Glen B. Alleman, Michael Henderson, and Ray Seggelke. Making agile development work in a government contracting environment - measuring velocity with earned value. In *2003 Agile Development Conference (ADC 2003), 25-28 June 2003, Salt Lake City, UT, USA*, pages 114–119. IEEE Computer Society, 2003. doi: 10.1109/ADC.2003.1231460. URL https://doi.org/10.1109/ADC.2003.1231460.

Samar Alsaqqa, Samer Sawalha, and Heba Abdel-Nabi. Agile software development: Methodologies and trends. *Int. J. Interact. Mob. Technol.*, 14(11):246–270, 2020. doi: 10.3991/IJIM.V14I11.13269. URL https://doi.org/10.3991/ijim.v14i11.13269.

Ana Cláudia Amorim, Miguel Mira da Silva, Rúben Pereira, and Margarida Gonçalves. Using agile methodologies for adopting COBIT. *Inf. Syst.*, 101:101496, 2021. doi: 10.1016/j.is.2020.101496. URL https://doi.org/10.1016/j.is.2020.101496.

Areti Ampatzoglou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. The financial aspect of managing technical debt: A systematic literature review. *Inf. Softw.*

Technol., 64:52–73, 2015. doi: 10.1016/j.infsof.2015.04.001. URL https://doi.org/10.1016/j.infsof.2015.04.001.

Vard Antinyan and Henrik Sandgren. Software safety analysis to support ISO 26262-6 compliance in agile development. *IEEE Softw.*, 38(3):52–60, 2021. doi: 10.1109/MS.2020.3026145. URL https://doi.org/10.1109/MS.2020.3026145.

Jason Ard, Kristine Davidsen, and Terril Hurst. Simulation-based embedded agile development. *IEEE Software*, 31(2):97–101, 2014.

Charles Ashbacher. Succeeding with agile: Software development using scrum, by mike cohn. *J. Object Technol.*, 9(4):0, 2010. doi: 10.5381/jot.2010.9.4.r1. URL https://doi.org/10.5381/jot.2010.9.4.r1.

Pascal Aurlane. The Ultimate Guide to BDD Test Automation Frameworks. https://cucumber.io/blog/bdd/the-ultimate-guide-to-bdd-test-automation-framewor/, 2019.

Arnon Axelrod. *Unit Tests and TDD*, pages 395–424. Apress, Berkeley, CA, 2018. ISBN 978-1-4842-3832-5.

Jakob Axelsson, Efi Papatheocharous, Jaana Nyfjord, and Martin Törngren. Notes on agile and safety-critical development. *ACM SIGSOFT Softw. Eng. Notes*, 41(2):23–26, 2016. doi: 10.1145/2894784.2894796. URL https://doi.org/10.1145/2894784.2894796.

Tom Axford. Concurrency in software engineering. *Encyclopedia of Software Engineering*, 2002.

Deepika Badampudi, Samuel Fricker, and Ana María Moreno. Perspectives on productivity and delays in large-scale agile projects. In Hubert Baumeister and Barbara Weber, editors, *Agile Processes in Software Engineering and Extreme Programming - 14th International Conference, XP 2013, Vienna, Austria, June 3-7, 2013. Proceedings*, volume 149 of *Lecture Notes in Business Information Processing*, pages 180–194. Springer, 2013. doi: 10.1007/978-3-642-38314-4_13. URL https://doi.org/10.1007/978-3-642-38314-4_13.

Magdalena Balazinska, Ettore Merlo, Michel Dagenais, Bruno Laguë, and Kostas Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the Seventh Working Conference on Reverse Engineering, WCRE'00, Brisbane, Australia, November 23-25, 2000*, pages 98–107. IEEE Computer Society, 2000. doi: 10.1109/WCRE.2000.891457. URL https://doi.org/10.1109/WCRE.2000.891457.

Lingfeng Bao, Xin Xia, David Lo, and Gail C Murphy. A large scale study of long-time contributor prediction for github projects. *IEEE Transactions on Software Engineering*, 2019.

Rafael Fazzolino Pinto Barbosa. Feature-trace: an approach to generate operational profile and to support regression testing from bdd features. Master's thesis, Universidade de Brasília, 2020.

Claude Baron and Vincent Louis. Towards a continuous certification of safety-critical avionics software. *Computers in Industry*, 125:103382, 2021.

Arlinta Christy Barus. The implementation of atdd and bdd from testing perspectives. *Journal of Physics: Conference Series*, 1175(1):012112, 2019.

Richard L Baskerville. Investigating information systems with action research. *Communications of the association for information systems*, 2(1):19, 1999.

Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 56–65. IEEE, 2012.

Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave W. Binkley. Are test smells really harmful? an empirical study. *Empir. Softw. Eng.*, 20(4):1052–1094, 2015. doi: 10.1007/s10664-014-9313-0. URL `https://doi.org/10.1007/s10664-014-9313-0`.

Kent Beck and Cynthia Andres. *Extreme Programming Explained*. XP Series. Addison Wesley/Pearson Education, second edition, February 2005.

Kent Beck, Mike Beedle, Arie van Bennekum andw Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. The agile manifesto. Available at `http://agilemanifesto.org`, 2001a.

Kent Beck, Mike Hendrickson, and Martin Fowler. *Planning extreme programming*. Addison-Wesley Professional, 2001b.

Kent L. Beck. *Test-driven Development - by example*. The Addison-Wesley signature series. Addison-Wesley, 2003. ISBN 978-0-321-14653-3.

Ron Bell. Safety critical systems - a brief history of the development of guidelines and standards. In *Proceedings of the Twenty-fifth Safety-Critical Systems Symposium*, 2017.

Herbert D. Benington. Production of large computer programs. *Annals of the History of Computing*, 5(4):350–361, October 1983.

Leonard Peter Binamungu. *Detecting and correcting duplication in behaviour driven development specifications*. PhD thesis, University of Manchester, 2020.

Leonard Peter Binamungu, Suzanne M Embury, and Nikolaos Konstantinou. Detecting duplicate examples in behaviour driven development specifications. In *2018 IEEE Workshop on Validation, Analysis and Evolution of Software Tests (VST)*, pages 6–10. IEEE, 2018a.

Leonard Peter Binamungu, Suzanne M. Embury, and Nikolaos Konstantinou. Maintaining behaviour driven development specifications: Challenges and opportunities. In Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 175–184. IEEE Computer Society, 2018b. doi: 10.1109/SANER.2018.8330207. URL `https://doi.org/10.1109/SANER.2018.8330207`.

Leonard Peter Binamungu, Suzanne M. Embury, and Nikolaos Konstantinou. Characterising the quality of behaviour driven development specifications. In Viktoria Stray, Rashina Hoda, Maria Paasivaara, and Philippe Kruchten, editors, *Agile Processes in Software Engineering and Extreme Programming - 21st International Conference on Agile Software Development, XP 2020, Copenhagen, Denmark, June 8-12, 2020, Proceedings*, volume 383 of *Lecture Notes in Business Information Processing*, pages 87–102. Springer, 2020. doi: 10.1007/978-3-030-49392-9_6. URL `https://doi.org/10.1007/978-3-030-49392-9_6`.

Elizabeth Bjarnason and Markus Borg. Aligning requirements and testing: Working together toward the same goal. *IEEE Softw.*, 34(1):20–23, 2017. doi: 10.1109/MS.2017.14. URL `https://doi.org/10.1109/MS.2017.14`.

Elizabeth Bjarnason, Michael Unterkalmsteiner, Markus Borg, and Emelie Engström. A multi-case study of agile requirements engineering and the use of test cases as requirements. *Inf. Softw. Technol.*, 77:61–79, 2016. doi: 10.1016/j.infsof.2016.03.008. URL `https://doi.org/10.1016/j.infsof.2016.03.008`.

Sue Black, Paul P Boca, Jonathan P Bowen, Jason Gorman, and Mike Hinchey. Formal versus agile: Survival of the fittest. *Computer*, 42(9):37–45, 2009.

Barry Boehm. Get ready for agile methods, with care. *IEEE Computer*, 35(1):64—69, January 2002.

Barry Boehm and Richard Turner. *Balancing Agility and Discipline: A Guide for the Perplexed*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

Rodrick Borg and Martin Kropp. Automated acceptance test refactoring. In Danny Dig and Don S. Batory, editors, *Fourth Workshop on Refactoring Tools 2011, WRT '11, Waikiki, Hon-*

olulu, HI, USA, May 22, 2011, pages 15–21. ACM, 2011. doi: 10.1145/1984732.1984736. URL https://doi.org/10.1145/1984732.1984736.

Johan Borgenstierna. Behave and pyunit: A testers perspective, 2018.

Neil C. Borle, Meysam Feghhi, Eleni Stroulia, Russell Greiner, and Abram Hindle. Analyzing the effects of test driven development in github. *Empir. Softw. Eng.*, 23(4):1931–1958, 2018. doi: 10.1007/s10664-017-9576-3. URL https://doi.org/10.1007/s10664-017-9576-3.

Judy Bowen, Benjamin Weyers, and Bowen Liu. Creating formal models from informal design artefacts. *Int. J. Hum. Comput. Interact.*, 39(15):3141–3158, 2023. doi: 10.1080/10447318.2022.2095833. URL https://doi.org/10.1080/10447318.2022.2095833.

Tyson R Browning and Ralph D Heath. Reconceptualizing the effects of lean on production costs with evidence from the f-22 program. *Journal of operations management*, 27(1):23–44, 2009.

Robert Carlson and Richard Turner. Review of agile case studies for applicability to aircraft systems integration. *Procedia Computer Science*, 16:469–474, 2013.

S. E. Carpenter and A. Dagnino. Is agile too fragile for space-based systems engineering? In *2014 IEEE International Conference on Space Mission Challenges for Information Technology*, pages 38–45, 2014. doi: 10.1109/SMC-IT.2014.13.

Álvaro Carrera, Carlos Angel Iglesias, and Mercedes Garijo. Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development. *Inf. Syst. Frontiers*, 16(2):169–182, 2014. doi: 10.1007/s10796-013-9438-5. URL https://doi.org/10.1007/s10796-013-9438-5.

Oisín Cawley, Xiaofeng Wang, and Ita Richardson. Lean/agile software development methodologies in regulated environments - state of the art. In Pekka Abrahamsson and Nilay V. Oza, editors, *Lean Enterprise Software and Systems - First International Conference, LESS 2010, Helsinki, Finland, October 17-20, 2010. Proceedings*, volume 65 of *Lecture Notes in Business Information Processing*, pages 31–36. Springer, 2010.

Oisín Cawley, Ita Richardson, Xiaofeng Wang, and Marco Kuhrmann. A conceptual framework for lean regulated software development. In Dietmar Pfahl, Reda Bendraou, Richard Turner, Marco Kuhrmann, Regina Hebig, and Fabrizio Maria Maggi, editors, *Proceedings of the 2015 International Conference on Software and System Process, ICSSP 2015, Tallinn, Estonia, August 24 - 26, 2015*, pages 167–168. ACM, 2015.

Z. Chaczko, R. Braun, L. Carrion, and J. Dagher. Design of unit testing using xunit.net. In *2014 Information Technology Based Higher Education and Training (ITHET)*, pages 1–9, Sep. 2014. doi: 10.1109/ITHET.2014.7155685.

Adwait Chandorkar, Nitish Patkar, Andrea Di Sorbo, and Oscar Nierstrasz. An exploratory study on the usage of gherkin features in open-source projects. In *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*, pages 1159–1166. IEEE, 2022. doi: 10.1109/SANER53432.2022.00134. URL `https://doi.org/10.1109/SANER53432.2022.00134`.

Roderick Chapman. Industrial experience with agile in high-integrity software development. In *Safety Critical Systems Club*, 2016.

Roderick Chapman, Neil White, and Jim Woodcock. What can agile methods bring to high-integrity software development? *Commun. ACM*, 60(10):38–41, September 2017. ISSN 0001-0782. doi: 10.1145/3133233. URL `http://doi.acm.org/10.1145/3133233`.

David Chelimsky, Dave Astels, Bryan Helmkamp, Dan North, Zach Dennis, and Aslak Hellesoy. *The RSpec book: Behaviour driven development with Rspec, Cucumber, and Friends*. O'Reilly UK Ltd, 2010.

Emmanuel Chenu. Agility and lean for avionics. Paper Presented at Lean, Agile Approach to High Integrity Software, Paris, 2009. `http://manu40k.free.fr/AgilityAndLeanForAvionics1.pdf`, 2009.

Emmanuel Chenu. Agile & lean software development for avionic software. In *6th European Congress on Real Time Software and Systems*, pages 1–3, Toulouse, France, February 2012. Association Aéronautique Astronautique de France.

Jacques M Chevalier and Daniel J Buckles. *Participatory action research: Theory and methods for engaged inquiry*. Routledge, 2019.

Chun Yong Chong and Sai Peck Lee. Can commit change history reveal potential fault prone classes? A study on github repositories. In Marten van Sinderen and Leszek A. Maciaszek, editors, *Software Technologies - 13th International Conference, ICSOFT 2018, Porto, Portugal, July 26-28, 2018, Revised Selected Papers*, volume 1077 of *Communications in Computer and Information Science*, pages 266–281. Springer, 2018. doi: 10.1007/978-3-030-29157-0_12. URL `https://doi.org/10.1007/978-3-030-29157-0_12`.

Jan Chong, Robert Plummer, Larry J. Leifer, Scott R. Klemmer, Ozgur Eris, and George Toye. Pair programming: When and why it works. In *Proceedings of the 17th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2005, Brighton, UK, June 29 -*

July 1, 2005, page 5. Psychology of Programming Interest Group, 2005. URL `https://ppig.org/papers/2005-ppig-17th-chong/`.

Heting Chu and Qing Ke. Research methods: What's in the name? *Library & Information Science Research*, 39(4):284–294, 2017.

John Ciliberti. Test-driven development with asp. net core mvc. In *ASP. NET Core Recipes*, pages 221–250. Springer, 2017.

Inta Cinite and Linda E Duxbury. Measuring the behavioral properties of commitment and resistance to organizational change. *The Journal of Applied Behavioral Science*, 54(2):113–139, 2018.

Paul Clarke, Marion Lepmets, Fergal McCaffery, Anita Finnegan, Alec Dorling, and Derek Flood. Mdevspice-a comprehensive solution for manufacturers and assessors of safety-critical medical device software. In *International Conference on Software Process Improvement and Capability Determination*, pages 274–278. Springer, 2014.

Alistair Cockburn. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley Professional, October 2004.

David J Coe and Jeffrey H Kulick. A model-based agile process for do-178c certification. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, page 1, 2013.

Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

CollabNet VersionOne. 7th annual state of agile report. `https://www.stateofagile.com`, May 2013.

CollabNet VersionOne. 8th annual state of agile report. `https://www.stateofagile.com`, May 2014.

CollabNet VersionOne. 9th annual state of agile report. `https://www.stateofagile.com`, May 2015.

CollabNet VersionOne. 10th annual state of agile report. `https://www.stateofagile.com`, May 2016.

CollabNet VersionOne. 11th annual state of agile report. `https://www.stateofagile.com`, May 2017.

CollabNet VersionOne. 12th annual state of agile report. `https://www.stateofagile.com`, May 2018.

CollabNet VersionOne. 13th annual state of agile report. `https://www.stateofagile.com`, May 2019.

CollabNet VersionOne. 14th annual state of agile report. `https://www.stateofagile.com`, MAY 2020.

Kieran Conboy. Agility from first principles: Reconstructing the concept of agility in information systems development. *Information Systems Research*, 20(3):329–354, 2009.

