CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Enhancing Software Quality in SMEs: A Holistic Approach to Testing Framework Development

Master's thesis in Computer science and engineering

Edenia Isaac Galan
Moa Berglund

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2024

Master's thesis 2024

Enhancing Software Quality in SMEs: A Holistic Approach to Testing Framework Development

Edenia Isaac Galan
Moa Berglund

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2024

Enhancing Software Quality in SMEs: A Holistic Approach to Testing Framework Development
EDENIA ISAAC GALAN, MOA BERGLUND

Supervisor: Robert Feldt, Department of Computer Science and Engineering
Examiner: Richard Torkar, Department of Computer Science and Engineering

Master's Thesis 2024
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2024

Enhancing Software Quality in SMEs: A Holistic Approach to Testing Framework Development

EDENIA ISAAC GALAN
MOA BERGLUND
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Numerous testing frameworks have been created to facilitate the pivotal process of software testing. The suitability of frameworks is contingent upon the size of an organization and this study focuses on Small to Medium-sized enterprises (SMEs). Regarding testing frameworks targeted to SMEs, none has proven to cover all aspects important for enhancing a software testing process. Therefore, the purpose of this Master Thesis is to develop a comprehensive software testing framework tailored for SMEs. The resulting framework builds upon a framework proposed by Argüello *et al.* [1], which has been modified based on relevant research about software testing and observations of a software company. What makes this study noteworthy is the recognition of human aspects related to the software testing process and the advances toward a more complete testing framework. The framework presents twelve key axes prominent in software testing. All axes are divided into three maturity stages, which can be achieved by following tailored improvement activities. The study was performed in collaboration with a Software-as-a-Service company, where the framework could be tested and evaluated in a real-world context. However, the entire framework could not be tested due to time constraints, which leaves a more thorough evaluation as an opportunity for future work. The practical application of the framework suggests that it is not only theoretically sound but also applicable and successfully improves the testing process. The framework contributes to filling the gap of a complete testing framework, which contributes to the broad mission of ensuring software solutions' accuracy, quality, and reliability.

Keywords: Software Testing, Testing Framework, Code Quality, Test Automation, Test Management, Test Environment, Test Framework Development, Framework Application, Software Process Improvement, Human Aspects

Acknowledgements

We would like to sincerely thank our supervisor Robert Feldt, who has guided us continuously throughout this study by providing great expertise in the software engineering research field.
We would also like to thank the participating company, for making this study possible and enabling a good collaboration throughout the process. We wish to express our special gratitude to all the personnel at the company engaging in the workshop, interviews, and framework application. Lastly, our sincere thanks go to our supervisor at the participating company. He has been a dedicated support during the work, helping us with the resources required and providing consolidation when needed.

Edenia Isaac Galan and Moa Berglund, Gothenburg, June 2024

Contents

List of Figures

List of Tables

1
Introduction

In the rapidly evolving landscape of technology where software becomes increasingly integral to our daily lives and businesses, software testing emerges as a critical phase of the software development process. Thorough software testing helps ensure the accuracy, quality, and reliability of software solutions, safeguarding against potential failures. Its significance can not be overstated, as it directly impacts user satisfaction and operational efficiency. It remains a critical investment for any organization aiming to deliver high technological solutions in the competitive landscape. It is estimated that the cost of poor software quality in the US in 2022 reached up to 2.41 trillion dollars [2]. Still, the inferior state of many software systems today indicates that software testing is often overlooked or not prioritized [3].

To address the issue of insufficient software testing, several artifacts have been produced to improve testing processes. Examples of such artifacts are frameworks and methodologies based on Testing Process Improvement (TPI) activities. These aim to increase the quality of the software development process by providing guidelines for best practices. The successful implementation of TPI models can offer several operational, technical, and business benefits. Minimized test cycle time, development of adequate training for test personnel, better software quality, fewer high-severity defects, improved test automation, improved test-case design, increased customer satisfaction, and increased business opportunities are a few examples [4]. There exist numerous TPI models, such as the Testability Maturity Model (TMM) [5], the Test Improvement Model (TIM) [6] and the Test Process Improvement Model (TPI) [7]. However, these present challenges for adoption in small and medium-sized enterprises (SMEs) due to the frameworks' extensive nature and the required resources.

Small and Medium-sized enterprises can be defined as companies that employ fewer than 250 people, have an annual turnover not exceeding EUR 50 million, and/or an annual balance sheet total not exceeding EUR 43 million [8]. SMEs looking to improve their testing processes through TPI activities face several challenges, many of which are tied to the specific characteristics of an SME. Kituyin and Amulen [9] discuss some of these challenges, naming how SMEs' primary business objective is to survive and must, therefore, carefully balance improvement efforts and ongoing development activities. They also state how employees in smaller organizations often possess a unique mindset characterized by a strong belief in their competence. This belief, in turn, creates a culture where formal training is frequently undervalued, in terms of time and financial investment. Additionally, Kituyin and Amulen describe how adherence to predefined rules or procedures is often considered unnecessary. Moreover, the environment in small organizations is typically characterized by short

cycle times and high-stress levels. This constant pressure to deliver results quickly sets the focus on immediate problem-solving instead of improvement activities. Kituyin and Amulen continue to describe how SMEs face several barriers to entry, such as a lack of core competencies, difficulties in affording the high cost of process improvements, and a flat organizational hierarchy where roles and responsibilities are not clearly defined. Many organizations find ongoing expenses, substantial human resource allocation, and considerable time dedication to be prohibitive factors. The limitations on available resources affect SMEs' ability to implement and maintain sophisticated testing processes. Additionally, the rapid pace of development applied in many SMEs can make it challenging to establish rigorous testing processes without hindering product timelines.

Several frameworks have been proposed to meet the specific needs of SMEs. As will be further discussed in Section 3, the suggested frameworks all have drawbacks or room for improvement. Areas such as automatic testing, requirement engineering, and test prioritization are examples of improvement suggestions from expert evaluations. Notably, a common omission in the frameworks revised in this study is the consideration of human aspects. The emerging field of Behavioral Software Engineering (BSE) addresses a research area concerned with human aspects, broadly recognizing their significant impact on software development practices [10]. In fact, the majority of issues in software development processes can be linked to human factors [10]. Since software testing is a human-intensive activity [11], it is essential to consider the human aspects as a critical part of the testing process, which makes the existing gap in the revised frameworks significant.

The existing frameworks all bring new aspects to the table, building on each other's gaps. However, none has proven to be able to cover all aspects essential for enhancing the software testing processes. The absence of the before-mentioned considerations and improvement suggestions propose areas ripe for further research. It becomes clear that a more complete testing framework for SMEs has yet to be developed.

1.1 Purpose and Research Questions

The purpose of this Master's Thesis is to develop a comprehensive software testing framework tailored for Small and Medium-sized Enterprises. Given the vast amount of pre-existing frameworks that cumulatively build upon each other's findings, this study builds upon an existing testing framework rather than developing a new framework from scratch. The study was done in cooperation with a small Swedish Software-as-a-Service company. The three research questions that guided the work toward fulfilling this purpose are listed below.

RQ 1: What improvements and extensions can be effectively integrated into existing software testing frameworks to enhance their applicability and effectiveness in SME environments?

RQ 2: In what ways can Behavioral Software Engineering principles be incorporated into software testing frameworks to address human and social aspects in SME environments?

RQ 3: What is the impact of the refined framework on the software testing process and overall product quality in SMEs when applied in a real-world setting?

1.2 Limitations and Delimitations

Working with a company allowed to test the resulting framework in a real-world setting. However, it also imposed limitations within the context. Given the time constraints and the available resources at the company, the entire framework could not be tested. Rather, parts assessed as most beneficial and applicable at the participating company were tested. The framework application was also delimited to a specific code module and was implemented by a single designated person at the company. The participating company does not adhere to standardized testing and coding strategies, leading to inconsistencies in code quality and divergent testing practices among developers. Such disparities could significantly have impacted the results observed from deploying the framework.
Another limitation of the study is that it is not based on an extensive literature review, such as a systematic mapping study. Instead, the snowballing technique is utilized to find relevant literature in a comprehended amount of time. Snowballing is a generally accepted technique that can reach high internal validity as well.
In the participating company, most developers work directly with customers, leading to a significant portion of the code being uniquely tailored to individual customer needs. However, a subset of the development team focuses on the company's internal code, which is not client-specific but crucial for the product's core functionality. The automated tests present within the company are designed exclusively for this internal code. Given this operational context, the scope of this study has been intentionally narrowed to focus solely on the internal development processes, with the application of the study's findings being confined to this segment of the code base.

1.3 Significance

The study is relevant to both practitioners and researchers. The practical application of the resulting framework aims to validate its efficiency and relevance in an actual business context, ensuring that it is not only theoretically sound but also practically viable and beneficial. It is anticipated that the results of this study benefit software development in SMEs by providing a comprehensive testing methodology, thereby enhancing their overall testing process and product quality. Research in the field benefits since this study endeavors to fill the gap of missing aspects in existing testing frameworks.

1.4 Structure of Thesis

The structure of the rest of the report is as follows: Chapter 2 presents the theory that forms the theoretical basis for the study. Chapter 3 continues with presenting existing work in the field by introducing existing testing frameworks and how these are not complete according to this study's scope. The chapter narrows down to presenting the framework chosen for extension in Section 3.2. Chapter 3 ends with highlighting the limited previous work of human aspects concerning testing frameworks. Chapter 4 explains the methodology used to execute this study. Continuing, the result of this study is presented in three chapters. It begins with Chapter 5, which displays the results of the activities performed before the testing framework of this study was completed. Chapter 6 presents the proposed framework. Then, Chapter 7 demonstrates the framework application at the company and presents the results of the testing process by comparing the process before and after the framework application. Chapter 8 provides a discussion about the work, including exploring threats to validity and suggestions for future work. Finally, Chapter 9 summarizes the work and highlights the outcome.

2
Background

This chapter covers the theoretical concepts that are the foundation of the study. The chapter begins by exploring the concept of software process improvement, which describes the general approach to streamlining a software process. Continuing, the chapter outlines basic software testing theory concepts; testing levels, static and dynamic testing, testing techniques, and test optimization. Continuing, a selection of metrics for assessing software testing is presented. Following, the concepts of continuous integration and continuous delivery are explicitly covered. Moving into the human aspects of software testing, Section 2.10 outlines the fundamentals, which include presenting several relevant cognitive biases. Finally, the chapter ends with covering the concept of code quality.
Many concepts explained in this chapter are based on the broadly accepted ISO/IEC/IEEE 29119 standard [12]. Using the standard as the foundation of theory ensures high quality due to the nature of international standards provided by experts in the field. If the information or concepts presented are based on other resources, those are explicitly cited in the text.

2.1 Software Process Improvement

Improving the software development process can lead to several advantages, such as decreased costs, higher code quality, and more reliable products. Taipale and Smolander [13] claim that software process improvement (SPI) is one of the most important methods to streamline development and testing processes. However, they explain that SPI models not only bring success but can also be challenging to apply since they imply organizational change. To utilize the possibilities of SPI, the efficiency, effectiveness, and quality of the current process must be determined to trace areas of improvement. This is called Software Process Assessment (SPA). The cycle of SPA and SPI is illustrated in Figure 2.1. Narrowing down the scope to only improving the test process, the activities are called Software Testing Process Improvement (STPI - often shortened TPI) and Test Maturity Assessment (TMA) respectively [4].

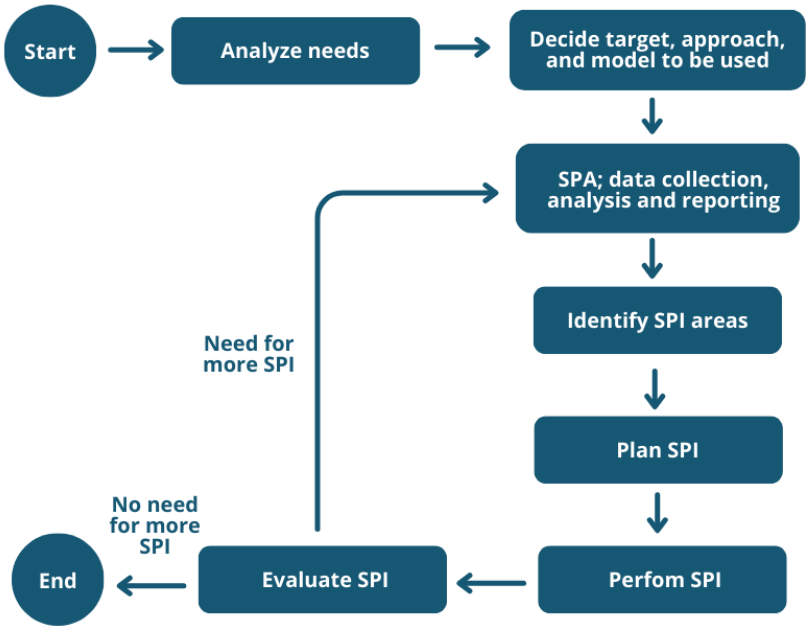

Figure 2.1: An overview of the SPA and SPI process. The diagram is inspired by a more detailed version of TMA and TPI created by Garousi *et al.* [4].

Software Process Improvement (SPI) models aim to increase software development quality by giving guidelines for best practices. There are numerous SPI models created, and Taipale and Smolander [13] problematize that a general perception of these is that they are not concrete enough to be applicable and, in contradiction, not abstract enough to suit variations of organizations. Afzal *et al.* [14] also problematize the application of SPI models since the SPI needs to fulfill an organization's specific needs to be suitable.

2.2 Levels of Testing

It is possible to test software on different levels. Each level has a subsequent purpose that aims to target certain risks. The generally acknowledged hierarchy of levels is unit testing, integration testing, system testing, and acceptance testing. It is common to model the levels as a pyramid to showcase how they are the foundation of each other, see Figure 2.2. The pyramid also showcases how the number of test cases generally decreases as the levels advance.

Unit testing is the lowest level and aims to test a small isolated piece of code at the time, for example, a method or class. The goal is to verify the specific functionality of the unit under test without any dependencies on other parts of the code. Torkar and Mankefors-Christiernin [15] highlights that unit testing is often considered crucial at the beginning of software testing since it hinders faults from propagating to the other levels. Software mocking can be used to achieve isolated units of code. Mocking means mimicking the interface of the actual dependencies but providing predefined responses to method calls, allowing developers to control the behavior of these dependencies during testing. If mocking is not used and the different code components are allowed to interact during the test, the test is on the integration level. The objective of this test level is to verify that the interactions between components are working correctly. Correcting faults on the integration testing level is still fairly easy as the tests are positioned close to the artifacts. When moving upwards in the pyramid, the artifacts are combined and more complex, making it more difficult to detect the fault.

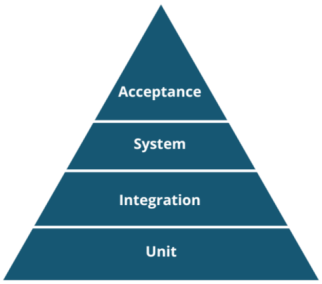

Figure 2.2: The "testing pyramid".

System tests deviate from unit and integration tests as they concern the system outcome and do not cover the specific implementation of components. This means that system tests can also involve API calls to another source, making the faults more challenging for the developer to correct. However, these tests are important as they are more realistic than the previous levels, yet still possible to perform from the developer site. The testing pyramid is completed with Acceptance testing, which aims to test the system in its intended context. End users perform this, and it is thereby impossible to automate, making it an expensive yet vital test.

2.3 Static and Dynamic Testing

The ISO/IEC/IEEE 29119 standard [12] explains static and dynamic testing as two complementary approaches to evaluate code. Dynamic testing executes code, and static testing does not. Due to this difference, the approaches are suitable in different stages of the software development lifecycle. Static testing can be performed at any point in the process as long as there is some document or source code to test. Static evaluation identifies problems before execution and is vital for improving code quality, readability, and maintainability. Static testing can be performed utilizing

various techniques, such as inspection tools, walkthroughs, and reviews. Dynamic testing aims to ensure the software functionality during runtime and can thereby not be performed before there is some executable code. However, the whole software must not be complete to begin dynamic testing since system units can be tested separately, as mentioned in 2.2.

2.4 Testing Techniques

Numerous techniques exist that aim to facilitate the process of software testing. What they have in common is, in general, that they are methodologies expressed abstractly and generically, making it possible to apply them in various contexts. Below, some acknowledged testing techniques are explained. What these have in common is the shared goal of enhancing testing effectiveness. Some focus on streamlining the testing process (Regression Testing, Sanity testing), others on identifying defects (Exploratory testing, Property-based testing, Risk-based testing), and one focuses on the quality of the test cases (Mutation testing). The next section, Section 2.5, delves deeper into deriving test cases by exploring test specification techniques.

Regression Testing refers to retesting software after modifications. The aim is to verify the original behavior of the software so that the modification does not intrude on existing functionality or introduce new issues. It is a powerful approach to catch unintended introduced side effects. It is common to specify a regression test suite that exercises the most important functions without detailed testing. It is advised to have tests of different levels in the regression suit. Every commit could automatically trigger regression tests through the Continuous Integration (CI) pipeline, further described in Section 2.8. [16]

Sanity Testing extends regression testing by adding initial checks before the comprehensive regression testing is executed to be able to stop the build before the full suite is executed. Sanity testing can also be conducted in other software lifecycle stages, such as before a demo or user acceptance testing. The aim is to run a few critical tests to ensure fundamental functionality. [17]

Exploratory Testing is an experience-based test technique. These are generally informal and include little planning, making them simple to conduct. However, it is difficult to measure the coverage of an experience-based test technique since the full set to test must not be defined. Exploratory testing means that testers interact with the product creatively, discovering what it can and cannot do and potentially revealing bugs or flaws. The tester bases the behavior on existing experience and knowledge about user behavior and common types of failure. [16]

Property-based Testing advances the lower level of testing by allowing a tool to generate random inputs for the test case and assert that the input obeys a particular property instead of verifying a specific output as many traditional testing techniques would do. The tool generates inputs with the ambition to try to find values that break the property, to reveal faults in the code. The explanation of this technique is

based on the conference paper [18] provided by Karlsson *et al.* They give a clarifying example to illustrate property-based testing:
"*Imagine that we test a search function. The search function requires a search string as input and returns a list of items that match the search string. Instead of using a predefined search string with a predefined outcome, we instead use a string generator for the input string and formulate the invariant properties of the resulting list of items. Such properties include (1) the name of the resulting entities should always contain the search string as a substring in their name, (2) the returned entities should always conform to a given format, and (3) the function should always return a successful result, i.e., not crash.* "

Risk-based Testing is a technique where the activities performed are based on the analyzed risk of performing or not performing the test. Risks can belong to either the product or the project. Potential risks must first be identified, and then the potential impact and likelihood are analyzed. A typical driver in this testing technique is the risk that the product does not meet the stakeholder demand. The ISO/IEC/IEEE standard [12] recommends risk-based testing as the approach to strategies and management testing.
Mutation Testing addresses the quality of the tests. The theory in this paragraph is based on Su *et al.*'s conference paper [19]. The core of mutation testing is that the developer actively inserts a bug in the code and then runs the test connected to it to verify that the test reveals the fault. If the fault is not revealed, the test case should be revised. Mutant operators assist in guiding how to insert a bug in the code. A mutant operator is a guideline for modifying a statement in the code into a faulty statement. The operators are defined with the target to simulate acknowledged programming errors. There exist numerous mutant operators. One example is to replace the operator $>$ with $>=$. Many others are shown on the webpage [20] that hosts the mutation testing tool Pitest [21] for the Java framework.

2.5 Test Specification Techniques

Test specification techniques are methods used to derive and select test cases. These techniques can be broadly categorized into Specification-based testing and Code-based testing. The first category includes techniques that use the program requirements as testing input. In contrast, the second category is techniques that use the source code to guide the testing activities. These techniques complement each other, and both should be used to find a complete test suite. The following sections will delve deeper into some well-known test specification techniques. The concepts explained are based on Aniche's work "Effective Software Testing: A Developer's Guide" [22].

Equivalence Partitioning is used to reduce the number of test cases by dividing input data into partitions from which test cases can be derived. Although the number of possible program inputs is nearly infinite, certain groups of inputs will result in the same program behavior, regardless of the specific input value. In test terminology, the set of inputs triggering the same behavior are equivalent, and

therefore, one specific input value can be chosen to represent the entire partition.
To apply this technique, one can start by reviewing the software's specifications and requirements to identify the appropriate input data. The next step involves separating the identified test data into equivalence partitions. These partitions should cover valid and invalid input data to ensure test cases can detect correct behavior and errors. Finally, a representative value from each partition should be selected, and the test cases should be developed using the value as input data.
For example, consider having a simple function to register a user if an accepted username is provided. The username has only one criterion to make it valid: it is within a certain length, say between 8-14 characters. This means that the test case has three equivalent partitions:

- Invalid: username less than 8 characters long
- Valid: username between 8 and 14 characters long
- Invalid: username more than 14 characters long

So, in this simple case, there are only three different cases that need to be derived.

Boundary Value Analysis focuses on the values at the boundaries of the input domains. The rationale behind this approach is that errors are more likely to occur at the boundaries of input ranges rather than in the center of the input range. Boundaries are the places where the program suddenly changes its behavior, often between partition classes. Whenever there is a boundary, two points should be tested - one belonging to each partition. To explore boundaries, looking at all the partitions and considering the inputs between them is recommended.
In order to apply this technique, the partition classes should be identified. Once this is done, the boundary values can be identified by selecting consecutive values that trigger the behavioral change between the classes. Finally, test cases using boundary values as inputs should be developed.
Continuing on the example provided in the Equivalence Partitioning section, there are two places where the function changes its behavior and inputs cross the partition classes: at 8 and 15 characters. The values from both sides of the boundary of these changing points should be considered. Therefore, the following test cases should be considered:

- Invalid: Username with 7 characters
- Valid: Username with 8 characters
- Valid: Username with 14 characters
- Invalid: Username with 15 characters

Code-based testing can utilize **code coverage** criteria to gain confidence that the program works as expected. Different types of code coverage exist that should be considered when designing test cases to augment the test suite.
Line coverage measures the extent to which the source code of a program is executed when a test suite runs. It represents the percentage of individual lines of code that are executed. If a test touches a line of code in any way, that line is counted as covered.
Branch Coverage considers how the program behaves differently based on the evaluation of branching instructions like "if," "for," and "while" statements. A branch

represents a distinct path in the execution flow stemming from decisions. For example, an "if" statement can be evaluated as true or false, opening up two distinct branches. To reach full branch coverage, each possible path in the program's control flow should be executed when running the test suite.
Condition Coverage considers each condition in a branch statement. For example, an "if" statement can consist of two individual conditions (if a AND b) and both need to be evaluated to be true for the entire statement to be evaluated to true. To reach condition coverage, the test suite should exercise each of those conditions being evaluated to true and false at least once.
To reach full *Path Coverage*, all the possible paths of program execution should be covered. This means that every possible combination of conditions and loops in the code has been evaluated in the test suite. Ideally, this is the strongest criterion, but it can be complicated and expensive to achieve since the number of test cases needed to cover all possible combinations rapidly increases with the number of conditions and loops in the code.

2.6 Test Optimization

One approach to reduce testing costs is to use different test case optimization techniques. Three widely accepted strategies are Test Case Selection, Test Case Minimization, and Test Case Prioritization. The first refers to choosing an appropriate subset of the test suite based on knowledge about the software. The second aims to increase efficiency by removing redundant tests and shrinking the test suite to a minimal subset that still reaches the desired adequacy criterion. Test case prioritization does not reduce the test suite compared to the two mentioned. Instead, the test suite is ordered to execute the most important tests first. In this way, critical errors are detected earlier, which optimizes time since it is possible to start addressing faults before the whole suite is executed. Meanwhile, if the test execution is abrupt, the most valuable tests have a higher chance of being run. [23]

2.7 Software Testing Metrics

The use of metrics is a fundamental practice in mathematics and physical sciences, and it has been successfully applied to the field of software engineering [24]. The purpose of software metrics is to establish and use quantitative measurements to manage the software development process, assist the organization in understanding its current capabilities and challenges, and objectively indicate the success of software process improvements. Different categories of software testing metrics exist. Quality metrics, efficiency metrics, effectiveness metrics, coverage metrics, and process improvement metrics are examples of a few. The relevant metrics for this study are discussed in detail in this section.

Code Coverage is not only used as a test specification technique but is also an extensively used metric. Code coverage can be used to evaluate the comprehensiveness of the test suite. It quantifies the percentage of code that is executed during testing.

A higher code coverage indicates that an extensive part of the code has been tested, increasing the likelihood of faults being detected. This metric facilitates identifying areas of the code that are not properly tested. [25]

Code Churn vs Test Churn. Code Churn is a software development metric that indicates the level of code changes over time. The changes can be additions, deletions, or edits. Measuring code churn provides insights into the maintainability of the code base [26]. On the other hand, test churn refers to changes within the test suite. Comparing these metrics helps assess how well the test suite is maintained. A high ratio of code churn to test churn might indicate insufficient testing in response to code changes.

Flakiness Score measures the consistency of test outcomes over multiple executions. A test is considered flaky if it yields different results under the same conditions across runs [27]. A high flakiness score undermines confidence in the test suite and can waste resources on investigating false fails. Their nondeterministic nature makes debugging difficult for developers and can translate into issues for end users.

2.8 Continuous Integration

Continuous Integration (CI) is a solution that automates building a software project upon every commit. The main goal is to ensure the health of the software early in the development lifecycle. A continuous integration pipeline can consist of various components, including compilation, rebuilding the database, running tests, inspections, and feedback mechanisms. In the context of software testing, it is common to integrate tests into the CI pipeline to automate the execution of tests. One benefit of CI is that these automated tests increase confidence and reduce repetitive manual work. To achieve this, a build script must be shared in the version control system instructing what tests should run on every build. GitHub Actions [28] is a feature that, among other functionalities, facilitates the use of build scripts in GitHub by providing a solution to write and execute them. Static analysis can be introduced in the build script, ensuring code inspections on every commit. The book [29] highlights CI as a "centerpiece for quality". The mentioned book serves as the primary source of knowledge for the stated information about CI. The ISO/IEC/IEEE standard highlights the usage of CI when describing automated tests.

2.9 Continuous Delivery

Continuous Delivery (CD) is an extension of Continuous Integration, still alleviating the process of automated testing. CD aims to ensure that the software is reliably released at any time, by automatically deploying code changes after the build stage. However, the deployment still needs approval before release and is not fully automated. Automatically deploying software upon every code change is called Continuous Deployment. [30]

TeamCity [31] is a software platform that facilitates workflows of CI/CD. Regarding CD, TeamCity enables simple software deployment from the version control system

with one click. It is also possible to configure software tests to run on this development.

2.10 Human Aspects related to Testing

MohamadSaleh and Alzahrani [10] highlight that the majority of issues in the software development process are related to human factors. However, the research area has been given lesser attention compared to the technical field [32]. Behavioral Software Engineering (BSE) is the study of cognitive, behavioral, and social aspects of software engineering performed by individuals, groups, or organizations [32]. The research in the area encompasses individual and organizational concepts such as motivation, communication, cognitive biases, and organizational culture. BSE is a growing area of research, but the existing research is unbalanced. BSE concepts have been applied to a limited number of Software Engineering areas and are not frequently connected to a specific phase of the Software development process. One Software Engineering area that has been heavily underrepresented in BSE research is Software Testing, with around 1 % being connected to this area [32]. Since software testing is a social-intensive activity [11], human aspects should not be neglected as a critical part of the testing process. Cognitive biases are a phenomenon in the human aspects scene, which heavily affects the process of software engineering and, thereby, software testing. The next section outlines fundamental knowledge about cognitive biases and how they relate to software engineering. After covering cognitive biases, another human aspect is explained in relation to software testing; Motivation.

2.10.1 Cognitive Biases

Cognitive biases are systematic deviations from optimal reasoning. They can be explained as recurring errors in thinking or patterns of bad judgment [33]. This is a universal phenomenon observable in different social contexts. Cognitive biases explain many common software engineering problems in diverse areas, including design, testing, requirements engineering, project management, etc. While there is no definitive list, over 200 cognitive biases have been identified in psychology, sociology, and management research. The following section describes some of the most well-known cognitive biases and their implications for software testing. The concepts explained are based on the work of R.Mohanani et al.[33], "a systematic literature review of Cognitive biases in the area of Software Engineering".

Confirmation bias is the tendency to pay more attention to information that agrees with perceptions. In the area of Software testing, confirmation bias can explain the tendency of software testers to design and run tests that confirm the expected execution of the program rather than ones that could reveal failures. During debugging, confirmation bias sometimes leads programmers to misidentify the reasons for program failure.
Optimism bias is the tendency to produce unrealistically optimistic estimates, judgments, and predictions. This bias can, for example, contribute to heavily underestimating the time needed to write tests.

Anchoring and adjustment is a common heuristic in which one makes estimates by adjusting an initial value called an anchor. **Anchoring bias** is the tendency to stick too closely to the anchor. This phenomenon can lead to errors in the anchors propagating to the new artifacts. In the context of software testing, anchoring bias can explain why developers may re-use tests written for similar code instead of writing an entirely new test.

Availability bias is the tendency for easy-to-recall information to influence preconceptions or judgments unduly. Software professionals tend to rely on their experiences or prior knowledge to seek information, ignoring unfamiliar and less-used keywords and locations. Availability bias is associated with over-representing memorable code features and certain controversies, such as preferring a familiar but less suitable programming language. In the context of Software Testing, availability bias may affect decisions made about the testing process and testing environment. For example, the decisions about what tools or techniques to use may be based on a lack of knowledge, going for the easy-to-find solutions.

Overconfidence bias is the tendency to overestimate one's skills and abilities. In software testing, this bias can explain why some developers consider testing unnecessary and heavily under-prioritize this activity. Developers may feel so confident about their code that they decide that some or no testing is good enough.

Adjustment bias can lead individuals to persevere with established or familiar opinions despite superior information, arguments, or conditions. In software testing, this could take the form of not changing to more suitable testing tools or techniques because of comfort with the old solution.

Representativeness bias is the tendency to make a judgment by fitting an object into a stereotype or model based on a few properties. Due to representativeness, people expect to obtain results that are considered representative or typical. This may lead to problems during the software testing phase. Consider a developer who has to decide whether or not a specific test case should be eliminated. Such a decision depends on whether the particular test case is considered representative of the entire population of test cases that produce errors. This may explain why test cases that produce errors may be overlooked, leading to increased software defect density.

2.10.2 Motivation

The concept of motivation has been given multiple definitions across a variety of research areas. Work motivation has been referred to as the desire to work, signaled by the individual's engagement, focus, and attitudes towards the work [34]. According to Hackman's [35], work motivation refers to being engaged in one's work because of the positive internal feelings generated by performing well. Motivation is also defined in the Goal Setting Theory [36], as the willingness to strive for the goals of a particular organization. According to this theory, four elements represent motivated behavior: Direction - as the direct attention and action; Effort - as the amount of effort mobilized in proportion to the perceived requirements of the goal or task; Persistence - as the directed effort extended over time and Strategy Development - as the development of strategies or action plans for attaining one's

goals.
Another theory, the "Motivation-Hygiene Theory" [37] classifies motivational factors as either extrinsic or intrinsic. Extrinsic motivation means the activity is a means to achieve a determined goal or desired result, for example, material gains or a promotion. While intrinsic motivation means that the reward lies in the action itself. According to Steers [38], if it were possible to synthesize the different concepts of motivation, they would all have a common characteristic: they are all primarily concerned with factors or events that energize, channel and sustain human behavior over time.
In Software Engineering, motivation has been reported to have the most significant impact on productivity and software quality management. However, it still faces the challenge of being undermined [39]. In Software Testing, low motivation can lead to poor testing and overlooking of software defects [40]. Shah *et al.* [41] concluded that testing quality is affected by testers' motivation and appreciation of testing efforts. Fraser and Straubinger [3] highlight in their paper "A Survey on What Developers Think About Testing" that nearly 60 % of the participants would test more if they were better recognized. Recognition has been conceptualized as constructive and genuine feedback acknowledging individuals and their expertise and contributions. It is seen as a return on an employee's effort, dedication, and results [42].

2.11 Code Quality

Quality assurance and testing are closely interrelated, where testing is a method to perform quality control and thereby facilitates increasing the code quality [12]. In return, testing benefits from software quality activities since higher code quality reduces testing costs [10]. This is mirrored in the study [13], where the authors highlight how poor quality causes extra work in testing activities. Toroi *et al.* [43] underscore the importance of continuous improvement of the software test process since it streamlines the code quality, which is claimed to be an important factor for competitiveness in the market. Among others, some critical aspects for achieving high code quality are adhering to code standards, principles, and design patterns, reducing complexity, and maintaining the code over time. Depending on the context, different guidelines are suitable to achieve high code quality. For example, the SOLID Design Principles are widely recommended if object-oriented programming is performed. The SOLID principles are presented lightweight in a blog post [44] by Samuel Oloruntoba. Hassan and Singh [45] conducted a study where they applied the SOLID principles to a software product, compared it to not following the principles, and found that they increased code quality significantly.
Pair programming is a technique where two developers work together on one computer on the same programming task. Williams *et al.* [46] demonstrate that this is not only a technique that is often evaluated as enjoyable but also ensures a higher quality of code in a shorter time compared to independent programming, no matter the expertise level.
There exist various ways to measure code quality from different views. One common way is to use metrics generated from various software tools. Following, some broadly accepted and commonly used metrics are presented. There exist variations on how

to calculate the metrics, and below are the definitions for this study.

Code smells are signals of issues in the code that do not cause any direct errors. These can lead to trouble anyhow since they limit the codebase's maintainability, readability, and extensibility. Often, code smells indicate a deeper issue in software design, but they can also be a syntactical smell.
Cyclomatic complexity refers to the number of linearly independent paths in a function. Whenever the control flow of a function divides, the cyclomatic complexity is increased by one. For example, the cyclomatic complexity of a function that consists of a conditional construct (if-else) is two.
Cognitive complexity refers to how difficult it is to understand the code's paths from a human perspective. The metric increases with nested blocks and complex operators.