Kieran Conboy and Noel Carroll. Implementing large-scale agile frameworks: challenges and recommendations. *IEEE Software*, 36(2):44–50, 2019.

Michael Coram and Shawn Bohner. The impact of agile methods on software project management. In *12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS'05)*, pages 363–370. IEEE, 2005.

Lucas C. Cordeiro, Raimundo S. Barreto, Rafael Barcelos, Meuse N. Oliveira Jr., Vicente Lucena, and Paulo Romero Martins Maciel. TXM: an agile HW/SW development methodology for building medical devices. *ACM SIGSOFT Softw. Eng. Notes*, 32(6), 2007. doi: 10.1145/1317471.1317476. URL `https://doi.org/10.1145/1317471.1317476`.

Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with github. *IEEE Access*, 5:7173–7192, 2017. doi: 10.1109/ACCESS.2017.2682323. URL `https://doi.org/10.1109/ACCESS.2017.2682323`.

Daniela S. Cruzes and Tore Dybå. Recommended steps for thematic synthesis in software engineering. In *Proceedings of the 5th International Symposium on Empirical Software Engineering and Measurement, ESEM 2011, Banff, AB, Canada, September 22-23, 2011*, pages 275–284. IEEE Computer Society, 2011. doi: 10.1109/ESEM.2011.36. URL `https://doi.org/10.1109/ESEM.2011.36`.

CucumberStudio. Anti-patterns. `https://cucumber.io/docs/guides/anti-patterns/?lang=java`, 2016.

CucumberStudio. Anti-patterns. `https://cucumber.io/docs/gherkin/reference/#scenario-outline`, 2019a.

CucumberStudio. Writing better Gherkin. `https://cucumber.io/docs/bdd/better-gherkin/`, 2019b.

Karina Curcio, Tiago Navarro, Andreia Malucelli, and Sheila S. Reinehr. Requirements engineering: A systematic mapping study in agile software development. *J. Syst. Softw.*, 139:32–50, 2018. doi: 10.1016/j.jss.2018.01.036. URL `https://doi.org/10.1016/j.jss.2018.01.036`.

Karen Sue Danley and Marsha Langer Ellison. *A handbook for participatory action researchers*. Boston University, 1999.

Dai Davis. Legal aspects of safety critical systems. In Gerhard Rabe, editor, *14th International Conference on Computer Safety, Reliability and Security, Safecomp 1995, Belgirate, Italy, October 11-13, 1995*, pages 156–170. Springer, 1995. doi: 10.1007/978-1-4471-3054-3_12. URL https://doi.org/10.1007/978-1-4471-3054-3_12.

Hugo Sica de Andrade, Eduardo Santana de Almeida, and Ivica Crnkovic. Architectural bad smells in software product lines: an exploratory study. In Anna Liu, John Klein, and Antony Tang, editors, *Proceedings of the WICSA 2014 Companion Volume, Sydney, NSW, Australia, April 7-11, 2014*, pages 12:1–12:6. ACM, 2014. doi: 10.1145/2578128.2578237. URL https://doi.org/10.1145/2578128.2578237.

Jose Luis de la Vara, Alejandra Ruiz, Katrina Attwood, Huáscar Espinoza, Rajwinder Kaur Panesar-Walawege, Ángel López, Idoya del Río, and Tim Kelly. Model-based specification of safety compliance needs for critical systems: A holistic generic metamodel. *Inf. Softw. Technol.*, 72:16–30, 2016. doi: 10.1016/j.infsof.2015.11.008. URL https://doi.org/10.1016/j.infsof.2015.11.008.

Elder Vicente de Paulo Sobrinho, Andrea De Lucia, and Marcelo de Almeida Maia. A systematic literature review on bad smells-5 w's: Which, when, what, who, where. *IEEE Trans. Software Eng.*, 47(1):17–66, 2021. doi: 10.1109/TSE.2018.2880977. URL https://doi.org/10.1109/TSE.2018.2880977.

Pedro Lopes de Souza, Antônio Francisco do Prado, Wanderley Lopes de Souza, Sissi Marilia dos Santos Forghieri Pereira, and Luís Ferreira Pires. Combining behaviour-driven development with scrum for software development in the education domain. In Slimane Hammoudi, Michal Smialek, Olivier Camp, and Joaquim Filipe, editors, *ICEIS 2017 - Proceedings of the 19th International Conference on Enterprise Information Systems, Volume 2, Porto, Portugal, April 26-29, 2017*, pages 449–458. SciTePress, 2017. doi: 10.5220/0006336804490458. URL https://doi.org/10.5220/0006336804490458.

Ian Dees, Matt Wynne, and Aslak Hellesoy. *Cucumber Recipes: Automate Anything with BDD Tools and Techniques*. Pragmatic Bookshelf, 2013.

Daniel Ryan Degutis. How to speed up bdd automated acceptance testing for safety-critical systems. B.S. thesis, University of Stuttgart, 2018.

Jeremy Dick, M. Elizabeth C. Hull, and Ken Jackson. *Requirements Engineering, 4th Edition*. Springer, 2017. ISBN 978-3-319-61072-6. doi: 10.1007/978-3-319-61073-3. URL https://doi.org/10.1007/978-3-319-61073-3.

Melanie Diepenbeck, Ulrich Kühne, Mathias Soeken, Daniel Große, and Rolf Drechsler. Behaviour driven development for hardware design. *IPSJ Trans. Syst. LSI Des. Methodol.*, 11: 29–45, 2018. doi: 10.2197/ipsjtsldm.11.29. URL `https://doi.org/10.2197/ipsjtsldm.11.29`.

Kim-Karol Dikert, Maria Paasivaara, and Casper Lassenius. Challenges and success factors for large-scale agile transformations: A systematic literature review. *J. Syst. Softw.*, 119:87–108, 2016. doi: 10.1016/j.jss.2016.06.013. URL `https://doi.org/10.1016/j.jss.2016.06.013`.

Torgeir Dingsøyr and Casper Lassenius. Emerging themes in agile software development: Introduction to the special section on continuous value delivery. *Information and Software Technology*, 77:56–60, 2016.

Simone do Rocio Senger de Souza, Maria A. S. Brito, Rodolfo A. Silva, Paulo Sergio Lopes de Souza, and Ed Zaluska. Research in concurrent software testing: a systematic review. In João Lourenço and Eitan Farchi, editors, *Proceedings of the 9th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2011, Toronto, ON, Canada, July 17-21, 2011*, pages 1–5. ACM, 2011. doi: 10.1145/2002962.2002964. URL `https://doi.org/10.1145/2002962.2002964`.

Ernani César dos Santos and Patricia Vilain. Automated acceptance tests as software requirements: An experiment to compare the applicability of fit tables and gherkin language. In Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar, editors, *Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018, Porto, Portugal, May 21-25, 2018, Proceedings*, volume 314 of *Lecture Notes in Business Information Processing*, pages 104–119. Springer, 2018. doi: 10.1007/978-3-319-91602-6_7. URL `https://doi.org/10.1007/978-3-319-91602-6_7`.

Osama Doss and Tim P. Kelly. The 4+1 principles of software safety assurance and their implications for scrum. In Helen Sharp and Tracy Hall, editors, *Agile Processes, in Software Engineering, and Extreme Programming - 17th International Conference, XP 2016, Edinburgh, UK, May 24-27, 2016, Proceedings*, volume 251 of *Lecture Notes in Business Information Processing*, pages 286–290. Springer, 2016. doi: 10.1007/978-3-319-33515-5_27. URL `https://doi.org/10.1007/978-3-319-33515-5_27`.

Bruce Powel Douglass. *Agile Systems Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2016.

Ana Marcia Debiasi Duarte, Denio Duarte, and Marcello Thiry. Tracebok: Toward a software requirements traceability body of knowledge. In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*, pages

236–245. IEEE Computer Society, 2016. doi: 10.1109/RE.2016.32. URL `https://doi.org/10.1109/RE.2016.32`.

Aleksander Grzegorz Duszkiewicz, Jacob Glumby Sørensen, Niclas Johansen, Henry Edison, and Thiago Rocha Silva. On identifying similar user stories to support agile estimation based on historical data. In Palash Bera, Fabiano Dalpiaz, and Yves Wautelet, editors, *Short Paper Proceedings of the First International Workshop on Agile Methods for Information Systems Engineering (Agil-ISE 2022) co-located with the 34th International Conference on Advanced Information Systems Engineering (CAiSE 2022), Leuven, Belgium, June 6, 2022*, volume 3134 of *CEUR Workshop Proceedings*, pages 21–26. CEUR-WS.org, 2022. URL `https://ceur-ws.org/Vol-3134/paper-4.pdf`.

Tore Dybå and Torgeir Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and software technology*, 50(9-10):833–859, 2008.

Amel DŽANIĆ, Amel Toroman, and Alma DŽANIĆ. Agile software development: Model, methods, advantages and disadvantages. *Acta Technica Corviniensis-Bulletin of Engineering*, 15(4), 2022.

Christof Ebert, Marco Kuhrmann, and Rafael Prikladnicki. Global software engineering: Evolution and trends. In *11th IEEE International Conference on Global Software Engineering, ICGSE 2016, Orange County, CA, USA, August 2-5, 2016*, pages 144–153. IEEE Computer Society, 2016.

Henry Edison, Xiaofeng Wang, and Kieran Conboy. Comparing methods for large-scale agile software development: A systematic literature review. *IEEE Trans. Software Eng.*, 48(8): 2709–2731, 2022. doi: 10.1109/TSE.2021.3069039. URL `https://doi.org/10.1109/TSE.2021.3069039`.

Abigail Egbreghts. A literature review of behavior driven development using grounded theory. In *27th Twente Student Conference on IT. Available at: https://pdfs. semanticscholar. org/4f03/ec0675d08cfd1ecdbaac3361a29d756ce656. pdf*, 2017.

Heba Elshandidy, Sherif Mazen, Ehab Hassanein, and Eman Nasr. Using behaviour-driven requirements engineering for establishing and managing agile product lines. *International Journal of Advanced Computer Science and Applications*, 12(2), 2021.

Dennis G Erwin and Andrew N Garman. Resistance to organizational change: linking research and practice. *Leadership & Organization Development Journal*, 2010.

Tor Erlend Fægri and Nils Brede Moe. Re-conceptualizing requirements engineering: findings from a large-scale, agile project. In Maria Paasivaara, editor, *Scientific Workshop Proceedings*

of the XP2015, Helsinki, Finland, May 25-29, 2015, page 4. ACM, 2015. doi: 10.1145/2764979.2764983. URL `https://doi.org/10.1145/2764979.2764983`.

Viktor Farcic and Alex Garcia. *Test-Driven Java Development: Invoke TDD principles for end-to-end application development*. Packt Publishing Ltd, 2018.

Henning Femmer, Daniel Méndez Fernández, Stefan Wagner, and Sebastian Eder. Rapid quality assurance with requirements smells. *J. Syst. Softw.*, 123:190–213, 2017. doi: 10.1016/j.jss.2016.02.047. URL `https://doi.org/10.1016/j.jss.2016.02.047`.

Brian Fitzgerald, Gerard Hartnett, and Kieran Conboy. Customising agile methods to software practices at intel shannon. *EJIS*, 15(2):200–213, 2006.

Brian Fitzgerald, Klaas-Jan Stol, Ryan O'Sullivan, and Donal O'Brien. Scaling agile methods to regulated environments: an industry case study. In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 863–872. IEEE Computer Society, 2013. doi: 10.1109/ICSE.2013.6606635. URL `https://doi.org/10.1109/ICSE.2013.6606635`.

Bent Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2): 219–245, 2006.

Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5: 1–38, 2012. doi: 10.5381/jot.2012.11.2.a5. URL `https://doi.org/10.5381/jot.2012.11.2.a5`.

Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley object technology series. Addison-Wesley, 1999. ISBN 978-0-201-48567-7. URL `http://martinfowler.com/books/refactoring.html`.

Martin Fowler. *Domain-Specific Languages*. The Addison-Wesley signature series. Addison-Wesley, 2011. ISBN 978-0-321-71294-3. URL `http://vig.pearsoned.com/store/product/1,1207,store-12521_isbn-0321712943,00.html`.

Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

Samuel A Fricker, Rainer Grau, and Adrian Zwingli. Requirements engineering: best practice. In *Requirements Engineering for Digital Health*, pages 25–46. Springer, 2015.

Davide Fucci, Hakan Erdogmus, Burak Turhan, Markku Oivo, and Natalia Juristo. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *IEEE*

Trans. Software Eng., 43(7):597–614, 2017. doi: 10.1109/TSE.2016.2616877. URL https://doi.org/10.1109/TSE.2016.2616877.

Davide Fucci, Cristina Palomares, Xavier Franch, Dolors Costal, Mikko Raatikainen, Martin Stettinger, Zijad Kurtanovic, Tero Kojo, Lars Koenig, Andreas A. Falkner, Gottfried Schenner, Fabrizio Brasca, Tomi Männistö, Alexander Felfernig, and Walid Maalej. Needs and challenges for a platform to support large-scale requirements engineering: a multiple-case study. In Markku Oivo, Daniel Méndez Fernández, and Audris Mockus, editors, *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM 2018, Oulu, Finland, October 11-12, 2018*, pages 19:1–19:10. ACM, 2018. doi: 10.1145/3239235.3240498. URL https://doi.org/10.1145/3239235.3240498.

Barbara Gallina, Faiz Ul Muram, and Julieth Patricia Castellanos Ardila. Compliance of agilized (software) development processes with safety standards: a vision. In Ademar Aguiar, editor, *Proceedings of the 19th International Conference on Agile Software Development, XP 2019, Companion, Porto, Portugal, May 21-25, 2018*, pages 14:1–14:6. ACM, 2018. doi: 10.1145/3234152.3234175. URL https://doi.org/10.1145/3234152.3234175.

Taghi Javdani Gandomani, Hazura Zulzalil, AA Ghani, Abu Bakar Md Sultan, and Khaironi Yatim Sharif. How human aspects impress agile software development transition and adoption. *International Journal of Software Engineering and its Applications*, 8(1):129–148, 2014.

Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Toward a catalogue of architectural bad smells. In Raffaela Mirandola, Ian Gorton, and Christine Hofmeister, editors, *Architectures for Adaptive Software Systems, 5th International Conference on the Quality of Software Architectures, QoSA 2009, East Stroudsburg, PA, USA, June 24-26, 2009, Proceedings*, volume 5581 of *Lecture Notes in Computer Science*, pages 146–162. Springer, 2009. doi: 10.1007/978-3-642-02351-4_10. URL https://doi.org/10.1007/978-3-642-02351-4_10.

Vahid Garousi and Baris Küçük. Smells in software test code: A survey of knowledge in industry and academia. *J. Syst. Softw.*, 138:52–81, 2018. doi: 10.1016/j.jss.2017.12.013. URL https://doi.org/10.1016/j.jss.2017.12.013.

Vahid Garousi, Baris Kucuk, and Michael Felderer. What we know about smells in software test code. *IEEE Software*, 36(3):61–73, 2018.

Kevin Gary, Andinet Enquobahrie, Luis Ibáñez, Patrick Cheng, Ziv Yaniv, Kevin Cleary, Shylaja Kokoori, Benjamin Muffih, and John Heidenreich. Agile methods for open source safety-critical software. *Softw. Pract. Exp.*, 41(9):945–962, 2011. doi: 10.1002/spe.1075. URL https://doi.org/10.1002/spe.1075.

Xiaocheng Ge, Richard F. Paige, and John A. McDermid. An iterative approach for development of safety-critical software and safety arguments. In Sallyann Freudenberg and Joseph Chao, editors, *2010 Agile Conference, AGILE 2010, Orlando, Florida, USA, August 9-13, 2010*, pages 35–43. IEEE Computer Society, 2010. doi: 10.1109/AGILE.2010.10. URL `https://doi.org/10.1109/AGILE.2010.10`.

Daniel Gerster and Christian Dremel. Agile contracts: Learnings from an autonomous driving sourcing project. In Jan vom Brocke, Shirley Gregor, and Oliver Müller, editors, *27th European Conference on Information Systems - Information Systems for a Sharing Society, ECIS 2019, Stockholm and Uppsala, Sweden, June 8-14, 2019*, 2019. URL `https://aisel.aisnet.org/ecis2019_rip/1`.

Martin Glas and Sven Ziemer. Challenges for agile development of large systems in the aviation industry. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 901–908. ACM, 2009.

Luis Alberto Cisneros Gómez. Analysis of the impact of test based development techniques (tdd, bdd, and atdd) to the software life cycle. Master's thesis, Instituto Politecnico de Leiria (Portugal), 2018.

Gildarcio Sousa Goncalves, Glaydson Luiz Bertoze Lima, Rene Esteves Maria, Ramiro Tadeu Wisnieski, Mayara Valeria Morais dos Santos, Manasseis Alves Ferreira, Alexandre Chaves da Silva, Andre Olimpio, Andre Gomes Lamas Otero, Luiz Eduardo Guarino de Vasconcelos, et al. An interdisciplinary academic project for spatial critical embedded system agile development. In *2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC)*, pages 8C3–1. IEEE, 2015.

Janusz Górski and Katarzyna Lukasiewicz. Assessment of risks introduced to safety critical software by agile practices - a software engineer's perspective. *Comput. Sci.*, 13(4):165–182, 2012. doi: 10.7494/csci.2012.13.4.165. URL `https://doi.org/10.7494/csci.2012.13.4.165`.

Raman Goyal, Gabriel Ferreira, Christian Kästner, and James Herbsleb. Identifying unusual commits on github. *Journal of Software: Evolution and Process*, 30(1):e1893, 2018.

Giovanni Grano, Cristian De Iaco, Fabio Palomba, and Harald C. Gall. Pizza versus pinsa: On the perception and measurability of unit test code quality. In *IEEE International Conference on Software Maintenance and Evolution, ICSME 2020, Adelaide, Australia, September 28 - October 2, 2020*, pages 336–347. IEEE, 2020. doi: 10.1109/ICSME46990.2020.00040. URL `https://doi.org/10.1109/ICSME46990.2020.00040`.

Julián Grigera, Alejandra Garrido, and José Matías Rivero. A tool for detecting bad usability smells in an automatic way. In Sven Casteleyn, Gustavo Rossi, and Marco Winckler, editors, *Web Engineering, 14th International Conference, ICWE 2014, Toulouse, France, July 1-4, 2014. Proceedings*, volume 8541 of *Lecture Notes in Computer Science*, pages 490–493. Springer, 2014. doi: 10.1007/978-3-319-08245-5_34. URL `https://doi.org/10.1007/978-3-319-08245-5_34`.

Thomas Gruber, Egbert Althammer, and Erwin Schoitsch. Field test methods for a co-operative integrated traffic management system. In Erwin Schoitsch, editor, *Computer Safety, Reliability, and Security*, pages 183–195, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

Jaber F Gubrium and James A Holstein. *Handbook of interview research: Context and method*. Sage Publications, 2001.

Ilhan Gunbayi. Action research as a mixed methods research: Definition, philosophy, types, process, political and ethical issues and pros and cons. *Journal of Mixed Methods Studies*, (2), 2020.

Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn B. Seaman. Exploring the costs of technical debt management - a case study. *Empir. Softw. Eng.*, 21(1):159–182, 2016. doi: 10.1007/s10664-014-9351-7. URL `https://doi.org/10.1007/s10664-014-9351-7`.

Abhimanyu Gupta. Generation of multiple conceptual models from user stories in agile. In *REFSQ Workshops*, 2019.

A. Hajou, Ronald S. Batenburg, and Slinger Jansen. An insight into the difficulties of software development projects in the pharmaceutical industry. *Lecture Notes in Software Engineering*, 3(4):267– 275, November 2015.

Ali Hajou, Ronald Batenburg, and Slinger Jansen. How the pharmaceutical industry and agile software development methods conflict: A systematic literature review. In Bernady O. Apduhan, Ana Maria A. C. Rocha, Sanjay Misra, David Taniar, Osvaldo Gervasi, and Beniamino Murgante, editors, *2014 14th International Conference on Computational Science and Its Applications, Guimaraes, Portugal, June 30 - July 3, 2014*, pages 40–48. IEEE Computer Society, 2014.

M Hammersley, P Foster, and R Gomm. *Case study and generalisation*. Sage, 2000.

Dawson R Hancock and Bob Algozzine. *Doing case study research: A practical guide for beginning researchers*. Teachers College Press, 2017.

Jo Erskine Hannay, Tore Dybå, Erik Arisholm, and Dag I. K. Sjøberg. The effectiveness of pair programming: A meta-analysis. *Inf. Softw. Technol.*, 51(7):1110–1122, 2009. doi: 10.1016/J.

INFSOF.2009.02.001. URL https://doi.org/10.1016/j.infsof.2009.02.001.

Kirsten Mark Hansen, Anders P. Ravn, and Victoria Stavridou. From safety analysis to software requirements. *IEEE Trans. Software Eng.*, 24(7):573–584, 1998. doi: 10.1109/32.708570. URL https://doi.org/10.1109/32.708570.

Geir Kjetil Hanssen, Børge Haugset, Tor Stålhane, Thor Myklebust, and Ingar Kulbrandstad. Quality assurance in scrum applied to safety critical software. In Helen Sharp and Tracy Hall, editors, *Agile Processes, in Software Engineering, and Extreme Programming - 17th International Conference, XP 2016, Edinburgh, UK, May 24-27, 2016, Proceedings*, volume 251 of *Lecture Notes in Business Information Processing*, pages 92–103. Springer, 2016. doi: 10.1007/978-3-319-33515-5_8. URL https://doi.org/10.1007/978-3-319-33515-5_8.

John Hatcliff, Alan Wassyng, Tim Kelly, Cyrille Comar, and Paul L. Jones. Certifiably safe software-dependent systems: challenges and directions. In James D. Herbsleb and Matthew B. Dwyer, editors, *Proceedings of the on Future of Software Engineering, FOSE 2014, Hyderabad, India, May 31 - June 7, 2014*, pages 182–200. ACM, 2014. doi: 10.1145/2593882.2593895. URL https://doi.org/10.1145/2593882.2593895.

Bi He, Bin Wang, Li Guo, Tongyao Yang, and Xin Xiong. A hierarchical modeling method based on model-driven development in real-time control system design. In *Proceedings of the 32nd Chinese Control Conference*, pages 5357–5362. IEEE, 2013.

Lise Tordrup Heeager. How can agile and documentation-driven methods be meshed in practice? In Giovanni Cantone and Michele Marchesi, editors, *Agile Processes in Software Engineering and Extreme Programming - 15th International Conference, XP 2014, Rome, Italy, May 26-30, 2014. Proceedings*, volume 179 of *Lecture Notes in Business Information Processing*, pages 62–77. Springer, 2014. doi: 10.1007/978-3-319-06862-6_5. URL https://doi.org/10.1007/978-3-319-06862-6_5.

Lise Tordrup Heeager and Peter Axel Nielsen. A conceptual model of agile software development in a safety-critical context: A systematic literature review. *Inf. Softw. Technol.*, 103:22–39, 2018. doi: 10.1016/j.infsof.2018.06.004. URL https://doi.org/10.1016/j.infsof.2018.06.004.

Ville T. Heikkilä, Maria Paasivaara, Casper Lassenius, Daniela E. Damian, and Christian Engblom. Managing the requirements flow from strategy to release in large-scale agile development: a case study at ericsson. *Empir. Softw. Eng.*, 22(6):2892–2936, 2017. doi: 10.1007/s10664-016-9491-z. URL https://doi.org/10.1007/s10664-016-9491-z.

Aslak Hellesøy. BDD is not test automation. `https://cucumber.io/blog/bdd/bdd-is-not-test-automation/`, 2020.

James D. Herbsleb and Audris Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Trans. Software Eng.*, 29(6):481–494, 2003.

Philipp Hohl, Jil Klünder, Arie van Bennekum, Ryan Lockard, James Gifford, Jürgen Münch, Michael Stupperich, and Kurt Schneider. Back to the future: origins and directions of the "agile manifesto" - views of the originators. *J. Softw. Eng. Res. Dev.*, 6:15, 2018. doi: 10.1186/s40411-018-0059-z. URL `https://doi.org/10.1186/s40411-018-0059-z`.

James A Holstein et al. *Handbook of interview research: Context and method*. Sage, 2002.

Philip M Huang, Ann G Darrin, and Andrew A Knuth. Agile hardware and software system engineering for innovation. In *2012 IEEE Aerospace Conference*, pages 1–10. IEEE, 2012.

D. Laurie Hughes, Nripendra P. Rana, and Antonis C. Simintiras. The changing landscape of IS project failure: an examination of the key factors. *J. Enterp. Inf. Manag.*, 30(1):142–165, 2017. doi: 10.1108/JEIM-01-2016-0029. URL `https://doi.org/10.1108/JEIM-01-2016-0029`.

Irum Inayat, Lauriane Moraes, Maya Daneva, and Siti Salwah Salim. A reflection on agile requirements engineering: solutions brought and challenges posed. In Maria Paasivaara, editor, *Scientific Workshop Proceedings of the XP2015, Helsinki, Finland, May 25-29, 2015*, page 6. ACM, 2015a. doi: 10.1145/2764979.2764985. URL `https://doi.org/10.1145/2764979.2764985`.

Irum Inayat, Siti Salwah Salim, Sabrina Marczak, Maya Daneva, and Shahaboddin Shamshirband. A systematic literature review on agile requirements engineering practices and challenges. *Comput. Hum. Behav.*, 51:915–929, 2015b. doi: 10.1016/j.chb.2014.10.046. URL `https://doi.org/10.1016/j.chb.2014.10.046`.

Javed Iqbal, Rodina B Ahmad, Muzafar Khan, Sultan Alyahya, Mohd Hairul Nizam Nasir, Adnan Akhunzada, and Muhammad Shoaib. Requirements engineering issues causing software development outsourcing failure. *PloS one*, 15(4):e0229785, 2020.

Mohsin Irshad, Ricardo Britto, and Kai Petersen. Adapting behavior driven development (BDD) for large-scale software systems. *J. Syst. Softw.*, 177:110944, 2021. doi: 10.1016/j.jss.2021.110944. URL `https://doi.org/10.1016/j.jss.2021.110944`.

Mohsin Irshad, Jürgen Börstler, and Kai Petersen. Supporting refactoring of BDD specifications - an empirical study. *Inf. Softw. Technol.*, 141:106717, 2022. doi: 10.1016/j.infsof.2021.106717. URL `https://doi.org/10.1016/j.infsof.2021.106717`.

Gibrail Islam and Tim Storer. A case study of agile software development for safety-critical systems projects, 2020a. URL `https://doi.org/10.1016/j.ress.2020.106954`.

Gibrail Islam and Tim Storer. A case study of agile software development for safety-critical systems projects. *Reliab. Eng. Syst. Saf.*, 200:106954, 2020b. doi: 10.1016/j.ress.2020.106954. URL `https://doi.org/10.1016/j.ress.2020.106954`.

Michael A. Jackson. *Problem Frames - Analysing and Structuring Software Development Problems*. Pearson Education, 2000. ISBN 978-0-2015-9627-4. URL `http://www.pearsoned.co.uk/Bookshop/detail.asp?item=100000000004768`.

Samireh Jalali and Claes Wohlin. Systematic literature studies: database searches vs. backward snowballing. In *Proceedings of the 2012 ACM-IEEE international symposium on empirical software engineering and measurement*, pages 29–38. IEEE, 2012.

Hanne-Gro Jamissen. The challenges to the safety process when using agile development models. Master's thesis, Østfeld University College, 2012.

Andrew P Johnson. *A short guide to action research*. Allyn and Bacon, 2008.

Danielle L. Jones and Scott D. Fleming. What use is a backseat driver? A qualitative investigation of pair programming. In Caitlin Kelleher, Margaret M. Burnett, and Stefan Sauer, editors, *2013 IEEE Symposium on Visual Languages and Human Centric Computing, San Jose, CA, USA, September 15-19, 2013*, pages 103–110. IEEE Computer Society, 2013. doi: 10.1109/VLHCC.2013.6645252. URL `https://doi.org/10.1109/VLHCC.2013.6645252`.

Henrik Jonsson, Stig Larsson, and Sasikumar Punnekkat. Agile practices in regulated railway software development. In *23rd IEEE International Symposium on Software Reliability Engineering Workshops, ISSRE Workshops, Dallas, TX, USA, November 27-30, 2012*, pages 355–360. IEEE Computer Society, 2012. doi: 10.1109/ISSREW.2012.80. URL `https://doi.org/10.1109/ISSREW.2012.80`.

Brendan Julian, James Noble, and Craig Anslow. Agile practices in practice: Towards a theory of agile adoption and process evolution. In Philippe Kruchten, Steven Fraser, and François Coallier, editors, *Agile Processes in Software Engineering and Extreme Programming - 20th International Conference, XP 2019, Montréal, QC, Canada, May 21-25, 2019, Proceedings*, volume 355 of *Lecture Notes in Business Information Processing*, pages 3–18. Springer, 2019. doi: 10.1007/978-3-030-19034-7_1. URL `https://doi.org/10.1007/978-3-030-19034-7_1`.

Natalia Juristo Juzgado and Ana María Moreno. *Basics of software engineering experimentation*. Kluwer, 2001. ISBN 978-0-7923-7990-4.

Veerapaneni Esther Jyothi and K Nageswara Rao. Effective implementation of agile practices. *International Journal of Advanced Computer Science and Applications*, 2(3), 2011.

Matti Kaisti, Ville Rantala, Tapio Mujunen, Sami Hyrynsalmi, Kaisa Könnölä, Tuomas Mäkilä, and Teijo Lehtonen. Agile methods for embedded systems development - a literature review and a mapping study. *EURASIP J. Emb. Sys.*, 2013:15, 2013.

Martin Kalenda, Petr Hyna, and Bruno Rossi. Scaling agile in large organizations: Practices, challenges, and success factors. *J. Softw. Evol. Process.*, 30(10), 2018. doi: 10.1002/smr.1954. URL https://doi.org/10.1002/smr.1954.

Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. Germán, and Daniela E. Damian. An in-depth study of the promises and perils of mining github. *Empir. Softw. Eng.*, 21(5):2035–2071, 2016. doi: 10.1007/s10664-015-9393-5. URL https://doi.org/10.1007/s10664-015-9393-5.

Hanna Kallio, Anna-Maija Pietilä, Martin Johnson, and Mari Kangasniemi. Systematic methodological review: developing a framework for a qualitative semi-structured interview guide. *Journal of advanced nursing*, 72(12):2954–2965, 2016.

Rashidah Kasauli, Eric Knauss, Benjamin Kanagwa, Agneta Nilsson, and Gul Calikli. Safety-critical systems and agile development: A mapping study. In Tomás Bures and Lefteris Angelis, editors, *44th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2018, Prague, Czech Republic, August 29-31, 2018*, pages 470–477. IEEE Computer Society, 2018a. doi: 10.1109/SEAA.2018.00082. URL https://doi.org/10.1109/SEAA.2018.00082.