3
Related Work

There has been a considerable amount of research in the area of testing frameworks and SPI models, and many frameworks have been proposed. However, these mainly cover technical or procedural aspects, and only a few of them include human aspects of the testing process. This chapter begins with presenting relevant existing software testing frameworks. After delivering a broad background regarding existing frameworks, the chosen framework for extension is explained in more detail. The chapter ends with highlighting the limited previous work in the field of human aspects in relation to software testing.

3.1 Existing Frameworks

Some of the most acknowledged SPI models are the CMM [47] and TMM [48] family, they found the majority of SPI models created [4]. CMM is an acronym for the Capability Maturity Model, which facilitates evaluating the maturity of a software process and provides a set of guidelines to enhance the software development process. CMM is based on defined key process areas, such as Software Project Planning and Defect Prevention. The key process areas compose five maturity levels to be achieved sequentially. CMM engages in all testing levels mentioned in Section 2.2. CMMI (the Capability Maturity Model Integration) is the updated model of CMM, which enhanced the model's adaptability by providing more flexibility and guidance on improving key processes. Even though CMMI exists, it has still received criticism for being too strategic, formal, and impractical [49].
A consensus emerges among research in the field that CMM is not appropriate for Small and Medium-sized companies (SMEs) due to its extensive nature and the amount of resources required [50], [49], [51], [52], [53], [54], [43]. Lin [50] even states that it is unrealistic for SMEs to practice all reports CMM implies. Even though CMM was still the most used SPI model year 2007 [51]. The same literature review also showed that SMEs struggle with progressing through the maturity levels of CMM, which indicates that the standards are not suitable for the nature of SMEs.
TMMI, the Testability Maturity Model Integration, is a derivative of CMMI focusing specifically on testing. It was developed by the TMMI Foundation [55], a non-profit organization. TMMI provides a framework for assessing and improving the software testing process through five maturity levels. Its predecessor, TMM, has been criticized for its abstract nature and lack of specific guidelines for progressing through the levels. CMMI and TMMI have around 20 process areas, with slight variations depending on the version.

The CMMI and TMMI models are staged approaches. Afzal *et al.* [14] explains that a staged representation implies that to mature to the next level, all requirements for all process areas must be fulfilled. In comparison, each process area can be matured individually using a continuous approach. Note that CMMI can be represented with both approaches. It is unusual for companies to achieve the higher maturity levels of the mentioned models [14] [51], which Afzal *et al.* suggest could be due to the staged approaches.

Due to the lack of suitability for SMEs to adopt these vastly acknowledged models, several models targeted at SMEs have been created. Among others is [9], which proposes an adoption model for software capability maturity. It is based on the aspects of CMM that were discovered to be pertinent for SMEs in developing countries. The proposed model is, however, not a framework to be applied in isolation since it is just a helping tool for applying CMM. The argument that CMM requires many resources is, therefore, still prominent. However, they concluded several findings about essential factors to consider when applying a SPI. In the background study, it is highlighted that within small organizations, it is common for employees to exhibit a high degree of self-trust and to place significant emphasis on personal responsibility, sometimes to the extent of feeling exempt from adhering to established rules. This is a mentality that hinders the adoption of frameworks. Lastly, the study also highlights that organizational culture and recognition are important factors to consider when applying SPI strategies.

Karlström *et al.* [49] proposes a minimal testing framework, which was concluded to be useful to small and emerging organizations at the beginning of their test process improvement. It is composed of five categories, each with three-level phases. The key areas for the testing framework are Organisation, Planning and Tracking, Test Cases, Testware, and Reviews. They conclude that most of the problems related to software testing are not technical but instead of an organizational nature. Each level of the categories is based on the size of the company. This implies that to reach the next maturity level, the company needs to grow in size. This hinders companies that do not aim to recruit to use this model to improve their testing process.

Hasan *et al.* [53] identified challenges SMEs encounter when practicing CMMI to improve their testing of service-related processes. They tried to address the challenges by suggesting a framework that pinpoints the critical practice areas and how to improve them. They formed the framework iteratively by suggesting a review after each phase to evaluate the progress instead of the level format that is common among other previously mentioned frameworks. This study aims to facilitate the adaption of CMMI and does not extend the process with any new factors. The lack of crucial aspects, such as human factors in CMMI, consequentially makes this an incomplete framework.

Taipale and Smolander [13] suggest several factors significant for improving a software testing process. Among others, they highlight that it is encouraged to include testability from an early stage by considering it already in the selection of components of the product. In this way, testing costs can decrease, and efficiency has been proven to increase. Another proposition they suggest is to use risk-based testing, which follows the ISO/IEC/IEEE standard [12]. The argument favoring this testing technique is that it works as a solution to tight schedules since it allows minimiz-

ing the test suite by risk priority. Their case study showed that risk-based testing helped avoid ad hoc decisions regarding testing since a predefined testing strategy was in place. However, it can be criticized that this result would probably arise even if other testing techniques were implemented interchangeably since the argument is not based on any of the characteristics of risk-based testing itself but on the nature of implementing a testing technique in general.
All of the mentioned frameworks and improvement suggestions have drawbacks or reasons for not being optimal in this study's scope. They fill in each other by bringing new propositions to the table. However, none is superior, covering all important aspects of enhancing a software testing process. In the following subsection, the chosen framework will be presented. This is not optimal either; however, it is judged to be the most appropriate foundation for this study.

3.2 Proposal to Improve Software Testing in Small and Medium Enterprises

The paper "Proposal to Improve Software Testing in Small and Medium Enterprises" [1] proposes a testing framework focusing on SMEs. This framework offers a structured approach to enhance software testing, characterized by its simplicity and adaptability, making it particularly suitable for SME environments. The framework is structured around a series of axes, each representing a critical aspect of the testing process: Plan definition, Affected Test Levels, Definition of Test Cases, Test Activities, Traceability, Test Environment, Testing Tools, and Feedback. Each axis is evaluated at three levels of implementation: No Implementation, Partial Implementation, and Full Implementation. This tiered structure allows for a clear assessment of the current state.
The **Plan Definition** axis aims to establish clear testing methodologies within the organizational constraints. The implementation level in this axis ranges from no plan, through informal planning, to a formal, well-documented plan that is communicated across stakeholders.
The **Affected Test Levels** axis aims to ensure comprehensive testing at all necessary levels; unit, integration, systems, and acceptance testing, to detect and correct defects early.
The **Definition of Test Cases** axis defines "how to test" through designing, prioritizing, and specifying test cases. It ranges from no test cases to high-level test cases, to fully specified test cases with detailed inputs and expected outcomes.
The **Test Activities** axis encompasses static and dynamic testing, and its level of evolution is based on the ability to perform both types of tests.
The **Traceability** axis aims to maintain connection across the testing process for coverage analysis, impact, and assessment. The level of traceability between the test basis and the work products determines the evolution of this axis.
The **Test Environment** axis aims to reach an environment that mimics the production environment as closely as possible to ensure relevant and effective testing. It ranges from non-specific environments to fully representative test environments.
The **Testing Tools** axis aims to leverage tools to enhance testing efficiency, consis-

tency, and reliability. It ranges from basic tools to comprehensive, integrated tool sets, supporting the entire testing process.
The **Feedback** axis aims to reach a point where the collection and utilization of data from testing activities continuously contributes to an improved process.
The axes outlined in the framework serve two fundamental objectives: diagnostic and directional, forming a comprehensive strategy for assessing and guiding software testing improvements. The diagnostic capability is crucial for TMA. A diagnosis matrix is provided to facilitate the analysis of the level of evolution for each of the defined axes. The matrix can be seen in Table 3.1. On the other hand, the directional objective of the axis guides the necessary improvements by specifying the conditions needed to progress to the next level. Table 3.2 and 3.3 show evolution matrices indicating the necessary activities for evolving to the next level of each axis.

Axis	**No Implementation**	**Partial Implementation**	**Full Implementation**
Plan Definition	No plan	Informal plan	Formal Plan
Affected Test Levels	No testing	Unit and system testing	Unit tests, Integration tests and higher levels
Definition of Test Cases	No test cases	High-level test cases	Fully specified test cases
Test Activities	No testing	Static or dynamic tests	Static and dynamic tests combined
Traceability	No traceability	Partial traceability	Full traceability
Test Environment	No specific testing environment	Managed and controlled environment	Testing in a suitable environment
Testing Tools	Non-specific tools	Specific tools for monitoring and executing tests	Extensive process automation
Feedback	No reports	Internal defect reports	Progress reports, activities and priority defects

Table 3.1: Axis diagnostic matrix proposed by Argüello *et al.* [1].

Axis	**Improvement activities**
Plan Definition	What do we test? What did we not test? And why do we do it? Define the general approach to testing
Affected Test Levels	Analyze the available objects from the unit test level and system test level test. Identify the features to be tested from the previously analyzed test objects, as well as the conditions under which it will be tested
Definition of Test Cases	Create high-level test cases. A checklist is often recommended as a first approximation
Test Activities	Perform static review tests, e.g., code reviews or inspections, walkthroughs, algorithm analysis, and functional documentation reviews. Perform exploratory dynamic tests manually
Traceability	Maintain traceability between test objects and test cases
Test Environment	Build a test environment outside of the development and production environment. Prepare it with test data, where possible, that is similar to production data
Testing Tools	Incorporate requirements and incident monitoring tools. As a first approximation, they can be carried in a spreadsheet, but it is advisable to use specific tools. Incorporate test execution tools that allow them to be automated
Feedback	Generate reports of internal defects. Communicate the quality level of the product under test. Keep a count of the test cases executed, the result obtained and the defects reported once the derived adjustments have been made. Estimate the resources / hours used to carry out this task

Table 3.2: Evolution matrix from No Implementation level, proposed by Argüello *et al.* [1].

Axis	**Improvement activities**
Plan definition	Document the plan and distribute it to the development team and stakeholders. Integrate and coordinate testing activities with the development lifecycle. Estimate people and resources to carry out the planned activities. Generate a calendar with specific dates or in the context of each iteration. Make an estimated time budget required to carry out the test activities
Affected test levels	Analyze the available integration test level objects and higher level tests such as acceptance tests. Study the need for tests at specialized levels such as Installation tests. Identify the features to be tested in the different levels. Define and prioritize the conditions for each level while maintaining traceability with the previous objects
Definition of test cases	Generate test cases identifying: the possible input data, the execution conditions and the expected results. Prioritize them based on the features to be tested Specify and record them for follow-up
Test activities	Perform static analysis tests guided by tools such as source code analysis, model checking and business process simulation. Automate dynamic tests
Traceability	Maintain traceability between requirements, test objects and test cases, so that it is possible to determine the coverage of the tests
Test environment	Include in the test environment: test harness, service virtualization, simulators and other necessary infrastructure elements Prepare test data from production data and keep it up to date. The result should be an environment as similar to the productive environment within the possibilities and restrictions of the business
Testing tools	Incorporate tools for test management and test products Incorporate test design and implementation tools, including test cases, test procedures, and test data. Incorporate tools for the execution and registration of tests Incorporate tools for performance measurement and dynamic analysis. Incorporate tools for specialized testing needs. For example: security tests, portability, and accessibility and others
Feedback	Generate progress reports to evaluate the evolution of the tests and track associated activities Prepare defect reports to communicate the testing process and product quality. Allow interested parties to provide feedback about the quality achieved and priorities for future releases

Table 3.3: Evolution matrix from Partial Implementation level, proposed by Argüello *et al.* [1].

The framework recommends a practical approach for its application, consisting of Diagnosis, detection of improvement points, implementation of improvement activities, and verification of results. The process is illustrated in Figure 3.1. During the diagnosis, the current state of the organization's testing process must be established, by deciding the current level for each axis. Once the diagnosis has been made, the axes that need to be improved should be detected, which are those that are not at the highest level. It is recommended to start with the less developed ones in the order they appear in the tables. The process continues with the implementation of the improvement activities. Finally, the results are verified, and the degree of evolution of the axis is re-measured using the diagnosis matrix again.

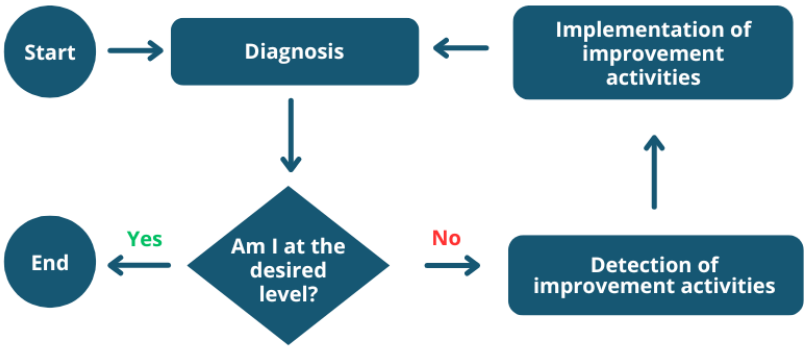

Figure 3.1: Illustration of the framework application process. The figure summarizes the application process figure presented in the framework [1].

The framework is praised for its simplicity and practicality, which makes it adaptable to various development models. However, multiple suggestions for improvement have been made. Integration testing and metrics such as code coverage are suggested to be added. Human aspects are not considered in the framework at all. Additionally, the framework has not been tested in a real-world context, which could allow for adaptations and refinement.

The decision to base this study on this framework was guided by its promising evaluation results, remarkable ease of implementation, and clarity of axes and evaluation levels. The framework's structured yet flexible approach makes it an excellent candidate for addressing the nuanced demands of software testing in SMEs. Furthermore, the fact that the framework has not been verified in a real-world context provided a clear opportunity for contributing to the field by addressing it in this study. The company selected for implementing the framework presents an ideal case study due to its size and the current state of its testing environment. The company's profile aligns with the framework's intended audience, offering a unique opportunity to observe its practical application and effectiveness in real-world settings. The fact that there are clear areas of improvement and gaps to be filled reinforces our choice, intending to develop a more complete testing framework.

3.3 Human Aspects related to Testing

In the paper "Behavioral software engineering: A definition and systematic literature review" [32], over 10,000 publications were screened, and a final set of 250 was chosen for further analysis. The results from this paper indicated that the existing research in BSE has focused on only a few of the Software engineering knowledge areas, with only 1% of the selected papers exploring human aspects in the context of Software testing. This result highlights how BSE research is heavily unbalanced.

Even though software testing as a human activity tends to be under-researched relative to technical aspects [56], there is some relevant work regarding the role of motivation. The paper "Challenges and strategies for motivating software testing personnel" [11] aimed to cast light on how professional software testers can be motivated. The research investigated the strategies companies use to stimulate their testers while considering motivational and demotivational factors. The results show that combining testing responsibilities with development and ensuring various engaging, challenging tasks increases motivation. On the same subject, Straubinger and Fraser [3] conducted a comprehensive survey to assess the factors influencing developer's inclination towards software testing. The survey received 284 responses, uncovering that the reasons for motivation to write tests encompass not only a general pursuit of software quality but also personal satisfaction. Providing better recognition for developers' testing efforts also emerged as a factor in increasing motivation to test.

Another human factor that has been researched in relation to software testing is cognitive biases. The paper "Cognitive biases in Software Quality and Testing" [57] aimed to explore this phenomenon in detail by investigating the factors in an organizational context triggering cognitive biases, ultimately affecting software quality. Which of the cognitive biases occur in software quality and testing was also one of the research questions in this paper. The ultimate goal was to help identify how software quality and testing could be improved by considering a human-centric approach.

Most existing research on cognitive biases has focused on confirmation bias, how it occurs and affects software testing [57]. The paper "Analyses of factors related to positive test bias in software testing" [58] conducted two parallel studies on confirmation bias and testing. The first study considered how expertise level, completeness of the software specification, and the existence of errors in the program could influence confirmation bias. The results showed strong evidence of confirmation bias regardless of the conditions above. On the other hand, the effects of confirmation bias appeared to be mitigated by increasingly higher levels of expertise and more complete specifications. The second study explored the kind of hypothesis testers generate during testing. The results emphasized the existence of confirmation bias and supported the notion that program specification drives the tester's hypothesis. It was concluded that confirmation bias is a critical concern in software testing and may have a seriously detrimental effect on the quality of testing. Additionally, the importance of complete program specification to enhance effective testing was highlighted. Another study on the same subject is "Empirical Analyses of the Fac-

tors Affecting Confirmation Bias and the Effects of Confirmation Bias on Software Developer/Tester Performance" [59]. This study analyzed factors affecting confirmation bias to discover methods to circumvent it. The results of investigating the relationship between code defect density and confirmation bias levels of software developers and testers showed a direct correlation between confirmation bias and defect proneness of the code. Other studies, such as Calikli *et al.* [60], also mention the possible impact of representativeness, anchoring, and adjustment biases in the software testing process.
Addressing the need to study the human aspects of software testing, Feldt *et al.* [61] touched on the cognitive aspects of software testing by investigating testing practices from a problem-solving approach. This research focused on developing a model of the tester's cognitive processes in order to understand the mechanisms by which human testers choose, design, implement, and evaluate test cases. The research experimented with five human subjects, resulting in a cyclical software testing model consisting of eight stages: *Identify the Test Goal, Define the test Goal, Analyze Knowledge, Form Strategy, Organize information, Allocate Resources, Monitor Progress, and Evaluate.* The proposed model provides a framework to investigate the tester's knowledge and expertise and serves as a reference for future investigation concerning the tester's cognitive processes. Furthermore, the model is argued to potentially enhance the test design process and contribute to better alignment of testing tools with the cognitive needs of testers. This research was built upon in the paper "Understanding Problem Solving in Software Testing: An Exploration of Tester Routines and Behavior" [62]. The last mentioned paper aimed to better understand the routine and behavior of human testers and their mental models when performing testing. The focus was put on understanding the personal decision-making processes of testers. The investigation was conducted by surveying 38 software testers and developers in Sweden. The survey results describe tester practices and identification of insights regarding testing as a problem-solving process. These results were utilized to further enhance the cognitive model of software testing. The enhanced model argues for enabling clear guidance on performing effective and efficient testing, potentially leading to the development of more human-like test-generation tools.
Although research on human aspects in the area of Software testing is limited, there is still enough evidence that human factors affect the testing process.

4
Methods

The study is a solution-seeking research whose primary goal is to develop a more complete testing framework for SMEs. To achieve this objective, the study was designed mainly as a Field study, but it also included elements of Action Research. Combining different research methodologies facilitated the investigation of the topic through the triangulation of strategies. This approach can help overcome the inherent limitations of the selected methodologies [63]. The participating company served as a case study for data collection and implementation and evaluation of the proposed framework. This study can be classified as cross-sectional since the resulting framework was evaluated at a single point in time.
This study's philosophical stance combines Pragmatism and Applied Research. Creating a software testing framework based on literature and company observation suggests a pragmatic approach, as the goal is to develop a practical solution that addresses specific needs and challenges in software testing. Applied Research is involved since the study aims to apply the theoretical outcome (the testing framework) in a real context.
The method of the study is illustrated in Figure 4.1. It was separated into five distinct phases, each explained in depth in the sections of this chapter. The design of the methodology guarantees research data from both literature and the case study, providing a justified and realistic depiction of the software testing process of SMEs today. Figure 4.2 illustrates the data collection process. First, a Focused Literature Review was conducted to examine existing research in the area. Then, the testing process of the participating company was analyzed. The analysis phase was designed to be conducted as a Field study since the main goal was to understand the process and challenges faced by the participatory company. The analysis was performed in a real-world context without any deliberate modifications to the research setting. Two different approaches were utilized to mitigate mono-method bias. The approaches used were interviews and software repository mining. The findings of the analysis, together with findings from the literature, served as the foundation for enhancing the chosen testing framework. To verify the results, a workshop session with practitioners from the company was conducted. The framework was then further refined with the feedback given from the workshop.

Figure 4.1: Illustration of the applied methodology in the study.

To evaluate the resulting framework, the Action Research methodology proposed by Avison *et al.* [64] was adopted. The extended framework was applied in a natural setting at the participating company, and the results from the implementation were evaluated. During the implementation phase, a developer at the company was asked to conduct testing following the recommendations given by the framework. The goal was to test the proposed framework in the natural setting by providing recommendations for software testing improvement. This constituted manipulation of the research setting in order to observe the effects of the introduced changes. The results of the changes were evaluated to get feedback from this experience, and the framework was further refined based on the observations and feedback.
Before covering the details of each research phase in the corresponding order they were performed, the participating company is presented in the first upcoming section to provide the industrial context.

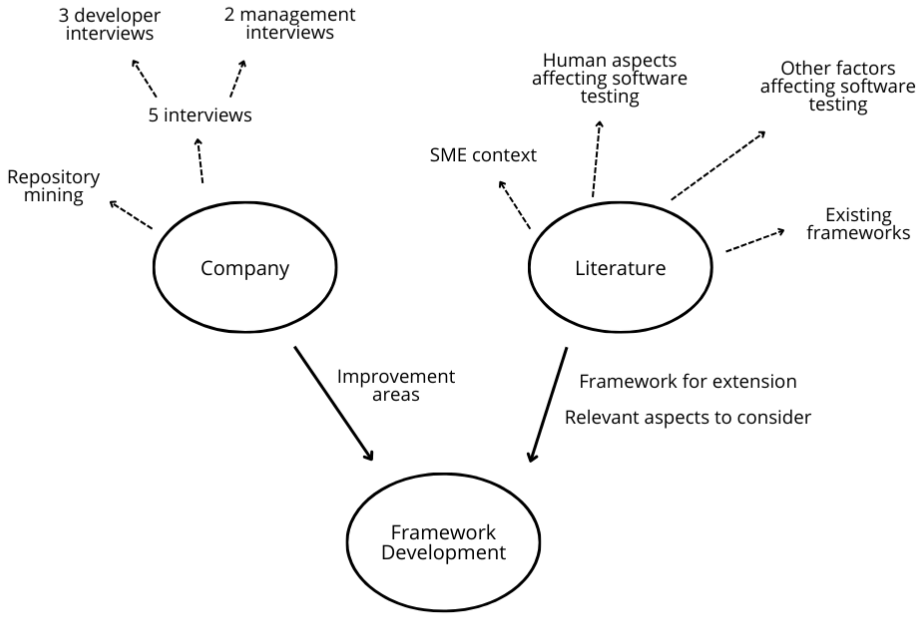

Figure 4.2: Illustration of the data collection process.

4.1 Industrial Context

The company is a Swedish SaaS (Software as a service) company that has been active for around 16 years. It is categorized as a small company with 40 employees. The specific company was selected based on non-probability sampling. The opposite, probability sampling, was infeasible since it requires a proper sampling frame, which was not accessible in this study. A proper sampling frame, in this case, could be a complete list of SMEs in Sweden that produce software. Instead, the sampling technique was a combination of purposive and convenience sampling. Baltes and Ralph [65] explain both sampling methods. Purposive sampling refers to when items are carefully selected based on some criteria. In this case, the criteria were that the company needed to be an SME, possess an immature testing process, be willing to participate in the study, be committed to the resources required, and need to be easily contacted during the process. Convenience sampling refers to when items are selected based on availability. In this case, the study's authors already had contacts in the company which made it easily accessed. The fact that the company is located in the same city where the study was conducted also facilitated the decision.
The participating company encompasses a flat organizational hierarchy. The developers collectively work towards shared objectives. The company does not have separate departments for the different stages in the software development lifecycle, such as requirement engineering, development, and testing. This is directly raised as a challenge to apply CMM [9].
The testing processes at the company have broadly remained unchanged since the company's foundation. The company's instituted testing process consists of unit, integration, and system-level tests. They practice both manual and automated testing, as well as utilize both static and dynamic testing. The company uses Git for version control and GitHub to host the repository. Git, as a version control tool, was adopted around a yer ago. The existing automatic tests are integrated in the CI pipeline through TeamCity [31], upon committing to the main branch in Git. A selection of these tests is also run by GitHub actions, triggered when a PR is made. Even though this might give the impression of a sound testing process, the components are vaguely established. The written tests in the code are few, and the decisions around the test environment and approach lack motivation. The testing techniques are not fully implemented, and the test levels are not fully covered. The high potential for improvement and the company's rapid growth incentivized the company to participate in this study.

4.2 Focused Literature Review

The Focused Literature Review was divided into three separate areas, targeting the research questions of this study from different views. The different objectives of the areas were:

1. Examine existing software testing frameworks and choose one for extension (RQ1)
2. Uncover which human aspects affect software testing (RQ2)
3. Identify other factors, mainly technical, that affect the process of software testing (RQ1)

Which framework that was chosen for extension was based on how well it covered relevant aspects for software testing, together with how recently the framework was proposed to allow advantages of new technology. Additionally, how easy the framework was to follow and understand was an important factor since this is important for frameworks to be used in SMEs [50].
The literature review was conducted using snowball sampling. It means to use the reference list of a paper (*backward* snowballing) or the citations to it (*forward* snowballing) to uncover relevant research [66]. Snowballing can be a good alternative to the use of database searches [66]. Snowballing was employed in this study to achieve depth in particular research areas. Further, snowballing reduces the risk of publication bias compared to database searches, where public and indexed papers are given priority. However, a risk of snowball sampling is that it can lead to sampling in a small, highly interconnected subset of publications [65]. To mitigate this risk, the snowball sampling started from multiple points within the relevant publications' population. Table 4.1 lists the starting points for each of the three areas.
The literature review process was limited by approximately two weeks, during which both authors worked full-time. This timeline was supplemented by the author's assessment that the field had been sufficiently explored to a reasonable depth that provided the most essential and contemporary insights.

4.3 Analysis at the Company

The analysis at the company included two primary activities: 1) performing a Test Maturity Assessment to identify areas of improvement in the company's test process and 2) gathering data for guidelines in the final framework. Both activities target both RQ1 and RQ2, and the data was also fundamental in answering RQ3, as it was later used to compare the impact of the applied framework. The data collection was composed of different methods that varied in degree of required human intervention. The choice of methods was based on Lethbridge et al.'s proposed taxonomy [67] since it includes threats to validity for the different methods. For example, they explain that a direct data collection approach in a social setting can make the respondents modify their behavior in response to the awareness of being observed or studied. Therefore, this study also examined code to ensure the artifact mirrored the same situation as the humans indicated.

The analysis began with semi-structured interviews. After the interviews, software repository mining was performed. This included analyzing data from GitHub and independent analysis of artifacts. Each of the three methods is explained fully in the following sections.

4.3.1 Semi-structured Interviews

Five semi-structured interviews were conducted at the company. Semi-structured interviews comprise both closed- and open-ended questions [68]. The nature of semi-structured interviews allows the interviewer to ask follow-up questions to explore the answer fully. The participants for these interviews were selected based on their role and experience in the company, to be able to delve deeply into the various experiences and perspectives, providing a comprehensive understanding of the testing process from multiple viewpoints. Purposive sampling was used to guarantee that participants knew the area of interest. Klein and Myers claim that this sampling is appropriate when handling a small population [69]. Each interview was recorded, with permission from the participants, to be able to be transcripted for further analysis. It was clarified that the interviews were anonymous so the participants would feel safe to express their opinions. Both authors of this study participated in the sessions. One interviewed, and the other took notes. However, both could deviate from the prepared interview questions and interpolate new related questions if such arose.

With inspiration from a master thesis study at Lund University that analyzed technical debt at a company [70], both informative and in-depth interviews were conducted to get a rich understanding of the working environment at the company under analysis. Informative interviews were held with three developers engaged in daily operations and who had various working experiences. Appendix A contains the questions prepared for these interviews. The purpose of these interviews was to understand how everyday operations involve testing and how testing is experienced by developers at the company. The aim was also to understand how the company's attitude towards testing propagates to the developers. The in-depth interviews were held in two different settings. One with personnel responsible for the current established test process and one with the Head of the Software Development Department. The in-depth interviews aimed to delve deeper into technical aspects and decisions regarding the testing process. The questions prepared for these interviews can be found in Appendix B and Appendix C.

All interview questions were produced by first establishing the purpose of the interview. Then, the key topics to be covered were established, and finally, concrete questions were created within these topics. The questions were intentionally formulated in an open-ended way, encouraging elaborating answers. Before the interviews were performed, the prepared questions were evaluated through pilot testing. Teijlingen and Hundley [71] describe pilot testing as a smaller version of a specific study, which serves as a preparatory test of the research instrument. This was done to ensure the interview flow and that the questions were communicated clearly and logically. Additionally, this created an opportunity to practice interview skills as well as estimate the interview time. The pilot testing mainly showed that some

technical questions needed further explanations to allow the participant to answer the question even though they might not have the theoretical knowledge. The pilot testing also resulted in a few reformulations of some of the interview questions.

4.3.2 Themtic Analysis

All five semi-structured interviews were compiled using Thematic Analysis. This method systematically analyzes qualitative data by defining themes (recurring concepts) in the dataset. The process was done according to Braun and Clarke's description of a six-phase thematic analysis found in [72]. However, the final phase, "Phase 6: Producing the report," was omitted since a thematic analysis report was not considered valuable for this study. The goal was rather to define themes, which is accomplished in phase 5, and that was thereby the ending phase of this study. The applied phases are:

- Phase 1: Familiarizing yourself with the data
- Phase 2: Generating initial codes
- Phase 3: Searching for themes
- Phase 4: Reviewing potential themes
- Phase 5: Defining and naming themes

First, both authors of this study independently familiarized themselves with the data. This was done by reading notes from the interviews, generating transcriptions, and listening to the audio records. Next, both authors analyzed each interview and created codes for the data content, still independently. The codes, which also could be described as labels or tags, were inductively created as they were found in the data. However, as Braun and Clarke mention, it is difficult to not bring any preassumptions to the analysis. Therefore, it was a suitable approach for both authors to analyze all interviews to sort out the inconveniences. A combination of both a latent and semantic approach was used, where the codes were allowed to be both interpretations of the data and purely descriptive. The authors joined for sessions where all codes were merged into a large code set representing a summary of the subjects discussed during all interviews. It was verified that the interview outcome reached a saturation point. If this were not true, more interviews would be held to uncover important information. The merging process resulted in creating a table comprising compiled codes, omitting any duplicate or redundant codes. Continuing, the codes were examined to classify them into themes. Finally, the themes were carefully defined, and a summary of each team and the codes within it was created.

4.3.3 Software Repository Mining

The software repository mining was the first interaction with the code base under analysis in this study. The aim was to provide empirical insights into the current testing process quality and efficiency. Due to the large size of the code repository and the time resources for this study, only a module of the code was selected for analysis. The selection of the module was based on how instantiable it was, since it was planned that the module would be compared to a similar one that would be created with adherence to the proposed framework. The module was also chosen

based on which employee wanted to participate in this study and what part of the code he worked with. This was because the person later applying the framework should provide an evaluation of it.
The focus of the repository mining was on a set of selected metrics. The metrics were selected based on the ability to correspond to the aspects addressed in the selected testing framework and on previous knowledge about assessing software quality and testing of the authors. The interviews provided insights into what aspects of the code contained problems, and some metrics were selected to investigate the mentioned problems. The chosen metrics are; Code Churn vs Test Churn, Number of Crashes in TeamCity, Code Coverage, Flakiness Score, Number of Requested Changes/Comments in Code Review, Documentation of Tests/Code, and a broad set of Code Quality Issues. How each metric was elicited from the code base will now be presented.
The **Code Churn** was determined by analyzing the number of lines added and deleted in the relevant PRs for the selected module. Regarding the **Test Churn**, prior information was received that no tests had been written for the module. The **Number of Crashes in TeamCity** had previously been measured by an internal report at the company. The report was written in January 2023 and analyzed the last 20 failures that caused the continuous integration pipeline to crash. There have not been any significant changes in the testing process since this report was written, which makes it still relevant. **Code Coverage** was manually decided since no tests were written for the module. The **Flakiness Score** was measured by running the existing tests locally and comparing the results from multiple runs to observe if the test failed/passed arbitrarily. The tests were run six times. The **Number of Requested Changes/Comments in the Code Review** was decided by examining the pull request reviews for the module. The **Documentation of Tests and Code** was found by counting the comments in all files in the module. Since the module had no tests, no tests were examined. **Code Quality Issues** were assessed using the Visual Studio extension Resharper [73]. This tool was chosen based on managerial input from the company regarding resource limitations. To highlight the variety of the code quality in the whole code base, another module was also analyzed through Resharper. This other component was chosen based on suggestions from developers at the company who were asked to name a class of lower code quality in the code base. This was done since the primary chosen module was not assessed as representative of the code base quality because the developer of the chosen module has a higher personal interest in code quality than the general perception of the developers at the company.

4.4 Framework Development

Once the literature review and the analysis at the company were performed, the chosen framework was revised based on the gained insights. First, the original framework was carefully examined, with the ambition to extract components found to be unnecessary and to identify absence of vital components. Both authors of this study separately analyzed the framework to later discuss the findings to reach a common ground. Some parts were directly removed, some were reformulated to

achieve the desired clarity, and some extensions were proposed. The framework axes, both the old and the newly proposed ones, were split up among the authors to formulate the axis explanation as well as the improvement activities. Then, peer review was conducted between the authors on all parts until the framework reached a state where both authors found it complete. During the analysis at the company, it was noted that fundamental testing knowledge is a crucial factor for the entire testing process, and the testing knowledge at this company was fairly vague. Therefore, it was decided to supplement the proposed framework with a theory file to support practitioners with explanations of fundamental concepts needed to be able to apply the framework.

To improve the framework, a workshop session was performed with ten developers at the company. The participants were chosen solely based on the availability of the employees at the company and their willingness to participate. The number of participants was ten since this number ensures a diversity of perspectives while still allowing for meaningful discussions. A workshop setting was chosen to encourage conversations about the suggested framework and gather qualitative data. This was a time-effective method to get a rich understanding of the participant's beliefs about the subject.

A few days before the workshop, the framework was sent to the participants, with the exhortation to read it before the session. The session began with a short recap of the framework, as a reminder, but also to ensure a common baseline for all participants. It was emphasized that the presented framework was a first version that should be refined, hopefully reducing the social bias of not wanting to criticize the work in front of the authors, and instead encouraging collaboration by providing feedback. Then, the participants were divided into groups of two or three and asked to freely discuss and revise three assigned axes in the framework. Physical copies of the framework were provided together with creative tools such as markers and post-it papers to encourage a creative environment where ideas and feedback could be expressed. After this, the room was gathered again, and a monitored discussion began. All axes of the framework were discussed in subsequent order. The assigned group expressed their revision, and the rest of the participants were encouraged to express their opinions. Once all axes were thoroughly discussed, general opinions about the framework were discussed with the two leading questions: "What was most challenging to understand in the framework?" respectively, "Did you find something missing in the framework?". The discussions were documented by one author taking notes.

Ending the workshop session, an online questionnaire was distributed. This was done to collect quantitative data mirroring the opinions about the framework in a way where all answers were given the same weight. A combination of qualitative and quantitative data is generally suggested in the literature, among others is [49]. Using qualitative and quantitative data can help reach a true insight into the opinions since the different methods can mitigate each other's drawbacks, reducing mono-method bias. For example, in a workshop session, it can be the case that one person takes more space in the discussion, causing the discussion to circle his specific opinions, whereas a questionnaire gives objective numbers.

The questionnaire included seven questions to evaluate the completeness of the

framework. The answers were in Likert scale format, with the options "Strongly agree," "Agree," "Not sure," "Disagree," and "Strongly disagree." The questionnaire ended with a section where the respondents were asked to practice the prioritization technique of Cumulative Voting [74]. The respondents were asked to distribute 100 dollars over the twelve axes, mirroring the importance of and desire for each axis. This was included to find out if some axes were assessed as unnecessary.

4.5 Framework Application

Having the final proposed framework outlined, the company's current testing process could be diagnosed to determine which axes were targets for improvements. Due to the time constraints, all axes could not be addressed in the framework application of this study. Which to include was decided upon discussion with the supervisor at the company. All axes were treated as candidates for application, regardless of whether they were added by this study or created in the original framework. This was because none of them had been tested in a natural environment, making all axes interesting to evaluate. Notably, many axes include concepts that are not feasible to measure progress within a few weeks, which was the timeframe for this phase. The axes selected for the framework application phase are presented in the result chapter, Section 7.2.

The framework application was composed of two parts: one where a developer interacted with the practical suggestions of the framework and one where the management group was presented with suggestions for improvement concerning organizational aspects. The developer was chosen based on availability and willingness to participate in the study.

To begin with, the developer was asked to put parts of the framework into practice by testing a chosen code module. Previously in this study, it had been planned to have the developer apply the framework while developing a new feature, similar to the one analyzed during the repository mining. This ended up not being possible since there was no comparable module to be developed during the period reserved for the framework application. As an alternative, an existing module lacking in testing efforts was chosen to be tested. The developer was handed a framework application document describing what concrete changes he needed to encompass. He was further encouraged to ask questions if any ambiguity arose. Additionally, another developer was invited to perform a code review on the produced tests. He was given instructions from the framework to use when reviewing the artifacts.

For the other application part, concerning the organizational aspect, the authors of this study compiled customized suggestions based on the framework's improvement activities that could not be tested by a single developer solely. The suggestions were presented to the management team in a meeting. Some customized suggestions were also implemented on a smaller scale to demonstrate the possible outcome.

4.6 Framework Evaluation

After implementing or presenting the improvement suggestions, an assessment of the result was conducted in order to answer RQ3. This phase included evaluating both application methods; the outcome generated by the developer who practically applied portions of the framework, and the response to the recommendations handed to the management team. The framework evaluation phase further resulted in refinements of the framework. The methodology used for the evaluation of both of the application processes is explained in the following sections.

4.6.1 Practical Framework Evaluation

A semi-structured interview was conducted with the developer who applied parts of the framework. The choice of a semi-structured interview was deliberate, given its great flexibility, allowing for follow-up questions [68]. This approach enables an in-depth exploration of the responses, thereby gathering rich, qualitative insights into the developer's experience. The evaluation aimed to assess the developer's opinions and experiences to understand the feasibility and practicality of the framework's recommendations. With the participant's consent, the interview was recorded to ensure accurate responses were captured. Anonymity was guaranteed to encourage openness and honesty in the feedback provided. The interview process involved both authors of the study. One author led the conversation and posed the questions, while the other took detailed notes.
The preparation of the interview questions mirrored the rigorous methodology applied in the analysis phase at the company, found in Section 4.3.1. The questions asked during the interview can be found in Appendix D.
To complement the subjective experience results from the interview, the artifacts produced were analyzed and evaluated. The primary goal was to investigate how the artifacts mirrored the recommendations in the framework application document and how well the developer's perception matched the outcome. The guidelines in the framework application document served as evaluation criteria against which to evaluate the artifacts. The following aspects were analyzed:

- Adherence to requirements
- Correctness of chosen test levels
- Completeness of high-level test cases
- Equivalence class partitioning and Boundary-value analysis
- Adherence to exit criteria

4.6.2 Evaluation with Management

One week before the meeting where the recommendations were presented to the management team, the recommendations were sent to the participants to encourage reading them ahead to facilitate their understanding. The meeting format was a combination of a presentation and a semi-structured interview. First, a recommendation was presented, and then an open discussion was encouraged, with some bullet point questions presented to inspire the discussion. The supplementary questions

can be found in Appendix E. The participants were asked to provide feedback on the suggestions as if they were applicable and adequate. This procedure continued until all recommendations were covered. Both authors participated during the meeting, taking turns in presenting recommendations and leading the discussion. The meeting was recorded, and the compiled feedback is presented in Section 7.3.3.

4.6.3 Software Repository Mining after Framework Application

Once the framework had been applied, the module under test was examined using the predefined metrics. This was done to be able to compare the previously measured metrics to investigate if the framework application left quantitative results. The predefined metrics were; Code Churn vs Test Churn, Number of Crashes in TeamCity, Code Coverage, Flakiness Score, Number of Requested Changes/Comments in Code Review, Documentation of Tests/Code, and a broad set of Code Quality Issues. They were elicited in the same way as described in Section 4.3.3. However, four metrics were omitted due to the framework application process.
Code Churn vs Test Churn was not measured due to the change in the framework application methodology, where the developer were asked to test a module instead of developing a new one. With this new approach, the metric was no longer valuable since the authors intruded to much by asking for tests rather than observing how the tests are maintained during development. The Number of Crashes in TeamCity and Flakiness Score were not measured, since no actions were performed to improve these metrics. The framework suggests activities to address them, but those were decided not to be applied in the partial framework application at the company. This was decided simply because the time did not allow all framework activities to be performed, and the supervisor at the company did not prioritize these aspects. Code Coverage was neither measured since the authors of this study intruded too much on the framework application for this metric to provide any valuable insight. The authors asked the developer to test specific methods of the selected module, so the developer was never asked to test exhaustively.

Category	Article Title	Authors	Year
Software testing frameworks	"A Minimal Test Practice Framework for Emerging Software Organizations"	D. Karlström and P. Runeson and S. Nordén	2005
	"Proposal to Improve Software Testing in Small and Medium Enterprises"	M. Argüello and C. A. C. Pietroboni and K. E. Cedaro	2021
	"A Unified Testing Process Framework for Small and Medium-Sized Tech Enterprises with Incorporation of CMMI-SVC to Improve Maturity of Software Testing Process"	M. Hasan and others	2022
Human aspects affecting software testing	"Cognitive biases in software quality and testing"	I. Salman	2016
	"A Survey on What Developers Think About Testing"	P. Straubinger and G. Fraser	2023
	"Cognitive biases in Software Engineering: A systematic Mapping Study"	R. Mohanani and I. Salman and B. Turhan and P. Rodriguez and P. Ralph	2018
	"Behavioral software engineering: A definition and systematic literature review"	P. Lenberg and R. Feldt and L. G. Wallgren	2015
	"Motivation and Satisfaction of Software Engineers"	C. França and F. Q. B. da Silva and H. Sharp	2020
Other factors affecting software testing	"Software and systems engineering – Software testing – Part 1: General concepts"	ISO/IEC/IEEE	2022
	"Identifying Process Improvement Targets in Test Processes: A Case Study"	T. Toroi and A. Raninen and L. Väätäinen	2013
	"Improving Software Testing by Observing Practice"	O. Taipale and K. Smolander	2006
	"A survey on testing and reuse"	R. Torkar and S. Mankefors-Christiernin	2003

Table 4.1: The starting point papers for the snowballing method, categorized by the three objectives of the literature review.

5

Results: Pre-intervention analysis and adaptations

This chapter covers the results of the activities performed before the testing framework of this study was completed. The chapter begins by describing the company's testing process before applying the framework. The testing process was analyzed through interviews and repository mining. First, the result of the thematic analysis conducted on the interviews is presented. Second, the metrics and insights from the repository mining are showcased. Lastly, the results from the workshop performed at the company before the framework was completed are displayed.

5.1 Thematic Analysis

This section presents the result of the first interview phase of the project. The results are presented in a Thematic Analysis Coding Tree found in Appendix F, Figures F.1 F.2. The identified themes are: Current Testing Environment, Current Testing Process, Fast Development Pace, Guidelines, Organization, Culture, Product, Opinions, and Feelings. Each theme is presented below by first defining the theme and then delivering the findings on the theme. Lastly, the thematic analysis is summarized with some general and concluding findings.

5.1.1 Current Testing Environment

The Current Testing Environment theme aims to provide a holistic overview of the existing testing landscape, serving as a critical snapshot of the technical foundation upon which software testing is built at the company.
One of the main findings derived during the interviews was that the company relies significantly on iterative manual testing, utilizing demonstration systems and customer database copies for carrying out informal System Testing. The demonstration systems are copies of the live product provided to each developer for testing proposes, ensuring that tests accurately reflect how changes behave in a production environment. This approach involves developers actively interacting with the systems through its graphical user interface, simulating expected customer behavior. Even though most testing is done manually, some unit tests have been written for the core functionality. In this product, the core functionality is the fundamental components forming the product skeleton. All systems have the core functionality but are later decorated with customer-specific features. The tests for the core func-

tionality are integrated into the CI/CD pipeline through automated testing. This approach highlights a strategic focus on ensuring the reliability of critical system components. Automation is facilitated through GitHub actions and TeamCity. All tests are executed on TeamCity, while a subset also runs on GitHub upon PRs. The GitHub tests exist to catch bugs early on in the development process, while the tests run on TeamCity should be the final check before reaching production. When an error is caught in TeamCity, the changes in the failed build can not be applied to the systems. This insight points to areas where the testing infrastructure could be improved for better reliability and efficiency.

Some written tests are run against a live test database, providing a dedicated environment for testing activities. However, the fact that the tests make changes to the same database, leading to potential data overriding, raises concerns about the isolation and integrity of test scenarios. The main issue is flaky tests, underscoring a challenge to test reliability. Given this challenge, the subset of tests that run on GitHub were selected based on the tests not being connected to the live test database.

Another critical finding was the absence of maintenance of the test environment, suggesting potential stagnation and degradation of testing capabilities over time. The transition from an old version control system to Git about a year ago was also brought to light as a factor influencing the current testing practices. More concretely, the old version control system did not have any support similar to PRs in GitHub, which caused the company not to adopt PRs when migrating. Lastly, it was noted that the testing process is not noticeable in everyday work.

5.1.2 Current Testing Process

The Current Testing Process relates to the procedural aspects and methodologies of the testing practices currently implemented within the company. This theme focuses on the sequence of actions, standards, and practices that define how testing is conducted, including the planning, execution, and evaluation phases of the testing life cycle.

A prominent finding is the overall low level of knowledge and experience with testing within the company. This gap suggests a need for enhanced training to build testing competencies. Another important finding is that the company has unconsciously adopted risk-based testing by prioritizing testing efforts based on the potential risk of failure and its impact on the system. This prioritization is primarily left to the individual developer and his risk assessment capabilities. Testing is also done with a reactive approach, where testing activities are initiated mainly in response to issues rather than being proactive and preventive. The practice of pushing code directly to the main branch without thoroughly testing is also widely spread at the company, raising concerns about the potential for introducing bugs into the production environment. When bugs are discovered in production, the ability to lock system updates becomes a critical control mechanism, preventing further updates until the issue is resolved. While this practice is crucial for maintaining system stability, it also emphasizes the importance of robust testing prior to deployment to minimize disruption. An informal procedure exists for reporting and managing

bugs, which will be discussed in the next section, given that it was considered a sub-category of this theme.
Another valuable insight gathered from the interviews was the absence of metrics for assessing the efficiency of the testing process. The same is true about testing methodologies, given that no test techniques are consciously applied during the testing process. It was also found that customers are sometimes involved in the testing process by being asked to test and give feedback about the feature under development. Overall, the interviews shed light on several areas of concern within the company's current testing process.

5.1.2.1 Bug Reporting

The Bug Reporting theme sheds light on the practices and challenges surrounding how errors found in the product are documented and tracked within the company. These findings reveal insights into the sources of bug reports, the utilization of support tools, and the visibility of bug impact.
During the interviews, it became clear that bugs are primarily reported by customers, suggesting that detection mechanisms within the development and testing phases only capture some issues before release.
The company utilizes bug reporting support within the Customer Relationship Management (CRM) system. Nevertheless, this system is not used consistently and is mainly left to situations where multiple customers are affected by an error. The CRM system offers the capability to see which systems were affected by a specific bug, which was considered a valuable aid in assessing the scope and impact of the issues.

5.1.3 Product

The theme refers to the company's software product itself. It reveals several key insights that offer a nuanced understanding of the development workflow of the product, the product quality, and product aspects leading to customer satisfaction.
An important finding derived from the interviews is that the product uses a unified code base for all customers, a streamlined approach to product maintenance and updates. It was also brought up that tests written for core functionality are considered critical, underscoring the importance placed on ensuring the stability of the product's foundational elements. While updates and customer-specific features are considered important, maintaining the integrity of the core structure is paramount to the product's overall performance and reliability.
From the managerial side, the product was considered to be of high quality, which is reflected in how it meets customers' expectations. Nevertheless, bugs in the live system were considered relatively common, but these bugs are usually not considered critical.
The balance between development speed and quality is a critical focus area discussed during several interviews. The company aims to deliver rapid feature changes without compromising product quality, but finding the right balance was considered a challenge. Long-term customer satisfaction and the company's commitment to its

users were key findings from the interviews and were considered the main contributing factors to their competitive advantage. These findings collectively paint a picture of a well-regarded product in terms of quality but not without its challenges.

5.1.3.1 Code Quality

The theme Code Quality focuses on the aspects of the product that pertain to standards, practices, and characteristics defining the code base. Code Quality was categorized as a sub-theme of Product, delving into the specifications of the software's construction, best practices, readability, maintainability, and scalability.
A predominant issue identified during interviews was the unstructured nature of the code, leading to difficulties in understanding the code base's flow and logic. While some parts of the code base have been developed more recently and follow a clear structure, others were developed several years ago in a more unstructured manner. The code was also considered to lack modularity, which hinders the testing process considerably, given components can not be tested in isolation. This was considered a critical threshold for introducing more testing activities in the company.
The absence of adherence to coding conventions was also brought up as a critical issue impacting readability and maintenance. The code was generally considered hard to comprehend, which was seen as a consequence of the previously mentioned issues.

5.1.4 Fast Development Pace

This theme summarizes a key strategy of the company: there should be a short time between a customer request and delivery.
This theme arose from attendees saying that testing is not done rigorously since that would slow down the process. The interview with the team responsible for the testing process revealed that they had not established PRs as mandatory since they sometimes wanted so fast delivery on urgent feature requests that they did not want to wait for a PR to be approved. One developer highlighted this theme by stating that the rapid pace gives a market advantage against competitors.
This theme was touched upon many times in the interview with the Head of Software Development. For example, they explained that the automated testing in their pipeline must be fast; let the developer do the commit, and be able to move on to the next task rapidly.

5.1.5 Guidelines

The company guidelines regarding the current test process were a key topic discussed during the interviews. The theme Guidelines summarize procedures the company has communicated as their conventional methodology for executing software testing. What was brought up during this topic were mainly uncertainties. First, there is no information regarding the testing process in the onboarding process. This is no surprise since the company has no complete documented testing guidelines. However, PRs together with code reviews were brought up several times as the standard way of implementing changes that affect not only one customer system. This standard is

not always put into practice, rather it is up to the individual to decide when to do a PR and when to push directly to main, which emphasizes personal responsibility. A developer said there is an ongoing discussion about introducing obligatory PRs.
It was also noticed that there is a lack of communicated expectations regarding the testing process. Some interviewees said that everyone should build their code before committing. However, another developer revealed that the repository often crashes due to building errors. This mirrors that the mentioned expectation is either not always followed or not communicated well.

5.1.6 Organization

Organization was elicited as a theme that gathers all aspects related to the corporate structure of testing. It includes the distribution of responsibilities, the decision-making process, and how they manage the developers' work.
To begin with, there is no person responsible for the test process in isolation. Rather, it is considered part of the Head of Software Development's responsibility. However, they do not proactively undertake any action to address this matter. The testing environment was decided at the company start-up, and since then, nothing has changed significantly. The manager said testing has not been prioritized or discussed in recent years. They believe this is because the product works as smoothly as it is. However, they see an advantage in improving the testing process even though it is not seen as an urgent issue.
In general, the company does not encourage its employees to achieve greater knowledge in software testing, and there is no effort to keep the company updated in the field. Additionally, there is a lack of recognition from the managers towards the developers for testing endeavors.

5.1.7 Culture

Company culture in the case of this thematic analysis means the norms, shared values, behaviors, and attitudes that are practiced in the organization. These do not necessarily need to have been internally communicated, rather it is the shared belief of methodologies and behavior that form the working environment at the company.
The company culture at the organization was explained from many angles during the interviews. To begin with, there is a general standard to have high trust in colleagues producing high-quality and working code. This feeds the feeling that testing is not that necessary, and development is given almost all focus. Due to this strong trust, code reuse is practiced often without questioning whether that code works.
Developers are not expected to write tests, but the code pushed to the main branch is expected to work. This argument shows how the company culture relies heavily on personal responsibility, which was frequently raised during the interviews, and was thereby elicited as its own sub-theme.
The vast focus on fast customer delivery also propagates to the company culture and influences the methodologies used. The developing cycle can be seen as stripped

down to only the essential tasks for being able to deliver code. Most often, no formal plan or time estimation is conducted for developing tasks. This may sound like a stressful working environment, but this is not how the developers experience it.
Another vital aspect of the company culture is learning by doing. Interviews with recently employed developers revealed that they emphasize imitating how senior developers carry out their work.
Finally, it was raised in several interviews that there is a common understanding of what parts of the code different developers are allowed to make changes in and not in the company culture.

5.1.7.1 Personal Responsibility

Personal responsibility was defined as a theme since the company culture indicated this as a core aspect of the organization. Personal responsibility in this case means that it is up to each developer to decide what to test, when, and how. This includes that the developer has the freedom of completely neglecting testing. Regarding code reuse, a developer in the interview expressed self-trust to the extent that if they knew they were the authors of the code to be reused, the code is trusted.

5.1.8 Opinions

This theme summarizes the participants' perspectives and judgments regarding the test process expressed during the interviews.
"It works fine as it is" is a quote from an interview that summarizes this theme well. According to the manager interview, it is due to the high quality of the developers. However, it was briefed that bugs reaching the live systems could have been caught earlier. On the other hand, a developer said that there are not so many bugs in production in relation to the commit rate. Of course, this is a subjective number, and the question should consider how severe the bugs are.
From both the developer and managerial sides, it was pointed out that the testing is not optimal. Still, it is a trade-off with the opinion that testing should not result in a slower process for releasing code. However, concerns were raised that the current test process is not scalable as the company grows, and testing should be considered more important.
Lastly, this theme also indicates that the company has room for improvement regarding its testing procedure. Many interviewees responded that they experience testing as difficult, and two even quoted "necessary painful".

5.1.8.1 Suggestions for Improvements

This theme captures a range of recommendations and actionable insights to enhance the current testing process that the interviewees proposed.
A common concern was the speed of the testing process. It was considered a priority to ensure that tests were efficient and quick to run to avoid delays in the development cycle. There was also a clear interest in expanding the depth of testing, including more API tests and a general increase in the number of tests conducted. Regular

test execution, for example, every night, was suggested as a strategy to catch issues early and ensure the code base remains stable.

The suggestion to introduce a formal testing policy or process arose in almost every interview. It was seen as a fundamental change to ensure consistency and quality in testing efforts and potentially reduce the threshold to write tests. The importance of Regression Testing was also emphasized, underlying the need to ensure new updates do not adversely affect existing functionality. Additionally, there was a strong call for improving test consistency and reducing flakiness. In order to ensure a stable testing environment, a suggestion to mock the test database was proposed as a strategy to isolate tests and avoid data override.

Making informed decisions on which test should run on Git vs. TeamCity was suggested in order to leverage the different platforms' strengths. This could involve using Git for early-stage testing and TeamCity for final-stage testing. The ambition would be that TeamCity never fails to build. These suggestions underscore the importance of early detection of issues to prevent failures and deployment delays.

There was also an aspiration to add more steps before changes reached production, such as adding a development branch to move towards a more refined development workflow. This could help manage changes better and ensure quality before merging into the main branch. An ongoing discussion about introducing guidelines for obligatory PRs was also mentioned as a way to ensure code reviews and successful testing before code merges.

A desire for selective testing was also expressed. The concept of not necessarily running all tests but choosing a smart subset was mentioned to help make testing more targeted and efficient, optimally reducing testing time. The suggestions collectively were aimed to refine and enhance the testing process, making it more efficient, reliable, and integrated with the development life cycle.

5.1.9 Feelings

Many emotions related to the current testing process were expressed during the interviews. Thus, it serves as the last theme of the thematic analysis.

Insecurity is a feeling that was brought up regarding the quality of the code. Developers felt nervous about doing wrong, which could be mitigated by the help of a clear testing process to rely on. Additionally, both managers and developers said that increased testing would probably increase the feeling of safety. Embarrassment was raised as a negative feeling that appears when a customer finds a critical bug since this should have been identified before going live.

The most dominant feeling expressed by the participators was uncertainty, and there was a lot of confusion and ambiguity during the interviews. In general, there is a lack of knowledge about the current process, making the answers partly unclear or vague. One developer expressed that they guess the way they should test since there are no clear guidelines to follow.

5.1.10 Summary of Thematic Analysis

Overall, the findings from the interviews point to a testing environment that, while having foundational strengths, faces several challenges. Figure 5.1 summarizes the results by highlighting the strengths, weaknesses, opportunities, and threats regarding the testing process at the company. It was highlighted that there are many unknowns in the current testing process and environment. The developers, and even managers, are unsure why things are in the way they are. Even so, there is an overall impression that the company does not suffer critically because of the neglected testing process. However, the interviews highlighted that the organization is close to a turning point where it outgrows its immature testing process. In general, the answers to the theoretical and technical questions also indicated that the company lacks sufficient testing knowledge.
The results from all five interviews were mainly consistent and provided a similar picture of the topic, while each provided some new angle. Common for all participants was stating that the goal of the testing process is to ensure no bugs are released to customers. As mentioned in the sections above, this is not achieved currently. However, there were some differences in the answers as well. The most distinct one was when the participants were asked to say how often the production environment crashes due to some bug in the code. The answers ranged from almost every day to every other week. Hence, this was therefore examined further and is described in Section 5.2.1.
Finally, internal motivation and personal responsibility seem to be the driving factor of testing at the company under observation. The quote, "If I feel that a function is important, I write tests for it," exemplifies this.

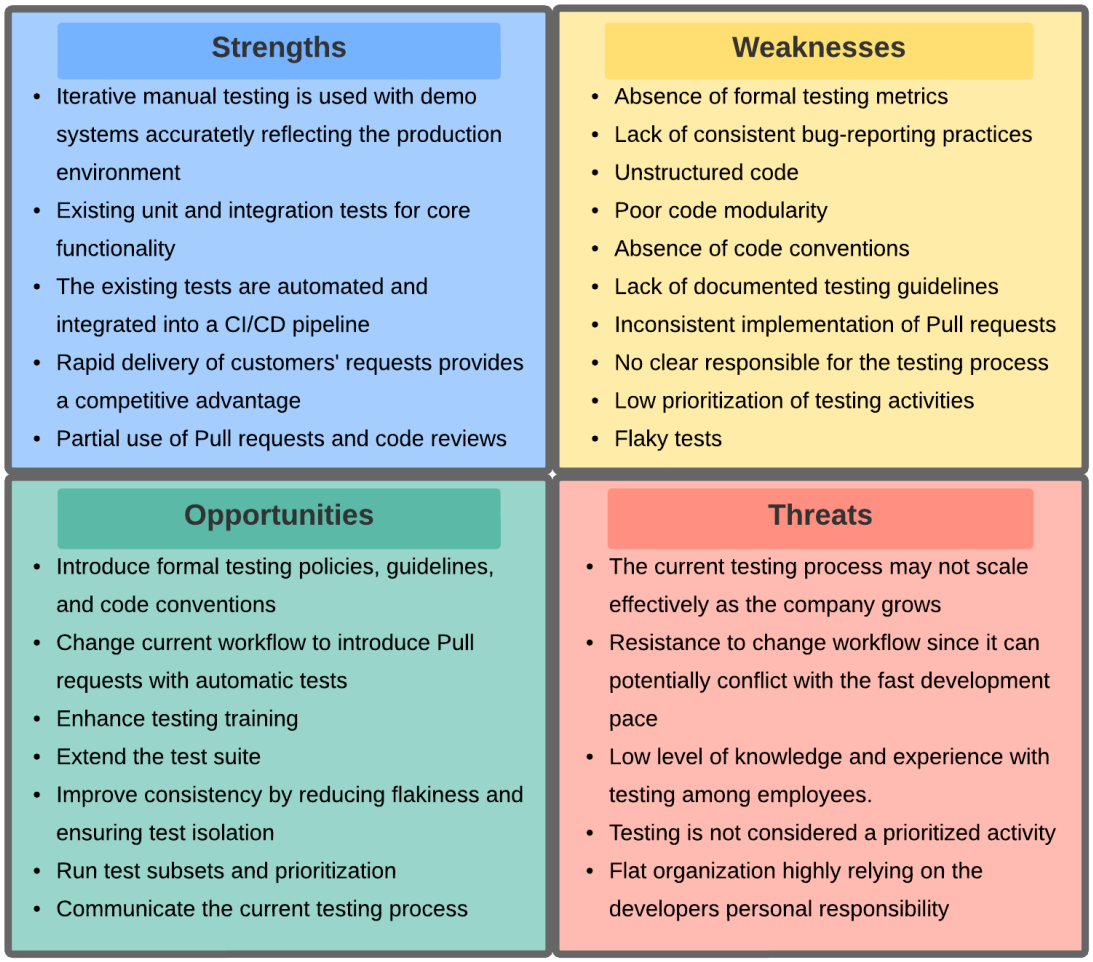

Figure 5.1: SWOT matrix summarizing the results from the Thematic Analysis.

5.2 Software Repository Mining

This section presents the results from the repository mining, which was conducted as a part of the analysis of the company's current testing process. The primary goal of conducting repository mining was to systematically extract and analyze data from the software repository to uncover insights. It was brought up during the interviews that the testing was hindered by the high coupling of the code. The findings during the software repository mining supported this claim. Other findings elicited from the software repository mining are presented in Table 5.1. Some table entries acquire an explanatory discussion, which is provided in the following sections.

5.2.1 Crashes in TeamCity

A report was written internally by the company analyzing the last 20 failures in TeamCity over a period of 13 days. The results of the report found that, on average, the system updates crashed around two times per day. The most common reason for the failures was compilation errors, having caused 16 of the 20 analyzed failures.

This result highlights compiling errors as the most significant issue, which was also confirmed during the interviews.

5.2.2 Flakiness Score

The results of running the tests repeatedly showed the same outcome five of the six times. These results show some degree of unpredictability in the test suite, especially considering the tests were run during the evening when no other automatic tests were run. As explained in the previous chapter, all tests interact with the same test database, which can result in data overriding and tests failing unexpectedly when run in parallel. The company asked us to perform the measurements outside of regular office hours in order to avoid unexpected errors in the continuous integration pipeline. The fact that our measurement could not be done reflecting the real work environment should be considered when interpreting the implications of these results.

5.2.3 Code Quality Issues

Component 1 is the chosen module that was analyzed for the other metrics as well, and it consists of seven classes. Component 2 is a complementary class from another module that was analyzed to mirror the code quality standard in many other components in the repository. The quantity metric is a summary of different quality issues identified by Resharper. Different tools would probably give different results in this aspect since they include and classify different code quality aspects. However, Resharper covers widely acknowledged ones such as redundant code, inaccessible code, syntax style, possibility for exceptions thrown, and spelling issues.

Metric	**Quantity**
Code Churn vs Test Churn	PR 1: +2408 loc PR 2: +253, -51 loc 0 lines of test code edited in both PRs
Crashes in TeamCity	2 per day
Code Coverage	0 % (no tests for this module exist)
Flakiness Score	same outcome 5 out of 6 times
Documentation of Tests/Code	4 comments / 651 lines of code
Number of requested changes /comments in code review	0 requested changes. One comment per PR: "Looks good!"
Code Quality Issues Component 1	110 code issues per 663 loc
Code Quality Issues Component 2	113 code issues per 280 loc

Table 5.1: Metrics from mining a module of the repository under analysis.

5.3 Workshop

The compiled notes from the workshop can be found in Appendix G. The notes showcase that the framework's structure was well received, justifying adhering to the structure proposed by Argüello *et al.* [1]. The workshop discussions mainly circled around what participants did not understand in the framework, and a desire for examples was repeatedly expressed. It was revealed that some participants had not read the framework before the workshop, even though they were asked to. This affected the feedback in some cases where participants expressed a lack of explanations of some concepts that actually were addressed in the framework.
The result of the workshop questionnaire can be found in Appendix H. These results indicate that even though there was potential for enhancement, the framework was headed in the right direction and was generally well-received. Again, the structure of the framework proved to be appreciated. It was confirmed that the theory chapter did provide value. The cumulative voting proved that all axes are relevant and no axis stands out as unnecessary. It is worth mentioning that the four axes this study added to the framework were evaluated as the four most important axes, which evidently motivates the additions and relevance to the company.
The refinements in the framework addressing the workshop results mainly consisted of reformulating concepts or sections to reach higher clarity. The concrete changes performed are listed below.

General:

- Hands-on suggestions and examples were added in several axes, such as examples of tools for traceability
- An appendix containing examples was added to meet the inquiry for concrete and hands-on examples

Plan definition:

- An explanation of the question "Where do we test?" and the concept "Exit Critera" was added
- "Define the general approach to testing" was removed
- The activity "Integrate and coordinate testing activities with the development lifecycle" was added

Affected test levels:

- "Available objects" was clarified with "Available test objects"
- The text was reformulated with the ambition to reduce abstractness

Definition of test cases:

- The concept of bi-directionally traceable test cases was explained
- "multiple test cycles" was removed and replaced by "various stages of the software development life cycle and across different versions of the product"

Test activities:

- "functional documentation reviews" was removed

Traceability:

- The first bullet point in improvement activities from partial implementation was reformulated to clarify the difference from the previous improvement activity

Test environment:

- An explanation of "test harness" was added

Feedback:

- Examples of metrics were added
- Emphasis on automation was added in the axis explanation together with discussing the potential administrative overhead

Theory file:

- A more rigorous explanation of the testing pyramid was added

6

Results: Final Framework

This chapter presents the final testing framework this study resulted in. It is based on the testing framework proposed by Argüello *et al.* [1]. Their framework, described in Section 3.2, has been revised and extended. The resulting framework has the same structure as the underlying one and is structured around a series of axes, each representing a critical aspect of the testing process. The original framework presented eight axes: Plan Definition, Affected Test Levels, Definition of Test Cases, Test Activities, Test Environment, Testing Tools, and Feedback. Four new axes were added for enhancement: Testing Knowledge, Code Quality, CI/CD Pipeline, and Developer Motivation.

Each axis is evaluated at three levels of testing maturity: No Implementation, Partial Implementation, and Full Implementation. This allows for a clear assessment of the current testing state. The axes outlined in the framework serve two fundamental objectives: diagnostic and directional, forming a comprehensive strategy for assessing and guiding software testing improvements. Three matrices are provided to support the objectives. The first matrix, shown in 6.1, supports the diagnostic capability and describes the testing process aspects considered in the testing framework. Yet, the aspects are condensed to be easily followed. This matrix is complemented by a detailed description of each aspect. The other two matrices support the directional objective, outlining how to mature in each aspect. However, these two transition matrices have been partitioned so that each axis displays improvement activities directly after the axis explanation to facilitate the reading flow. However, if interested in studying the complete matrices, they are presented in Appendix I and J.

As in the original framework, it is recommended to approach the development of the axes in the order in which they are found in the matrices. This arrangement is deliberate, as they are organized according to the phase of the development process in which these testing activities are involved. The final proposed framework can be found in isolation at Zenodo: https://doi.org/10.5281/zenodo.11391129, to be easily distributed to the industry.

Two additional artifacts have been added to the framework in Zenodo. The first is a supplementary theory file that aims to explain concepts raised in the framework that might not be known to every actor applying the framework. The theory file is, however, entirely supplementary and can be overlooked if the concepts in the framework are familiar. The framework refers to the theory file when the covered concepts are presented. In this report, the concepts are explained in the background chapter (Chapter 2), and it is the same explanations in the theory file in the framework. The other artifact is an appendix of examples. This can be found in Appendix K.