Rashidah Kasauli, Grischa Liebel, Eric Knauss, Swathi Gopakumar, and Benjamin Kanagwa. Requirements engineering challenges in large-scale agile system development. In Matthias Tichy, Eric Bodden, Marco Kuhrmann, Stefan Wagner, and Jan-Philipp Steghöfer, editors, *Software Engineering und Software Management 2018, Fachtagung des GI-Fachbereichs Softwaretechnik, SE 2018, 5.-9. März 2018, Ulm, Germany*, volume P-279 of *LNI*, pages 133–135. Gesellschaft für Informatik, 2018b. URL https://dl.gi.de/20.500.12116/16327.

Rashidah Kasauli, Eric Knauss, Jennifer Horkoff, Grischa Liebel, and Francisco Gomes de Oliveira Neto. Requirements engineering challenges and practices in large-scale agile system development. *J. Syst. Softw.*, 172:110851, 2021. doi: 10.1016/j.jss.2020.110851. URL https://doi.org/10.1016/j.jss.2020.110851.

Mohamad Kassab. The changing landscape of requirements engineering practices over the past decade. In Richard Berntsson-Svensson, Maya Daneva, Neil A. Ernst, Sabrina Marczak, and Nazim H. Madhavji, editors, *2015 IEEE Fifth International Workshop on Empirical Requirements Engineering, EmpiRE 2015, Ottawa, ON, Canada, August 24, 2015*, pages 1–8. IEEE Computer Society, 2015. doi: 10.1109/EmpiRE.2015.7431299. URL `https://doi.org/10.1109/EmpiRE.2015.7431299`.

Tim Kelly. Software certification: Where is confidence won and lost? *Addressing Systems Safety Challenges, T. Anderson, C. Dale (Eds), Safety Critical Systems Club*, 2014.

Stephen Kemmis, Robin McTaggart, and Rhonda Nixon. *The action research planner: Doing critical participatory action research*. Springer Science & Business Media, 2013.

Dave Kennedy. Smelly Cucumbers. `https://www.sitepoint.com/smelly-cucumbers/`, 2012.

Mary M Kennedy. Generalizing from single case studies. *Evaluation quarterly*, 3(4):661–678, 1979.

Elizabeth Keogh. Bdd: A lean toolkit. In *Processings of Lean Software & Systems Conference, Atlanta*, 2010.

PM Khan and MMS Sufyan Beg. Extended decision support matrix for selection of sdlc-models on traditional and agile software development projects. In *2013 Third International Conference on Advanced Computing and Communication Technologies (ACCT)*, pages 8–15. IEEE, 2013.

V. Khorikov. *Unit Testing*. MANNING PUBN, 2020a. ISBN 9781617296277. URL `https://books.google.co.uk/books?id=CbvZyAEACAAJ`.

Vladimir Khorikov. *Unit Testing Principles, Practices, and Patterns*. Manning Publications, 1 edition, 2020b. ISBN 1617296279;9781617296277;.

Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, pages 1–11, 2012.

Sara Kindon, Rachel Pain, and Mike Kesby. *Participatory action research approaches and methods: Connecting people, participation and place*, volume 22. Routledge, 2007.

Barbara A. Kitchenham, Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen G. Linkman. Systematic literature reviews in software engineering - A systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, 2009. doi: 10.1016/j.infsof.2008.09.009. URL `https://doi.org/10.1016/j.infsof.2008.09.009`.

Andrew Knight. Bad Gherkin. https://automationpanda.com/tag/gherkin/, 2017.

John C Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, pages 547–550. ACM, 2002.

Sascha Konrad and Michael Gall. Requirements engineering in the development of large-scale systems. In *16th IEEE International Requirements Engineering Conference, RE 2008, 8-12 September 2008, Barcelona, Catalunya, Spain*, pages 217–222. IEEE Computer Society, 2008. doi: 10.1109/RE.2008.31. URL https://doi.org/10.1109/RE.2008.31.

Andy Koronios, Michael Lane, and Glen Van Der Vyver. Facilitators and inhibitors for the adoption of agile methods. In *Systems Analysis and Design: People, Processes, and Projects*, pages 43–62. Routledge, 2015.

Tomaz Kosar, Sudev Bohra, and Marjan Mernik. Domain-specific languages: A systematic mapping study. *Inf. Softw. Technol.*, 71:77–91, 2016. doi: 10.1016/J.INFSOF.2015.11.001. URL https://doi.org/10.1016/j.infsof.2015.11.001.

Valsa Koshy. *Action research for improving practice: A practical guide*. Sage, 2005.

Aapo Koski and Tommi Mikkonen. Rolling out a mission critical system in an agilish way. reflections on building a large-scale dependable information system for public sector. In Matthias Tichy, Jan Bosch, Michael Goedicke, and Brian Fitzgerald, editors, *2nd IEEE/ACM International Workshop on Rapid Continuous Software Engineering, RCoSE 2015, Florence, Italy, May 23, 2015*, pages 41–44. IEEE Computer Society, 2015. doi: 10.1109/RCoSE.2015.15. URL https://doi.org/10.1109/RCoSE.2015.15.

Philippe Kruchten. Contextualizing agile software development. *Journal of software: Evolution and Process*, 25(4):351–361, 2013.

Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a better understanding of software features and their characteristics: A case study of marlin. In Rafael Capilla, Malte Lochau, and Lidia Fuentes, editors, *Proceedings of the 12th International Workshop on Variability Modelling of Software-Intensive Systems, VAMOS 2018, Madrid, Spain, February 7-9, 2018*, pages 105–112. ACM, 2018. doi: 10.1145/3168365.3168371. URL https://doi.org/10.1145/3168365.3168371.

Wolfgang Kuchinke, Christian Krauth, and Töresin Karakoyun. Agile software development requires an agile approach for computer system validation of clinical trials software products. In *eChallenges e-2014 Conference Proceedings*, pages 1–8. IEEE, 2014.

Konstantin Kudryashov. The beginner's guide to bdd. *Dan North Q & A. https://inviqa.com/blog/bdd-guide*, 2015.

Alessandro Landi and Mark Nicholson. Arp4754a/ed-79a-guidelines for development of civil aircraft and systems-enhancements, novelties and key topics. *SAE International Journal of Aerospace*, 4(2011-01-2564):871–879, 2011.

Phillip A Laplante. *Requirements engineering for software and systems*. CRC Press, 2017.

Amanda Lee, Jeffrey C. Carver, and Amiangshu Bosu. Understanding the impressions, motivations, and barriers of one time code contributors to FLOSS projects: a survey. In Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 187–197. IEEE / ACM, 2017. doi: 10.1109/ICSE.2017.25. URL `https://doi.org/10.1109/ICSE.2017.25`.

Shou-Yu Lee, W. Eric Wong, and Ruizhi Gao. Software safety standards: Evolution and lessons learned. In *2014 International Conference on Trustworthy Systems and their Applications, TSA 2014, Taichung, Taiwan, June 9-10, 2014*, pages 44–50. IEEE, 2014. doi: 10.1109/TSA.2014.16. URL `https://doi.org/10.1109/TSA.2014.16`.

Dean Leffingwell. *SAFe 4.0 Reference Guide: Scaled Agile Framework for Lean Software and Systems Engineering: Scaled Agile Framework for Lean Software and Systems Engineering*. Addison-Wesley Professional, first edition, 2016.

Timo O. A. Lehtinen, Mika Mäntylä, Jari Vanhanen, Juha Itkonen, and Casper Lassenius. Perceived causes of software project failures - an analysis of their relationships. *Inf. Softw. Technol.*, 56(6):623–643, 2014. doi: 10.1016/j.infsof.2014.01.015. URL `https://doi.org/10.1016/j.infsof.2014.01.015`.

Howard Lei, Farnaz Ganjeizadeh, Pradeep Kumar Jayachandran, and Pinar Ozcan. A statistical analysis of the effects of scrum and kanban on software development projects. *Robotics and Computer-Integrated Manufacturing*, 43:59–67, 2017.

Per Lenberg, Robert Feldt, Lars Göran Wallgren Tengberg, and Lucas Gren. Behavioral aspects of safety-critical software development. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 173–176. ACM, 2020. doi: 10.1145/3387940.3392227. URL `https://doi.org/10.1145/3387940.3392227`.

Maurizio Leotta, Maura Cerioli, Dario Olianas, and Filippo Ricca. Hamcrest vs assertj: An empirical assessment of tester productivity. In *International Conference on the Quality of Information and Communications Technology*, pages 161–176. Springer, 2019.

Marion Lepmets, Fergal McCaffery, and Paul M. Clarke. Piloting mdevspice: the medical device software process assessment framework. In Dietmar Pfahl, Reda Bendraou, Richard

Turner, Marco Kuhrmann, Regina Hebig, and Fabrizio Maria Maggi, editors, *Proceedings of the 2015 International Conference on Software and System Process, ICSSP 2015, Tallinn, Estonia, August 24 - 26, 2015*, pages 9–16. ACM, 2015. doi: 10.1145/2785592.2785598. URL `https://doi.org/10.1145/2785592.2785598`.

Nancy G Leveson. *Engineering a safer world: Systems thinking applied to safety*. The MIT Press, 2016.

Mikael Lindvall, Victor R. Basili, Barry W. Boehm, Patricia Costa, Kathleen Coleman Dangle, Forrest Shull, Roseanne Tesoriero Tvedt, Laurie A. Williams, and Marvin V. Zelkowitz. Empirical findings in agile methods. In Don Wells and Laurie A. Williams, editors, *Extreme Programming and Agile Methods - XP/Agile Universe 2002, Second XP Universe and First Agile Universe Conference Chicago, IL, USA, August 4-7, 2002, Proceedings*, volume 2418 of *Lecture Notes in Computer Science*, pages 197–207. Springer, 2002.

Google LLC. Google Scholar. `https://scholar.google.com/`, 2004.

Mike Lowery and Marcus Evans. Scaling product ownership. In Jutta Eckstein, Frank Maurer, Rachel Davies, Grigori Melnik, and Gary Pollice, editors, *AGILE 2007 Conference (AGILE 2007), 13-17 August 2007, Washington, DC, USA*, pages 328–333. IEEE Computer Society, 2007.

Garm Lucassen, Fabiano Dalpiaz, Jan Martijn E. M. van der Werf, and Sjaak Brinkkemper. The use and effectiveness of user stories in practice. In Maya Daneva and Oscar Pastor, editors, *Requirements Engineering: Foundation for Software Quality - 22nd International Working Conference, REFSQ 2016, Gothenburg, Sweden, March 14-17, 2016, Proceedings*, volume 9619 of *Lecture Notes in Computer Science*, pages 205–222. Springer, 2016. doi: 10.1007/978-3-319-30282-9_14. URL `https://doi.org/10.1007/978-3-319-30282-9_14`.

Katarzyna Lukasiewicz and Janusz Górski. Agilesafe - a method of introducing agile practices into safety-critical software development processes. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS 2016, Gdańsk, Poland, September 11-14, 2016*, volume 8 of *Annals of Computer Science and Information Systems*, pages 1549–1552. IEEE, 2016. doi: 10.15439/2016F360. URL `https://doi.org/10.15439/2016F360`.

Katarzyna Lukasiewicz and Janusz Górski. Introducing agile practices into development processes of safety critical software. In Ademar Aguiar, editor, *Proceedings of the 19th International Conference on Agile Software Development, XP 2019, Companion, Porto, Portugal, May 21-25, 2018*, pages 13:1–13:8. ACM, 2018. doi: 10.1145/3234152.3234174. URL `https://doi.org/10.1145/3234152.3234174`.

D. Ma'ayan. The quality of junit tests: An empirical study report. In *2018 IEEE/ACM 1st International Workshop on Software Qualities and their Dependencies (SQUADE)*, pages 33–36, May 2018.

Linda A. Macaulay. *Requirements engineering*. Applied computing. Springer Science & Business Media, 2012.

Cathy MacDonald. Understanding participatory action research: A qualitative research methodology option. *The Canadian Journal of Action Research*, 13(2):34–50, 2012.

Daniel Maciel, Ana C. R. Paiva, and Alberto Rodrigues da Silva. From requirements to automated acceptance tests of interactive apps: An integrated model-based testing approach. In Ernesto Damiani, George Spanoudakis, and Leszek A. Maciaszek, editors, *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering, ENASE 2019, Heraklion, Crete, Greece, May 4-5, 2019*, pages 265–272. SciTePress, 2019. doi: 10.5220/0007679202650272. URL `https://doi.org/10.5220/0007679202650272`.

Edward J Mango. Safety characteristics in system application software for human rated exploration missions. *Journal of Space Safety Engineering*, 3(3):104–110, 2016.

Rafaela Mantovani Fontana and Sabrina Marczak. Characteristics and challenges of agile software development adoption in brazilian government. *Journal of technology management & innovation*, 15(2):3–10, 2020.

Mika Mantyla. *Bad smells in software-a taxonomy and an empirical study*. PhD thesis, PhD thesis, Helsinki University of Technology, 2003.

Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. A taxonomy and an initial empirical study of bad smells in code. In *19th International Conference on Software Maintenance (ICSM 2003), The Architecture of Existing Systems, 22-26 September 2003, Amsterdam, The Netherlands*, pages 381–384. IEEE Computer Society, 2003. doi: 10.1109/ICSM.2003.1235447. URL `https://doi.org/10.1109/ICSM.2003.1235447`.

Hafiza Maria Maqsood, Eduardo Martins Guerra, Xiaofeng Wang, and Andrea Bondavalli. Patterns for development of safety-critical systems with agile: Trace safety requirements and perform automated testing. In *EuroPLoP '20: European Conference on Pattern Languages of Programs 2020, Virtual Event, Germany, 1-4 July, 2020*, pages 3:1–3:6. ACM, 2020. doi: 10.1145/3424771.3424800. URL `https://doi.org/10.1145/3424771.3424800`.

Johnny Marques and Adilson Cunha. A reference method for airborne software requirements. In *Digital Avionics Systems Conference (DASC), 2013 IEEE/AIAA 32nd*, pages 7A2–1. IEEE, 2013.

Luiz Eduardo Galvão Martins and Tony Gorschek. Requirements engineering for safety-critical systems: A systematic literature review. *Inf. Softw. Technol.*, 75:71–89, 2016. doi: 10.1016/j.infsof.2016.04.002. URL `https://doi.org/10.1016/j.infsof.2016.04.002`.

Luiz Eduardo Galvão Martins and Tony Gorschek. Requirements engineering for safety-critical systems: Overview and challenges. *IEEE Softw.*, 34(4):49–57, 2017. doi: 10.1109/MS.2017.94. URL `https://doi.org/10.1109/MS.2017.94`.

Tom McBride and Marion Lepmets. Quality assurance in agile safety-critical systems development. In Mark C. Paulk, Ricardo J. Machado, Miguel A. Brito, Miguel Goulão, and Vasco Amaral, editors, *10th International Conference on the Quality of Information and Communications Technology, QUATIC 2016, Lisbon, Portugal, September 6-9, 2016*, pages 44–51. IEEE Computer Society, 2016. doi: 10.1109/QUATIC.2016.016. URL `http://doi.ieeecomputersociety.org/10.1109/QUATIC.2016.016`.