In Section 6.1, the application process of the framework is presented. Following,

Section 6.2 presents the different components of the framework. The refinements added to the matrices by this study are presented in *italics* to distinguish the work from the underlying framework. After the presentation of the framework, Section 6.3 motivates the axes and the changes and additions done to the original framework. The presentation and motivation of the framework are separated to allow for overlooking the motivation if only wanting advice on how to streamline an immature testing process of an SME.

6.1 Application Process

The framework implementation consists of five steps that should be applied in brief iterative cycles. The steps are Diagnosis, Detection of improvement activities, Contextual adjustment, Implementation of improvement activities, and Verification of results. Figure 6.1 provides a visual representation of the process.
During the diagnosis, the current state of the organization's testing process must be established. This is done by deciding the current level of each axis in the diagnosis matrix. Once the diagnosis is performed, the axes to be improved should be detected. It is recommended to start with the axes that are less developed and on areas that offer the most significant return on investment. However, the axes are presented in a deliberate order, which should be considered when addressing them. The next step of the framework implementation is the Contextual adjustment. It aims to tailor the recommendations to fit the company's specific context. During this step, companies should reflect on their available resources and existing processes. The considered resources could be available personnel, budget, and time for testing efforts. The processes adopted could be agile practices, among others. Based on the insights gained from reflecting on the company's available resources, constraints, and adopted processes, companies should then adapt the improvement recommendations to fit their unique context. The objective of reaching the highest level for all axes may not be immediately feasible for all companies; instead, it could be a long-term goal. To guide the Contextual Adjustment step, the following questions could be considered:

- What is the available budget for testing activities within the organization?
- What is the available personnel that can be allocated for testing activities, and to what extent (full-time, part-time, etc.)?
- What is the available time frame to introduce the chosen testing activities?
- What are the available resources to maintain the testing activities in the long term?
- How is the available testing knowledge within the company? Is the necessary expertise available within the organization?
- What are the current practices and processes within the organization, and how can they be integrated with the chosen testing activities?
- What are the benefits vs the cost of implementation of the chosen testing activities?

Only those activities that are considered reasonable to implement should be chosen. The implementation process continues with the implementation of the improvement activities for each axis. Finally, the results are verified, and the degree of evolution

of the axis is re-measured using the diagnosis matrix, which effectively goes back to the diagnosis step. This process is to be repeated iteratively until all the axes reach the desired level.

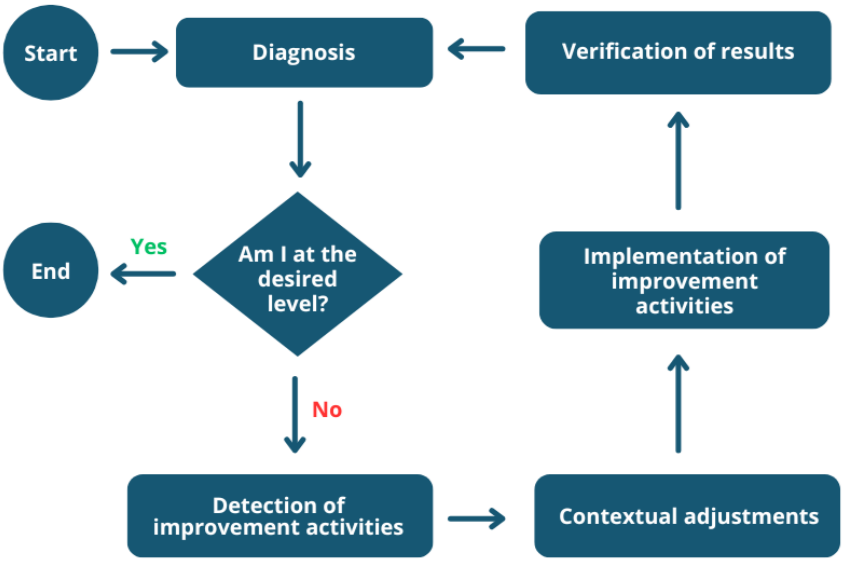

Figure 6.1: Illustration of the framework application process. The diagram is inspired by the original framework [1].

6.2 Presentation of Framework

Axis	No Implementation	Partial Implementation	Full Implementation
Testing knowledge	*Limited*	*High-level understanding, not maintained*	*Continuously shared and maintained within the organization*
Plan definition	No plan	Informal plan	Formal Plan. *The plan is continuously maintained and evaluated.*

Affected test levels	No testing	*Deliberate use of test levels. Mainly unit testing, some integration and system testing*	*Comprehensive* unit, integration and *system* tests and higher level tests
Definition of test cases	No definition *of test cases*	High-level *definition of* test cases	Fully specified test cases
Test activities	No testing	Static or dynamic tests. *Some testing techniques applied*	Static and dynamic tests combined. *Carefully selected testing techniques*
Traceability	No traceability	Partial traceability	Full traceability
Test environment	No specific testing environment	Managed and controlled environment	Testing in a suitable environment
Testing tools	Non-specific tools	Specific tools for monitoring and executing tests	Extensive process automation
Code quality	*Not considered or evaluated*	*Introduced guidelines*	*Full codebase adhere to guidelines*
CI/CD pipeline	*No tests included*	*Includes static and dynamic tests*	*Optimized tests*
Feedback	No reports	*Some internal and/or external defect reports*	*Extensive internal and external defect reports. Progress and process reports and activities*
Developer motivation	*No efforts have been made*	*Activities to enhance testing motivation have been introduced*	*Developers are motivated and recognized for testing activities*

Table 6.1: Final proposed framework: Axis diagnostic matrix.

Testing Knowledge. Testing knowledge in this framework refers to the level of un-

derstanding of the different aspects and processes that software testing encompasses. This axis ensures knowledge in the area at the first maturity level, verifying that every developer possesses fundamental testing knowledge through simple means. This could be done by providing reports generated by trustworthy parties that suit the context, such as the testing guidelines provided by Microsoft: [75]. Continuing, the next maturity level encourages advancing the testing skills and knowledge sharing. This could be done by enrolling developers in testing courses of various ranges, from free online courses to academic or institutional sessions. The total improvement activities for each level are presented in Table 6.2.
In parallel with technical knowledge, an understanding of cognitive biases in relation to software testing is essential to be able to mitigate the phenomena [33]. This framework provides a condensed explanation of cognitive biases, found in Section 2.10.1. The mentioned resource is encouraged to spread within the company for educational purposes. Finally, the distribution of knowledge within the organization could be achieved by, for example, providing communication channels assigned to discussing software testing or arranging simple presentations, engaging workshops, and conferences.

Testing knowledge	
Maturity stage	**Improvement activities**
From No Implementation	• *Include basic testing principles as well as the organizational approach regarding testing in the onboarding process* • *Provide simple resources to achieve fundamental testing knowledge* • *Emphasize unit testing* • *Clarify advantages of software testing*
From Partial Implementation	• *Provide courses in testing, with the possibility to advance in expertise level* • *Educate about cognitive biases that affect testing* • *Ensure fundamental testing skills before employment* • *Provide an opportunity to share knowledge with colleagues*

Table 6.2: Evolution matrix for Testing knowledge.

Plan Definition. The Plan Definition axis involves activities that outline the test objectives and the strategies for reaching those objectives, ultimately resulting in the creation of a straightforward and easy-to-follow testing process. This axis encompasses aspects such as the definition of clear Exit Criteria, which determines the

set of conditions that should be met to consider the testing phase as concluded. An example of how an Exit Criteria could be formulated is shown in Appendix K. Other aspects included are clearly defining roles and responsibilities related to testing at the company and estimating time and resources destined for testing activities. Decisions about how the testing and development activities should be integrated should also be made. This could be, for example, deciding on a test-driven development process where the tests are written before the actual code or deciding on continuous collaboration between developers and testers by adopting agile methodologies.
The degree of evolution along this axis is determined by the level of detail and how the plan is communicated and formalized. In this context, formalization means that the testing process has been documented and distributed to all stakeholders. The transition suggestions are found in Table 6.3.
Ideally, test planning should begin early in the development process, simultaneously with obtaining requirements. The system's requirements should be clearly stated to enable traceability and monitoring of results. An example of system requirements is shown in Appendix K. The established testing process should be regularly maintained and evaluated to track its progress and be able to take corrective measures when necessary. Feedback from testing activities should be used to fine-tune the testing plan.
In the context of this axis the questions of when and where to test means making decisions on how often and under which circumstances test activities will take place. The answer to when to test could, for example, be: running the test suite every night or running them on every commit. The answer to where to test could be decisions on running the test suite locally or as part of a continuous integration pipeline. The specialized testing needs of the company refer to aspects of the software not directly related to a specific function or user action, such as performance, usability, reliability, and security.
When estimating the time needed for testing activities, the estimation technique Planning Poker is recommended. This technique consists of participants independently estimating tasks and simultaneously revealing their estates. This process is done iterative until all participants have reached a consensus. It is recommended that companies balance testers' work schedules, allowing them to engage in other activities such as development.

Plan Definition	
Maturity stage	**Improvement activities**
From No Implementation	• Answer the questions: What do we test? What do we not test? And why do we do it? *When do we test? Where do we test?* • *Define and communicate roles and responsibilities for testing activities. Assign someone responsible for the software testing* • *Integrate and coordinate testing activities with the development lifecycle* • *Define product requirements* • *Define Exit Criteria*
From Partial Implementation	• Document the plan and distribute it to the development team and stakeholders • Make an estimated time budget required to carry out the test *activities and add buffer time for testing activities* • Generate a calendar with specific dates or in the context of each iteration • Estimate people and resources to carry out the planned activities. If possible, make a tentative assignment • *Decide specialized testing needs at the company* • *Create opportunities for participatory decision-making* • *Hold retrospective meetings destined to maintain and evaluate the chosen testing process* • *Comprehend feedback from testing activities to fine-tune the testing plan* • *Follow up if product requirements have been fulfilled*

Table 6.3: Evolution matrix for Plan definition.

Affected Test Levels. This axis addresses the question "What to test?" in terms of different test levels. If not familiar with the testing levels, read Section 2.2 that swiftly covers the four most common testing levels; unit, integration, system, and acceptance. To effectively answer the primary question proposed by this axis, an analysis of the relevant modules to be tested in the code base should be undertaken. This involves determining which parts of the code are pertinent for testing at each

level. The different maturity levels cover the order in which the testing levels should be addressed. If starting from a case where test levels have not been considered at all, the first level to prioritize is unit tests. Then, integration and system test levels are naturally followed. Low-level testing allows defects to be addressed earlier in the testing process. The final maturity stage covers the top of the testing levels, as in acceptance tests. This should establish confidence in the product quality, ensuring its completeness and expected performance. The concrete steps for maturing are shown in Table 6.4.
In most cases, SMEs do not have a complete test suite. It is essential to prioritize testing time wisely when extending the suite. Encompassing a wide range of levels enables the early detection of deviations. To decide what tests should be performed at each level, it is suggested to analyze the existing test suite and identify gaps to fill.

Affected test levels	
Maturity stage	**Improvement activities**
From No Implementation	• Analyze the available test objects from the unit, *integration* and system-level test *perspectives* • Identify the features to be tested from the previously analyzed test objects, as well as the conditions under which they will be tested • *A rule of thumb is to focus the density of tests in the order of the test pyramid, with most tests in the lower levels*
From Partial Implementation	• Analyze the higher-level test objects, such as acceptance tests • Identify the features to be tested in the previously added test levels.

Table 6.4: Evolution matrix for Affected test levels.

Definition of Test Cases. The axis Definition of Test Cases answers the question "How to test?". The involved activities extend from designing test cases to identifying input data, execution conditions, and expected results. This axis is considered to be at the lowest maturity level when no definition of test cases is generated, the middle level when high-level test cases have been identified, and at the highest level when the test cases have a complete specification of their inputs, execution conditions, and expected outcomes. In this context, a fully specified test case is defined as one that not only includes detailed information on the expected positive outcomes but also includes the negative outcomes and conditions which the software is not designed to handle or should explicitly fail under. The concrete steps for maturing

are shown in Table 6.5.
Generally, it is good practice to start with designing high-level test cases. These are expressed without concrete values for the input data and the expected results, also known as checklists. An example of a high-level test case is shown in Appendix K. High-level test cases can be reused in various stages of the software development life cycle and across different versions of the product. It should be noted that the generation of test cases also leads to the identification of other needs early on, such as the necessary infrastructure and tools.
It is recommended to apply a test specification technique when designing test cases since they provide a systematic approach to testing, helping to ensure test coverage and increasing the likelihood of identifying defects. Section 2.5 delves into some established test specifications techniques, providing the bases for their implementation.
Ideally, each test case should be traceable bi-directionally to the specific test condition it addresses. This means starting from the test condition, often derived from requirements, tracing forward to the corresponding test cases designed to verify that the condition is met, and the same the other way around. When defining test cases, it is important to remember that each test case should be independent. This ensures consistency in the test process, mitigating the risk of cascading failures, where the failure of one test case affects the outcome of subsequent ones.
Another aspect to consider when generating test cases is the effects of cognitive biases. In particular, confirmation bias is a critical concern in software testing and is seriously detrimental to testing quality. One debasing technique that mitigates this bias is explicitly asking developers to seek evidence of problems rather than the system working correctly. Another is asking developers to play devil's advocate and consider why the solution may be wrong. These direct questions can be added as a checklist.

Definition of test cases	
Maturity stage	**Improvement activities**
From No Implementation	• *Agree on a standard format for test case specification* • Create high-level test cases. A checklist is often recommended as a first approximation

From Partial Implementation	• *Choose appropriate test specification technique(s) to generate test cases* • *Utilize debasing techniques such as playing the devil's advocate* • Generate test cases identifying: the possible input data, the execution conditions, and the expected results • *Document the fully specified test cases for follow-up*

Table 6.5: Evolution matrix for Definition of test cases.

Test Activities. Software testing activities refer to the actions done in a testing process. Static and dynamic testing are the major broad categories within test activities. The fundamental difference between the categories is that dynamic testing encompasses code execution, whereas static testing does not. They complement each other by finding different types of defects, making it essential to perform both types of tests. Therefore, the axis's evolution targets the application of both test activities. The concrete steps for maturing are shown in table 6.6.
Automated test execution tools are suggested when performing dynamic tests. Some benefits are reduced repetitive manual work, time efficiency, greater consistency, objectivity, and repeatability. On the other hand, static testing relies on the manual evaluation of work products. However, static code analysis can be facilitated with the use of tools. Code reviews are a commonly practiced static test activity. Practicing code reviews enhances the quality of the software product while also fostering teamwork. An example of a clear guide to follow when performing code review activities can be found in Appendix K.
Within static and dynamic testing, different techniques can be applied. This framework will not cover the exhaustive list of testing techniques, but a few inspirational and commonly applied ones are; Risk-based testing, Regression testing, Sanity testing, Exploratory testing, Property-based testing, and Mutation testing. These are explained in Section 2.4. Property-based testing is strongly dependent on a tool to assist the input generation. For the Java Framework, jqwik [76] is a commonly used and suggested tool. Mutation testing can also rely on testing tools. For Java, Pitest [21] is a common tool to use for mutation testing, and MutPy [77] for Python programmers. What technique and tool to choose depends on the context and present specialized testing needs.
Another testing activity that should be considered at the higher level of maturity is optimization techniques. Three common testing techniques in this regard are described in Section 2.6, which all aim to reduce testing costs.

Test activities	
Maturity stage	**Improvement activities**
From No Implementation	• Perform static review tests, e.g., code reviews or inspections and walkthroughs • Perform exploratory dynamic tests manually • *Consider basic testing techniques such as Regression testing and Sanity testing*
From Partial Implementation	• Perform static analysis tests guided by tools such as source code analysis *(examples are given in the Code Quality axis)* • Automate dynamic tests • *Consider more advanced testing techniques such as Search-based testing, Property-based testing, Mutation testing* • *Optimize tests*

Table 6.6: Evolution matrix for Test activities.

Traceability. This axis involves the monitoring and control activities of the developed testing plan. Test monitoring refers to continuously evaluating the real progress against the predefined test plan. Meanwhile, test control refers to implementing necessary measures to achieve the goals outlined in the test plan.
The degree of evolution across this axis is determined by the level of traceability between the test suite and the work products. To achieve the highest maturity level, requirements should be traced to the specific test case, highlighted in the transition matrix in Table 6.7.
Maintaining traceability during the entire testing process is crucial for successful test monitoring and control. Effective traceability enables the evaluation of test coverage. Furthermore, it enables the analysis of the impact of modifications and supplies data to appraise product quality, process capability, and advancement of business goals. To support the traceability of software testing efforts, test management tools such as QTest[78] and Zephyr[79] can be used. Such tools allow organizing test cases, planning test executions and tracking of results. They often integrate with requirements management and defect tracking tools such as Jira[80] to provide end-to-end traceability, maintaining a transparent and auditable trail from requirements to test cases and outcomes.

Traceability	
Maturity stage	**Improvement activities**
From No Implementation	• Maintain traceability between test objects and test cases
From Partial Implementation	• *Add traceability between requirements and the test objects and test cases* • *Determine the coverage of the tests* • *Introduce test management tools*

Table 6.7: Evolution matrix for Traceability.

Test Environment. The test environment axis covers the setup under which the tests are run. Various test categories can require different test environments. System-level testing should preferably be conducted in the system's operating environment since detecting failures under those conditions becomes irrelevant if the testing environment differs significantly from the operating environment. However, due to resource constraints and practical limitations, it is often unfeasible to test in the production environment. Consequently, test environments should closely mirror real-world conditions as often as possible.
This particular axis aims to assess the level of suitability of the testing environment and how well it mirrors reality. The lowest maturity level represents the absence of an established testing environment. The next level is reached when a dedicated, incomplete environment for testing is available. This could be incomplete for various reasons, such as missing integrations or poorly defined data in the database. The final level is achieved when the testing environment fully mirrors the production environment. The transition matrix encouraging maturity of this axis is found in Table 6.8.
The highest maturity level includes a test harness, which contains all the information needed to run test cases in automated or integration testing. This could, for example, be configuration settings, managing test data, controlling the flow of test execution, and cleaning up after test execution. The final level also encourages the use of service virtualization. This is the practice of simulating dependencies and emulating the behavior of dependent systems to be able to remove constraints from the software under test. This enables testing earlier in the lifecycle since all of the software connections do not need to be available. Additionally, virtual services can save resources if the original components are expensive to interact with.
It is recommended to let a CI/CD pipeline be a part of the test environment by using it to trigger tests at different stages of the lifecycle. This is explained in greater detail in its axis further below.
A critical problem regarding the test environment is the risk of producing flaky

tests. This means that the tests yield different results under the same conditions across runs. This could be caused by several testing nodes accessing and modifying the same database simultaneously. To mitigate this, mocking the database interactions for unit-level tests is a solution suggested in the final maturity matrix. Mocking means mimicking the interface of the actual dependencies but providing predefined responses to method calls, allowing developers to control the behavior of dependencies during testing.

Test environment	
Maturity stage	**Improvement activities**
From No Implementation	• Build a test environment outside of the development and production environment • Prepare it with shared test data, where possible, that is similar to production data • *Integrate triggering tests in the CI/CD pipeline*
From Partial Implementation	• Include in the test environment: test harness, service virtualization, simulators, and other necessary infrastructure elements • Prepare test data from production data and keep it up to date • *Mock external dependencies as often as possible for lower-level tests*

Table 6.8: Evolution matrix for Test environment.

Testing Tools. Numerous tools exist with the ambition to support different test activities. Overall, tools reduce the possibility of including errors. Testing tools can also support changing workflows by enforcing adherence to new guidelines and hindering old routines. In the meantime, only implementing a particular tool does not promise success. Every new tool introduced into the test process requires effort and proper management of people using the tool to achieve real and lasting benefits. It is important not to be tempted to apply complex tools only because they promise high results since it must be achievable to learn the tool given the resources. The maturity flow of this axis suggests when it is suitable to incorporate different types of tools in the testing process. The concrete steps for maturing in this axis are shown in Table 6.9.
Some examples of tools that support unit testing are JUnit[81] for Java and NUnit[82] for .NET languages. Together with GitHub Actions[28], unit tests could be automated if the repository is hosted on GitHub. The automation process is closely related to the CI/CD Pipeline. TeamCity[31] is an example of a tool that supports testing at the delivery stage. A tool to assess the coverage of the test is suggested.

For instance, JaCoCo[83] is a tool for Java development that outlines different kinds of code coverage, such as statement, branch, line, method, and class coverage. A suggested testing tool for integration tests is PostMan[84], a software facilitating integration testing by enabling API call testing. Jira[80] and GitHub Issues[85] are examples of tools that facilitate bug reporting and management of the test process.

Testing tools	
Maturity stage	**Improvement activities**
From No Implementation	• Incorporate requirements and incident monitoring tools. As a first approximation, they can be carried in a spreadsheet, but it is advisable to use specific tools for this type of task • Incorporate test execution tools that allow them to be automated • *Apply tools supporting unit testing*
From Partial Implementation	Incorporate tools for: • test management and test products • test design and implementation, including test cases, test procedures, and test data • execution and registration of tests • performance measurement and dynamic analysis • *specialized testing needs such as security tests, portability, and accessibility*

Table 6.9: Evolution matrix for Testing tools.

Code Quality. This axis aims to streamline the testing process by increasing the code quality of the code base. The transition suggestions are found in Table 6.10. The first transition establishes a shared perception of what defines the desired code quality and how to achieve it, as well as enforces application in the development routines. The final transition covers ensuring coherent code quality in the already existing code as well.
To increase code quality, adhering to code standards is essential. These include syntactical decisions such as naming conventions, formatting rules, code architecture, and design pattern guidelines. The large development ecosystems have their suggested conventions - to mention a few; .NET: [86], Java: [87], and Python: [88]. The specific conventions that suit the development context should be chosen. For example, if the system under development is an object-oriented product developed in the .NET framework, it would be suitable to follow the SOLID principles as well as .NET's coding style: [86]. Code standards are often dynamic and constantly revised, so it is essential to stay updated. It is suggested to facilitate adherence

to code standards by integrating a formatting tool. This could be integrated into the IDE. Again, what specific tools to use are context-dependent. As inspiration; ReSharper [73] developed by JeantBrains for C# and Pylint[89] for Python could be utilized. SonarQube[90] is another recommended static analysis tool that complies with several programming languages. SonarQube can help detect other code quality issues, such as code smells and complexity (described in Section 2.11).

Code quality	
Maturity stage	**Improvement activities**
From No Implementation	• *Chose code standard(s) to follow and establish a routine for staying updated with the guidelines (suggestion: workshop once a year)* • *Introduce automated tools for continuous assessment of code quality* • *Integrate the code quality aspects to code reviews* • *Communicate and educate the importance of modular code* • *Pair programming is encouraged* • *Comment the code*
From Partial Implementation	• *Do a full scan of the code base and fix existing code quality issues. This could be a good opportunity for someone with lower domain expertise or experience at the company to familiarize themselves with the code base (e.g., a summer job or a project for a new employee)* • *The code quality of every new change should be evaluated to meet the code quality guidelines*

Table 6.10: Evolution matrix for Code quality.

CI/CD Pipeline. This axis defines how to integrate testing into an existing CI/CD pipeline. It is generically formulated since a pipeline is highly dependent on the context, as in what tools and frameworks are applied in the current project. The goal of the maturity in this axis is to transition into a CI/CD pipeline where optimized testing is seamlessly integrated through automated processes. Table 6.11 presents the improvement activities for each of the three maturity stages.
The No Implementation level assumes a basic production environment; a version control system, a platform for hosting the repository, and a deployment stage, but without any testing activities. If a pipeline is not set up, it is recommended to browse the internet to see what solutions fit the context. For example, if working with GitHub, numerous guides exist provided by the engaged community. Among

others is this blog post: [91].
A development pipeline, including tests, can be of various ranges, and the comprehensiveness often reflects the execution time. However, the more rigorous testing in the pipeline, the higher the chance of catching faults, desirably as early as possible. Automated tools are suggested because they make the testing activities less disruptive, leaving the execution time not to affect the developer so much. For example, SonarQube, mentioned in the Code Quality axis, can be integrated seamlessly to be run on every commit if the repository is cloud-based, with the use of SonarCloud [92]. Running linters can also be automatically triggered in the CI/CD pipeline.

CI/CD pipeline	
Maturity stage	**Improvement activities**
From No Implementation	• *Set up a pre-commit or pre-push command that runs formatting and simple tests* • *Create a build script in the repository that states which set of tests should be run on every commit/push. Update the chosen set regularly* • *Integrate static code analysis tools for performing code quality checks* • *Create a build script for the deployment, including what tests to run at this stage*
From Partial Implementation	• *Prioritize the order of the test cases in the build scripts* • *Perform risk assessment and run the critical first* • *Examine the different tests to not cover redundant tests* • *Ensure to not run the same tests in several build scripts if not needed*

Table 6.11: Evolution matrix for CI/CD pipeline.

Feedback. There are different types of feedback involved in the testing process, including internal and external reports. Internal feedback includes information about when a test level is completed, how the testing is progressing, results from code reviews and test execution, and how well the tests cover the software under test. Also, providing feedback on the process and its activities is important. It can contribute to building good relationships since feedback creates a space for dialogue that allows visibility of progress and leads to continuous improvement. External feedback includes, among others, feedback from users, acceptance testing, and bug reports. Many of the internal and external reports can be elicited from metric-based tools.

Some examples are code coverage tools (for example JaCoCo [83], test execution tools (for example Resharper [73]) and testing management tools (e.g., Jira [80]).
For this axis to provide value to the company, the feedback reports must be carefully considered as the initialization for improvement actions. Involving feedback in the development lifecycle adds administrative overhead, but it can also be added with small means. Therefore, many of the improvement activities suggested rely on automation.
The maturity in this axis is based on how many aspects of the process are evaluated with feedback. The transition matrix is found in Table 6.12. The No Implementation level covers the scenario when no feedback reports are provided. The second maturity level includes some internal and/or external reports, whereas the final maturity level also incorporates process and progress evaluation. Reports in this axis refer to any artifact containing feedback results and do not explicitly mean a report in paper format.

Feedback	
Maturity stage	**Improvement activities**
From No Implementation	• *Establish metrics, such as code coverage, testing time, and number of failures* • *Integrate tools for extracting the metrics* • *Gather feedback from users* • *Keep a count of the test cases executed, the results obtained, and the defects reported*
From Partial Implementation	• Generate progress reports to evaluate the process of testing • Allow interested parties to provide feedback about the quality achieved and priorities for future releases

Table 6.12: Evolution matrix for Feedback.

Developer Motivation. The axis Developer Motivation highlights the factors influencing human motivation to perform software testing. This encompasses both motivational and de-motivational factors as well as proposed strategies to increase motivation.
The degree of maturity evolution across this axis is determined by the efforts made to stimulate and motivate software developers to test. The company is considered at the lowest level when no effort has been made to motivate the developers to test. The company is considered at the intermediate level when activities to enhance testing motivation have been consciously introduced. This indicates that motivation has been acknowledged as a relevant aspect of the testing process. The highest level

is determined by developers feeling motivated and recognized for testing activities. Concrete steps to reach a higher maturity level are displayed in Table 6.13.

To increase motivation it is recommended to combine testing responsibilities with development activities. In companies with a designated testing department, the testers should be involved during the entire development process and not only at the end. Recognition has a clear role in software professionals' motivation toward testing activities. Making developers feel recognized can be achieved by actively giving positive feedback related to software activities and by working towards raising the status of testing at the company. Testing activities should be considered as crucial as development activities. Finding bugs should be encouraged for testers to feel motivated to do their job. If testers feel unpopular when opening critical defects, they could become reluctant even if it is a severe defect [11].

Developer motivation	
Maturity stage	**Improvement activities**
From No Implementation	• *Increase the testing status at the company* • *Combine testing responsibilities with development activities* • *Ensure a variety of engaging and challenging tasks* • *Encourage a good relationship with management and colleagues* • *Set aside time destined for only testing activities* • *Ensure the testing environment holds a high-quality* • *Provide a clear testing strategy to follow* • *Provide clear test examples to follow*
From Partial Implementation	• *Introduce ways to provide positive feedback when discovering faults* • *Ensure active influence in decisions concerning the testing process* • *Make use of code quality metrics* • *Adopt pair programming practices*

Table 6.13: Evolution matrix for Developer motivation.

6.3 Motivation of Framework

The framework in general is motivated by both the focused literature review as well as the analysis of the participatory company. The literature highlights the importance of a high-quality test suite since it facilitates the evolvement and maintenance of the software [3]. The analysis of the company clearly expressed a need for guidelines regarding the testing process to manifest confidence in the codebase, especially when the codebase grows rapidly.
The original framework has been extended with new aspects found to be essential for more a complete testing framework, and some elements have been removed when not supported by literature or the company analysis. The aspects were reformulated when objectives were not clearly communicated, and several factors were explained more concretely with examples, with the ambition to facilitate easy adoption of the framework for industrial companies. However, the original framework generally received positive feedback and was concluded to be easily followed, which is why components were edited with caution. The format of the framework was kept due to the feedback about its simplicity and practicality. The concrete changes performed on the framework matrices are shown in a color-coded version highlighting the revisions in Appendix L.
Examples of tools, processes, and relevant literature were added to the framework to inspire companies with little experience in testing. Additionally, techniques can often debias tasks faster than awareness of the human can [33], which makes it a fast track to take in parallel with educating the developers about potential biases.
The following section is organized as follows: First, the refinement of the implementation process is motivated. Then, all axes are motivated in the order they are presented in the framework.

6.3.1 Contextual Adjustment

When attempting to implement Test Process Improvement (TPI) models, SMEs encounter distinct challenges that differ from those larger organizations face. SMEs are often limited by resources in terms of finance, available time, and skilled personnel within specific areas, such as software testing. These limitations can affect their ability to implement and maintain rigorous testing processes [9]. Additionally, significant variability exists within the companies that fit the SME definition, encompassing companies with up to 250 employees. The specific processes utilized at the company can vary considerably, as one organization can operate in an agile environment while others use the waterfall approach.
In order to address the mentioned difficulties and the great diversity within the SME landscape, it is fundamental that companies adjust to the models as necessary to fit their unique context and constraints. The Contextual Adjustment step was added to the original implementation process to facilitate a pragmatic and tailored approach to the proposed framework. This step ensures that the proposed improvements are not only theoretically sound but also practically viable, taking into account the company's specific operational environment, resources, workflow methodologies, and strategic goals. In essence, the Contextual Adjustment step aims to make

the framework not just a set of best practices but a dynamic tool that respects the unique nature of each organization, facilitating a more thoughtful and effective implementation of software testing improvements.

6.3.2 Testing Knowledge

Possessing knowledge about software testing is fundamental to being able to perform any testing activities. Without understanding the advantages of testing or the drawbacks of neglecting it, spending resources on testing activities is unreasonable. Therefore, this aspect is added to the framework, placing it in the beginning to serve as a foundation for the framework. Ng *et al.* [93] presents that the primary factor hindering organizations from adopting software testing methodologies is the lack of testing knowledge. Toroi *et al.* [43] agrees and emphasizes that developers need more testing expertise. Kituyin and Amulen [9] points out lack of competence in specific software engineering areas as a common prohibitive factor when applying test process improvement in SME environments. These are only some example studies that highlight the importance of testing knowledge. Fraser and Straubinger [3] highlight how knowing the significance of testing activities can motivate developers to perform software testing. The focus on unit testing knowledge is because of several resources highlighting the importance of unit testing, among others Toroi *et al.* [43] and Torkar and Mankefors-Christiernin [15]. Torkar and Mankefors-Christiernin explain that unit testing is crucial since it ensures a stable foundation in the code if appropriately done, hindering errors from propagating throughout the software. The suggestion about testing courses as a solution to achieve testing expertise is inspired by [3] that highlights how developers who have completed software testing courses see the benefits of software testing and are generally more motivated to perform software testing. Finally, education about cognitive biases is suggested since [33] and [58] highlight how knowledge about biases serves as a mitigation to the bias.

6.3.3 Plan Definition

The main structure and most of the recommendations included in the Plan Definition axis were kept as presented in the original framework. This axis's importance was supported by the literature and interviews conducted at the company. During the interviews, the need to decide on testing procedures and communicate them through the organization was clearly expressed. There was a general uncertainty about the testing process, resulting in feeling unsure about the quality of the code. Straubinger and Fraser present a solution by stating that thorough testing gives developers peace of mind, providing a feeling of security [3]. This was supported during the interviews where developers expressed they would feel safer if more testing was introduced at the company.

Even though the plan definition axis was deemed appropriate in the original framework, several aspects were added to enhance the activities. Some of the added aspects have their foundation in the current research in Behavioral Software engineering, considering how human aspects affect the planning process. Participatory

decision-making and retrospective meetings were recommended since they have been shown to ameliorate the effects of availability bias on effort estimation and decision-making [33]. It has also been shown that decision-makers struggle to adjust to new or changed environments because they are anchored on their outdated understanding of the context [33]. The practice of retrospective meetings can also reduce this difficulty. The interviews revealed a lack of space to bring up ideas for improvements and reconsider previous decisions. This problem is also tackled by the recommendations on creating opportunities for participatory decision-making, as well as defining clear roles and responsibilities for the testing process.

Another aspect added was the recommendation to use the estimation technique Planning Poker when estimating a time budget for testing activities since it has been shown to mitigate anchoring and adjustment biases [33].

Furthermore, the recommendation to formulate a clear Exit Criteria was made, given that the lack of specific quality goals can result in misguided assumptions about code quality and the overall necessity of testing [3]. It was also recommended to encourage testing activities early in the development process since this contributes to decreasing testing costs and improving efficiency [13]. Introducing traceability early on was recommended since it will ensure that all requirements are covered when applying the defined testing process, facilitating the identification of any missing or incomplete tests [94]. Another recommendation is to combine testing and development activities since individuals who are not only involved in testing activities have been shown to have lower confirmation bias, making them better testers [59]. Concerns about when and where to test were revealed during the interviews. These aspects were not included in the original framework but were considered necessary. Therefore, "When do we test?" and "Where do we test?" were added.