Martin McHugh, Oisín Cawley, Fergal McCaffery, Ita Richardson, and Xiaofeng Wang. An agile v-model for medical device software development to overcome the challenges with plan-driven software development lifecycles. In John Knight and Craig E. Kuziemsky, editors, *Proceedings of the 5th International Workshop on Software Engineering in Health Care, SEHC 2013, San Francisco, California, USA, May 20-21, 2013*, pages 12–19. IEEE Computer Society, 2013. doi: 10.1109/SEHC.2013.6602471. URL `https://doi.org/10.1109/SEHC.2013.6602471`.

Martin McHugh, Fergal McCaffery, and Garret Coady. An agile implementation within a medical device software organisation. In Antanas Mitasiunas, Terry Rout, Rory V. O'Connor, and Alec Dorling, editors, *Software Process Improvement and Capability Determination - 14th International Conference, SPICE 2014, Vilnius, Lithuania, November 4-6, 2014, Proceedings*, volume 477 of *Communications in Computer and Information Science*, pages 190–201. Springer, 2014. doi: 10.1007/978-3-319-13036-1_17. URL `https://doi.org/10.1007/978-3-319-13036-1_17`.

Juliana Medeiros, Alexandre M. L. de Vasconcelos, Carla Silva, and Miguel Goulão. Requirements specification for developers in agile projects: Evaluation by two industrial case studies. *Inf. Softw. Technol.*, 117, 2020. doi: 10.1016/j.infsof.2019.106194. URL `https://doi.org/10.1016/j.infsof.2019.106194`.

Ben Swarup Medikonda and P Seetha Ramaiah. Software safety analysis to identify critical software faults in software-controlled safety-critical systems. In *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India-Vol II*, pages 455–465. Springer, 2014.

Hossein Mehrfard and Abdelwahab Hamou-Lhadj. The impact of regulatory compliance on agile software processes with a focus on the FDA guidelines for medical device software. *Int. J. Inf. Syst. Model. Des.*, 2(2):67–81, 2011. doi: 10.4018/jismd.2011040104. URL `https://doi.org/10.4018/jismd.2011040104`.

Hossein Mehrfard, Heidar Pirzadeh, and Abdelwahab Hamou-Lhadj. Investigating the capability of agile processes to support life-science regulations: The case of XP and FDA regulations with a focus on human factor requirements. In Roger Y. Lee, Olga Ormandjieva, Alain Abran, and Constantinos Constantinides, editors, *Software Engineering Research, Management and Applications 2010 [selected papers from the 8th ACIS International Conference on Software Engineering Research, Management and Applications, SERA 2010, Montreal, Canada, May 24-26, 2010]*, volume 296 of *Studies in Computational Intelligence*, pages 241–255. Springer, 2010. doi: 10.1007/978-3-642-13273-5_16. URL `https://doi.org/10.1007/978-3-642-13273-5_16`.

Ines Mergel, Yiwei Gong, and John Bertot. Agile government: Systematic literature review and future research. *Gov. Inf. Q.*, 35(2):291–298, 2018. doi: 10.1016/j.giq.2018.04.003. URL `https://doi.org/10.1016/j.giq.2018.04.003`.

Ines Mergel, Sukumar Ganapati, and Andrew B Whitford. Agile: A new way of governing. *Public Administration Review*, 2020.

Craig A Mertler. *Action research: Teachers as researchers in the classroom*. Sage, 2009.

Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

Faïda Mhenni, Nga Nguyen, and Jean-Yves Choley. Safesyse: A safety analysis integration in systems engineering approach. *IEEE Syst. J.*, 12(1):161–172, 2018. doi: 10.1109/JSYST.2016.2547460. URL `https://doi.org/10.1109/JSYST.2016.2547460`.

Jakub Miler and Paulina Gaida. On the agile mindset of an effective team - an industrial opinion survey. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *Proceedings of the 2019 Federated Conference on Computer Science and Information Systems, FedCSIS 2019, Leipzig, Germany, September 1-4, 2019*, volume 18 of *Annals of Computer Science and Information Systems*, pages 841–849, 2019. doi: 10.15439/2019F198. URL `https://doi.org/10.15439/2019F198`.

Roy Miller and Christopher T Collins. Acceptance testing. *Proc. XPUniverse*, 238, 2001.

Myint Myint Moe. Comparative study of test-driven development (tdd), behavior-driven development (bdd) and acceptance test–driven development (atdd). *International Journal of Trend in Scientific Research and Development*, pages 231–234, 2019.

Adel Hamdan Mohammad, Tariq Alwada'n, et al. Agile software methodologies: strength and weakness. *International Journal of Engineering Science and Technology*, 5(3):455, 2013.

Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating github for engineered software projects. *Empir. Softw. Eng.*, 22(6):3219–3253, 2017. doi: 10.1007/s10664-017-9512-6. URL `https://doi.org/10.1007/s10664-017-9512-6`.

Arbesë Musliu and Xhelal Jashari. Software automated testing using bdd approach with cucumber framework. In Hajrizi Edmond, editor, *2021 UBT INTERNATIONAL CONFERENCE*, 2021.

Colin Myers, Tracy Hall, and Dave Pitt. *The responsible software engineer: Selected readings in IT professionalism*. Springer Science & Business Media, 2012.

Thor Myklebust. Safety standards, software and improved development of safety equipment, 01 2008.

Thor Myklebust, T Stålhane, GK Hanssen, and B Haugset. Change impact analysis as required by safety standards, what to do. In *Probabilistic Safety Assessment & Management conference (PSAM12), Honolulu, USA*, 2014a.

Thor Myklebust, Tor Stålhane, Geir Kjetil Hanssen, Tormod Wien, and Børge Haugset. Scrum, documentation and the iec 61508-3: 2010 software standard. In *International Conference on Probabilistic Safety Assesment and Management (PSAM). PSAM, Hawaii*, 2014b.

Thor Myklebust, T Stålhane, and GK Hanssen. Important considerations when applying other models than the waterfall/v-model when developing software according to iec 61508 or en 50128. In *33rd International System Safety Conference (Aug. 2015)*, 2015.

Thor Myklebust, Tor Stålhane, and Narve Lyngby. An agile development process for petrochemical safety conformant software. In *2016 Annual Reliability and Maintainability Symposium (RAMS)*, pages 1–6. IEEE, 2016.

NASA. *NASA Software Safety Guidebook*. NASA HQ, Office of Safety and Mission Assurance, Washington, D.C, USA, 2004.

Nicolas Nascimento, Alan R. Santos, Afonso Sales, and Rafael Chanin. Behavior-driven development: A case study on its impacts on agile development teams. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 109–116. ACM, 2020a. doi: 10.1145/3387940.3391480. URL `https://doi.org/10.1145/3387940.3391480`.

Nicolas Nascimento, Alan R. Santos, Afonso Sales, and Rafael Chanin. Behavior-driven development: An expert panel to evaluate benefits and challenges. In Everton Cavalcante,

Francisco Dantas, and Thaís Batista, editors, *SBES '20: 34th Brazilian Symposium on Software Engineering, Natal, Brazil, October 19-23, 2020*, pages 41–46. ACM, 2020b. doi: 10.1145/3422392.3422460. URL https://doi.org/10.1145/3422392.3422460.

Alexander Newman, Yuen Lam Bavik, Matthew Mount, and Bo Shao. Data collection via online platforms: Challenges and recommendations for future research. *Applied Psychology*, 2021.

Dan North. What's in a story? https://dannorth.net/whats-in-a-story, 2019.

Dan North. We need to talk about testing. https://dannorth.net/2021/07/26/we-need-to-talk-about-testing/, 2021.

Dan North et al. Introducing bdd. *Better Software*, 12, 2006.

Chris Northwood. Testing. In *The Full Stack Developer*, pages 141–157. Springer, 2018.

Jesper Pedersen Notander, Martin Höst, and Per Runeson. Challenges in flexible safety-critical software development - an industrial qualitative survey. In Jens Heidrich, Markku Oivo, Andreas Jedlitschka, and Maria Teresa Baldassarre, editors, *Product-Focused Software Process Improvement - 14th International Conference, PROFES 2013, Paphos, Cyprus, June 12-14, 2013. Proceedings*, volume 7983 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2013. doi: 10.1007/978-3-642-39259-7_23. URL https://doi.org/10.1007/978-3-642-39259-7_23.

Bashar Nuseibeh and Steve M. Easterbrook. Requirements engineering: a roadmap. In Anthony Finkelstein, editor, *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*, pages 35–46. ACM, 2000. doi: 10.1145/336512.336523. URL https://doi.org/10.1145/336512.336523.

Gabriel Oliveira and Sabrina Marczak. On the understanding of BDD scenarios' quality: Preliminary practitioners' opinions. In Erik Kamsties, Jennifer Horkoff, and Fabiano Dalpiaz, editors, *Requirements Engineering: Foundation for Software Quality - 24th International Working Conference, REFSQ 2018, Utrecht, The Netherlands, March 19-22, 2018, Proceedings*, volume 10753 of *Lecture Notes in Computer Science*, pages 290–296. Springer, 2018. doi: 10.1007/978-3-319-77243-1_18. URL https://doi.org/10.1007/978-3-319-77243-1_18.

Gabriel Oliveira, Sabrina Marczak, and Cassiano Moralles. How to evaluate BDD scenarios' quality? In Ivan do Carmo Machado, Rodrigo Souza, Rita Suzana Pitangueira Maciel, and Cláudio Sant'Anna, editors, *Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES 2019, Salvador, Brazil, September 23-27, 2019*, pages 481–490. ACM,

2019. doi: 10.1145/3350768.3351301. URL https://doi.org/10.1145/3350768.3351301.

Marco Ortu, Tracy Hall, Michele Marchesi, Roberto Tonelli, David Bowes, and Giuseppe Destefanis. Mining communication patterns in software development: A github analysis. In *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering*, PROMISE'18, pages 70–79, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6593-2. doi: 10.1145/3273934.3273943. URL http://doi.acm.org/10.1145/3273934.3273943.

Viktor Österholm. Overview of behaviour-driven development tools for web applications. Master's thesis, Åbo Akademi University (Finland), 2021.

Özden Özcan-Top and Fergal McCaffery. Conformance to medical device software development requirements with xp and scrum implementation. In *Proceedings of the International Conference on Software Engineering Research and Practice (SERP)*, 2018.

Necmettin Ozkan. Imperfections underlying the manifesto for agile software development. In *2019 1st International Informatics and Software Engineering Conference (UBMYK)*, pages 1–6. IEEE, 2019.

Maria Paasivaara, Benjamin Behm, Casper Lassenius, and Minna Hallikainen. Large-scale agile transformation at ericsson: a case study. *Empir. Softw. Eng.*, 23(5):2550–2596, 2018. doi: 10.1007/s10664-017-9555-8. URL https://doi.org/10.1007/s10664-017-9555-8.

Frauke Paetsch, Armin Eberlein, and Frank Maurer. Requirements engineering and agile software development. In *12th IEEE International Workshops on Enabling Technologies (WETICE 2003), Infrastructure for Collaborative Enterprises, 9-11 June 2003, Linz, Austria*, pages 308–313. IEEE Computer Society, 2003.

Richard F. Paige, Ramon Charalambous, Xiaocheng Ge, and Phillip J. Brooke. Towards agile engineering of high-integrity systems. In Michael D. Harrison and Mark-Alexander Sujan, editors, *Computer Safety, Reliability, and Security, 27th International Conference, SAFECOMP 2008, Newcastle upon Tyne, UK, September 22-25, 2008, Proceedings*, volume 5219 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 2008. doi: 10.1007/978-3-540-87698-4_6. URL https://doi.org/10.1007/978-3-540-87698-4_6.

Richard F. Paige, Andy Galloway, Ramon Charalambous, Xiaocheng Ge, and Phillip J. Brooke. High-integrity agile processes for the development of safety critical software. *IJCCBS*, 2(2): 181–216, 2011.

Kamalendu Pal and Bill Karakostas. Software testing under agile, scrum, and devops. In *Agile Scrum Implementation and Its Long-Term Impact on Organizations*, pages 114–131. IGI Global, 2021.

Stephen R. Palmer. *A Practical Guide to Feature-Driven Development*. Prentice Hall, February 2002.

Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *Proceedings of the 9th international workshop on search-based software testing*, pages 5–14. ACM, 2016.

Robert E Park, Wolfhart B Goethert, and William A Florac. Goal-driven software measurement. a guidebook. Technical report, Carnegie-Mellon Univ Pittsburgh Pa Software Engineering Inst, 1996.

Richard R Parry. Programmable electronic safety systems. In *Proceedings of International Conference on Particle Accelerators*, pages 2225–2227. IEEE, 1993.

Lauriane Pereira, Helen Sharp, Cleidson de Souza, Gabriel Oliveira, Sabrina Marczak, and Ricardo Bastos. Behavior-driven development benefits and challenges: Reports from an industrial study. In *Proceedings of the 19th International Conference on Agile Software Development: Companion*, XP '18, pages 42:1–42:4, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6422-5. doi: 10.1145/3234152.3234167. URL `http://doi.acm.org/10.1145/3234152.3234167`.

Anthony Peruma, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. An exploratory study on the refactoring of unit test files in android applications. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 350–357. ACM, 2020. doi: 10.1145/3387940.3392189. URL `https://doi.org/10.1145/3387940.3392189`.

Mark Petticrew and Helen Roberts. *Systematic reviews in the social sciences: A practical guide*. John Wiley & Sons, 2008.

Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit*. Addison-Wesley Professional, May 2003.

Naveen Prakash and Deepika Prakash. Model-driven user stories for agile data warehouse development. In Peri Loucopoulos, Yannis Manolopoulos, Oscar Pastor, Babis Theodoulidis, and Jelena Zdravkovic, editors, *19th IEEE Conference on Business Informatics, CBI 2017, Thessaloniki, Greece, July 24-27, 2017, Volume 1: Conference Papers*, pages 424–433. IEEE

Computer Society, 2017. doi: 10.1109/CBI.2017.67. URL https://doi.org/10.1109/CBI.2017.67.

Evgeny Pyshkin, Maxim Mozgovoy, and Mikhail Glukhikh. On requirements for acceptance testing automation tools in behavior driven software development. In *Proceedings of the 8th Software Engineering Conference in Russia (CEE-SECR)*, 2012.

Fumin Qi, Xiao-Yuan Jing, Xiaoke Zhu, Xiaoyuan Xie, Baowen Xu, and Shi Ying. Software effort estimation based on open source projects: Case study of github. *Information and Software Technology*, 92:145–157, 2017.

Mikko Raatikainen, Tomi Männistö, Teemu Tommila, and Janne Valkonen. Challenges of requirements engineering - A case study in nuclear energy domain. In *RE 2011, 19th IEEE International Requirements Engineering Conference, Trento, Italy, August 29 2011 - September 2, 2011*, pages 253–258. IEEE Computer Society, 2011. doi: 10.1109/RE.2011.6051629. URL https://doi.org/10.1109/RE.2011.6051629.

Ramon Radnoci. Methods for testing concurrent software, 2009.

Mazedur Rahman and Jerry Gao. A reusable automated acceptance testing architecture for microservices in behavior-driven development. In *2015 IEEE Symposium on Service-Oriented System Engineering, SOSE 2015, San Francisco Bay, CA, USA, March 30 - April 3, 2015*, pages 321–325. IEEE Computer Society, 2015. doi: 10.1109/SOSE.2015.55. URL https://doi.org/10.1109/SOSE.2015.55.

Balasubramaniam Ramesh, Lan Cao, and Richard Baskerville. Agile requirements engineering practices and challenges: an empirical study. *Information Systems Journal*, 20(5):449–480, 2010.

Rod Rasmussen, Tim Hughes, J. R. Jenks, and John Skach. Adopting agile in an FDA regulated environment. In Yael Dubinsky, Tore Dybå, Steve Adolph, and Ahmed Samy Sidky, editors, *2009 Agile Conference, Chicago, IL, USA, 24-28 August 2009*, pages 151–155. IEEE Computer Society, 2009. doi: 10.1109/AGILE.2009.50. URL https://doi.org/10.1109/AGILE.2009.50.

Derek Rayside, Aleksandar Milicevic, Kuat Yessenov, Greg Dennis, and Daniel Jackson. Agile specifications. In Shail Arora and Gary T. Leavens, editors, *Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA*, pages 999–1006. ACM, 2009.

Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *J. Object Technol.*, 6(9):231–251, 2007. doi: 10.5381/jot.2007.6.9.a12. URL `https://doi.org/10.5381/jot.2007.6.9.a12`.

William Richardson, Kalen Bennett, Douglas Dempster, Philippe Dumas, Caroline Leprince, Kim Richard Nossal, David Perry, Elinor Sloan, and Craig J Stone. *Toward agile procurement for national defence: Matching the pace of technological change*. Canadian Global Affairs Institute Calgary, 2020.

Hans-Gerd Ridder. The theory contribution of case study research designs. *Business Research*, 10(2):281–305, 2017.

Tracy Riley and Roger Moltzen. Learning by doing: Action research to evaluate provisions for gifted and talented students. *Kairaranga*, 12(1):23–31, 2011.

Peter Ritchie. Testing. In *Practical Microsoft Visual Studio 2015*, pages 169–193. Springer, 2016.

Thiago Rocha Silva. Towards a domain-specific language to specify interaction scenarios for web-based graphical user interfaces. In *Companion of the 2022 ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 48–53, 2022.

Tobias Roehm, Daniel Veihelmann, Stefan Wagner, and Elmar Juergens. Evaluating maintainability prejudices with a large-scale study of open-source projects. In *International Conference on Software Quality*, pages 151–171. Springer, 2019.

Mary Beth Rosson and John M. Carroll. *Usability engineering - scenario-based development of human-computer interaction*. The Morgan Kaufmann series in interactive technologies. Elsevier Morgan Kaufmann, 2002. ISBN 978-1-55860-712-5.

Johann Rost and Robert L. Glass. *The dark side of software engineering: evil on computing projects*. John Wiley & Sons, 2011.

Pieter Adriaan Rottier and Victor Rodrigues. Agile development in a medical device company. In Grigori Melnik, Philippe Kruchten, and Mary Poppendieck, editors, *Agile Development Conference, AGILE 2008, Toronto, Canada, 4-8 August 2008*, pages 218–223. IEEE Computer Society, 2008. doi: 10.1109/Agile.2008.52. URL `https://doi.org/10.1109/Agile.2008.52`.

RTCA. *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 1150 18th St, NW, Suite 910. Washington DC 20036-3816 USA., December 2011.

Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

Daniel Russo, Gerolamo Taccogna, Paolo Ciancarini, Angelo Messina, and Giancarlo Succi. Contracting agile developments for mission critical systems in the public sector. In Valérie Issarny and Schahram Dustdar, editors, *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Society, ICSE (SEIS) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 47–56. ACM, 2018. doi: 10.1145/3183428.3183435. URL `https://doi.org/10.1145/3183428.3183435`.

J. Ghoshal S. Bose, M. Kurhekar. Agile methodology in requirements engineering. *Computer*, 42(9):37–45, February 2010.

SAE. *International. Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment: ARP4761 [S/OL]*, 2017.

Victor Travassos Sarinho. "bdd assemble!": A paper-based game proposal for behavior driven development design learning. In Erik D. Van der Spek, Stefan Göbel, Ellen Yi-Luen Do, Esteban Clua, and Jannicke Baalsrud Hauge, editors, *Entertainment Computing and Serious Games - First IFIP TC 14 Joint International Conference, ICEC-JCSG 2019, Arequipa, Peru, November 11-15, 2019, Proceedings*, volume 11863 of *Lecture Notes in Computer Science*, pages 431–435. Springer, 2019. doi: 10.1007/978-3-030-34644-7_41. URL `https://doi.org/10.1007/978-3-030-34644-7_41`.

Maggi Savin-Baden and Katherine Wimpenny. Exploring and implementing participatory action research. *Journal of Geography in Higher Education*, 31(2):331–343, 2007.

André Scandaroli, Rodrigo Leite, Aléxis H Kiosia, and Sandro 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 *2019 ACM/IEEE 14th International Conference on Global Software Engineering (ICGSE)*, pages 105–110. IEEE, 2019.

Ingo Schnabel and Markus Pizka. Goal-driven software development. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*, pages 59–65. IEEE Computer Society, 2006. doi: 10.1109/SEW.2006.21. URL `https://doi.org/10.1109/SEW.2006.21`.

Eva-Maria Schön, Jörg Thomaschewski, and María José Escalona. Agile requirements engineering: A systematic literature review. *Comput. Stand. Interfaces*, 49:79–91, 2017a. doi: 10.1016/j.csi.2016.08.011. URL `https://doi.org/10.1016/j.csi.2016.08.011`.

Eva-Maria Schön, Dominique Winter, María José Escalona, and Jörg Thomaschewski. Key challenges in agile requirements engineering. In Hubert Baumeister, Horst Lichter, and Matthias Riebisch, editors, *Agile Processes in Software Engineering and Extreme Programming - 18th*

International Conference, XP 2017, Cologne, Germany, May 22-26, 2017, Proceedings, volume 283 of *Lecture Notes in Business Information Processing*, pages 37–51, 2017b. doi: 10.1007/978-3-319-57633-6_3. URL https://doi.org/10.1007/978-3-319-57633-6_3.

Nancy Van Schooenderwoert and Ron Morsicato. Taming the embedded tiger - agile test techniques for embedded software. In *2004 Agile Development Conference (ADC 2004), 22-26 June 2004, Salt Lake City, UT, USA*, pages 120–126. IEEE Computer Society, 2004.

Ken Schwaber and Mike Beedle. *Agile Software Development with SCRUM*. Prentice Hall, 2001.

Philip Sedgwick and Nan Greenwood. Understanding the hawthorne effect. *Bmj*, 351, 2015.

Abhishek Sharma, Ferdian Thung, Pavneet Singh Kochhar, Agus Sulistya, and David Lo. Cataloging github repositories. In *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, pages 314–319, 2017.

Sheetal Sharma, Darothi Sarkar, and Divya Gupta. Agile processes and methodologies: A conceptual study. *International journal on computer science and Engineering*, 4(5):892, 2012.

Tushar Sharma, Marios Fragkoulis, Stamatia Rizou, Magiel Bruntink, and Diomidis Spinellis. Smelly relations: measuring and understanding database schema quality. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, pages 55–64, 2018.

Helen Sharp and Hugh Robinson. Three 'c's of agile practice: collaboration, co-ordination and communication. In *Agile software development: current research and future directions*, pages 61–85. Springer, 2010.

Ajit Ashok Shenvi. Navigating the maze: journey towards an optimal process framework for regulated medical software. In Dharanipragada Janakiram, Koushik Sen, and Vinay Kulkarni, editors, *7th India Software Engineering Conference, Chennai, ISEC '14, Chennai, India - February 19 - 21, 2014*, pages 21:1–21:6. ACM, 2014. doi: 10.1145/2590748.2590769. URL https://doi.org/10.1145/2590748.2590769.

Baruch Shimoni. What is resistance to change? a habitus-oriented approach. *Academy of Management Perspectives*, 31(4):257–270, 2017.

Farhad Shokraneh. Reproducibility and replicability of systematic reviews. *World Journal of Meta-Analysis*, 7(3), 2019.

Lubna Siddique and Bassam A Hussein. Practical insight about choice of methodology in large complex software projects in norway. In *2014 IEEE International Technology Management Conference*, pages 1–4. IEEE, 2014.

Thiago Rocha Silva. Definition of a behavior-driven model for requirements specification and testing of interactive systems. In *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China, September 12-16, 2016*, pages 444–449. IEEE Computer Society, 2016. doi: 10.1109/RE.2016.12. URL https://doi.org/10.1109/RE.2016.12.

Thiago Rocha Silva and Brian Fitzgerald. Empirical findings on BDD story parsing to support consistency assurance between requirements and artifacts. In Ruzanna Chitchyan, Jingyue Li, Barbara Weber, and Tao Yue, editors, *EASE 2021: Evaluation and Assessment in Software Engineering, Trondheim, Norway, June 21-24, 2021*, pages 266–271. ACM, 2021. doi: 10.1145/3463274.3463807. URL https://doi.org/10.1145/3463274.3463807.

Thiago Rocha Silva and Marco Winckler. A scenario-based approach for checking consistency in user interface design artifacts. In Isabela Gasparini, Lara S. G. Piccolo, Luciana A. M. Zaina, and Roberto Pereira, editors, *Proceedings of the XVI Brazilian Symposium on Human Factors in Computing Systems, IHC 2017, Joinville, Brazil, October 23-27, 2017*, pages 3:1–3:10. ACM, 2017. doi: 10.1145/3160504.3160506. URL https://doi.org/10.1145/3160504.3160506.

Thiago Rocha Silva, Marco Winckler, and Hallvard Trætteberg. Ensuring the consistency between user requirements and graphical user interfaces: A behavior-based automated approach. In Sanjay Misra, Osvaldo Gervasi, Beniamino Murgante, Elena N. Stankova, Vladimir Korkhov, Carmelo Maria Torre, Ana Maria A. C. Rocha, David Taniar, Bernady O. Apduhan, and Eufemia Tarantino, editors, *Computational Science and Its Applications - ICCSA 2019 - 19th International Conference, Saint Petersburg, Russia, July 1-4, 2019, Proceedings, Part I*, volume 11619 of *Lecture Notes in Computer Science*, pages 616–632. Springer, 2019a. doi: 10.1007/978-3-030-24289-3_46. URL https://doi.org/10.1007/978-3-030-24289-3_46.

Thiago Rocha Silva, Marco Winckler, and Hallvard Trætteberg. Ensuring the consistency between user requirements and GUI prototypes: A behavior-based automated approach. In David Lamas, Fernando Loizides, Lennart E. Nacke, Helen Petrie, Marco Winckler, and Panayiotis Zaphiris, editors, *Human-Computer Interaction - INTERACT 2019 - 17th IFIP TC 13 International Conference, Paphos, Cyprus, September 2-6, 2019, Proceedings, Part I*, volume 11746 of *Lecture Notes in Computer Science*, pages 644–665. Springer, 2019b. doi: 10.1007/978-3-030-29381-9_39. URL https://doi.org/10.1007/978-3-030-29381-9_39.

Thiago Rocha Silva, Marco Winckler, and Hallvard Trætteberg. Extending behavior-driven development for assessing user interface design artifacts (S). In Angelo Perkusich, editor, *The 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019, Hotel Tivoli, Lisbon, Portugal, July 10-12, 2019*, pages 485–623. KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019c. doi: 10.18293/SEKE2019-054. URL `https://doi.org/10.18293/SEKE2019-054`.

Thiago Rocha Silva, Marco Winckler, and Cédric Bach. Evaluating the usage of predefined interactive behaviors for writing user stories: an empirical study with potential product owners. *Cogn. Technol. Work.*, 22(3):437–457, 2020a. doi: 10.1007/s10111-019-00566-3. URL `https://doi.org/10.1007/s10111-019-00566-3`.

Thiago Rocha Silva, Marco Winckler, and Hallvard Trætteberg. Ensuring the consistency between user requirements and task models: A behavior-based automated approach. *Proc. ACM Hum. Comput. Interact.*, 4(EICS):77:1–77:32, 2020b. doi: 10.1145/3394979. URL `https://doi.org/10.1145/3394979`.

John Ferguson Smart. *BDD in Action*. Manning Publications, 2014.

John Ferguson Smart and Jan Molak. *BDD in Action: Behavior-driven development for the whole software lifecycle*. Simon and Schuster, 2023.

Jonathan Smart. To transform to have agility, dont do a capital a, capital T agile transformation. *IEEE Softw.*, 35(6):56–60, 2018. doi: 10.1109/MS.2018.4321245. URL `https://doi.org/10.1109/MS.2018.4321245`.

Hannah Snyder. Literature review as a research methodology: An overview and guidelines. *Journal of Business Research*, 104:333–339, 2019.

Rational Software. The rational unified process. best practices for software development teams. white paper TP026B, The Rational Corporation, 2003.

Carlos Solis and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387. IEEE, 2011.

Bridget Somekh. *Action research: a methodology for change and development: a methodology for change and development*. McGraw-Hill Education (UK), 2005.

Tjerk Spijkman, Fabiano Dalpiaz, and Sjaak Brinkkemper. Back to the roots: Linking user stories to requirements elicitation conversations. In *30th IEEE International Requirements Engineering Conference, RE 2022, Melbourne, Australia, August 15-19, 2022*, pages 281–287. IEEE, 2022. doi: 10.1109/RE54965.2022.00042. URL `https://doi.org/10.1109/RE54965.2022.00042`.

Cary Spitzer, Uma Ferrell, and Thomas Ferrell. *Digital avionics handbook*. CRC press, 2017.

Manuel Stadler, Raoul Vallon, Martin Pazderka, and Thomas Grechenig. Agile distributed software development in nine central european teams: Challenges, benefits, and recommendations. *International Journal of Computer Science & Information Technology (IJCSIT) Vol*, 11, 2019.

T Stålhane, T Myklebust, and GK Hanssen. Safety standards and scrum–a synopsis of three standards. *SafeScrum. no, GK Hanssen, Editor*, 2013.

Tor Stålhane and Thor Myklebust. The role of CM in agile development of safety-critical software. In Floor Koornneef and Coen van Gulijk, editors, *Computer Safety, Reliability, and Security - SAFECOMP 2015 Workshops, ASSURE, DECSoS, ISSE, ReSA4CI, and SASSUR, Delft, The Netherlands, September 22, 2015, Proceedings*, volume 9338 of *Lecture Notes in Computer Science*, pages 386–396. Springer, 2015. doi: 10.1007/978-3-319-24249-1_33. URL `https://doi.org/10.1007/978-3-319-24249-1_33`.

Tor Stålhane, Thor Myklebust, and Geir Hanssen. The application of safe scrum to iec 61508 certifiable software. In *11th International Probabilistic Safety Assessment and Management Conference and the Annual European Safety and Reliability Conference 2012, 25-29 June 2012, Helsinki, Finland*. Curran Associates Inc., 2012.

Tor Stålhane, Geir Kjetil Hanssen, Thor Myklebust, and Børge Haugset. Agile change impact analysis of safety critical software. In Andrea Bondavalli, Andrea Ceccarelli, and Frank Ortmeier, editors, *Computer Safety, Reliability, and Security - SAFECOMP 2014 Workshops: ASCoMS, DECSoS, DEVVARTS, ISSE, ReSA4CI, SASSUR. Florence, Italy, September 8-9, 2014. Proceedings*, volume 8696 of *Lecture Notes in Computer Science*, pages 444–454. Springer, 2014.

George E. Stark, Paul W. Oman, Alan Skillicorn, and Alan Ameele. An examination of the effects of requirements changes on software maintenance releases. *J. Softw. Maintenance Res. Pract.*, 11(5):293–309, 1999.

Jan-Philipp Steghöfer, Eric Knauss, Jennifer Horkoff, and Rebekka Wohlrab. Challenges of scaled agile for safety-critical systems. In Xavier Franch, Tomi Männistö, and Silverio Martínez-Fernández, editors, *Product-Focused Software Process Improvement - 20th International Conference, PROFES 2019, Barcelona, Spain, November 27-29, 2019, Proceedings*, volume 11915 of *Lecture Notes in Computer Science*, pages 350–366. Springer, 2019. doi: 10.1007/978-3-030-35333-9_26. URL `https://doi.org/10.1007/978-3-030-35333-9_26`.