The findings in the literature revealed a common problem in small organizations: employees exhibit a high degree of self-trust and place significant emphasis on personal responsibility [9]. This phenomenon can manifest to the extent of employees feeling exempt from adhering to rules. This characteristic was also present in the analyzed company, where much emphasis was put on personal responsibility. The literature also indicates the common existence of flat organizational hierarchies in SME environments, where roles and responsibilities are often unclear. This characteristic was also observed at the participatory company, where testing responsibilities were not assigned to any specific person. These issues are yet another reason for creating a clear testing plan.

6.3.4 Affected Test Levels

This axis is only slightly adjusted compared to the original framework. This is because this study's findings align with this aspect in the original framework, strengthening this axis even further. Since testing on different levels targets different potential problems, this is a valid axis to include. The interviews at the participatory company highlighted that they miss testing on some levels, for example, testing API calls, which also argues the importance of including a reminder about different testing levels in a complete framework. The original framework included suggestions about specialized testing needs and considered prioritization of test cases. These

two suggestions were removed from this axis since they were assessed to belong to other axes instead; Plan Definition and Test Activities, respectively.
A rule of thumb on how to prioritize the amount of testing performed on each level was added to the transition matrix from No Implementation level. This was because the literature review showed that lower-level testing is the fundamental core of testing and should be carried out in the early stages of testing [15]. It was also adjusted to include integration tests at the Partial Implementation level since the original framework received feedback from an expert group that integration tests should be carried out in conjunction with the early unit tests.

6.3.5 Definition of Test Cases

The Definition of Test Cases axis was also to a large extent kept as in the original framework. Nevertheless, some recommendations were added to enhance the existing framework. Once again, research in the field of BSE and considerations of how human aspects affect the testing process stand as the foundation for many of the changes introduced in this axis.
There is strong evidence of confirmation bias among software developers regardless of their level of expertise [58]. This bias explains the tendency among software testers to create and execute test cases that affirm the anticipated functioning of the program rather than conducting tests that uncover defects [33]. To mitigate this bias, the concept of a fully specified test case was introduced. This recommendation was based on the discoveries of the paper "Analyses of factors related to positive test bias in Software Testing" [58]. This study states that the effects of confirmation bias appear to be mitigated by using more complete specifications. Using debiasing techniques, such as playing devil's advocate, was recommended based on the findings in the paper "Cognitive Biases in Software Engineering: A Systematic Mapping Study" [33]. This paper recommends using direct questions through a checklist to mitigate overconfidence and optimism bias. These recommendations align with trying to break the system instead of trying to write perfect test cases, which is also encouraged in the mentioned study. The recommendation of agreeing on a standard format for test specification early on was also added, given that a standard format makes test cases reusable and self-sufficient [43].

6.3.6 Test Activities

This axis was extended with new aspects compared to the original framework. The original framework only highlighted static and dynamic testing, which are fundamental test activities but are not hands-on themselves. This framework extends the suggestion by including examples of appropriate testing techniques to apply in the testing activities to inspire more concrete activities. The interviews highlighted the importance of regression testing, which is why it was added to the first transition matrix. Risk-based testing was also added in the first transition matrix because of the incitement provided by the ISO/IEC/IEEE standard [12]. Mutation testing was included since the literature review highlighted that developers tend to desire feedback on how good their tests are [15]. Exploratory testing was added in the

first transition matrix since it is easy to conduct and is, therefore, a good start for testing. However, this methodology is prone to human biases, and the tester could miss important aspects, which is why we suggest several other techniques as well, with the ambition that the techniques can compensate for each other's limitations.

6.3.7 Traceability

The traceability axis was kept as in the original framework, and no considerable charges were deemed necessary. The existing literature supports the critical role of traceability in software maintenance and evolution activities. The literature reveals that companies utilizing traceability were more efficient in completing maintenance tasks and generated more accurate solutions. This evidence suggests that traceability can reduce the workload associated with maintenance activities and enhance the overall quality of software maintenance [95].

6.3.8 Test Environment

This axis was extended compared to the original one. It is a critical axis to include since a testing environment is fundamental for executing tests on a broader scale and to be able to automate them in the shared repository. Test harness and service virtualization were explained since these concepts are not guaranteed to be understood by the intended reader, which was revealed during the company workshop. The company interviews highlighted the problem of flaky tests, which is why the axis was extended with a suggestion about mocking external dependencies, such as a test database. It was added that the database should be shared since [43] pinpoints this to be valuable since several developers have the same needs regarding testing data. A hint about integrating tests into the CI/CD pipeline was added since automation is considered a critical part of the test environment, according to the literature. More elaborated motivations about automation and CI/CD pipelines are presented in Section 6.3.11.

6.3.9 Testing Tools

Argüello *et al.* [1] highlight several benefits of testing tools: they improve the efficiency of testing activities by automating the manual, repetitive, or resource-intensive tasks (for example, running regression tests); they support manual testing activities throughout the testing process; increase the reliability of tests; they improve quality by allowing more consistent testing and a higher level of defect reproducibility. Additionally, the literature review suggested keeping the tools simple and easy to learn [15] [43], which is why the axis was extended with this proposal. Toroi *et al.* [43] also pinpoints that proper management of people is important when applying tools, which is why it is addressed in the framework. Due to the emphasis on unit testing in the literature, tools for automation of unit testing were explicitly brought up in the first transition matrix. Tools supporting different types of code coverage were discussed in the axis to shred light onto different aspects that indicate how well the tests cover the product under test since this was concluded a desire in Torkar and S. Mankefors-Christiernin's study [15].

6.3.10 Code Quality

The code quality directly influences the codebase's testability, and poor code quality increases testing costs. Code quality contributes to the code's comprehensibility, a fundamental factor for testing activities. Therefore, code quality should be considered when streamlining a testing process and was added as an axis in the framework. Fraser and Straubinger [3] highlight that a common opinion among developers is that they are reluctant to test other developers' code since it can be hard to understand someone else's code. Similarly, the company interviews revealed that developers found it hard to comprehend the code, and the company has no clearly communicated code style. This is why the framework forces standardization of code style in the first maturity transition. Tools are suggested since they can help the developer follow code standards. Pair programming is recommended since experimental results made by Williams *et al.* [46] indicate that it achieves higher code quality, no matter the expertise level of the developer. The importance of modular code was explicitly added since the interviews at the participatory company revealed that the testing was prohibited because of monolithic code architecture.

6.3.11 CI/CD Pipeline

This axis was added to the framework since the literature and the participating company highlight the importance of making testing as close to non-events as possible, with automation being a trending topic of interest. The interviews revealed that developers at the company do not want to take time and effort from development to testing, which encourages leaving the testing to the CI/CD pipeline. Literature about motivation among developers explicitly encourages integrating tests into the pipeline since many developers think testing is tedious and challenging to do manually [3], causing it to decrease work satisfaction. Kituyin and Amulen [9] also mentioned the difficulties balancing improvement efforts and ongoing development activities in SME environments. Automating the testing process can contribute to alleviating these difficulties. Literature also highlights how code quality and DevOps are closely related [10], leaving DevOps to be closely related to testing since code quality and testing are intertwined. CI/CD pipelines are practices within the DevOps framework, which makes it an approach to engage with code quality.
The analysis at the company resulted in an understanding that test execution time is a critical factor. To target this, optimization techniques are (again) encouraged to shrink the test suite execution time. The advice to ensure that the same tests do not run multiple times, if not having arguments for it, was also added because of the company analysis since this was a contributing factor to the company's test execution time. Additionally, interviews at the company led to the addition of regularly updating the set of tests selected for automation. This aims to keep the tests updated with the evolution of the code base.

6.3.12 Feedback

The Feedback axis was highly inspired by the original framework since it received no constructive feedback. However, it was reformulated to address the problems

of SMEs more explicitly. More concrete actions are suggested in this extended framework instead of the fairly abstract ones provided by the original framework. For example, "Generate reports of internal defect" was reformulated into two hands-on steps; 1) "Establish metrics, such as code coverage and number of failures" and 2) "Integrate tools or procedures for extracting the metrics". This was done based on the literature review showing how SMEs struggle to apply abstract guidelines. Additionally, it was explicitly addressed in the description of the axis that the term "reports" does not necessarily mean reports in paper format. This was done since using the word without explanation leaves the interpretation to the ambiguity of the word, and the literature review highlighted how extensively paper format reports do not suit SMEs since they do not have the resources to manage a lot of extensive documents.
The middle maturity level was adjusted to include external reports, given the evidence of the participating company's substantial integration of customer feedback juxtaposed with its limited incorporation of internal feedback. Therefore, it was concluded that it is possible to practice external reports without being at the top maturity level.

6.3.13 Developer Motivation

Motivation has been reported to have a pivotal impact on productivity and software quality management. However, motivation continues to be undervalued [39]. Low motivation can result in inadequate testing and the overlooking of software flaws [40]. This perspective is corroborated by Shah *et al.* [36], showing that testing quality is significantly influenced by the motivation of testers and the recognition of their testing efforts. Given the importance of motivation for successful testing activities, the axis developer motivation was added to the framework.
Many of the recommendations proposed in this axis aim to increase recognition of testing efforts. Recognition is pivotal when applying SPI strategies [9]. Straubinger and Fraser found in their study "A Survey on What Developers Think About Testing" [3] that nearly 60% of participants would test more if better recognized. Working towards increasing the status of testing activities and providing positive feedback when failures are reported are recommended in the framework as impact strategies to offer recognition.
The study "Challenges and strategies for motivating software testing personnel" [11] proposed a series of motivational factors in software testing that were added as recommendations in this framework. Recommendations such as combining testing responsibilities with development, encouraging good relationships between management and colleagues, and ensuring various engaging and challenging tasks were all found to be contributing factors to developer motivation. Pair programming is considered an enjoyable activity for both student and professional programmers, improving their job satisfaction and overall confidence in their work [46]. Given the advantages of pair programming, this practice is also recommended in this framework to increase motivation.
Recommendations aimed to facilitate the testing efforts were also added to the framework with the ambition to increase developer motivation. A need to reduce

the threshold for testing was expressed during the interviews. Some of the suggestions made to achieve this were providing test examples to follow and having a test strategy defining what and how to test. Supporting this, findings in the literature point out that many developers would engage more in testing if it were technically easier [3]. The utilization of code quality metrics was also considered a motivational factor in the testing process, given that motivation to test can be influenced by the awareness of the quality of the code [3]. The recommendation to set aside time destined for testing activities was also supported by the literature as a motivational factor. It was corroborated during the workshop session when a group of the participants reflected on how they would disregard testing if they already felt stressed by development work. Setting specific time aside for testing was considered an important measure.

7

Results: Evaluation of company after Framework application

This chapter presents the progress of the company's testing process once the final proposed framework was outlined. First, Section 7.1 presents the diagnosis of the state of the company's testing process before the framework was implemented. Continuing, Section 7.2 expresses what aspects of the framework were targeted for improvement in this specific study. Finally, Section 7.3 presents the evaluation of the framework application together with the refinements the evaluation triggered, by providing results of software metrics and the outcome of interviews performed after the application. Additionally, a diagnosis matrix of the testing process after the framework application is presented, to highlight the progress throughout the maturity levels. The chapter ends with a summary of the evaluation phase.

7.1 Diagnosis before Framework application

The analysis of the participating company resulted in an understanding of the testing process at the site. Table 7.1 showcases the assessment of the process in terms of the framework.

Axis	**No Implementation**	**Partial Implementation**	**Full Implementation**
Testing knowledge	Limited	High-level understanding, not maintained	Continuously shared and maintained within the organization
Plan definition	No plan	Informal plan	Formal Plan. The plan is continuously maintained and evaluated.

Affected test levels	No testing	Deliberate use of test levels. Mainly unit testing, some integration and system testing	Comprehensive unit, integration and system tests and higher level tests
Definition of test cases	No definition of test cases	High-level definition of test cases	Fully specified test cases
Test activities	No testing	Static or dynamic tests. Some testing techniques applied	Static and dynamic tests combined. Carefully selected testing techniques
Traceability	No traceability	Partial traceability	Full traceability
Test environment	No specific testing environment	Managed and controlled environment	Testing in a suitable environment
Testing tools	Non-specific tools	Specific tools for monitoring and executing tests	Extensive process automation
Code quality	Not considered or evaluated	Introduced guidelines	Full codebase adhere to guidelines
CI/CD pipeline	No tests included	Includes static and dynamic tests	Optimized tests
Feedback	No reports	Some internal and/or external defect reports	Extensive internal and external defect reports. Progress and process reports and activities
Developer motivation	No efforts have been made	Activities to enhance testing motivation have been introduced	Developers are motivated and recognized for testing activities

Table 7.1: The gray cells indicate at which maturity levels the company is assessed before framework application. When two levels are colored on the same axis, the company is partly in both. Some improvement activities are left to be performed at the light gray level before transitioning to the higher level.

The table illustrates that the testing process was fairly immature, with no axis at the "Full Implementation" level. This suggested several areas of improvement. Testing knowledge is assessed as "Limited", which could explain why the rest of the axes are not considered to a greater extent. Without knowledge about what software testing could achieve, it is not surprising that no rigorous testing process is established.

7.2 Framework Application

The framework axes applied at the company are listed below. As mentioned in the methodology chapter, not all axes could be applied during this study due to resource constraints.

- Plan definition
- Definition of test Cases
- Test Activities
- Code Quality
- CI/CD pipeline
- Developer motivation

Table 7.1 showed the diagnostic matrix of the company before the framework application, and the improvement activities for reaching the next maturity stage were applied at the selected axes. The only corresponding improvement activities that could not be performed in the covered axes due to the scope of the study were:

- Plan definition - "Define and communicate roles and responsibilities for testing activities. Assign someone responsible for the software testing"
- Code quality - "Pair programming is encouraged"
- Developer motivation - "Encourage a good relationship with management and colleagues"

The CI/CD pipeline improvement activities were only addressed on a small scale, in a separate branch. Two suggestions for utilizing build scripts were showcased to exemplify how different optimization techniques could be used (covered in the "Testing activities" axis). It was shown how GitHub action build scripts can be used to prioritize test cases. It was also demonstrated how selecting an appropriate subset of tests could be automated. Two options were provided as alternatives to choose a subset of tests, one based on the functionality that had been changed (Integration, Automation, etc) and the other based on the specific files that had been changed.

The instructions given to the developer performing the framework application contained a detailed description of how to address the testing. Requirements for the module, an example of a high-level test case, test specifications techniques and exit criteria to follow, what code quality tool to use, coding standards, and an explanation of how to use mutation testing were given. The mentioned instructions can be found in Appendix M. The code review instructions from the framework provided to the other developer reviewing artifacts can be found in Appendix N.
The customized suggestions presented to the management contained propositions covering overall decisions about the testing process. Topics involved were how to utilize GitHub actions respectively TeamCity efficiently, how to maintain structure in the testing process, and different optimization techniques in the build scripts were showcased and discussed, and recommendations regarding flakiness in the specific setup were given. The specific recommendations can be found in Appendix O. However, company-specific details such as detailed code, terms, and names are removed from the version provided in the appendix due to the company's privacy desire.

7.3 Framework Evaluation and Refinements

This section delves into the evaluation of the developed software testing framework and the refinements the evaluation initiated. The evaluation aimed to provide a holistic view of the framework's effectiveness, applicability, and impact on improving software testing practices within a collaborative company setting. The general feedback during the evaluation phase was positive. The different evaluation components hint towards a more complete testing framework, hence some areas of improvement were identified. These concerns were mainly found during the interview with the management team. The framework addressed the concerns, and a final framework was established. This section presents the evaluation results, together with the refinements performed as the suggested areas of improvement were raised. First, the insights gathered from the semi-structured interview with the developer that applied portions of the framework are presented. The interview is complemented by an analysis of the artifacts resulting from the developer's application of the framework. Following, the results of the semi-structured management interview are presented. The metrics collected post-framework application provide concrete data on the results from adopting the framework's recommendations, and they are presented in Section 7.3.4. Finally, the diagnosis matrix of the company's test process after the partial framework application is presented to illustrate the improvements.

7.3.1 Developer Interview

This section summarizes the feedback from the developer who implemented parts of the framework. Firstly, the overall reception of the given recommendations is discussed, followed by criticisms received and possible implementation challenges brought up by the developer. This section continues by discussing the adherence to the given recommendations and the observed deviations from the framework during implementation. Lastly, the feedback received about the recommended automation

testing strategies is discussed. This section ends with commenting that the framework was not refined based on this evaluation. The main takeaways are summarized in Table 7.2.

The developer provided generally positive feedback, appreciating the framework's guidance throughout the testing process. He noted, "It was useful to have a guide to follow". The recommendations were deemed complete, as the developer was not missing any specific steps. He expressed a willingness to continue using the framework, highlighting its role in enhancing code quality through testing, although at the expense of time. A highlight from the interview was the developer's satisfaction with the high-level test cases, which he deemed to facilitate organizing the work and provide a structured approach to testing. However, the developer expressed concern about whether others would deem this step too time-consuming and, therefore, skip it. The developer expressed that adopting the framework would lead to better testing of the codebase. Still, he also mentioned that the company needs more time to conduct thorough testing. He suggested that "on a small scale," certain parts could be effectively implemented. He also expressed how he felt more confident about the code after testing, believing more testing would provide a sense of security for developers.

Regarding the results from the framework implementation, the developer was generally pleased with the resulting test cases. Writing tests helped him assess the quality of the code, finding potential issues he would have liked to fix. Furthermore, the developer expressed how the framework encouraged motivation, like following a "to-do list," making the testing process more engaging.

While the feedback was predominantly positive, the developer identified specific challenges, particularly with ReSharper's standard settings not matching the company's recommended conventions. Additionally, the large codebase and previous lack of adherence to a coding standard posed difficulties in utilizing ReSharper effectively. This mismatch exemplifies how not all recommendations in the framework fit all companies and the great need to adapt the recommendations to the specific context. He expressed his willingness to recommend the framework to others, emphasizing its comprehensive coverage of testing aspects and its role in ensuring product functionality. The developer underscored the framework's accessibility, indicating that it did not present a high barrier to entry for him but that it could be a threshold for others lacking testing knowledge. This distinction highlights a crucial aspect of the framework's deployment; its usability is directly affected by the familiarity of those applying it with testing principles.

The developer's adherence to the given recommendations is vital to assess since it affects the relevance of his feedback. It will now be covered. The developer followed the framework's guidance on high-level test cases and unit testing but faced difficulties with integration testing due to time constraints and the necessity of setting up database connections. He employed techniques such as "breaking the system" and found equivalence classes helpful in generating a broad set of input data for testing. Nevertheless, it is interesting that he deliberately chose not to test all the test cases he could come up with, considering that certain cases are unlikely to occur in a real-world implementation. He pointed out that "happy testing" should also be highlighted as important since it is crucial to ensure the system works as intended.

This is already covered in the framework as it prompts testing both valid and invalid input data, so this feedback did not result in any changes to the framework.
The developer also shared valuable insights into his experience with mutation testing. The developer noted, "Testing by mutating the code and checking that the tests did not pass proved beneficial, especially when writing tests after the fact". Despite recognizing the utility of mutation testing in evaluating the test cases' effectiveness, the developer expressed a preference for Test-Driven Development (TDD). He acknowledged that while mutation testing remains valuable, its importance diminishes in a TDD context because "the tests would have failed from the beginning," highlighting the proactive detection of issues inherent in TDD. His favor for TDD is coherent with the framework as it suggests that the plan definition could include TDD if it suits the specific context.
The developer acknowledged he did not follow every step in the recommendations thoroughly. The Exit criteria were not fully fulfilled. He did express, "I should have followed the Exit criteria better; I missed commenting on my test." He also expressed that the requirements given in the Exit criteria were reasonable and that commenting on the code could help explain complicated flows.
In summary, the framework was highly appreciated by the developer and did not receive any concrete suggestions for improvements, hinting towards a more complete framework. However, he pointed out some challenges faced during the application process. Nevertheless, these challenges did not result in any refinements of the framework since the challenges were deemed too context-specific to be relevant for the broad audience of the framework.

Category	Feedback
Overall Reception	• Framework provided useful guidance • No concrete suggestions for improvement were received • The challenges faced were context-specific
Positive Feedback	• The given recommendations were complete, with no missing steps • The framework enhances code quality through testing • The use of high-level test cases was considered to provide a structured approach to testing • The testing activities resulted in increased developer confidence and a sense of security • Positive to the use of Mutation testing • The importance of Exit Criteria and code comments was acknowledged
Criticism and Challenges	• Concerns about time-consuming steps that others would potentially skip • The company lacks time for thorough testing, but the framework could be adopted on a small scale • The large code base and previous lack of coding standards pose difficulties for testing activities • The framework's usability could be affected by the developers' previous testing knowledge • Test-driven development (TDD) was preferred for proactive issue detection

Table 7.2: Summary of feedback received during Developer Interview.

7.3.2 Artifact Evaluation

The artifacts resulting from the developer implementing parts of the framework were two test methods and corresponding high-level test cases. In this section, the artifacts are evaluated to assess how well they mirrored the framework recommendations and the subjective experience of the developer. The evaluation of the artifacts is centered around five key aspects:

- **Adherence to Requirements:** The artifacts partially adhere to the specified requirements. The functional requirements for both tested methods were thoroughly assessed; however, the non-functional requirements were not explicitly addressed.
- **Correctness of chosen test levels:** The selection of test levels for the evaluated methods demonstrates a clear understanding of their scope, with both methods appropriately tested at the unit level. This decision is justified, considering the nature of the methods to be tested. Nevertheless, other recommended methods to test were omitted, given they required testing at the integration level.
- **Completeness of high-level test cases:** The high-level test cases generated during the evaluation process are comprehensive, adhering to the structured provided format of an objective, test steps, and expected results.
- **Equivalence class partitioning and Boundary value analysis:** The implementation of equivalence class partitioning and boundary value analysis was not fully realized. Certain edge cases, such as null or empty input parameters, were overlooked. An example is how testing a long string as input was disregarded since the case was considered unlikely to happen in a real-world scenario. This oversight suggests a potential for undetected errors in edge case scenarios, undermining the robustness of the artifacts.
- **Adherence to Exit Criteria:** The artifacts did not satisfactorily meet all the exit criteria. The lack of comments within the test methods and incomplete coverage of equivalence classes and boundary conditions indicate a deviation from the established guidelines. This deviation reveals a gap between the developer's intended testing strategy and the actual implementation. The developer acknowledged during the interview that he did not thoroughly follow the exit criteria to consider his testing efforts completed.

While the artifacts exhibited good quality, there was a gap between the framework recommendations and their implementation. Aligning more closely with the framework recommendations would have ensured a more thorough and effective validation of the system under test. It is essential to notice that the time available for this implementation was limited, which potentially affected the resulting artifacts.

7.3.3 Management Interview

The general perception expressed during this interview was positive. All participants clearly expressed the relevance of the recommendations. However, some were deemed more applicable than others. The aspect of resource limitations was a topic that permeated the whole interview, which was no surprise due to the nature of an SME. This appearance justifies the added stage of contextual arguments in the framework application process. During the interview, each recommendation was discussed separately in depth. The discussions are summarized in Table 7.3. Certain feedback initiated refinements of the framework when the feedback was not too context-specific. The refinements are:

- Advice about hindering old habits was added to the Testing Tools axis
- The chosen sets of tests that run on build scripts should be updated regularly to keep the test process updated was added to the CI/CD Pipeline axis
- The motivation of the Plan Definition axis was extended with the relevance of retrospective meetings from the company perspective

Category	**Feedback**
Overall Reception	• Generally positive feedback, all recommendations were deemed relevant, though some more applicable than others
Recommendation #1	• Agreement on the importance of clear file naming and organization • It was suggested to ensure following conventions by automatic checks, which is already proposed in the framework
Recommendation #2	• Well received in theory but practical challenges due to organization politics • There was resistance to introducing Pull Request in the standard workflow due to perceived delays in feature delivery • The resistance to PR's could be due to unfamiliarity with working with git. Discussing this again could lead to other decisions • Suggestion to introduce a pre-push command was rejected due to long building time • If they would begin to use PR's they would like to lock the main so that it would not be possible to push to it, to hinder old habits

Recommendation #3	• Well received, agreeing on the need to have independent test cases • The time necessary to implement this recommendation was acknowledged, but it was deemed justified by the long-term benefits
Recommendation #4	• Well received, agreeing on the need to have higher code coverage • It was suggested to assign simpler tasks, such as writing unit tests, as summer jobs
Recommendation #5	• The recommendation was highly encouraged • It was proposed to assign responsibility for regular review of test failure statistics
Recommendation #6	• The general recommendation of running a subset of tests was well-received • It was preferred to run a subset of tests based on the edited files • Test subsets based on branch naming were deemed unreliable
Recommendation #7	• The recommendation was praised for its applicability and value in prioritization test cases

Table 7.3: Summary of feedback received during Management Interview.

7.3.4 Software Repository Mining

Table 7.4 illustrates the results of mining the module under test during the framework application. Fewer metrics were gathered after the framework application compared to prior framework application, which is motivated in Section 4.6.3.

Metric	Quantity
Documentation of Tests/Code	0
Number of requested changes /comments in code review	5 requested changes. Comprehensive feedback on the overall code. The comment consisted of around 120 words.
Code Quality Issues	140 per 651 loc

Table 7.4: Metrics from mining a module of the repository after framework application.

The metric Documentation of Tests/Code quantifies what the developer confessed during the interview; he did not comment on the code he produced even though the framework suggested that he do. This leaves the aspect of commented code not improved by the framework. However, it could be the case that this result is not a consequence of the framework but rather a consequence of other intruding factors in the field experiment. For example, the developer was under time pressure, leaving stress a prominent factor for his behavior.
The Number of requested changes /comments in the code review shows a drastic improvement. The number of requested changes is increased compared to before the framework application, where zero requested changes were identified. Additionally, the 120 words compared to two in the previous code reviews indicate a more delicate review when using the framework.
Finally, it is difficult to tell if the amount of Code Quality Issues has decreased with the framework applied. If comparing with the code quality issues for component number 1 before the framework application, which had 110 per 663 loc, there is no improvement, rather an impairment with 4.9%. However, it suggests a clear improvement compared to component number 2 prior to framework application, which had 113 per 280 loc, making the module after framework implementation 20% better in terms of code quality issues.

7.3.5 Diagnosis after Framework application

Table 7.5 shows the diagnosis matrix evaluated after the framework application. It highlights how the company reached at least one maturity stage higher than before conducting the improvement activities in the respective axis.

Axis	**No Implementation**	**Partial Implementation**	**Full Implementation**
Testing knowledge	Limited	High-level understanding, not maintained	Continuously shared and maintained within the organization
Plan definition	No plan	Informal plan	Formal Plan. The plan is continuously maintained and evaluated.
Affected test levels	No testing	Deliberate use of test levels. Mainly unit testing, some integration and system testing	Comprehensive unit, integration and system tests and higher level tests
Definition of test cases	No definition of test cases	High-level definition of test cases	Fully specified test cases
Test activities	No testing	Static or dynamic tests. Some testing techniques applied	Static and dynamic tests combined. Carefully selected testing techniques
Traceability	No traceability	Partial traceability	Full traceability
Test environment	No specific testing environment	Managed and controlled environment	Testing in a suitable environment
Testing tools	Non-specific tools	Specific tools for monitoring and executing tests	Extensive process automation
Code quality	Not considered or evaluated	Introduced guidelines	Full codebase adhere to guidelines

CI/CD pipeline	No tests included	Includes static and dynamic tests	Optimized tests
Feedback	No reports	Some internal and/or external defect reports	Extensive internal and external defect reports. Progress and process reports and activities
Developer motivation	No efforts have been made	Activities to enhance testing motivation have been introduced	Developers are motivated and recognized for testing activities

Table 7.5: Note that three improvement activities, mentioned in Section 7.2, were overlooked.

7.3.6 Framework Evaluation Summary

The framework's implementation marked a shift towards more structured and effective testing practices. The feedback was generally positive, highlighting the framework's utility in providing clear guidelines. Post-implementation, there was an observable improvement in the testing process, with six axes advancing in their maturity level. However, not all axes could be implemented and evaluated, suggesting areas for future work. Based on the feedback and challenges encountered, several refinements were made to the framework. The refinements were not extensive as the framework was found to cover most of the necessary aspects already, demonstrating that the framework starts to convey toward completeness. These results hint toward a framework that not without its challenges, promises to be applicable and contribute to the industry.

8

Discussion

In this chapter, the method and result of this study are critically discussed. First, Research Question 1 (RQ1) and Research Question 2 (RQ2) are examined by discussing the final framework and how it responds to the questions but must not necessarily be the only possible solution. Following, the methodology is deliberately addressed in Section 8.2. How the approach was sometimes reconsidered to adapt to the context is discussed in Section 8.3. Research Question 3 is examined in Section 8.4 when the impact of the refined framework while applying it to the participating company is discussed. The subsequent section approaches threats to validity by addressing reliability, conclusion, internal, construct, and external validity. Following, Section 8.6 delves deeper into the aspect of generalizability by discussing the analytical generalizability of the findings. The chapter ends with encouragement for future work.

8.1 Final Framework

This study resulted in a testing framework for Small and Medium-sized Enterprises within the software developer industry. The evaluation highlights that practitioners in the industry encourage the framework. The framework relies heavily on a testing framework proposed by Argüello *et al.* [1]. Their framework received positive feedback from an expert group, which makes it a stable foundation for extension. This study complements their framework evaluation by applying part of the final framework in a real context. The final proposed framework is closer to a complete testing framework than the foundational framework, including human aspects, previously overlooked technical aspects, and further adjustments to the SME context. However, it can not be guaranteed to be fully complete. This is mainly because of the limited literature in the area of human aspects related to software testing.

BSE principles and human aspects were integrated into the testing framework from different angles. It was addressed in the existing axes when appropriate gaps were found. This is reasonable since human aspects affect every software development activity and is not an isolated topic that should be addressed in isolation. BSE is an emerging field of research, and the included BSE aspects are probably not the exhaustive list affecting the software testing process. This specific area is therefore encouraged to delve deeper into.

The analysis of the participating company, together with the literature review, clearly highlights the need for clear guidance in the software testing process, which underscores the importance of the resulting framework of this study. The study has

outlined twelve axes fundamental to consider when improving the software testing process. The outlining and definition of the axes are valuable for software companies since software testing is an emerging activity that needs to be streamlined as society expects high quality of today's software systems.

The final framework includes four axes added by this study: Testing knowledge, Code Quality, CI/CD Pipeline, and Developer Motivation. All of these were addressed since the analysis of the participating company revealed these prohibited the maturity of their testing process. However, this raises the question as if conducting this study in collaboration with another company, other axes might have been suggested instead. This reasoning motivates similar studies to be performed in different contexts.

Even though the framework aims to be concrete and hands-on, it is still abstract and general in some aspects. The abstractness prohibits the ease of applying it. Many aspects of the framework are context-specific, which makes it impossible to give concrete advice that suits several contexts. For example, the Plan definition axis needs to be customized depending on the work management at the specific site. A waterfall setting requires certain adaptions unsuitable for an Agile workflow. The abstractiveness could be solved by limiting the scope to be customized for a particular context. However, this was deemed to not provide sufficient value to either the research field or the industry to conduct a rigorous study for such a limited target.

Despite the framework being tailored for SME environments, some of the axes and their compatibility in the SME context could be debated. For instance, the Plan Definition axes suggest having a formalized testing plan, and the axis Definition of Test Cases recommends having test cases with complete specifications to reach the highest maturity level. These recommendations could be considered to conflict with the limited resources and the fast development pace in SME environments. However, the resulting framework aims towards flexibility, particularly by adding the Contextual adjustment step. This step allows companies to tailor improvement activities to their specific context, making the framework more practical and applicable to various SME settings. Moreover, the framework recognizes that the optimal goal of reaching the highest maturity level may not be reasonable for all contexts. Nevertheless, the recommendations mentioned were kept, given that the original framework was already tailored for SMEs and received positive feedback. Additionally, given the variety of companies that fall under the SME categorization, it is reasonable to assume some organizations will aim toward more formal and ambitious goals. The intent is not to enforce rigid standards but to provide a structured approach that can be scaled and modified as needed.

Ending this section, it is worth addressing the critics of CMM and TMM families and how it does not apply to the proposed framework. The main issue with CMM is that it is too complex for SMEs. It requires more resources than usually available in an SME context and is too documentation-heavy to be easily applied. The CMM and TMM families encompass five maturity levels and around 20 process areas in which to mature, with a vast amount of documentation for each. This results in several hundreds of pages to cover. The proposed framework of this study only consists of 3 maturity levels and 12 axes with concise documentation, ending up

with around 30 pages of documentation. This is heavily condensed in compression to the CMM and TMM families, with the ambition to fit the timely resources of SMEs. Additionally, it is observed that companies commonly fail to mature in the CMM and TMM levels, probably because of the staged approach [14]. The proposed framework of this study aims to ease the process of maturation by providing a continuous approach, where all axes must not be in the same maturity stage. As mentioned before, the Contextual adjustment step was also added to the implementation process, addressing a common challenge faced by SMEs: lack of resources. These adaptations, proposed by the resulting framework, are considered to facilitate testing process improvement activities compared to the CMM and TMM families.
Our final proposed framework has been evaluated to ensure clarity and that it can be easily followed by practitioners in the field. The contextual adjustment step, which includes a stage of resource assessment, has been added to the cycle of the framework application to include resources as the foundation of improvement activities. Additionally, the framework has been proposed in a lightweight way, aiming to leave only the core aspects of the framework and adding theory, examples, or other interesting artifacts as appendices that easily can be overlooked. The framework has only been partially applied in a natural context. Nevertheless, the application shows that it is possible to mature in the framework axes in a short time if the company adheres to it. The application phase in this study was only a few weeks with one developer, and the corresponding testing process matured in six axes.