Ernst Stelzmann. Contextualizing agile systems engineering. *IEEE Aerospace and Electronic Systems Magazine*, 27(5):17–22, 2012.

Jan Stenberg. Behaviour-Driven Development Anti-Patterns. https://www.infoq.com/news/2016/09/bdd-anti-patterns/, 2016.

ZR Stephenson, JA McDermid, and AG Ward. Health modelling for agility in safety-critical systems development. In *2006 1st IET International Conference on System Safety*. IET, 2006.

Christoph Johann Stettina and Werner Heijstek. Necessary and neglected?: an empirical study of internal documentation in agile software development teams. In Aristidis Protopsaltis, Nicolas Spyratos, Carlos J. Costa, and Carlo Meghini, editors, *Proceedings of the 29th ACM international conference on Design of communication, Pisa, Italy, October 3-5, 2011*, pages 159–166. ACM, 2011. doi: 10.1145/2038476.2038509. URL https://doi.org/10.1145/2038476.2038509.

Tim Storer and Ruxandra 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*, pages 228–237. IEEE, 2019. doi: 10.1109/SCAM.2019.00033. URL https://doi.org/10.1109/SCAM.2019.00033.

Viktoria Stray, Bakhtawar Memon, and Lucas Paruch. A systematic literature review on agile coaching and the role of the agile coach. In Maurizio Morisio, Marco Torchiano, and Andreas Jedlitschka, editors, *Product-Focused Software Process Improvement - 21st International Conference, PROFES 2020, Turin, Italy, November 25-27, 2020, Proceedings*, volume 12562 of *Lecture Notes in Computer Science*, pages 3–19. Springer, 2020a. doi: 10.1007/978-3-030-64148-1_1. URL https://doi.org/10.1007/978-3-030-64148-1_1.

Viktoria Stray, Nils Brede Moe, and Dag I. K. Sjøberg. Daily stand-up meetings: Start breaking the rules. *IEEE Softw.*, 37(3):70–77, 2020b. doi: 10.1109/MS.2018.2875988. URL https://doi.org/10.1109/MS.2018.2875988.

SW Suan. *An Automated Assistant for Reducing Duplication in Living Documentation*. PhD thesis, Masters Thesis, School of Computer Science, University of Manchester âĂę, 2015.

Thomas Sundberg. Cucumber Anti-Patterns. http://www.thinkcode.se/blog/2016/06/22/cucumber-antipatterns, 2016.

Anders Sundelin, Javier Gonzalez-Huerta, and Krzysztof Wnuk. Test-driving fintech product development: An experience report. In *International Conference on Product-Focused Software Process Improvement*, pages 219–226. Springer, 2018.

Girish Suryanarayana, Ganesh Samarthyam, and Tushar Sharma. *Refactoring for software design smells: managing technical debt*. Morgan Kaufmann, 2014.

Mikael Svahnberg, Tony Gorschek, Thi Than Loan Nguyen, and Mai Nguyen. Uni-repm: a framework for requirements engineering process assessment. *Requir. Eng.*, 20(1):91–118, 2015. doi: 10.1007/s00766-013-0188-1. URL `https://doi.org/10.1007/s00766-013-0188-1`.

M. Tanveer. Agile for large scale projects - a hybrid approach. In *2015 National Software Engineering Conference (NSEC)*, pages 14–18, Dec 2015. doi: 10.1109/NSEC.2015.7396338.

The Standish Group. Chaos report. `https://www.standishgroup.com/sample_research`, 2019.

Gary Thomas. *How to do your case study*. Sage, 2015.

S.K. Thompson. *Sampling*. CourseSmart. Wiley, 2012. ISBN 9781118162941. URL `https://books.google.de/books?id=-sFtXLIdDiIC`.

Daniel Todd and Ronald D Humble. *World aerospace: a statistical handbook*. Routledge, 2019.

Fernandez Tomas. 9 Ways To Make Slow Tests Faster. `https://semaphoreci.com/blog/make-slow-tests-faster`, 2022.

Thi Thu Hien Tran. Why is action research suitable for education? *VNU Journal of Science, Foreign Languages*, 2009.

K. Trektere, F. McCaffery, M. Lepmets, and G. Barry. Tailoring mdevspice for mobile medical apps. In *2016 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, pages 106–110, 2016. doi: 10.1109/ICSSP.2016.022.

Marina Trkman, Jan Mendling, and Marjan Krisper. Using business process models to better understand the dependencies among user stories. *Inf. Softw. Technol.*, 71:58–76, 2016. doi: 10.1016/j.infsof.2015.10.006. URL `https://doi.org/10.1016/j.infsof.2015.10.006`.

M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. An empirical investigation into the nature of test smells. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 4–15, Sep. 2016.

Dan Turk, Robert B. France, and Bernhard Rumpe. Limitations of agile software processes. *CoRR*, abs/1409.6600, 2014. URL `http://arxiv.org/abs/1409.6600`.

Daniel E. Turk, Robert B. France, and Bernhard Rumpe. Assumptions underlying agile software-development processes. *J. Database Manag.*, 16(4):62–87, 2005.

Ömer Uludag, Martin Kleehaus, Christoph Caprano, and Florian Matthes. Identifying and structuring challenges in large-scale agile development based on a structured literature review. In *22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2018, Stockholm, Sweden, October 16-19, 2018*, pages 191–197. IEEE Computer Society, 2018. doi: 10.1109/EDOC.2018.00032. URL `https://doi.org/10.1109/EDOC.2018.00032`.

Bart Van Rompaey, Bart Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.

Steven H. VanderLeest and A. Buter. Escape the waterfall: Agile for aerospace. In *2009 IEEE/AIAA 28th Digital Avionics Systems Conference*, pages 6.D.3–1–6.D.3–16, October 2009.

Christopher Vendome, Gabriele Bavota, Massimiliano Di Penta, Mario Linares Vásquez, Daniel M. Germán, and Denys Poshyvanyk. License usage and changes: a large-scale study on github. *Empir. Softw. Eng.*, 22(3):1537–1577, 2017. doi: 10.1007/s10664-016-9438-4. URL `https://doi.org/10.1007/s10664-016-9438-4`.

Dasari Venkatesh and Manik Rakhra. Agile adoption issues in large scale organizations: A review. *Materials Today: Proceedings*, 2020.

Leo R. Vijayasarathy and Charles W. Butler. Choice of software development methodologies: Do organizational, project, and team characteristics matter? *IEEE Softw.*, 33(5):86–94, 2016. doi: 10.1109/MS.2015.26. URL `https://doi.org/10.1109/MS.2015.26`.

Jéssyka Vilela, Jaelson Castro, Luiz Eduardo Galvão Martins, and Tony Gorschek. Integration between requirements engineering and safety analysis: A systematic literature review. *J. Syst. Softw.*, 125:68–92, 2017. doi: 10.1016/j.jss.2016.11.031. URL `https://doi.org/10.1016/j.jss.2016.11.031`.

Jéssyka Vilela, Jaelson Castro, Luiz Eduardo Galvão Martins, Tony Gorschek, and Camilo C. Almendra. Requirements communication in safety-critical systems. In Maria Lencastre, Marcela Ridao, and Henrique Prado de Sá Sousa, editors, *Anais do WER19 - Workshop em Engenharia de Requisitos, Recife, Brasil, August 13-16, 2019*. Editora PUC-Rio, 2019. URL `http://wer.inf.puc-rio.br/WERpapers/artigos/artigos_WER19/WER_2019_paper_7.pdf`.

Jéssyka Vilela, Jaelson Castro, Luiz Eduardo Galvão Martins, and Tony Gorschek. Safety practices in requirements engineering: The uni-repm safety module. *IEEE Trans. Software Eng.*, 46(3):222–250, 2020. doi: 10.1109/TSE.2018.2846576. URL `https://doi.org/10.1109/TSE.2018.2846576`.

Ungureanu Vlad. Difficulties in using TDD. `https://medium.com/@learnstuff.io/difficulties-in-using-tdd-41429cf1e6e3`, 2019.

John Huan Vu, Niklas Frojd, Clay Shenkel-Therolf, and David S. Janzen. Evaluating test-driven development in an industry-sponsored capstone project. In Shahram Latifi, editor, *Sixth International Conference on Information Technology: New Generations, ITNG 2009, Las Vegas, Nevada, USA, 27-29 April 2009*, pages 229–234. IEEE Computer Society, 2009. doi: 10.1109/ITNG.2009.11. URL `https://doi.org/10.1109/ITNG.2009.11`.

Matti Vuori. Agile development of safety-critical software. *Tampere University of Technology. Department of Software Systems; 14*, 2011.

Gerard Wagenaar, Sietse Overbeek, Garm Lucassen, Sjaak Brinkkemper, and Kurt Schneider. Working software over comprehensive documentation - rationales of agile teams for artefacts usage. *J. Softw. Eng. Res. Dev.*, 6:7, 2018. doi: 10.1186/S40411-018-0051-7. URL `https://doi.org/10.1186/s40411-018-0051-7`.

Nicholas Walliman. *Research methods: The basics*. Routledge, 2017.

Geoff Walsham. Interpretive case studies in is research: nature and method. *European Journal of information systems*, 4(2):74–81, 1995.

Shiyun Wang. Exploring a research method-interview. *Advances in Social Sciences Research Journal*, 2(7), 2015.

Xiaofeng Wang, Kieran Conboy, and Minna Pikkarainen. Assimilation of agile practices in use. *Information Systems Journal*, 22(6):435–455, 2012.

Yang Wang and Stefan Wagner. Toward integrating a system theoretic safety analysis in an agile development process. In Wolf Zimmermann, Lukas Alperowitz, Bernd Brügge, Jörn Fahsel, Andrea Herrmann, Anne Hoffmann, Andreas Krall, Dieter Landes, Horst Lichter, Dirk Riehle, Ina Schaefer, Constantin Scheuermann, Alexander Schlaefer, Sibylle Schupp, Andreas Seitz, Andreas Steffens, André Stollenwerk, and Rüdiger Weißbach, editors, *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar 2016*, volume 1559 of *CEUR Workshop Proceedings*, pages 156–159. CEUR-WS.org, 2016a. URL `http://ceur-ws.org/Vol-1559/paper19.pdf`.

Yang Wang and Stefan Wagner. Towards applying a safety analysis and verification method based on STPA to agile software development. In *Proceedings of the International Workshop on Continuous Software Evolution and Delivery, CSED@ICSE 2016, Austin, Texas, USA, May 14-22, 2016*, pages 5–11. ACM, 2016b. doi: 10.1145/2896941.2896948. URL `https://doi.org/10.1145/2896941.2896948`.

Yang Wang and Stefan Wagner. Combining STPA and BDD for safety analysis and verification in agile development: A controlled experiment. In Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar, editors, *Agile Processes in Software Engineering and Extreme Programming - 19th International Conference, XP 2018, Porto, Portugal, May 21-25, 2018, Proceedings*, volume 314 of *Lecture Notes in Business Information Processing*, pages 37–53. Springer, 2018. doi: 10.1007/978-3-319-91602-6_3. URL https://doi.org/10.1007/978-3-319-91602-6_3.

Yang Wang, Ivan Bogicevic, and Stefan Wagner. A study of safety documentation in a scrum development process. In Roberto Tonelli, editor, *Proceedings of the XP2017 Scientific Workshops, Cologne, Germany, May 22 - 26, 2017*, pages 22:1–22:5. ACM, 2017a. doi: 10.1145/3120459.3120482. URL https://doi.org/10.1145/3120459.3120482.

Yang Wang, Jasmin Ramadani, and Stefan Wagner. An exploratory study on applying a scrum development process for safety-critical systems. In Michael Felderer, Daniel Méndez Fernández, Burak Turhan, Marcos Kalinowski, Federica Sarro, and Dietmar Winkler, editors, *Product-Focused Software Process Improvement - 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29 - December 1, 2017, Proceedings*, volume 10611 of *Lecture Notes in Computer Science*, pages 324–340. Springer, 2017b. doi: 10.1007/978-3-319-69926-4_23. URL https://doi.org/10.1007/978-3-319-69926-4_23.

Yang Wang, Daniel Ryan Degutis, and Stefan Wagner. Speed up BDD for safety verification in agile development: a partially replicated controlled experiment. In Ademar Aguiar, editor, *Proceedings of the 19th International Conference on Agile Software Development, XP 2019, Companion, Porto, Portugal, May 21-25, 2018*, pages 12:1–12:8. ACM, 2018. doi: 10.1145/3234152.3234181. URL https://doi.org/10.1145/3234152.3234181.

Tom Wengraf. *Qualitative Research Interviewing: Biographic Narrative and Semi-Structured Methods*. SAGE Publications, 2001. URL https://books.google.co.uk/books?id=gj5rvAR1CYgC.

Karl Wiegers and Joy Beatty. *Software requirements*. Pearson Education, 2013.

Laurie A. Williams and Robert R. Kessler. *Pair Programming Illuminated*. Addison Wesley, 2003. ISBN 978-0-201-74576-4. URL http://www.informit.com/store/pair-programming-illuminated-9780201745764.

Andrew Wils, Stefan Van Baelen, Tom Holvoet, and Karel De Vlaminck. Agility in the avionics software world. In Pekka Abrahamsson, Michele Marchesi, and Giancarlo Succi, editors, *Extreme Programming and Agile Processes in Software Engineering, 7th International Conference, XP 2006, Oulu, Finland, June 17-22, 2006, Proceedings*, volume 4044 of *Lecture*

Notes in Computer Science, pages 123–132. Springer, 2006. doi: 10.1007/11774129_13. URL https://doi.org/10.1007/11774129_13.

Dietmar Winkler, Rory V. O'Connor, and Richard Messnarz, editors. *Integrating agile practices with a medical device software development lifecycle*, volume 301 of *Communications in Computer and Information Science*, 2012. Springer. ISBN 978-3-642-31198-7. doi: 10.1007/978-3-642-31199-4. URL https://doi.org/10.1007/978-3-642-31199-4.

Jason D Winningham, David J Coe, and Jeffrey H Kulick. Agile systems integration process. In *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*, page 149. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.

Sune Wolff. Scrum goes formal: agile methods for safety-critical systems. In Stefania Gnesi, Stefan Gruner, Nico Plat, and Bernhard Rumpe, editors, *Proceedings of the First International Workshop on Formal Methods in Software Engineering - Rigorous and Agile Approaches, FormSERA 2012, Zurich, Switzerland, June 2, 2012*, pages 23–29. IEEE, 2012. doi: 10.1109/FormSERA.2012.6229784. URL https://doi.org/10.1109/FormSERA.2012.6229784.

Matt Wynne, Aslak Hellesoy, and Steve Tooke. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2017.

Aiko Yamashita and Leon Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 682–691, Piscataway, NJ, USA, 2013. IEEE Press. ISBN 978-1-4673-3076-3. URL http://dl.acm.org/citation.cfm?id=2486788.2486878.

Aidan Z. H. Yang, Daniel Alencar da Costa, and Ying Zou. Predicting co-changes between functionality specifications and source code in behavior driven development. In Margaret-Anne D. Storey, Bram Adams, and Sonia Haiduc, editors, *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*, pages 534–544. IEEE / ACM, 2019. doi: 10.1109/MSR.2019.00080. URL https://doi.org/10.1109/MSR.2019.00080.