8.2 Research Methodology

In general, the research methodology of this study was a field study approach where a specific company served as the center of the work. Field studies encompass any research efforts undertaken in a specific, real-world environment to examine a particular phenomenon within software engineering [63]. To investigate all of the three research questions of the study, the field study approach was complemented with elements of Action Research during the implementation, validation, and refinement phases. Action Research is described as an iterative cycle where a researcher aims to try out a theory in a real-world context, gain feedback from this experience, and modify the theory based on the input obtained [64].
The analysis phase of the study aimed at developing a deep understanding of the current testing process at the company and the factors influencing this process. Following the Field study approach served this goal well, given that the authors intended to explore the natural setting without disturbing it.
On the other hand, during the implementation phase, the authors of this study manipulated the testing practices to observe an effect. The results from the implementation phase were evaluated, and feedback was obtained. Finally, the proposed framework was refined. This process can best be described as conducting Action Research. The decision to employ Action Research was considered suitable because it facilitated a participatory research environment where both researchers and practitioners could collaboratively engage in experimentation and learning. This collaborative stance was instrumental in ensuring that the modifications to the testing practices were grounded in the actual needs and realities of the company, thereby

enhancing the practical relevance and applicability of the research findings. Furthermore, the iterative nature of Action Research was ideally suited to the objectives of this study, which sought not only to assess the effectiveness of the proposed framework but also to refine it based on empirical evidence.
Even though the chosen methodology was considered appropriate, both the Field study and Action Research approach present inherent limitations, especially threatening the generalizability of the study. This is further discussed in Section 8.5.5. However, the research methodology was chosen to complement and balance the previous work on the original framework. Argüello *et al.* [1] evaluated the framework by a group of experts both from the knowledge area and from the industry, expressing their opinions about the feasibility of the framework. The framework was not practiced in a real context in their study, and they were further encouraged to test the proposal in a real environment as future work. In turn, our study focuses on a real context application and does not put as much emphasis on the width of the assessments to complement the previous work. Instead, an in-depth and detail-rich evaluation could be offered by our setting. However, since we enhanced the previous framework, a comprehensive sample study should be conducted to guarantee generalizability since it can not rely solely on the previous evaluation. Nevertheless, it was neglected in this study due to time constraints.

8.3 Methodological Adjustments

Some methodological adaptions were made during the study due to the research context. The adjustments are summarized in Table 8.1. Each adaption is presented in the text below in the same order as in the table.

Plan	**Actual implementation**
Observe developers during testing in the company analysis	Inspection of commits during software repository mining
Strong focus on software metrics and statistical generalizability	A few extracted metrics, more emphasis on analytical generalizability
Continuous assessment of developer experience during framework application	Thorough evaluation interview after framework application
Observe developer during framework application	Evaluate resulting artifacts after framework application

Table 8.1: Summary of the methodological adjustments performed in the study.

The plan to analyze the company's initial test process included observations of the conducted testing. The ambition was to witness how the tester worked in its natural context. However, after the interviews were performed, the methodology was reconsidered. This was because the interviews revealed that the typical workflow

in the company does not include any testing activities besides developers running the software and interacting with it, simulating end-user behavior without any guiding heuristics besides intuition. Because of this, conducting an observational study would likely not provide any valuable information compared to the effort administrating the method. However, due to this omitting, we may miss out on information not revealed during the interviews since humans are not guaranteed to explain how they work correctly [96]. To mitigate this concern and not solely rely on introspection, we closely inspected commits in GitHub during the software repository mining to verify no trace of testing activities, with confirmed assumptions.

Another methodological adjustment of this study is the usage of software metrics. At the beginning of the study, the plan was to base the verification of the framework partly on metrics. It was desired to show quantitative data justifying the effectiveness of the framework. This was performed, however, to a lesser extent than expected. As the study proceeded, it was revealed that guaranteeing a causal relationship between the framework application and the metrics is unfeasible due to the nature of a cross-sectional study. The results could be a consequence of other factors present when eliciting the metrics rather than solely the framework application. To guarantee causality, the variables must be observed over time and not only in a snapshot. For example, a metric we wanted to include was "Test Suite Execution Time". However, it was decided to be neglected since it does not provide value in a snapshot, since other factors are affecting the execution time, such as hardware resources of GitHub actions, caching, and the number of processes that run in parallel. Additionally, the fact that all axes could not be tested in this study due to resource constraints made some metrics less accessible. For example, we planned to evaluate the metric "Flakiness Score", since it could be addressed by the framework suggestion about mocking the test database. However, the participating company did not want to prioritize that suggestion in the application phase, and the suggestion was therefore omitted. Therefore, measuring the flakiness score after the framework application would not provide any value since no actions were performed to improve the metric.

Furthermore, it had originally been planned that continuous assessment would be performed during the framework implementation phase. It was thought developers would be asked to log their experiences via a short questionnaire. This step was meant to capture detailed and timely insights and mitigate the risk of recall bias. However, the framework was implemented by a single developer who focused on testing just one specific module of the codebase. The testing activities involved only one class and were completed in a concise time frame, spanning just one day. Given the limited scope of this testing effort and its short duration, the planned continuous assessment was deemed unnecessary and ultimately not executed. This decision was based on the rationale that the limited extent of testing, carried out over such a brief period, significantly reduced the potential for memory-related biases to influence the findings.

It had also been planned to carry out an observational study during the implementation phase to evaluate the framework. However, it was later decided that a better approach would be to assess the resulting artifacts since they would mirror how the recommendations were followed. The researchers also wanted to alleviate the

occurrence of the Hawthorne Effect, which refers to the alteration of behavior by the subjects of a study due to their awareness of being observed.

8.4 Impact of the final framework applied in a real-world setting

The results obtained in this study are promising since the evaluation shows that the proposed framework made a positive impact on the testing process. Positive feedback was given from practitioners, and the test cases produced following the framework were assessed to be of significantly higher quality than previous test cases. Additionally, managers at the participating company confirmed that the framework was also sound on an organizational level. However, the impact of the proposed framework in a real-world setting is difficult to claim due to the cross-sectional research setting of this study. A cross-sectional study can not guarantee stability in the results over time, making the results of this study unsure. Additionally, only one person applied the framework, leaving the data too scarce to guarantee conclusions.
It must be noted that when involving humans in an evaluation, other factors than the object under evaluation can affect the results. As mentioned in Section 7.3.4, stress could be a factor clouding the process of the framework application. For example, the developer got instructions on how to address the application. Nevertheless, he did not fulfill all the tasks. This leaves the results incused by his behavior, and the result might not be a consequence of the framework itself. Still, the results indicate that the parts he fulfilled received a positive evaluation.

8.5 Threats to Validity

Every research study has potential threats to validity. Some research settings are especially exposed to validity threats. Among them is the case study methodology used in this study. It is important to identify the threats, to be able to apply countermeasures, and to be transparent to the audience about the limitations possessed. Research constraints such as context and provided resources can hinder the ability to eliminate identified threats. In this study, the analysis of validity threats was guided by a checklist by Runeson and Höst [97].
The fact that this study builds upon an existing framework, makes the result vulnerable to the threats in the underlying framework. The underlying framework could include situational factors that are not present in the context in which this study was conducted. However, to mitigate the weaknesses of the foundational framework, we verified that it only included relevant factors, with support from literature and the experiences at the company. The axes were slightly edited in cases where we found the suggestion unsuitable. Still, the underlying framework has not been tested in isolation in a natural context, which leaves a concern of confidence in the findings of it. In the following sections, we transparently acknowledge specific threats to this study.

8.5.1 Reliability

Reliability concerns the extent to which the data collection, analysis, and interpretation of results are independent of the specific researchers. In this study, some threats to this aspect of validity are present. Interviews were broadly used during this study's data collection and evaluation process. When relying on interviews there is a risk that interview questions are unclear or worded in a way open to multiple interpretations, leading respondents to answer the questions differently. To mitigate this threat, the interview questions were evaluated through pilot testing. This process helped identify ambiguities, and the feedback received could be used to refine the question for clarity. Additionally, all interviews were semi-structured, allowing the interviewer to add questions or discussions ad hoc as they appeared appropriate in relation to the respondent's answers. Therefore, the interviews are difficult for another researcher to replicate, who acts with other respondents. The respondents' answers are also left to the researcher's perception, making it a threat to reliability. However, both authors of this study independently analyzed the recordings of the meetings to at least use two interpretations when claiming the results.

Another aspect that threatens reliability is the methodology used to categorize the collected data into themes during the thematic analysis. There is a risk that different researchers would have coded the same text differently, given the width of the subjects discussed during the interviews. To mitigate this threat, the two researchers of this study worked with the same material in parallel by coding the data separately. This process aimed at initially diversifying the interpretation of the data to capture a broader range of insights and reduce individual bias. The researchers then combined their findings in an iterative process to arrive at the final themes. This stage allowed for identifying any discrepancies and finally reaching a consensus. The iterative reconciliation helped in refining the themes to ensure they robustly represented the collected data. It is also worth mentioning the themes themselves are not directly correlated to the changes made to the original framework, which supports the claim that slight variations in the coding of the collected data would not have signified considerably different results in this study.

8.5.2 Conclusion Validity

To be able to draw reliable and correct conclusions, the data sample needs to mirror the real context identically. However, this is seldom possible due to the nature of a data sample. In this study, the testing process was reflected with the help of interviewed employees at the company, which leaves the description of the test process potentially subjective. Ideally, the participants should encompass a large random heterogeneity to achieve high conclusion validity. However, this is not the case at the company, where the subjects vary in degree of coding experience and knowledge. To try to give a nuanced depiction of the process under investigation, purposive sampling was used, as motivated in Section 4.3.1. It is a common suggestion to include more subjects to increase the statistical power, which also applies to this study. We decided to stop at a stage where data saturation was reached.

8.5.3 Internal Validity

When performing the analysis of the testing process prior to the framework application, quality metrics were assessed with the help of the software tool Resharper. The analysis is thereby affected by the tool and how it identifies and classifies quality issues, which can be seen as a threat to validity if the tool is not optimal. Other tools might have received different results. However, Resharper is largely acknowledged and provided by a trustworthy company, motivating the tool's usefulness.
The other part of the company analysis, the interviews, also includes threats to internal validity. In fact, the questions prepared for the interviews affect the topics discussed during the sessions. This, in turn, affects the assessment of the company's testing process and the final framework, as the interviews served as a data source for the framework. To reduce the intrusiveness, the interviews were semi-structured, allowing them to follow the participant's initiatives. However, some aspects were almost not touched upon in the interviews. This could be an affecting factor as to why some aspects were not edited notably from the original version. One example is the axis Traceability. Nevertheless, if concerns were not raised during the interviews, they are likely not critical, making the potentially missed revisions of the framework less prominent.
Regarding the workshop performed during the framework development, how it was executed might have affected the outcome and threatened internal validity. The axes were distributed so that each was discussed by one group. This leaves the quality of feedback to the group discussing the specific axis and also makes personal preferences shine unevenly on the axes. For example, one group gave the same improvement idea on all their assigned axes; to give more hands-on suggestions. This leaves the question of whether they had also left the same feedback on other axes if reviewing them. This could have been mitigated if all groups had reviewed all axes, but due to time constraints, this was omitted, and instead, it was assessed as sufficient to allow for an open discussion about each axis. The discussions were engaging and mainly limited by the timeframe. Therefore, a suggestion for future work is to conduct similar workshops to mine the framework even further. However, the flow of the discussions showed a pattern of strong opinions, and significant issues were raised in the beginning, moving on to a more detailed discussion. This motivates that the most critical feedback was received, even though the time hindered it from being complete. To cover for the compressed time, the participants were encouraged to contact the authors to express additional thoughts.
The internal validity of this study is also threatened by the fact that only one developer participated in testing the framework. This person brought his skills, opinions, and routines to the workflow, which could affect the result. For example, the code quality was improved after the framework application compared to one of the code modules analyzed before the framework application. This does not necessarily be an outcome of the framework application since the developer might have good practices in code quality compared to the developer who produced the code analyzed in the company analysis before the framework application, causing the result to look better than it is. This is an example of selection bias, where groups compared in a study (code by two different developers in this case) are not comparable.

Selection bias could also have been affecting the study from another angle. Namely, the modules compared before and after the framework application were not identical, and the deviations among them could hinder the results from being objective. For example, the code reviews performed on the different modules might not ultimately require the same amount of requested changes, leaving the absolute metrics incomparable. The first module may not need any changes to reach a flawless pull request, making the zero requested changes optimal. (However, we found potential for improvements in the module, making the code review imperfect.) On the other hand, the module analyzed after the framework application might as well have been 20 changes away from a flawless pull request, leaving the five proposed far from optimal. So, even though the code review process was improved with the framework application, from zero to five requested changes, the absolute numbers might not reveal the truth.
Another bias threatening the internal validity that probably existed during the study is the volunteer bias. The workshop participants and the developer who conducted the framework application were all selected based on their willingness to participate. A consequence of this is that the result gathered might represent extreme opinions since people possessing these are often more likely to volunteer for a study. However, the workshop session did not confirm this threat, as most participants did not express any strong opinions and were not showing signs of extraordinary commitment.

8.5.4 Construct Validity

One of the primary challenges that undermines the whole study is the scarcity of existing research on the human aspects and their impact on the software testing field. This poses a significant threat to construct validity since the foundation this study is built upon in the mentioned research fields is far from complete. The framework probably does not cover all essential factors when considering human aspects due to the absence of complete research. Cognitive biases and motivation were the two main aspects of this study because of theoretical availability. Including two human aspects is more than has been included in previous testing frameworks and contributes to the field, even though including all relevant human aspects is not done.
The fact that this study involves human actors implies some threats to construct validity since social threats may affect the actors. For example, evaluation apprehension may cause study participants to behave differently than ordinary because they are aware they are being observed. In the context of this study, this could occur during the interviews where the participants might frame their working practices in another way since they either want to mask flaws or polish their appearance, also known as Social Desirability Bias. To mitigate this, we tried to create a comfortable environment where the participants could feel safe and clarified at the beginning of the interview that the participants were anonymous.
Ending the discussion about construct validity threats, it is worth mentioning if the company used for this study is the correct data source to form a more complete testing framework. As discussed in Section 8.1, other problems might have been identified if other companies were included in the process, resulting in a different

framework. However, the company's suitability is apparent, and no refinements were made to the framework that was not deemed generally applicable. Additionally, no prompts in the framework are purely based on personal opinions or context-specific refinements from the company, without relying on literature as well.

8.5.5 External Validity

The prominent threat to external validity in this study is already presented briefly, namely that the framework was only extended and evaluated with the help of one company, making it impossible to ensure statistical generalizability of the results. However, this threat is not surprising since the software engineering research field, in general, is struggling with the generalizability aspect, as stated by Baltes and Ralph [65]. They claim that since non-probability sampling is seldom feasible in this research domain, generalizability is consequently prohibited. They elaborate that this is because accessing a suitable sampling frame is almost impossible in SE research, making representativeness an unattainable state.
No data saturation can be ensured with only one data source, which also threatens external validity since it can not be concluded that enough data was gathered to feed the framework creation. Therefore, a clear opportunity for future work is to engage more companies in the framework validation and verify that the proposed changes suit software SMEs. For example, the participating company does not work with the Agile workflow, leaving this part of the population unverified. This is a suitable opportunity for a sample study. However, this study relies upon a previous framework evaluated by expert groups from both industry and research, which increases the generalizability.
Another limitation of this study was that the entire framework could not be applied at the company. This threatens external validity since the parts not applied at the company are not evaluated in a real-world setting.

8.6 Generalizability of the Findings

Even though the results of this study can not argue for achieving statistical generalizability, the focus can be shifted toward analytical generalizability. The insights derived from addressing Research Questions 1 and 2 can be argued to hold relevance for software testing frameworks within Small and Medium-sized Enterprises (SMEs). The investigated case study provides a substantial basis for theorizing about integrating (BSE) aspects to refine software testing processes and the necessary adaptations for existing software testing frameworks to fit the SME context.
The available body of knowledge in the BSE research broadly supports the significant impact of human factors on the practices of software engineering [10], not least in the area of software testing [11]. Integrating these human-centric aspects into existing software testing frameworks emerges as a contribution of substantial transferability across a broad spectrum of companies. Moreover, the empirical investigations conducted within the collaborating company revealed a series of challenges intricately linked to human factors that could be directly related to the BSE literature. Since the collaborating company was chosen based on purposive sampling

and fulfilled the selection criteria of being representative of other SME companies in the software engineering field, the empirical observations could be trusted to inspire improvements in the original framework.
The above-mentioned is not only true for the aspects connected to BSE, in the contrary, this approach was followed for all enhancements made to the original framework. To only mention a few, the Code Quality axis was inspired by the analysis of the testing process at the company. It could be concluded from both the interviews and the repository mining that the quality of the code was a hindrance to incorporating better testing practices. Given that the literature extensively relates quality aspects such as code modularity facilitating testability [3], the Code Quality axis was added to the framework. This is an example of how both the empirical observations and the literature lead to the proposed framework. Another example is adding the Contextual Adjustment step to the application process. Several of the empirical observations of the test process at the company had their root in well-known challenges faced by SMEs and could also be easily connected to the literature. One of those challenges was the limited resources SMEs hold. Therefore, the Contextual Adjustment step was added to account for the analysis of the available resources that can realistically be used for testing efforts. This adaptation in the framework is also highly transferable since this is a common challenge for SMEs [9].
To summarize, it is essential to note that the improvements applied to the original framework were not solely predicated on empirical observations but were also deeply rooted in existing research. This dual foundation enhances the robustness of the recommendations proposed within the framework. Given this comprehensive approach, it is reasonable to contend that the suggested modifications will likely be both applicable and beneficial for similar enterprises within the industry, thereby arguing for the analytical generalizability of the study's findings.

8.7 Future Work

As mentioned in the covered sections, there are several opportunities for improvement, and future work involving the proposed framework is encouraged. Primarily, it would benefit from being applied in more companies and evaluating its applicability. The inherent limitations in establishing a direct causal link between the framework's application and the improvements observed at the company, given the cross-sectional nature of the study, have been discussed. Considering these insights, a compelling direction for future research is conducting a longitudinal study. Longitudinal case studies yield in-depth data examining how changes develop over time, crucial for understanding the intricate interplay within software systems [98]. Such a study would offer a more nuanced understanding of how the framework influences software testing process improvements within SME environments (RQ3). By tracking the evolution of key metrics and practices before, during, and after the framework's implementation, a longitudinal approach could mitigate the confounding factors encountered in cross-sectional analysis. A study of this nature could have been conducted at the collaborating company if the necessary time and resources had been available. Additionally, only a subset of the axes were implemented and evaluated at the collaborating company, given the existing constraints for this study.

For future work, it would also be beneficial to include the rest of the axes in an implementation and evaluation process.
Another inherent limitation of this study, given the chosen methodology, is the statistical generalizability of the findings. A sample study is left as future work to tackle the current limitations on generalizability. It is also a continuous work to keep the framework up to date. The aspects of the framework are time-dependent since future technological advancements might affect the dimensions relevant to a testing framework. For example, using AI to facilitate software testing could be interesting to investigate. Additionally, the importance of human aspects when performing SPI activities can not be overstated. Future work, including a broader selection of human aspects than those examined in this study, is highly recommended.

9
Conclusion

Within this Master Thesis, a comprehensive framework facilitating the improvement of software testing processes in Small to Medium-sized enterprises has been developed. The framework is based on previous work performed by Argüello *et al.* [1]. The original framework has been extended with new aspects found to be essential for a more complete testing framework. Human aspects are rarely included in existing research in the software testing framework field and are a significant focus of this study. Mitigation techniques to common cognitive biases in relation to software testing are applied in the framework, together with concrete methods to increase developers' motivation to engage in testing activities. Some elements have been removed from the original framework when not supported by literature or the company analysis performed at the participatory SaaS company. The original framework was reformulated when objectives were found unclear, and several factors have been explained more concretely with examples, with the ambition to facilitate easy adoption of the framework for industrial companies.
The evaluation of the proposed framework suggests that it is suitable and applicable in industry, and it evidently improved the testing process of the company participating in the study. Nevertheless, it should be noted that the framework has only been partly applied to one company during this study.
The results of this study benefit software development in SMEs by providing a comprehensive testing methodology that can potentially enhance their testing process and, thereby, product quality. Research in the field also benefits since this study endeavors to bridge the gap in existing testing frameworks.

Bibliography

[1] M. Argüello, C. A. C. Pietroboni, and K. E. Cedaro, "Proposal to improve software testing in small and medium enterprises," in *Applied Informatics*, ser. Communications in Computer and Information Science (CCIS), vol. 1455, 2021, pp. 497–512. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-89654-6_35.

[2] H. Krasner, *The cost of poor software quality in the us: A 2022 report*, Consortium for IT Software Quality (CISQ), Accessed on: 2023-11-28, 2022. [Online]. Available: https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf.

[3] P. Straubinger and G. Fraser, "A survey on what developers think about testing," in *Proc. 2023 IEEE 34th Int. Symp. Software Reliability Eng. (ISSRE)*, Florence, Italy, 2023, pp. 80–90. [Online]. Available: https://doi.org/10.1109/ISSRE59848.2023.00075.

[4] V. Garousi, M. Felderer, and T. Hacaloğlu, "What we know about software test maturity and test process improvement," *IEEE Software*, vol. 35, pp. 84–92, Feb. 2018. [Online]. Available: https://doi.org/10.1109/MS.2017.4541043.

[5] D. Gelperin, "A testability maturity model," in *Proceedings of the Fifth International Conference on Software Testing, Analysis and Review*, Orlando, FL, 1996.

[6] T. Ericson, A. Subotic, and S. Ursing, "Tim- a test improvement model," *Software Testing, Verification and Reliability*, vol. 7, pp. 229–246, 1997. [Online]. Available: https://doi.org/10.1002/stvr.317.

[7] T. Koomen and M. Pol, *Test Process Improvement: A Practical Step-By-Step Guide to Structured Testing*, 1st ed. Harlow, England; Reading, Mass: Addison-Wesley, 1999.

[8] E. Commission. "Sme definition." Accessed: 2024-02-14. (2024), [Online]. Available: https://single-market-economy.ec.europa.eu/smes/sme-definition_en (visited on 02/14/2024).

[9] G. M. Kituyi and C. Amulen, "A software capability maturity adoption model for small and medium enterprises in developing countries," *The Electronic Journal of Information Systems in Developing Countries*, vol. 55, pp. 1–19, Dec. 2012. [Online]. Available: https://doi.org/10.1002/j.1681-4835.2012.tb00389.x.

[10] A. MohamadSaleh and S. Alzahrani, "Development of a maturity model for software quality assurance practices," *Systems*, vol. 11, p. 464, 2023. [Online]. Available: https://doi.org/10.3390/systems11090464.

[11] A. Deak, T. Stålhane, and G. Sindre, "Challenges and strategies for motivating software testing personnel," *Information and Software Technology*, vol. 73, pp. 1–15, May 2016. [Online]. Available: https://doi.org/10.1016/j.infsof.2016.01.002.

[12] ISO/IEC/IEEE, *Software and systems engineering – software testing – part 1: General concepts*, ISO/IEC/IEEE 29119-1:2022(E), Published 27 Jan 2022, 2022, pp. 1–60. DOI: 10.1109/IEEESTD.2022.9698145.

[13] O. Taipale and K. Smolander, "Improving software testing by observing practice," in *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering (ISESE '06)*, New York, NY, USA: Association for Computing Machinery, Sep. 2006, pp. 262–271. [Online]. Available: https://doi.org/10.1145/1159733.1159773.

[14] W. Afzal, S. Alone, K. Glocksien, and R. Torkar, "Software test process improvement approaches: A systematic literature review and an industrial case study," *Journal of Systems and Software*, vol. 111, pp. 1–33, 2016, ISSN: 0164-1212. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121215001910.

[15] R. Torkar and S. Mankefors-Christiernin, "A survey on testing and reuse," in *Proceedings of the IEEE International Symposium on Software Testing*, Dec. 2003, pp. 164–173. [Online]. Available: https://doi.org/10.1109/SWSTE.2003.1245437.

[16] E. van Veenendaal, R. Black, and D. Graham, *Foundations of Software Testing: ISTQB Certification*, 4th ed. Cengage Learning EMEA, 2019.

[17] M. Chemuturi, *Mastering Software Quality Assurance - Best Practices, Tools and Techniques for Software Developers*. J. Ross Publishing, Inc., 2011. [Online]. Available: https://app.knovel.com/kn/resources/kpMSQABPTR/toc.

[18] S. Karlsson, A. Causevic, and D. Sundmark, "Automatic property-based testing of graphql apis," in *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, Madrid, Spain, 2021, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/document/9463000.

[19] R. Su, Z. Zhang, Y. Zhou, and Y. Yao, "Test generation for mutation testing by symbolic execution," in *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security Companion (QRS-C)*, Chiang Mai, Thailand, Oct. 2023, pp. 55–61. [Online]. Available: https://ieeexplore.ieee.org/document/10430005.

[20] PITest. "Mutators - pitest." Accessed on: 2024-02-29. (2024), [Online]. Available: https://pitest.org/quickstart/mutators/.

[21] PIT, [Software], Pitest.org.

[22] M. Aniche, "Structural testing and code coverage," in *Effective Software Testing: A Developer's Guide*. USA: Manning Publications, 2022.

[23] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, Oct. 2001. [Online]. Available: https://doi.org/10.1109/32.962562.

[24] G. O'Regan, "Test metrics and problem-solving," in *Concise Guide to Software Testing* (Undergraduate Topics in Computer Science), Undergraduate Topics

in Computer Science. Cham: Springer, Oct. 2019, ch. 9, pp. 153–179. [Online]. Available: https://doi.org/10.1007/978-3-030-28494-7_9.

[25] M. Tufano, S. Chandel, A. Agarwal, N. Sundaresan, and C. Clement, *Predicting code coverage without execution*, Jul. 2023. [Online]. Available: https://www.researchgate.net/publication/372625450_Predicting_Code_Coverage_without_Execution.

[26] S. Elbaum and J. Munson, "Code churn: A measure for estimating the impact of code change," in *Proceedings of the Conference on Software Maintenance*, 1998, pp. 24–31. [Online]. Available: https://doi.org/10.1109/ICSM.1998.738486.

[27] S. Rasheed, A. Tahir, J. Dietrich, N. Hashemi, and L. Zhang, *Test flakiness' causes, detection, impact and responses: A multivocal review*, Dec. 2022. [Online]. Available: https://www.researchgate.net/publication/366005041_Test_Flakiness'_Causes_Detection_Impact_and_Responses_A_Multivocal_Review.

[28] GitHub. "Github actions." Accessed on: 2024-02-15. (2024), [Online]. Available: https://github.com/features/actions (visited on 02/15/2024).

[29] P. Duvall, S. Matyas, and A. Glover, *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007.

[30] D. Farley and J. Humble, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, Jul. 2010, ISBN: 9780321670250.

[31] JetBrains. "Teamcity." (2024), [Online]. Available: https://www.jetbrains.com/teamcity/ (visited on 02/15/2024).

[32] P. Lenberg, R. Feldt, and L. G. Wallgren, "Behavioral software engineering: A definition and systematic literature review," *Journal of Systems and Software*, vol. 107, pp. 15–37, 2015. [Online]. Available: https://doi.org/10.1016/j.jss.2015.04.084.

[33] R. Mohanani, I. Salman, B. Turhan, P. Rodriguez, and P. Ralph, "Cognitive biases in software engineering: A systematic mapping study," *IEEE Transactions on Software Engineering*, vol. 46, no. 12, pp. 1318–1339, 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8506423/.

[34] C. França, F. Q. B. da Silva, and H. Sharp, "Motivation and satisfaction of software engineers," *IEEE Transactions on Software Engineering*, vol. 46, no. 2, pp. 118–140, Feb. 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2842201.

[35] J. R. Hackman and G. R. Oldham, "Development of the job diagnostic survey," *Journal of Applied Psychology*, vol. 60, no. 2, pp. 159–170, 1975. [Online]. Available: https://psycnet.apa.org/doi/10.1037/h0076546.

[36] E. A. Locke, "Toward a theory of task motivation and incentives," *Organizational Behavior and Human Performance*, vol. 3, no. 2, pp. 157–189, 1968. [Online]. Available: https://doi.org/10.1016/0030-5073(68)90004-4.

[37] F. Herzberg, B. Mausner, and B. B. Snyderman, *The Motivation to Work*. Revue Française de Sociologie, 1959, Review in Revue Française de Sociologie, 1.

[38] R. M. Steers, R. T. Mowday, and D. L. Shapiro, "Introduction to special topic forum: The future of work motivation theory," *Academy of Management Review*, vol. 29, pp. 379–387, 2004. [Online]. Available: https://doi.org/10.2307/20159049.

[39] T. Hall, N. Baddoo, S. Beecham, H. Robinson, and H. Sharp, "A systematic review of theory use in studies investigating the motivations of software engineers," *ACM Transactions on Software Engineering and Methodology*, vol. 18, no. 3, 10:2, 2009. [Online]. Available: https://doi.org/10.1145/1525880.1525883.

[40] T. A. Majchrzak, "Best practices for the organizational implementation of software testing," in *Proceedings of the 43rd Hawaii International Conference on System Sciences (HICSS)*, Conference held on January 5-8, 2010, Hawaii, USA, Jan. 2010, pp. 1–10. [Online]. Available: https://doi.org/10.1109/HICSS.2010.83.

[41] H. Shah, M. J. Harrold, and S. Sinha, "Global software testing under deadline pressure: Vendor-side experiences," *Information and Software Technology*, vol. 56, pp. 6–19, Jan. 2014. [Online]. Available: https://doi.org/10.1016/j.infsof.2013.04.005.

[42] P. Baskar, "A study on the impact of rewards and recognition on employee motivation," *International Journal of Science and Research (IJSR)*, vol. 4, Jan. 2013.

[43] T. Toroi, A. Raninen, and L. Väätäinen, "Identifying process improvement targets in test processes: A case study," in *2013 IEEE International Conference on Software Maintenance*, Eindhoven, Netherlands, 2013, pp. 11–19. [Online]. Available: https://doi.org/10.1109/ICSM.2013.12.

[44] S. Oloruntoba. "S.o.l.i.d: The first five principles of object-oriented design." Accessed on: 2024-02-19. (Nov. 2021), [Online]. Available: https://www.digitalocean.com/community/conceptual-articles/s-o-l-i-d-the-first-five-principles-of-object-oriented-design.

[45] H. Singh and S. I. Hassan, "Effect of solid design principles on quality of software: An empirical assessment," 4, vol. 6, 2015, pp. 1321–1324.

[46] L. Williams, R. R. Kessler, W. Cunningham, and R. Jeffries, "Strengthening the case for pair programming," *IEEE Software*, vol. 17, no. 4, pp. 19–25, Jul. 2000. [Online]. Available: https://doi.org/10.1109/52.854064.

[47] M. C. Paulk, B. Curtis, M. B. Chrissis, and C. V. Weber, "Capability maturity model, version 1.1," *IEEE Software*, vol. 10, no. 4, pp. 18–27, Jul. 1993. [Online]. Available: https://doi.org/10.1109/52.219617.

[48] TMMi Foundation. "Test maturity model integration (tmmi) release 1.0." Accessed on: 2024-02-08. (2012), [Online]. Available: https://www.tmmi.org/tmmi-model/.

[49] D. Karlström, P. Runeson, and S. Nordén, "A minimal test practice framework for emerging software organizations," *Software Testing, Verification and Reliability*, vol. 15, pp. 145–166, Sep. 2005. [Online]. Available: https://doi.org/10.1002/stvr.317.

[50] Q. Lin, "Research on software testing technology and methodology based on revised and modified capability maturity model: A novel approach," in *Proceed-*

ings of the 2015 International Conference on Education Technology, Management and Humanities Science, Atlantis Press, Mar. 2015, pp. 1396–1400, ISBN: 978-94-62520-61-5. [Online]. Available: https://doi.org/10.2991/etmhs-15.2015.309.

[51] F. J. Pino, F. García, and M. Piattini, "Software process improvement in small and medium software enterprises: A systematic review," *Software Quality Journal*, vol. 16, pp. 237–261, Jun. 2008. [Online]. Available: https://doi.org/10.1007/s11219-007-9038-z.

[52] A. Raninen, J. J. Ahonen, H.-M. Sihvonen, P. Savolainen, and S. Beecham, "Lappi: A light-weight technique for practical process modeling and improvement target identification," *Journal of Software: Evolution and Process*, vol. 25, no. 9, pp. 915–933, Sep. 2013. [Online]. Available: https://doi.org/10.1002/smr.1571.

[53] M. Hasan *et al.*, "A unified testing process framework for small and medium-sized tech enterprises with incorporation of cmmi-svc to improve maturity of software testing process," in *Proceedings of the 17th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, 2022, pp. 327–334. [Online]. Available: https://doi.org/10.5220/0011037400003176.

[54] H. Saiedian and N. Carr, "Characterizing a software process maturity model for small organizations," *SIGICE Bulletin*, vol. 23, no. 1, pp. 2–11, Jul. 1997. [Online]. Available: https://doi.org/10.1145/1031167.1031168.

[55] E. V. Veenendaal and B. Wells, *Test maturity model integration (tmmi)*, TMMI Foundation (http://www.tmmifoundation.org), 2012.

[56] N. P. G. Boumans, H. J. de Jong, and S. M. Janssen, "Age differences in work motivation and job satisfaction. the influence of age on the relationships between work characteristics and workers' outcomes," *International Journal of Aging and Human Development*, vol. 73, pp. 331–350, Feb. 2012. [Online]. Available: https://doi.org/10.2190/AG.73.4.d.