Robert K Yin. *Applications of case study research*. sage, 2011.

Robert K Yin et al. Design and methods. *Case study research*, 3, 2003.

Wanja Zaeske, Umut Durak, and Christoph Torens. Behavior driven development for airborne software engineering. In *AIAA Scitech 2021 Forum*, page 1917, 2021.

Fiorella Zampetti, Andrea Di Sorbo, Corrado Aaron Visaggio, Gerardo Canfora, and Massimiliano Di Penta. Demystifying the adoption of behavior-driven development in open source projects. *Inf. Softw. Technol.*, 123:106311, 2020. doi: 10.1016/j.infsof.2020.106311. URL `https://doi.org/10.1016/j.infsof.2020.106311`.

Appendix A

Interview Questions for use of Agile Methods

Background Questions	
1. What is your job title?	
2. What is the typical life span of a project that you have worked on recently?	
3. Tell us about your role in some recent projects.	
4. What category of system are you working in? e.g. Communications? Fuel? Landing? Other?	
5. What regulatory standards do you follow, if any? Example DO178C.	
Project organisation questions	
1. Who is the customer in your case e.g. company, individuals?	TQ1
2. What about other stakeholders and their involvement?	TQ1
3. How many teams are involved in a typical project?	TQ1
4. What is the typical team size?	TQ1
5. How are the teams' roles structured within the overall project: for example, is there a separate hardware, software and integration team, or some other arrangement?	TQ1
6. Are teams co-located?	TQ1
7. If not, how are teams distributed geographically (e.g. same campus but different buildings, different campuses)?	TQ1
8. How do you communicate between teams? : a) Formal In person b) Informal in person c) Electronic communication, e.g. IM, Email. d) Telephone e) Via issue tracking discussions f) Other: please state.	TQ1
9. Do meeting decisions get recorded and disseminated? If so, how?	TQ2
10. How often does each type of communication occur (many times a day, once or day a day, once or twice a week…)?	TQ1
11. How do you assign requirements to each team in a project? Who is responsible for assigning requirements?	TQ1
Individual Team organisation (Thinking about within your own team)	
1. What are the roles in each team, if any?	TQ1
2. What is the typical level of experience of each team member?	TQ1
3. Are all team members co-located?	TQ1

<table>
<tr><td>4. If not, what is the structure?</td><td>TQ1</td></tr>
<tr><td>5. How do you communicate within individual teams? :
a) Formal In person
b) Informal in person
c) Electronic communication, e.g. IM, Email.
d) Telephone
e) Via issue tracking discussions
f) Other: please state.</td><td>TQ1</td></tr>
<tr><td>6. Do meeting decisions get recorded and disseminated? If so, how?</td><td>TQ1</td></tr>
<tr><td>7. How often does each type of communication occur (many times a day, once or day a day, once or twice a week…)?</td><td>TQ1</td></tr>
<tr><td>8. How do you assign requirements to each individual in a team? Who is responsible for assigning requirements?</td><td>TQ1</td></tr>
<tr><td>9. Do team members participate in knowledge sharing activities such as pair programming or mentoring?</td><td>TQ1</td></tr>
<tr><td>Software Process</td><td></td></tr>
<tr><td>1. Does your team use an agile methodology, if so:
a) Which one does your organisation use (Scrum, XP, Crystal, mixture, parts of several).
b) Which phases of the project life cycle have you applied agile to (requirements, design, implementation, qa/certification, integration, delivery)?
c) Which phases would/are you considering applying agile to in the future? Why?
d) What benefits have you found?
e) What obstacles did you encounter (if any):
i) When changing from your previous SDLC?
ii) Integrating agile methods with avionics regulatory and certification requirements (process vs people/communication)?
iii) Projects that combine both hardware and software components?
f) Where agile methods are not employed, what process model (if any) do you follow?
g) Do you consider different SDLC models for different projects? If so, what reasoning is applied when selecting a model? Who makes the decision?
h) What compromises/customisations have you made to theoretical descriptions of agile processes?
i) What changes/additions are you planning to make to your software process, or do you think would be desirable?</td><td>TQ5</td></tr>
<tr><td>2. Who is involved in deciding the project schedule?
a) Team members
b) Team management
c) Customer
d) Other</td><td>TQ2</td></tr>
<tr><td>3. Who decides if the project schedule should be changed?</td><td>TQ2</td></tr>
</table>

Software Practices	
1. Do you follow sprints? If so: a) How long does a sprint last?	TQ2
2. Which of the following *ceremonies* do you practice? a) Product planning b) Daily stand-ups c) Customer demonstrations/meetings d) Retrospectives For each ceremony: a) If you don't use the ceremony explain why not (not tried yet, don't think it suitable…) b) What benefits have you found of applying it? c) What obstacles? d) What would you do differently to current practice? e) How frequently does the meeting take place? f) Who with (core team, customer or product owner, managers, wider project…) g) What do you do in the meetings? Examples for retrospectives are data gathering techniques, e.g. HSG, data analysis (RC techniques, e.g. 5 whys).	TQ2
3. What documentation is generated within the project process? • Plans • Requirements • Risk management • Certification or regulatory documentation • Other a) How is documentation managed? b) How frequently is it reviewed?	TQ1
Customer Involvement in the Project	
1. Agile methods emphasize on customer's involvement. a) Is the customer involved during the life-cycle of the project? If so: b) How frequent is customer involvement in the project?	TQ2
2. Which phases of the project is the customer involved in (requirements, design, implementation, quality assurance, certification, and delivery, other…)? What is the nature of the involvement in each phase?	TQ1
3. What measures do you use to ensure customer engagement, if any? For example, customer co-location, attendance at meetings, other communication.	TQ1
4. What methods do you use to solicit customer feedback when required (if any?). a) Are these methods formally defined by the customer/contract/wider organisation? How long do you have to wait before you get the feedback from the customer?	TQ1

5. Do you experience delays in delivering software to customers? If so: a) How often do you experience a delay? b) What are the major factors that delay product delivery? c) When a delay in delivering a feature occurs, do you normally: i) Extend the deadline for delivery ii) Postpone the feature (or lower priority features) to the next sprint	TQ4
6. Are multiple releases delivered to the customer during a project? If so: a) Are these releases all certified in the same way/to the same standard?	TQ1
Requirements	
1. Are requirements specifications delivered in a pre-defined structure/template/document? For example, does the template use: a) UML use case or class diagrams, or similar? b) Text c) Structured text d) Other? Can an example requirements document set be shared?	TQ1
2. Do requirements change during the life-cycle of a project? If so… a) What is the source of requirements change? i) External/customer ii) During design, implementation due to discovery of conflicts, feasibility issues iii) Need for further elaboration (e.g. due to ambiguous language or need for restructuring.) b) When does a need for change typically get discovered? i) Requirements analysis ii) Design iii) Integration with hardware iv) QA/Certification v) Delivery vi) Other?	TQ3
3. How are uncertainties regarding requirements, design and implementation resolved? For example: a) Discussion within team b) Negotiation with customer c) Decided by management d) Other 4. What proportion of the requirements specification requires change? a) How often does this occur? b) Are the requirements transformed into a different notation during this process?	TQ3
5. How is requirements cost measured? Story points, person time…?	TQ1
6. How are requirements allocated to successive sprints? Who is responsible for this?	TQ2
7. How many sprints in advance does your team plan?	TQ3

8. During a sprint, how many requirements allocated to a sprint change? a) For a current sprint b) For a future sprint	TQ3
9. How are requirements managed during elaboration/change/evolution? Is a requirements management tool employed?	TQ2
10. How often are requirements reviewed? How is this done?	TQ4
11. What are the average number of requirements (or cost of requirements) per release?	TQ1
12. Do hardware requirements/specification change during software development and vice/versa? a) If so, what affect does this have on either?	TQ4
13. Do requirements and design stabilise during the project? If so, at what point?	TQ1
Quality Assurance	
1. What practices do you employ to ensure that the project is maintainable in the future? For example: • Refactoring • Pair programming • Code review • Static analysis • Automated unit testing (e.g. using frameworks like CPPUnit) • Test-driven development • Other In each case: a) Indicate how much effort is applied to this practice per sprint. b) Which practice you think is most effective at detecting defects?	TQ4
2. What proportion of activity is spent on QA versus software development?	TQ4
3. Do you use automated metrics to analyse your code, for example: • Test code coverage • Normalised vulnerability scores • Mutation testing estimates • Static analysis warnings	TQ4
4. How does certification drive quality assurance practices?	TQ5
Certification/Change Management	
1. Is certification documentation i) Maintained manually ii) Generated from other source artifacts iii) Both	TQ5

2. If maintained manually: a) Does documentation get updated: i) Periodically ii) When a change is made to a dependent artefact. b) If periodically, how frequently is documentation checked? i) When a release is being prepared? ii) After a sprint iii) After a set period of time.	TQ5
Integration	
1. What are the different types of integration that need to be undertaken? (For example: software-software, software-hardware…)	TQ1
2. Who is responsible for integration between software and hardware components? (Separate team, collaborative between individual teams…)	TQ1
3. How often is integration performed (End of every sprint, end of every release…)?	TQ1
4. Does integration lead to feedback to individual project components? If so, how is this communicated and managed?	TQ1

Appendix B

Interview Questions for Investigating Use of BDD

RQ	CRQ	TQ	IQ
Learn about the feasibility of applying behaviour driven development to the development and maintenance of a system developed in a large-scale environment; to gain a deeper insight into the benefits and difficulties experienced when developing and maintaining avionics systems using behaviour driven development.	What is the extent of application of behaviour driven development to development and maintenance of the systems in the company?	TQ1. Background of interviewee	What is your job title?
			What is the life span of the project that you have worked on recently?
			Do you have an estimate of how long will it take to complete?
			Tell us about your role in some recent projects.
		TQ2. Familiarity and experience with agile and BDD?	Did you receive any formal training on using scrum or applying agile? • Has the organisation arranged any scrum/agile trainings? • If yes, Tell us about it
			Have you used agile method(s) before this project? If yes, tell us about it i.e. for how many years etc.
			When did you personally start using behaviour driven development?
		TQ3. Project organisation and Team Structure	Who is the customer in your case e.g. company, individuals?
			Who was the product owner?
			• What about other stakeholders and their involvement? • How many people were directly involved? • How many people were indirectly involved?
			What is the team size?
			What are the roles in each team, if any?
			What is the typical level of experience of each team member?
			Are all team members co-located?
			What different kinds of meetings did you have during the course of the project?
			• What time did you have the meetings? • Who participated in each type of meetings?

			• What did you discuss at the meetings? • Who spoke at the meetings? • What mile stones did you set? How many out of those achieved? • What wasn't achieved? The ones not achieved what were the reasons in your opinion?
			Did meeting decisions get recorded and disseminated? If so, how?
			How did you often communicate? How often did each type of communication occur (many times a day, once or day a day, once or twice a week…)?
			How are the requirements assigned to each individual in a team? Who is responsible for assigning requirements?
			What is the size of this project in terms of number of features?
			Do team members participate in knowledge sharing activities such as pair programming or mentoring?
		TQ4 (Customer Involvement in the Project)	BDD emphasizes on customer's involvement. a) Is the customer involved during the life-cycle of the project? If so: How frequent is customer involvement in the project?
			Who was involved in writing BDD specifications?
			Who was involved in discussing examples and writing scenarios?
			• What methods do you use to solicit customer feedback when required (if any?). • Were these methods formally defined by the customer/contract/wider organisation? • How long do you have to wait before you get the feedback from the customer? • How is the feedback recorded and where?

			• Could you talk about the feedback from the people who have used the system?
			Is there anything you would like to change about the customer interaction and communication, based upon your experience with this project?
		TQ5. Software Process	You use an agile methodology, if so: a) Which one did you use in this project (Scrum, XP, Crystal, mixture, parts of several). b) What was the main motivation behind adopting agile? c) What was the motivation behind adopting BDD? d) What benefits have you found of using BDD?
			Were you involved in specification writing prior to working on this project?
			What information or documentation did you have about the project before starting the project?
			What were the steps or strategy for capturing and documenting requirements in form of BDD features?
			What were the typical steps from conceiving a feature to implementation?
			How different were the user stories elaborated during the user stories workshop, from the ones implemented?
			How well did you understand the requirements?
			Did you need to make adjustment to requirements? if yes, what were the reasons? And where? Feature level/ scenario level? and why?
			How often the technology forced the requirements to change?
			Who was responsible for testing the application?
			Did you have a tester role assigned to a team member? • If not, who did the quality assurance and verification?
			How often did you run the tests?

			How often and what level of testing was performed? e.g. • unit testing, • regression testing, • Integration testing etc.
			What steps did you take when certain testing was unsuccessful?
			How long were the tests?
			How often did you have to change the tests?
			a) What compromises /customisations have you made to theoretical descriptions of agile processes? b) What changes/additions to your software process do you think would be desirable?
			Who is involved in deciding the project schedule? a) Team members b) Team management c) Customer Other
			Who decides if the project schedule should be changed?
			At what point did you start using BDD in this project? If not from the start then What difficulties did you face in switching from old methods and tools to BDD tools and specifications?
			How many iterations has it been?
			How long is an iteration?
			How many sprints?
			Did you have retrospectives or similar at the end of iteration? If yes, who was involved? Did BDD come up in the retrospectives? What did you discuss?
	Is it feasible to apply behaviour driven development to the development and maintenance of a system developed in a large-scale environment?	TQ6. How does BDD help in development of a system in a large-scale environment?	What tool are you using for automating BDD specifications? Have you found any problems with the tools?
			What other tools did you use for supporting the project?
			How many features have been implemented vs the features which have been documented?
			What are the total number of features and scenarios now?

			What is the ratio of duplication of steps in the scenarios?
			How did you handle duplication in scenarios within feature files? • across feature files?
			How much duplication is there in the code? What do you do about it?
		TQ7: How does BDD help in maintenance of a system developed in a large-scale environment?	• How frequent was a change or an update requested in a feature? • What steps did you follow to accommodate and manage a change or update to the specifications? • Who was involved in updating the features? • Do you think the use of BDD has made the specification difficult to understand, extend and change? • How did bdd affect design of the system
			What project related documentation do you have other than feature files? How often that documentation is used for guidance and consultation?
			Have you produced any documentation other than the Gherkin specification yet? Do you think they are all in sync?
			What other project related documentation do you need to produce for this project?
			How often did you revisit the features? For what reasons?
			How much time did you spend in maintenance activities vs implementation?
			Did you have to change the specification while refactoring the code?
		TQ8. Can you record all the requirements with BDD?	Are there any types of requirements that you did not document in BDD but implemented? Were there any UI requirements?
			What were the type of requirements which were difficult

			to document in BDD?
	What benefits and difficulties were experienced when developing avionics systems using behaviour driven development	TQ9 Benefits Difficulties	What would you say about learning BDD? How difficult or easy is it?
			What were your expectations from applying behaviour driven development?
			What are the benefits that you expected but have not achieved from application of BDD?
			In which phases of software development, BDD helped? and how?
			Which phases of software development do you think BDD lacks support for? and why?
			What are the problems that you have faced with BDD?
			Are there any problems that you expected and faced after application of BDD OR expected but did not face?
			How did you prioritize the features? Did the implementation drive the prioritization or the requirements themselves? Who was involved in prioritization?
			How often do you run the feature files and write the tests before actually starting the implementation?
			Were there occasions where you implemented some functionality and afterwards wrote BDD features and scenarios? If yes why do you think that happened?
			Were you happy with the pace of development?
			What were the pain points while using BDD and developing this project?
			What are the lessons learned from this project?