[57] I. Salman, "Cognitive biases in software quality and testing," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, Austin Texas, USA, May 2016, pp. 823–826. [Online]. Available: https://doi.org/10.1145/2889160.2889265.

[58] M. Leventhal, B. E. Teasley, and D. S. Rohlman, "Analyses of factors related to positive test bias in software testing," *International Journal of Human-Computer Studies*, vol. 41, pp. 717–749, Nov. 1994. [Online]. Available: https://doi.org/10.1006/ijhc.1994.1079.

[59] G. Calikli and A. Bener, "Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, Sep. 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1868328.1868344.

[60] G. Calikli, A. Basar, B. Caglayan, and A. Tosun, "Modeling human aspects to enhance software quality management," *International Conference on Information Systems, ICIS 2012*, vol. 5, pp. 4470–4480, Jan. 2012. [Online]. Available: https://www.researchgate.net/publication/282675080_Modeling_human_aspects_to_enhance_software_quality_management.

[61] E. Enoiu, G. Tukseferi, and R. Feldt, "Towards a model of testers' cognitive processes: Software testing as a problem solving approach," in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2020, pp. 272–279. [Online]. Available: https://doi.org/10.1109/QRS-C51114.2020.00053.

[62] E. Enoiu, G. Gay, J. Esber, and R. Feldt, "Understanding problem solving in software testing: An exploration of tester routines and behavior," in *IFIP International Conference on Testing Software and Systems*, 2023. [Online]. Available: https://doi.org/10.1007/978-3-031-43240-8_10.

[63] K.-J. Stol and B. Fitzgerald, "The ABC of software engineering research," *ACM Transactions on Software Engineering and Methodology*, vol. 27, pp. 1–51, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3241743.

[64] D. Avison, F. Lau, M. Myers, and P. Nielsen, "Action research," *Commun. ACM*, vol. 42, pp. 94–97, Jan. 1999. [Online]. Available: https://doi.org/10.1145/291469.291479.

[65] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *Empirical Software Engineering*, vol. 27, no. 4, Jul. 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10072-8.

[66] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, London, England, 2014, pp. 1–10. [Online]. Available: https://dl.acm.org/doi/10.1145/2601248.2601268.

[67] T. C. Lethbridge, S. E. Sim, and J. Singer, "Studying software engineers: Data collection techniques for software field studies," *Empirical Software Engineering*, vol. 10, pp. 311–341, Jul. 2005. [Online]. Available: https://doi.org/10.1007/s10664-005-1290-x.

[68] W. C. Adams, "Conducting semi-structured interviews," in *Handbook of Practical Program Evaluation*, K. E. Newcomer, H. P. Hatry, and J. S. Wholey, Eds., 2015. [Online]. Available: https://doi.org/10.1002/9781119171386.ch19.

[69] H. K. Klein and M. D. Myers, "A classification scheme for interpretive research in information systems," in *Qualitative Research in IS*, E. M. Trauth, Ed. Hershey, PA: Idea Group Publishing, 2001, pp. 218–239. [Online]. Available: https://doi.org/10.4018/978-1-930708-06-8.ch009.

[70] E. Schillström and D. Wahlin, "Identification of technical debt in code using software metrics," M.S. thesis, Lund University, Lund, Sweden, 2021.

[71] E. R. Teijlingen and V. Hundley, "The importance of pilot studies," *Social Research Update*, vol. 35, 2001. [Online]. Available: https://www.researchgate.net/publication/11173521_The_Importance_of_Pilot_Studies.

[72] V. Braun and V. Clarke, "Thematic analysis," in *APA Handbook of Research Methods in Psychology, Vol. 2, Research Designs: Quantitative, Qualitative, Neuropsychological, and Biological*, H. Cooper *et al.*, Eds., American Psychological Association, 2012, pp. 57–71. [Online]. Available: https://doi.org/10.1037/13620-004.

[73] ReSharper, Version 2024.1, [Software], Prague, Czech Republic: JetBrains, 2024.

[74] P. Berander and A. Andrews, "Requirements prioritization," in *Engineering and Managing Software Requirements*, A. Aurum and C. Wohlin, Eds. Berlin, Heidelberg: Springer, 2005, pp. 69–94, ISBN: 978-3-540-28244-0. [Online]. Available: https://doi.org/10.1007/3-540-28244-0_4.

[75] Microsoft Community, "Testing in .net core," Dec. 2023, Accessed on: 2024-02-22. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/core/testing/.

[76] jqwik - Property-based testing for Java, Version 1.8.4 [Software], jqwik Community.

[77] MutPy, Version 0.6.1, [Software], Wilmington, Delaware, United States: Python Software Foundation, 2019.

[78] QTest, [Software], Leverkusen, Germany: Testmo GmbH.

[79] Zephyr, [Software], Somerville, Massachusetts, USA: SmartBear.

[80] Jira, Version 9.15.1, [Software], Sydney, Australia: Atlassian, 2024.

[81] JUnit, Version 5.10.2, [Software], JUnit Community, 2024.

[82] NUnit, Version 4.1.0, [Software], Redmond, Washington, United States: .NET Foundation, 2024.

[83] JaCoCo, Version 0.8.10, [Software], Brussels, Belgium: Eclipse Public License, 2023.

[84] Postman, [Software], San Francisco, US: Postman Inc, 2024.

[85] GitHub Issues, [Software], San Francisco, US: GitHub, 2024.

[86] Microsoft Community, "Coding conventions," Jan. 2023, Accessed on: 2024-02-22. [Online]. Available: https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/coding-conventions.

[87] Oracle, "Code conventions for the java programming language," Apr. 1999, Accessed on: 2024-02-22. [Online]. Available: https://www.oracle.com/java/technologies/javase/codeconventions-contents.html.

[88] Python Software Community, *PEP 8 – Style Guide for Python Code*, Accessed on: 2024-02-22, Jul. 2001. [Online]. Available: https://peps.python.org/pep-0008/.

[89] Pylint, Version 3.1.0, [Software], Wilmington, Delaware, United States: Python Software Foundation, 2024.

[90] SonarQube, Version 10.5, [Software], Geneva, Switzerland: SonarSource SA, 2024.

[91] B. Douglas, "Build a ci/cd pipeline with github actions in four steps," *GitHub Blog*, Feb. 2022. [Online]. Available: https://github.blog/2022-02-02-build-ci-cd-pipeline-github-actions-four-steps/.

[92] SonarCloud, [Software], Geneva, Switzerland: SonarSource SA, 2024.

[93] S. P. Ng, T. Murnane, K. Reed, D. Grant, and T. Y. Chen, "A preliminary survey on software testing practices in australia," in *Proc. IEEE 2004 Australian Software Engineering Conference*, Melbourne, VIC, Australia, 2004, pp. 116–125. [Online]. Available: https://ieeexplore.ieee.org/document/1290464.

[94] H. Tufail, H. Masood, B. Zeb, and M. Wassem, "A systematic review of requirement traceability techniques and tools," in *Proceedings of the Interna-*

tional Conference on Software Reuse and Integration (ICSRS), 2017, pp. 450–454. [Online]. Available: https://doi.org/10.1109/ICSRS.2017.8272863.

[95] F. Tian, T. Wang, P. Liang, C. Wang, A. Khan, and M. Ali Babar, "The impact of traceability on software maintenance and evolution: A mapping study," *Journal of Software: Evolution and Process*, vol. 33, e2374, Aug. 2021. [Online]. Available: https://doi.org/10.1002/smr.2374.

[96] E. Schwitzgebel, "The unreliability of naive introspection," *The Philosophical Review*, vol. 117, pp. 245–273, 2 Apr. 2008. [Online]. Available: https://doi.org/10.1215/00318108-2007-037.

[97] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, pp. 131–164, 2009. [Online]. Available: https://doi.org/10.1007/s10664-008-9102-8.

[98] L. McLeod, S. G. MacDonell, and B. Doolin, "Qualitative research on software development: A longitudinal case study methodology," *Empirical Software Engineering*, vol. 16, pp. 430–459, 2011. [Online]. Available: https://link.springer.com/article/10.1007/s10664-010-9153-5.

A

Interview questions for interviews with developers

A.1 Background information

- What is your role in the company? How does it relate to testing?
- Do you develop mostly internal functionalities or towards clients?
- How long have you been working at the company?
- What is your previous experience related to testing? Worked with/studied?
- What is your opinion about software testing? Important or not? Enjoyable or not?

A.2 Testing today

- How do you test your code? Why?
- Can you walk us through a test cycle for a recently implemented feature?
- How do you choose your test cases and the data for them? Do you usually reuse tests?
- Do you feel motivated to test your code? Why or why not?
- Would you say you test a lot, moderately, or minimally? In comparison to regular development. Why?
- How often do you need to fix bugs discovered in production? Is it a stressful moment to interrupt what you were actually planning to do, or what do you think about it?
- In your opinion, what is the reason for writing tests?
- Do you estimate the time for a feature development? If so, do you include testing time?
- Do you reuse code in your daily work? How do you know it is reliable?

A.3 Test Techniques

- Are you familiar with black, white, and gray box testing? Which, if any, of these do you apply?
- Which level of Unit, Integration, System, and Acceptance level do you test on?

- What test techniques do you use? For example mutation testing or regression testing.
- How do you measure the effectiveness of the testing process? For example, do you use test coverage or other metrics for code quality? Tools?
- Do you optimize the tests in any way? (For example, prioritization or choosing a suitable subset
- How do you stay updated in software testing? For example continuing education, certifications, etc.

A.4 Guidelines

- Do the company have any guidelines to be followed regarding testing? For example, guidelines could include code reviews.
- How do you decide when tests should be written? Is it something you consciously always or never test?

A.5 Opinions about the testing process today

- How do you perceive the current test process? Are there any challenges? How are they handled?
- Do you experience any consequences in your daily work based on the company's testing process?
- Are there any parts of the process that you think work really well?

A.6 Areas for Improvement

- What areas are there for improvement regarding the current test process? Have you ever expressed this to your manager? Is there someone obvious to talk to about it?

Other thoughts?

B

Interview questions for interview with Head of the Software Development

B.1 Background information

- What is your role in the company? How does it relate to testing?
- How long have you been with working at the company?
- What is your previous experience related to testing? Worked with/studied?
- What is your opinion about software testing? Important or not? Enjoyable or not?

B.2 Testing today

- What does the company aim to achieve with the software testing today? Are there any requirements you want to ensure are met?
- Is there any test environment to test against that is identical to the production environment?
- Are there any specific areas that should receive more attention in the testing process, according to you? Both areas in the code, and testing aspects (e.g., security)
- Do you do anything to keep the developers motivated to test?
- Do you provide recognition for developers' testing work?

B.3 Decision making

- How has the management team influenced the testing process?
- Can you describe important decisions you have made regarding the testing process?
- How do you allocate resources between software development and testing? Is it a suitable partition according to you?

B.4 Opinions about the testing process today

- How do you perceive the current test process? Are there any challenges? How are they addressed?
- How important is software testing for the company? Why?
- Is testing an activity that affects the company's results? Results such as financial, customer satisfaction, and employee satisfaction.
- Whose responsibility is that software testing is performed? Who benefits from the tests?

B.5 Areas for Improvement

- What do you see as potential areas for improvements regarding the current test process?
- What do you think about the company's future related to testing? How will the testing process evolve? How do you want it to evolve?
- Have there been any initiatives to change the testing process lately? Were they carried out?

Other thoughts?

C

Interview questions for interview with Test Process Responsible

C.1 Background information

- What is your previous experience related to testing? Worked with/studied?
- What is your opinion about software testing? Important or not? Enjoyable or not?

C.2 Current testing process

- Describe your test environment. We've heard something about a test database. How does it work? How was the data generated?
- Can you describe which components are included in the test environment? Database, CI, libraries, reports, etc?
- Why does it look like this? How were the specific tools chosen?
- How did the software testing start? How was it taken from nothing to what you have today? What decisions were made? What drove the decisions?

C.3 Testing today

- What does the company aim to achieve with the software testing today? Are there any requirements you want to ensure are met?
- How is testing integrated with the development cycle? How do you test while developing?
- Can you walk us through a test cycle for a recently implemented feature?
- How is the test environment maintained? Who is responsible?
- What is the flow from a customer discovering a bug to it being resolved?

C.4 Test Techniques

- Are you familiar with black, white, and gray box testing? Which, if any, of these do you apply?
- Which level of Unit, Integration, System, and Acceptance level do you test on?

- What test techniques do you use? For example mutation testing or regression testing.
- How do you measure the effectiveness of the testing process? For example, do you use any test coverage or other metrics for code quality? Tools?
- Do you optimize the tests in any way? (For example, prioritization or choosing a suitable subset)
- How do you keep the company updated in software testing? For example continuing education, certifications, etc.

C.5 Guidelines

- Do the company have any guidelines to be followed regarding testing?
- How do you decide when tests should be written? Is it something you consciously always or never test?

C.6 Opinions about the testing process today

- How do you perceive the current test process? Are there any challenges? How are they addressed?
- How important is the software testing for the company? Why?
- Are there any parts of the process that you think work really well?

C.7 Areas for Improvement

- What do you see as potential areas for improvements regarding the current test process?
- What is the difference between the company's test process and an ideal case, in your opinion?
- What do you think about the company's future related to testing? How will the testing process evolve? How do you want it to evolve?

Other thoughts?

D

Interview questions for Framework Evaluation

D.1 General perception:

- How would you describe your experience using our recommendations?
- Were there any parts that felt particularly rewarding? In that case which and why?
- Were there any parts that felt unnecessary or redundant? In that case which and why?
- Did you expect any steps/recommendations that were missing? Is there another activity you would have liked to add to improve the testing process?
- Did the different steps feel realistic to integrate into your work? Did you think the recommendations were tailored for SME companies?
- How do you think the adoption of these steps would have affected the testing process?
- In what way has the framework been helpful during the testing process?
- Would you continue to use the framework for future testing needs? Why, why not?

D.2 Implementation

- What test levels have you tested at? Do you wish you had tested more? If yes, why didn't you do it?
- How did you chose your test cases? For example, did you test expected behavior, unexpected behavior, error handling etc?
- What did you think about using Equivalence Classes and Boundary Analysis? Is it something you think adds value?
- What did you think about using mutation testing? Is it something you think adds value?
- How did you experience working with Resharper? Is it something you think adds value?
- What do you think of using an Exit Criteria to guide when you are done with your testing? Is it something that adds value?

D.3 Challenges

- Did you encounter any challenges during the testing process?
- How did you deal with these challenges and do you have suggestions on how to overcome them in future iterations?
- Is there something that stopped you from reaching the desired quality during the testing process?

D.4 Experience

- Are you satisfied with your testing?
- Do you feel more confident about your code? More secure?
- Did you feel motivated to test?
- Did you experience a high threshold to begin your testing? If lower than before using the framework, what made it easier?
- Would you recommend this framework to others in your industry? Why or why not? How likely is it that the company will follow this framework?

E

Questions regarding recommendations to Company

- What was your first opinion about this recommendation?
- What are the benefits of this recommendation?
- What are the drawbacks or apprehensions of this recommendation?
- How feasible would it be to implement this recommendation?
- If you had to suggest a change/adaptation to the recommendations, what would it be?
- If different options are presented, which do you prefer and why?

F

Thematic Analysis

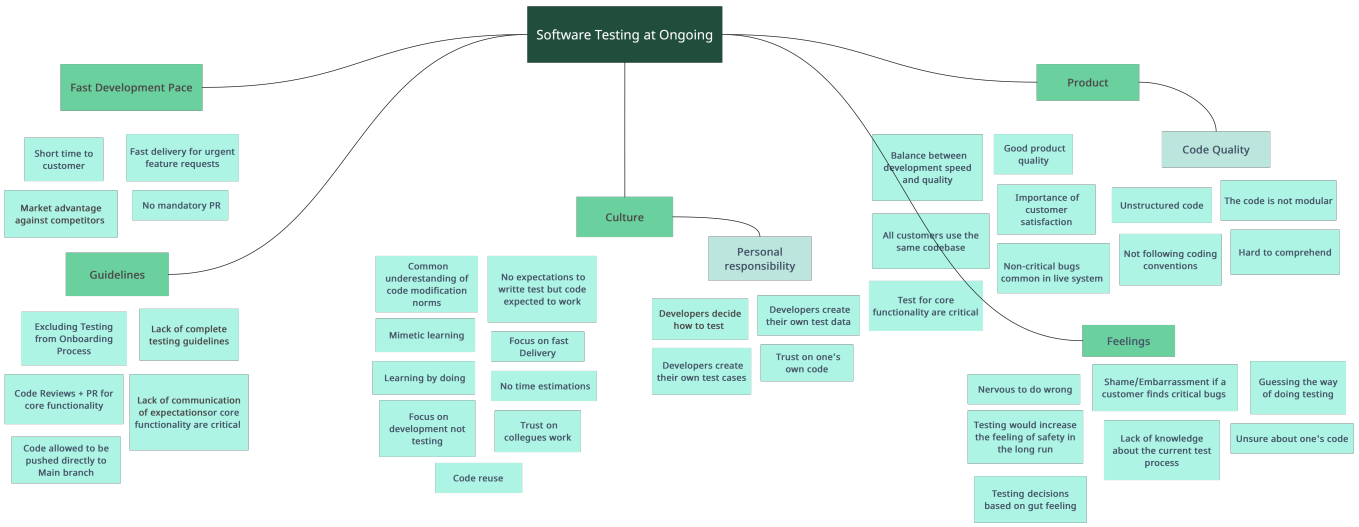

Figure F.1: Thematic analysis part 1.

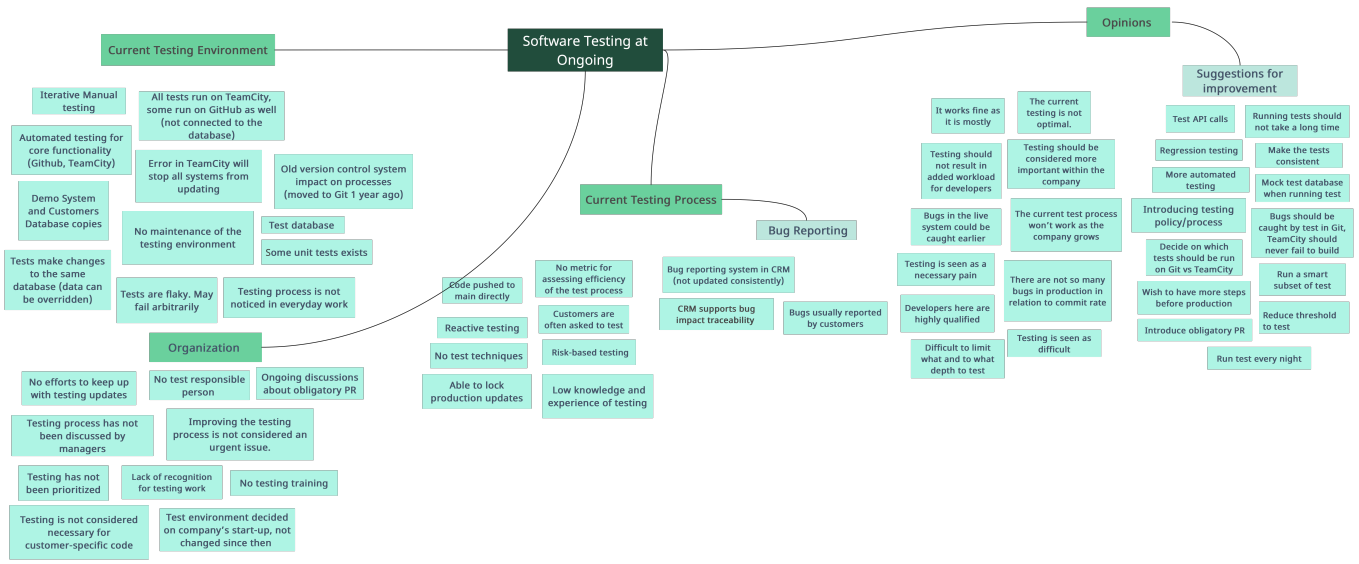

Figure F.2: Thematic analysis part 2.

G

Workshop Feedback

General feedback:

- Nice and clear structure.
- Easy to follow.
- Want more hands-on suggestions.
- Want examples.

Testing knowledge:

- The axis is relevant and adequate. Participants expressed that testing knowledge was lacking.
- It is important and good that it is raised that simple resources can be provided.
- It is important to distribute and update information about the company's testing process so everyone feels involved and on board with what is going on - good that this is included.

Plan definition:

- It is not clear what the question "Where do we test?" means - is it an environmental question as in where are the tests conducted (locally, pipeline, etc), or what parts in the code are tested.
- Not clear what is meant by "Define exit criteria".
- "Define the general approach to testing", does it mean "how do we test"?
- Wish to include if the testing should be continuous or not, and if the testing should be before development or not.

Affected test levels:

- What does "available objects" mean?
- Good that the importance of unit tests is expressed.
- There was a released bug the other day that could have been caught by unit tests.
- Text is generally too abstract.
- Want guidance about how the tests should be written.
- Regarding the testing pyramid - is it the number of cases that is decreased from bottom to top or is the pyramid showing the order of how to write tests?

Definition of test cases:

- What is meant by "High-level cases"?
- What does it mean that test cases should be traceable in both directions?
- Good that the importance of independent test cases is addressed.
- What does it mean by "multiple test cycles"?

Test activities:

- Some participants expressed that they did not have time to read the theory chapter and this axis was dependent on it because they did not possess

knowledge about several concepts within this axis. Therefore, difficult to give feedback.
- What is meant by “functional documentation reviews”?
- What is the difference between test activities and test techniques?

Traceability:
- Confused by the difference between test objects/cases/requirements.
- The first bullet point from No Implementation and Partial Implementation is very similar - what is the difference?
- Overall good.

Test environment:
- Good and important axis. Underscoring the importance of mocking.
- Still, want more hands-on suggestions.
- What is meant by "test harness"?

Testing tools:
- Really good.
- Good that it was highlighted not to use complicated tools - they want it to be simple to perform testing.
- Liked that automation was addressed.

Code quality:
- Easy to understand, and they relate to the need for better quality.
- What is meant by modular code?
- Pair programming can be difficult to apply in everyday work.
- The suggestion about doing a full code scan is good and they would like to apply it.

CI/CD pipeline:
- Good.

Feedback:
- Adequate.
- Want more examples of what to measure.
- Feedback adds administrative overhead, which makes the organization slower.
- Would be good to know how many bugs are released.
- Encourage automatic reports.

Developer Motivation:
- Important axis.
- Testing motivation is decreased when the developer experiences stress.
- Good that it is suggested to set aside time destined for only testing activities.
- Good that it is suggested to provide a clear testing strategy to follow.
- Good that it is suggested to provide clear test examples to follow.

H

Questionarie results

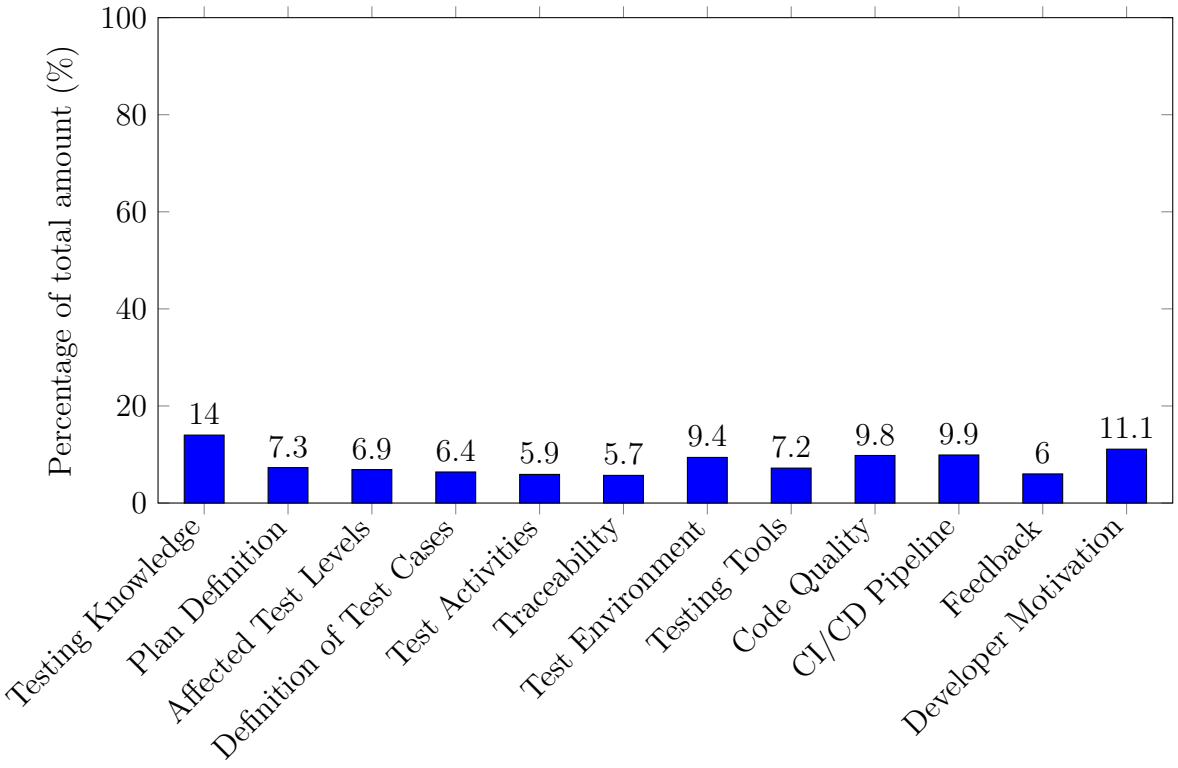

Figure H.1: Distribution of funds across the axes. Represented as the percentage of the total amount of dollars distributed by participants.

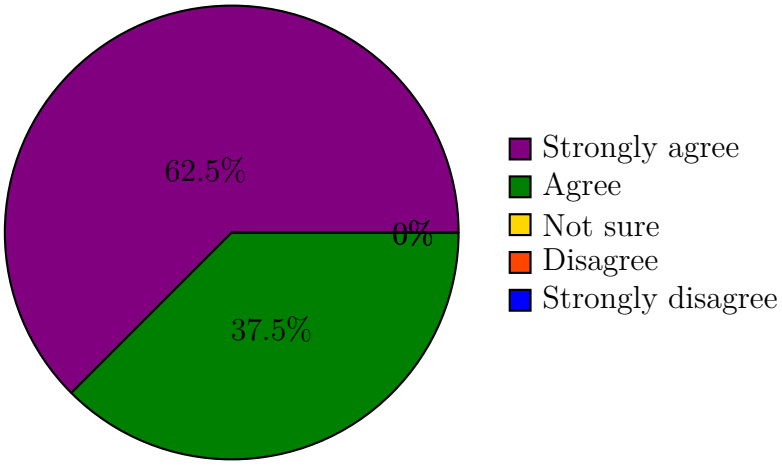

Figure H.2: The distribution of responses regarding the statement "The axis in the framework are adequate".

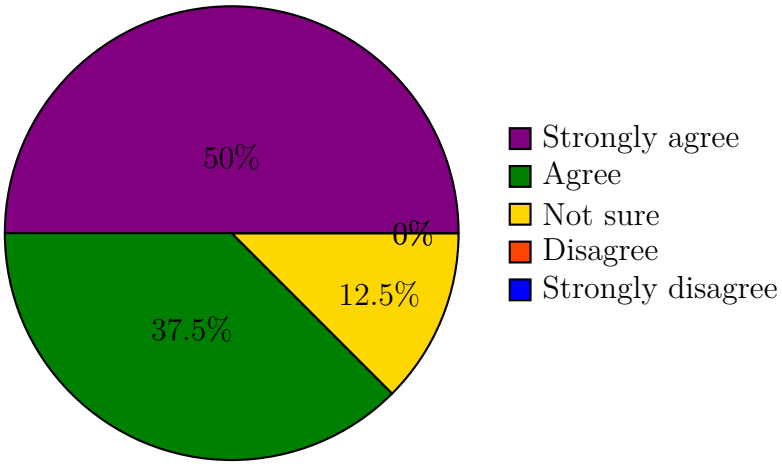

Figure H.3: The distribution of responses regarding the statement "The axes in the framework captures the key aspects in a software testing process".

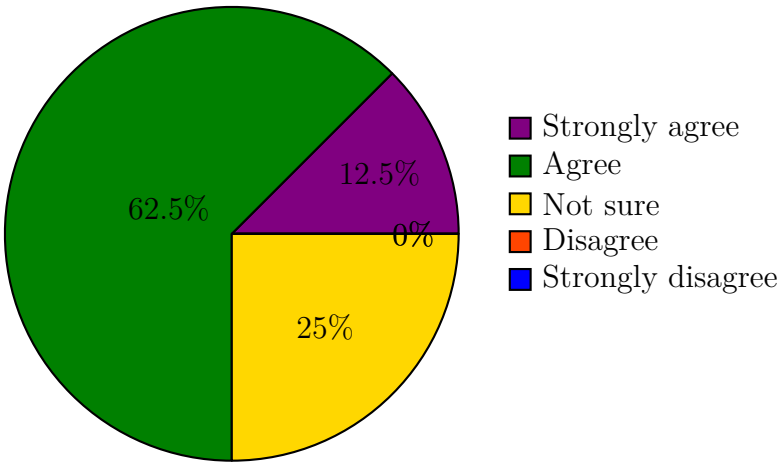

Figure H.4: The distribution of responses regarding the statement "The framework is applicable".

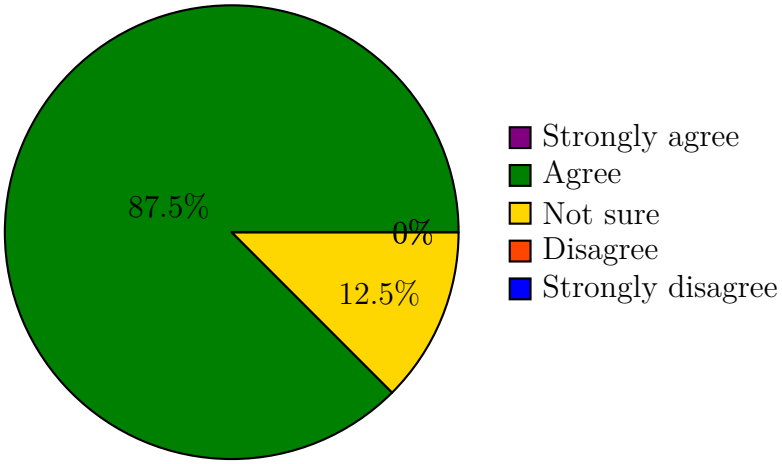

Figure H.5: The distribution of responses regarding the statement "The framework is easy to follow".

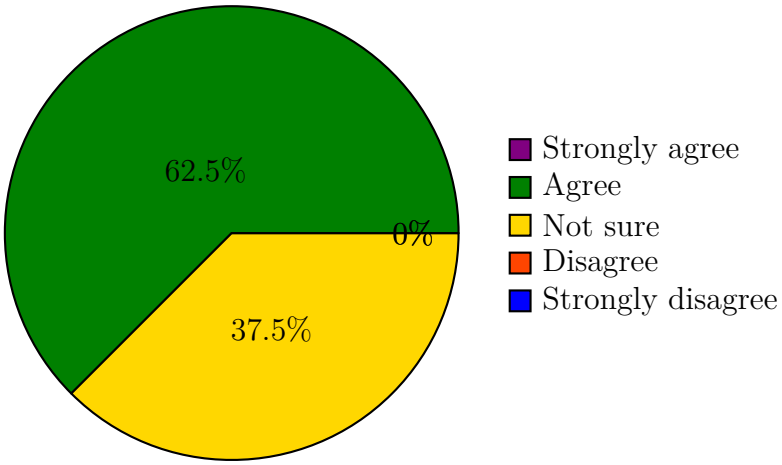

Figure H.6: The distribution of responses regarding the statement "The framework is complete".

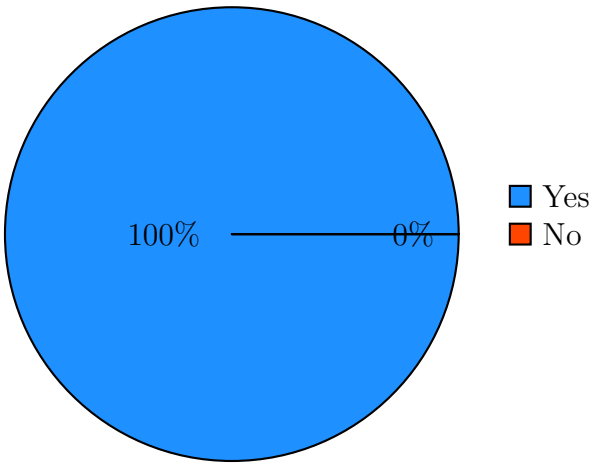

Figure H.7: The distribution of responses regarding the question "Does the theory chapter provide any value?".

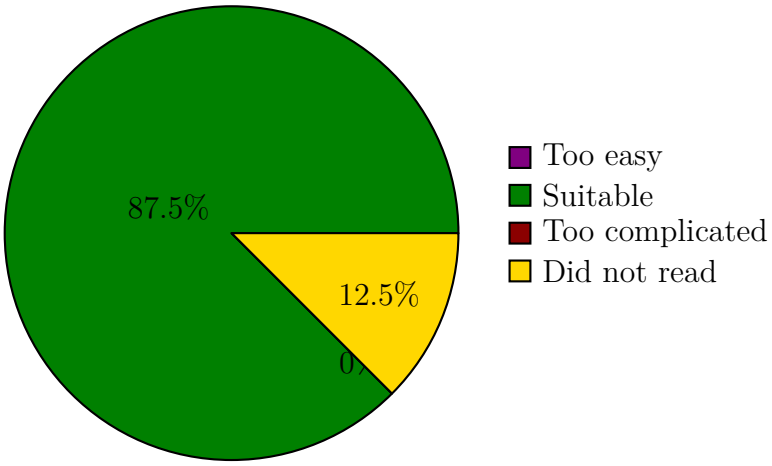

Figure H.8: The distribution of responses regarding the question "How would you rate the difficulty level of the theory chapter?".

I

Evolution matrix from No Implementation level

Axis	**Improvement activities**
Testing knowledge	• Include basic testing principles as well as the organizational approach regarding testing in the onboarding process • Provide simple resources to achieve fundamental testing knowledge • Emphasize unit testing • Clarify advantages of software testing
Plan definition	• Answer the questions: What do we test? What do we not test? And why do we do it? When do we test? Where do we test? • Define and communicate roles and responsibilities for testing activities. Assign someone responsible for the software testing • Integrate and coordinate testing activities with the development lifecycle • Define product requirements • Define exit criteria
Affected test levels	• Analyze the available test objects from the unit, integration and system-level test perspectives • Identify the features to be tested from the previously analyzed test objects, as well as the conditions under which they will be tested • A rule of thumb is to focus the density of tests in the order of the test pyramid, with most tests in the lower levels

Definition of test cases	• Agree on a standard format for test case specification • Create high-level test cases. A checklist is often recommended as a first approximation
Test activities	• Perform static review tests, e.g., code reviews or inspections and walkthroughs • Perform exploratory dynamic tests manually • Consider basic testing techniques such as Regression testing and Sanity testing
Traceability	• Maintain traceability between test objects and test cases
Test environment	• Build a test environment outside of the development and production environment • Prepare it with shared test data, where possible, that is similar to production data • Integrate triggering tests in the CI/CD pipeline
Testing tools	• Incorporate requirements and incident monitoring tools. As a first approximation, they can be carried in a spreadsheet, but it is advisable to use specific tools for this type of task • Incorporate test execution tools that allow them to be automated • Apply tools supporting unit testing
Code quality	• Chose code standard(s) to follow and establish a routine for staying updated with the guidelines (suggestion: workshop once a year) • Introduce automated tools for continuous assessment of code quality • Integrate the code quality aspects to code reviews • Communicate and educate the importance of modular code • Pair programming is encouraged • Comment the code

CI/CD pipeline	• Set up a pre-commit command that runs formatting and simple tests • Create a build script in the repository that states which set of tests should be run on every commit/push. Update the chosen set regularly • Integrate static code analysis tools for performing code quality checks • Create a build script for the deployment including what tests to run at this stage
Feedback	• Establish metrics, such as code coverage, testing time, and number of failures • Integrate tools for extracting the metrics • Gather feedback from users • Keep a count of the test cases executed, the results obtained, and the defects reported
Developer motivation	• Increase the testing status at the company • Combine testing responsibilities with development activities • Ensure a variety of engaging and challenging tasks • Encourage a good relationship with management and colleagues • Set aside time destined for only testing activities • Ensure the testing environment holds a high quality • Provide a clear testing strategy to follow • Provide clear test examples to follow

Table I.1: Final proposed framework: Evolution matrix from No Implementation level.

J

Evolution matrix from Partial Implementation level

Axis	**Improvement activities**
Testing knowledge	• Provide courses in testing, with the possibility to advance in expertise level • Educate about cognitive biases that affect testing • Ensure fundamental testing skills before employment • Provide an opportunity to share knowledge with colleagues
Plan definition	• Document the plan and distribute it to the development team and stakeholders • Make an estimated time budget required to carry out the test activities and add buffer time for testing activities • Generate a calendar with specific dates or in the context of each iteration • Estimate people and resources to carry out the planned activities. If possible, make a tentative assignment • Decide specialized testing needs at the company • Create opportunities for participatory decision-making • Hold retrospective meetings destined to maintain and evaluate the chosen testing process • Comprehend feedback from testing activities to fine-tune the testing plan • Follow up if product requirements have been fulfilled
Affected test levels	• Analyze the higher-level test objects, such as acceptance tests • Identify the features to be tested in the previously added test levels

Definition of test cases	• Choose appropriate test specification technique(s) to generate test cases • Utilize debasing techniques such as playing the devil's advocate. • Generate test cases identifying: the possible input data, the execution conditions, and the expected results. • Document the fully specified test cases for follow-up
Test activities	• Perform static analysis tests guided by tools such as source code analysis (examples are given in the Code Quality axis) • Automate dynamic tests • Consider more advanced testing techniques such as Search-based testing, Property-based testing, Mutation testing • Optimize tests
Traceability	• Add traceability between requirements and the test objects and test cases • Determine the coverage of the tests • Introduce test management tools
Test environment	• Include in the test environment: test harness, service virtualization, simulators, and other necessary infrastructure elements • Prepare test data from production data and keep it up to date • Mock external dependencies as often as possible for lower-level tests
Testing tools	Incorporate tools for: • test management and test products • test design and implementation, including test cases, test procedures, and test data • execution and registration of tests • performance measurement and dynamic analysis • specialized testing needs such as security tests, portability, and accessibility

Code quality	• Do a full scan of the code base and fix existing code quality issues. This could be a good opportunity for someone with lower domain expertise or experience at the company to familiarize themselves with the code base (e.g., a summer job or a project for a new employee) • The code quality of every new change should be evaluated to meet the code quality guidelines
CI/CD pipeline	• Prioritize the order of the test cases in the build scripts • Perform risk assessment and run the critical first • Examine the different tests to not cover redundant tests • Ensure to not run the same tests in several build scripts if not needed
Feedback	• Generate progress reports to evaluate the process of the testing • Allow interested parties to provide feedback about the quality achieved and priorities for future releases
Developer motivation	• Introduce ways to provide positive feedback when discovering faults • Ensure active influence in decisions concerning the testing process • Make use of code quality metrics • Adopt pair programming practices

Table J.1: Final proposed framework: Evolution matrix from Partial Implementation level.

K

Examples

K.1 Exit Criteria

- All test cases must be executed, with a minimum of 95% passing.
- All high-priority issues identified must be resolved, and medium-priority issues must either be resolved or have a planned resolution documented.
- The test coverage must meet or exceed the agreed threshold of 90%.
- The software's performance must align with the specified performance benchmarks (system uptime of 99.9%), confirming that it can handle the expected load and stress conditions.
- Documentation of the testing process, including test results, bug reports, and quality metrics, must be completed and reviewed.

K.2 System Requirements for an E-commerce System

Functional Requirements:

- **Registration:** Users can register a new account by providing the required information such as name, address, phone number, email, password etc.
- **Product Management:** Administrators can add, update or remove products.
- **Search and Filtering:** Users can search for products using keywords or filter products based on characteristics such as brand, price, color etc.
- **Shopping cart:** Users can add items to their shopping cart, edit quantities or remove items.
- **Payment:** The system should facilitated payment through integration with multiple payment services such as credit card processing and Klarna.
- **Order Management:** User can view their order history and track their order status.
- **Notifications:** Users receive notifications about order status, promotions and other relevant information via email or SMS.

Non-Functional Requirements:

- **Scalability:** The system should be scalable to adapt to growth in the number of users, products and transaction volumes.
- **Reliability:** The system should have an uptime of at least 99,9 %.

- **Usability:** The user interface should be intuitive and accessible, supporting various devices and screen sizes.
- **Security:** They system should comply with data protection regulations and follow security mechanism such as encryption and authentication.
- **Performance:** They system should be able to handle a high number of simultaneous users and transactions without degradation of performance.

K.3 High-level test case

Objective: Test positive number addition functionality **Steps:**

- Number A is typed in the first field
- Number B is typed in the second field.
- The "Calculate" button is clicked.

Expected Results:

- The Sum of A and B appears on screen.

K.4 Code Review Guide

- **Understand the Specified Functionality:**
  - Ensure the submitted code meets all the specified requirements and delivers the expected functionality.
- **Code Correctness and Quality:**
  - Check that the code includes passing unit tests which cover a range of scenarios, including edge cases.
  - Ensure the code adheres to the project's coding standards and best practices.
  - Review for logical correctness, algorithmic efficiency, and simplicity.
- **Documentation and Readability:**
  - Confirm that the code is well-documented, including clear comments, function descriptions, and inline comments where necessary.
  - Ensure the code is readable and maintainable, with a logical structure and consistent formatting.
- **Static Code Analysis:**
  - Ensure the code passes static code analysis with no critical issues.
  - Check for good maintainability ratings and absence of new code smells or technical debt.
- **Merge Readiness:**
  - Confirm there are no merge conflicts with the base branch, ensuring a smooth integration process.
- **Error Handling:**
  - Check that the code robustly handles errors and exceptions, providing clear, user-friendly error messages or logging where appropriate.
- **Extensibility and Scalability:**
  - Ensure the code is designed to allow for future extensions, modifications, and enhancements without significant rework.

  - Assess whether the code follows principles that support scalability, such as modularity and loose coupling.
- **Security Aspects:**
  - Examine the code for potential security vulnerabilities, ensuring that it adheres to industry-standard security practices.
  - Check for the proper implementation of data validation, authentication, and authorization mechanisms.

L

Changes done to the original framework

The additions are marked with green. The deletions are marked with red. Changes such as reformulations are marked with orange.

Axis	**No Implementation**	**Partial Implementation**	**Full Implementation**
Testing knowledge	Limited	High-level understanding, not maintained	Continuously shared and maintained within the organization
Plan definition	No plan	Informal plan	Formal Plan. The plan is continuously maintained and evaluated
Affected test levels	No testing	Deliberate use of test levels. Mainly unit testing, some integration and system testing	Comprehensive unit, integration and system tests and higher level tests
Definition of test cases	No definition of test cases	High-level definition of test cases	Fully specified test cases
Test activities	No testing	Static or dynamic tests. Some testing techniques applied	Static and dynamic tests combined. Carefully selected testing techniques

Traceability	No traceability	Partial traceability	Full traceability
Test environment	No specific testing environment	Managed and controlled environment	Testing in a suitable environment
Testing tools	Non-specific tools	Specific tools for monitoring and executing tests	Extensive process automation
Code quality	Not considered or evaluated	Introduced guidelines	Full codebase adhere to guidelines
CI/CD pipeline	No tests included	Includes static and dynamic tests	Optimized tests
Feedback	No reports	Some internal and/or external defect reports	Extensive internal and external defect reports. Progress and process reports and activities
Developer motivation	No efforts have been made	Activities to enhance testing motivation have been introduced	Developers are motivated and recognized for testing activities

Table L.1: Color coded axis diagnostic matrix highlighting the revisions of this study.

Axis	**Improvement activities**
Testing knowledge	• Include basic testing principles as well as the organizational approach regarding testing in the onboarding process • Provide simple resources to achieve fundamental testing knowledge • Emphasize unit testing • Clarify advantages of software testing

Plan definition	• Answer the questions: What do we test? What do we not test? And why do we do it? When do we test? Where do we test? • Define the general approach to testing • Define and communicate roles and responsibilities for testing activities. Assign someone responsible for the software testing • Integrate and coordinate testing activities with the development lifecycle • Define product requirements • Define exit criteria • Generate a high-level test schedule. When starts each test, when it ends and the most important milestones
Affected test levels	• Analyze the available test objects from the unit, integration and system-level test perspectives • Identify the features to be tested from the previously analyzed test objects, as well as the conditions under which they will be tested • A rule of thumb is to focus the density of tests in the order of the test pyramid, with most tests in the lower levels
Definition of test cases	• Agree on a standard format for test case specification • Create high-level test cases. A checklist is often recommended as a first approximation
Test activities	• Perform static review tests, e.g., code reviews or inspections, walkthroughs, algorithm analysis and functional documentation reviews • Perform exploratory dynamic tests manually • Consider basic testing techniques such as Regression testing and Sanity testing

Traceability	• Maintain traceability between test objects and test cases
Test environment	• Build a test environment outside of the development and production environment • Prepare it with shared test data, where possible, that is similar to production data • Integrate triggering tests in the CI/CD pipeline
Testing tools	• Incorporate requirements and incident monitoring tools. As a first approximation, they can be carried in a spreadsheet, but it is advisable to use specific tools for this type of task • Incorporate test execution tools that allow them to be automated • Apply tools supporting unit testing
Code quality	• Chose code standard(s) to follow and establish a routine for staying updated with the guidelines (suggestion: workshop once a year) • Introduce automated tools for continuous assessment of code quality • Integrate the code quality aspects to code reviews • Communicate and educate the importance of modular code • Pair programming is encouraged • Comment the code
CI/CD pipeline	• Set up a pre-commit command that runs formatting and simple tests • Create a build script in the repository that states which set of tests should be run on every commit/push. Update the chosen set regularly • Integrate static code analysis tools for performing code quality checks • Create a build script for the deployment including what tests to run at this stage

Feedback	• Generate reports of internal defects. The idea is to communicate the quality level of the product under test. It is suggested to keep a count of the test cases executed, the result obtained and the defects reported once the derived adjustments have been made. It is also important to be able to estimate the resources / hours used to carry out this task • Establish metrics, such as code coverage, testing time, and number of failures • Integrate tools for extracting the metrics • Gather feedback from users • Keep a count of the test cases executed, the results obtained, and the defects reported
Developer motivation	• Increase the testing status at the company • Combine testing responsibilities with development activities • Ensure a variety of engaging and challenging tasks • Encourage a good relationship with management and colleagues • Set aside time destined for only testing activities • Ensure the testing environment holds a high quality • Provide a clear testing strategy to follow • Provide clear test examples to follow

Table L.2: Color coded evolution matrix from No Implementation level, highlighting the revisions of this study.

Axis	**Improvement activities**
Testing knowledge	• Provide courses in testing, with the possibility to advance in expertise level • Educate about cognitive biases that affect testing • Ensure fundamental testing skills before employment • Provide an opportunity to share knowledge with colleagues

Plan definition	• Document the plan, either in a specific tool or spreadsheet, and distribute it to the development team and stakeholders • Integrate and coordinate testing activities with development lifecycle activities • Make an estimated time budget required to carry out the test activities and add buffer time for testing activities • Generate a calendar with specific dates or in the context of each iteration • Estimate people and resources to carry out the planned activities. If possible, make a tentative assignment • Decide specialized testing needs at the company • Create opportunities for participatory decision-making • Hold retrospective meetings destined to maintain and evaluate the chosen testing process • Comprehend feedback from testing activities to fine-tune the testing plan • Follow up if product requirements have been fulfilled
Affected test levels	• Analyze the available integration and higher-level test objects, such as acceptance tests • Study the need for tests at specialized levels such as Installation tests • Identify the features to be tested in the different levels • Define and prioritize the conditions for each level while maintaining, as far as possible, traceability with the previous objects
Definition of test cases	• Choose appropriate test specification technique(s) to generate test cases • Utilize debasing techniques such as playing the devil's advocate • Generate test cases identifying: the possible input data, the execution conditions, and the expected results • Prioritize them based on the features to be tested • Document the fully specified test cases for follow-up

Test activities	• Perform static analysis tests guided by tools such as source code analysis (examples are given in the Code Quality axis), model checking and business process simulation • Automate dynamic tests • Consider more advanced testing techniques such as Search-based testing, Property-based testing, Mutation testing • Optimize tests
Traceability	• Add traceability between requirements and the test objects and test cases • Determine the coverage of the tests • Introduce test management tools
Test environment	• Include in the test environment: test harness, service virtualization, simulators, and other necessary infrastructure elements • Prepare test data from production data and keep it up to date. The result should be an environment as similar to the productive environment within the possibilities and restrictions of the business • Mock external dependencies as often as possible for lower -level tests
Testing tools	Incorporate tools for: • test management and test products • test design and implementation, to create maintainable work products, including test cases, test procedures, and test data • execution and registration of tests • performance measurement and dynamic analysis • specialized testing needs such as security tests, portability, and accessibility

Code quality	• Do a full scan of the code base and fix existing code quality issues. This could be a good opportunity for someone with lower domain expertise or experience at the company to familiarize themselves with the code base (e.g., a summer job or a project for a new employee) • The code quality of every new change should be evaluated to meet the code quality guidelines
CI/CD pipeline	• Prioritize the order of the test cases in the build scripts • Perform risk assessment and run the critical first • Examine the different tests to not cover redundant tests • Ensure to not run the same tests in several build scripts if not needed
Feedback	• Generate progress reports to evaluate the process of the testing and track associated activities • Prepare defect reports to communicate the testing process and product quality • Allow interested parties to provide feedback about the quality achieved and priorities for future releases
Developer motivation	• Introduce ways to provide positive feedback when discovering faults • Ensure active influence in decisions concerning the testing process • Make use of code quality metrics • Adopt pair programming practices

Table L.3: Color coded evolution matrix from Partial Implementation level, highlighting the revisions of this study.

M

Developer Guide for Framework Application

M.1 Procedure

1. Install and integrate Resharper into your IDE, by starting a 30-day free trial.
2. During all coding, adhere to the company's coding standards.
3. For each method to be tested:
    - **Familiarize with the Provided Requirements:** Carefully review the requirements section to understand the specifications and expectations for each method to be tested.
    - **Analyze Test Levels:** Determine the appropriate test level (unit, integration, system, or acceptance) for each method, prioritizing the test pyramid's lower levels.
    - **Create High-Level Test Cases:** Develop high-level test cases for each method based on the analysis. An example is provided for reference.
    - **Specification Techniques:**
        - **Equivalence Classes and Boundary Analysis:** Utilize these techniques to create detailed test cases, ensuring comprehensive coverage and efficiency.
        - **Mutation Testing:** Apply mutation testing to assess and ensure the quality of the test cases developed.
    - **Exit Criteria:** Use the specified exit criteria to conclude that the testing phase is complete.
4. Contact us before the code review should be performed.

M.2 Requirements

Functional Requirements

- **FR1:** The system should enable appending suffixes from order numbers to support different integration scenarios.
- **FR2:** The system must aggregate results from multiple files or reports, summarising the overall success or failure.
- **FR3:** The system should provide detailed error logging for orders, including reasons for failure and the specific entities affected.
- **FR4:** The system should update the database row for the corresponding order with the error logging information.

Non-Functional Requirements

- **NFR1: Reliability:** Error handling should be robust, providing clear, actionable information and ensuring system stability in case of integration failures.
- **NFR2: Usability:** The code should be well-documented.
- **NFR3: Security:** Any data handling should comply with relevant data protection and privacy regulations.

M.3 Example of High-Level Test Case

Objective: To verify that the AggregateFileResults method accurately aggregates multiple file results into a single summarised result.
Test Steps:

1. Initialise three separate collections of FileResultClass objects, each representing different scenarios: all successes, all failures, and a mix of both.
2. For each collection, call the AggregateFileResults method separately, passing the collection and a specified FileResultAggregationType. Capture the returned object for each call.

Expected Results: The returned object should accurately reflect the aggregation of the input file results, specifically:

- The Success property should be true if all file results were successful, and false if any were unsuccessful.
- The Message property should contain a summary message corresponding to the input results' aggregation.

M.4 Techniques

M.4.1 Equivalence Classes

Equivalence Partitioning is a software testing technique used to reduce the number of test cases by dividing input data into partitions from which test cases can be derived, providing sufficient certainty about the correctness of the program. Although the number of possible program inputs is nearly infinite, certain groups of inputs will result in the same program behavior, regardless of the specific input value. In test terminology, the set of inputs triggering the same behaviour are equivalent and therefore one specific input value can be chosen to represent the entire partition.
To apply this technique one can start by reviewing the specification and requirements of the software to identify the appropriate input data. Continue by separating the identified test data into equivalence partitions. These partitions should cover both valid and invalid input data to ensure that test cases can detect both correct behavior and errors. Finally, select a representative value from each partition and develop the test cases using the value as input data.
For example, consider having a simple function to register a user if an accepted username is provided. The username has only one criterion to make it valid, and that is that it is within a certain length, say between 8-14 characters. This means that the test case has three equivalent partitions:

- **Invalid:** username less than 8 characters long
- **Valid:** username between 8 and 14 characters long
- **Invalid:** username more than 14 characters long

So, in this simple case, there are only three different cases that need to be derived.

M.4.2 Boundary Value Analysis

Boundary Value Analysis focuses on the values at the boundaries of the input domains. The rationale behind this approach is that errors are more likely to occur at the boundaries of input ranges rather than in the center of the input range. Boundaries can be defined as the place where the program suddenly changes its behaviour, often lying in between partition classes. Whenever there is a boundary, two points should be tested - one belonging to each partition. In order to explore boundaries it is recommended to look at all the partitions and think of inputs between them. In order to apply this technique the partition classes should be identified. Once this is done the boundary values can be identified by selecting consecutive values that trigger the behavioural change between the classes. Finally, test cases using boundary values as inputs should be developed. Continuing on the example provided in the Equivalence Partitioning section, there are two places where the function changes its behavior and inputs cross the partition classes: at 8 and 15 characters. The values from both sides of the boundary of these changing points should be considered. Therefore, the following test cases should be considered:

- **Invalid:** Username with 7 characters
- **Valid:** Username with 8 characters
- **Valid:** Username with 14 characters
- **Invalid:** Username with 15 characters

M.4.3 Mutation Testing

A testing technique that rather tests the quality of the tests is mutation testing. The core of mutation testing is that the developer actively inserts a bug in the code, and then runs the test connected to it to verify that the test reveals the fault. If the fault is not revealed, the test case should be revised. To guide how to insert a bug in the code, mutant operators assist. A mutant operator is a guideline for how to modify a statement in the code into a faulty statement. The operators are defined with the target to simulate acknowledged programming errors. There exist numerous mutant operators. One example is to replace the operator $>$ with $>=$. See link `https://pitest.org/quickstart/mutators/` for more mutant operator examples.

M.5 Exit Criteria

- All new test cases must successfully pass execution.
- All tests must be well-commented.
- External dependencies of the artefact under test have been verified to work as expected.

- The added changes have undergone a thorough code review process and have received approval.
- All equivalence classes and boundaries should be covered.
- Expected and unexpected behaviour should be tested.
- Invalid and valid input data should be tested in every test case.
- Mutation testing should be performed using at least three mutators.
- Corresponding requirements should be met.

N

Code Review Guideline

Code Review Guide

- **Understand the Specified Functionality:**
  - Ensure the submitted code meets all the specified requirements and delivers the expected functionality.
- **Code Correctness and Quality:**
  - Check that the code includes passing unit tests which cover a range of scenarios, including edge cases.
  - Ensure the code adheres to the project's coding standards and best practices.
  - Review for logical correctness, algorithmic efficiency, and simplicity.
- **Documentation and Readability:**
  - Confirm that the tests are well-documented, including clear comments, function descriptions, and inline comments where necessary.
  - Ensure the tests are readable and maintainable, with a logical structure and consistent formatting.
- **Merge Readiness:**
  - Confirm there are no merge conflicts with the base branch.
- **Error Handling:**
  - Check that the code robustly handles errors and exceptions, providing clear, user-friendly error messages or logging where appropriate.
- **Security Aspects:**
  - Examine the code for potential security vulnerabilities, ensuring that it adheres to industry-standard security practices.
  - Check for the proper implementation of data validation, authentication, and authorization mechanisms.

O

Recommendations to Company

Recommendation #1: Naming and file organization conventions

We recommend you establish a clear direction on how to name and organize test cases. This facilitates the maintenance of tests, and findings of specific tests, which mitigates the risk of test case duplication. Today, the naming and file organization is ambiguous in the solution. We recommend you to have a test project for each project in the solution, and then organize the test classes correspondingly to the project under test and name the test classes consistently. Name the test class containing tests for classNameA.cs to classNameATests.cs. Note: "Tests" not "Test".
For example:
Testing ProjectName/FolderA/ClassName.cs should be done in the test file:
ProjectName.**Tests**/ FolderA/ClassName**Tests**.cs
We recommend this pattern since the solution already partly follows it, but not fully - which generates confusion. We recommend fixing this and then adhering strictly to the conventions. This enables you to run tests for corresponding changed files automatically in GitHub, as will be described in recommendation #6.1.

Recommendation #2: Git Workflows

A Git workflow is a guideline for how to use Git to accomplish work consistently and productively. As it is today, the company is using the ***Centralised workflow***, which is suitable for teams transitioning from other version control tools. The centralised workflow uses a central repository to serve as the single point-of-entry for all changes to the project. All changes are committed into the ***main*** branch. The Centralized Workflow is great for small teams but it can become insufficient as your team scales in size. We recommend therefore exploring the benefits of the ***Feature Branch Workflow***.

The core idea behind the Feature Branch Workflow is that all development should take place in dedicated feature branches instead of the main branch. This encapsulation makes it easy for multiple developers to work on features without disturbing the main codebase. The goal for the main branch is to never contain broken code, which is a huge advantage for continuous integration environments. The Feature Branch Workflow assumes a central repository, and main represents the official project history. The feature branches can (and should) be pushed to the central repository by using ***Pull requests (PR)***.

Once someone completes a feature, they don't immediately push it into main. Instead, they push the feature branch to the central server and file a pull request asking to merge their additions into main. This gives other developers an opportunity to review the changes before they become a part of the main codebase. By using GitHub actions, automatic tests and builds can be run on every PR. The pull requests can be set up to be approved automatically upon the build and automatic tests passing, and the branch can be set to be automatically deleted when the changes have been merged into main. A protection rule can also be added to the main branch so that a push can only be made through pull requests. Once a pull request is accepted, the actual act of publishing a feature is much the same as in the Centralized Workflow.

At your company, this workflow would facilitate catching errors before they reach TeamCity. The main advantage of adopting this workflow is that the main branch will ultimately never contain broken code, leading to considerably fewer failures in TeamCity. The biggest disadvantage is that this will require changes in the current workflow and possibly longer time from a customer request to delivery. However, it is worth considering that this time could be worth not having failures in TeamCity, which causes delays today.

This recommendation can be used in combination with recommendation #6. For example, custom-specific development may not run any automatic tests but only the build script to check for compile errors. According to the fail data in TeamCity, checking for compiling errors should be highly prioritized, since this is the most common failure in TeamCity with a fail rate of 1.5 fails per day. Therefore we highly recommend at least running a build script on every merge to main to eliminate these errors in TeamCity. If there are automatic tests that you assess always should be run, these can also be run on Git using GitHub actions. We recommend setting up the branch protection rule so that no changes can be merged to the main without a PR. In this case, the test cases that run on every PR could be removed from TeamCity, which shortens the update time once the code is guaranteed successful.

Another possible alternative workflow is the ***Gitflow workflow***. Gitflow is an alternative Git branching model that involves the use of feature branches and multiple primary branches. Gitflow can be used for projects with a scheduled release cycle. Instead of a single main branch, this workflow uses two branches to record the history of the project. The ***main*** branch stores the official release history, and the ***develop*** branch serves as an integration branch for features. It's also convenient to tag all commits in the main branch with a version number. The develop branch will contain the complete history of the project, whereas main will contain an abridged version.
We do not recommend adopting Gitflow and a develop branch as a first step in the company's progression, but as a future step that may be considered as the company continues to grow. If working with product releases becomes relevant for the company in the future, adopting the Gitflow should then be considered.
For further reading about Git workflows see links:
`https://www.atlassian.com/git/tutorials/comparing-workflows/feature-branch-workflow`
`https://www.atlassian.com/git/tutorials/comparing-workflows`
`https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow`

#2.1 Pre-push command

As an alternative to adopting ***Feature Branch Workflow and Pull Request*** a pre-push command could be considered to decrease the number of errors in TeamCity. By utilizing Git hooks an appropriate script can be set up to build the project upon every push into the repository. This would stop pushes introducing compiling errors, and thereby avoiding this kind of errors in TeamCity. The main disadvantage of adopting this recommendation is that it will add extra time from customer requests to delivery, given the necessary time to build the project.

Recommendation #3: Make the tests consistent

This recommendation is already acknowledged within the organization, and actions to perform to achieve test consistency are already identified. Therefore, it was decided, upon discussion with the supervisor at the company, that this recommendation should not be investigated in detail. Rather, it serves as an inspirational and general recommendation since it affects the other recommendations as well.
The main takeaway should be to prioritize making the tests consistent. By consistent, we mean that each test case should be independent and be able to run in isolation without affecting the run of other test cases. Being able to make decisions on where to run the existing tests without the current limitations on whether the tests are connected to the test database or not is fundamental for test process optimizations. However, it is possible to apply the recommendations even if not addressing the current consistency issue, but without achieving the optimal results.
As a first step to make the existing tests isolated and consistent, we recommend mocking the database interactions when testing at the unit level. Utilizing mocking to simulate database interactions is a widely adopted practice in unit testing. It involves replacing the actual database calls with predefined behaviors or responses. This can make tests faster, and more reliable and avoid unnecessary reliance on the test database. In the case of your company, this would imply that more test cases could be moved to GitHub.
We recommend using mocking frameworks such as Moq. Let's look at a simple example.
Imagine you will be testing the following code:

```
01. using System;
02. using System.Collections.Generic;
03. using System.Linq;
04. using System.Text;
05. using System.Threading.Tasks;
06.
07. namespace TestProjectLibrary
08. {
09.     public class checkEmployee
10.     {
11.         public virtual Boolean checkEmp()
12.         {
13.             throw new NotImplementedException();
14.         }
15.     }
16.
17.     public class processEmployee
18.     {
19.         public Boolean insertEmployee(checkEmployee objtmp)
20.         {
21.             objtmp.checkEmp();
22.             return true;
23.         }
24.     }
25. }
```

As you can see, the checkEmployee class contains a checkEmp() function that is not implemented. We are sending an object of the checkEmployee class to the insertEmployee() function to check whether the employee already exists before it is inserted into the DB. Instead of getting the actual return value of the checkEmp() when performing tests, we can mock a desired value.
Here is an example code of the test implementation:

```
01. using System;
02. using Microsoft.VisualStudio.TestTools.UnitTesting;
03. using TestProjectLibrary;
04. using Moq;
05.
06. namespace UnitTest
07. {
08.     [TestClass]
09.     public class UnitTest
10.     {
11.         [TestMethod]
12.         public void TestMethod2()
13.         {
14.             Mock<checkEmployee> chk = new Mock<checkEmployee>();
15.             chk.Setup(x => x.checkEmp()).Returns(true);
16.
17.             processEmployee obje = new processEmployee();
18.             Assert.AreEqual(obje.insertEmployee(chk.Object), true);
19.         }
20.     }
21. }
```

We are defining a mock object associated with the checkCmployee class. Moq has a Setup() function by which we can set up the mock object. Whenever the unit test application encounters the checkEmp() function it will always return true without executing its code.
For reading on using the Moq framework for testing database connections using the Entity Framework, consider the following link `https://learn.microsoft.com/en-us/ef/ef6/fundamentals/testing/mocking`. However, the Moq framework only supports testing at unit level, since it does not test the actual database integration. For integration-level tests, several options can be considered:

#3.1 Mock database

One alternative is to have separate mocked instances of the test database for each test suite run. In other words, each test suite run should create an isolated copy of the test database and run against it. In this way, parallel runs will not interfere with each other. Once testing is completed, the isolated database instance should be deleted.

#3.2 Database transactions

Another alternative is to wrap each test in a database transaction and roll back the transaction at the end of the test. This ensures that each test is isolated in terms of database state, as changes made in one test will not affect others. Some tests currently work similarly, creating their own test data and deleting it upon test completion. The downside of this alternative is that it requires considerable effort since many of the existing tests will need to be rewritten to adopt this principle.

Recommendation #4: Extend test suite

We suggests extending the existing test suite to cover more of the product functionality, to ensure fewer bugs reach the customer. The company analysis showed that the developers aren't always confident in the functionality of their code, which imposes a feeling of insecurity. This feeling is not desirable in the matter of employment satisfaction. It was expressed that a rigorous test suite to rely on could mitigate this uncertainty. Another argument for this recommendation is that the company is growing rapidly, exhibiting new requirements on the reliability of the product and leaving ad-hoc bug resolving less suitable.
We recommend a risk-based approach, where the code base is examined and the most critical functions/modules for the system are identified. These should be listed together with the testing effort already performed for the module, to gain an overview of what parts of the system are actually being tested and identify critical parts of the system that are lacking in testing efforts. This should be carried out by a person who has a deep understanding of the code base and the system functionality. Further, the actual creation of the test cases could be done by someone with lower domain expertise or experience at the company to familiarise themselves with the code base (e.g. a summer job or a project for a new employee).

Recommendation #5: Move most likely-to-fail tests to GitHub

Table x and x (these are removed from the report due to company data restrictions) present the tests that have failed in TeamCity over the last 6 months. We recommend that the tests with the most failure occurrences should be moved to GitHub, to reduce the amount of failures in TeamCity. This change will facilitate catching common errors early in the development process, increasing the possibility of catching errors in a stage where it will not hinder any system update. The chosen test methods should be run through GitHub action upon every pull request. Note that this approach requires PR's to be run on every commit, as explained in recommendation #2. If not adhering to the new workflow, it is still possible to place the tests in GitHub to increase the chance of catching the faults early, but the tests can not be removed from TeamCity if it is not guaranteed that the tests are run on every commit to main.
Optimally, failure data should be gathered over a longer period to arrive at well-founded conclusions about which test cases are most likely to fail. In addition, the test method considered to be most likely-to-fail should be updated over time since this data could change as the codebase evolves. We suggest gathering and updating this data at least once a year.

Recommendation #6: Run a subset of the test suite

A way to decrease the test execution time is to apply the optimization technique of test case selection. We have two different recommendations to achieve this: either by running a subset of tests that belong to a corresponding feature in the code (for example standard Automation functionality), or by executing tests that correspond to the changed files in the commit. We will cover both options below.

#6.1 Run a subset based on feature

Test cases that concern a specific category of the code should be identified and these tests can then be executed only when the concerned part is changed. Categories should be decided within the company but they could, for example, be "Integration" and "Automation" etc.
This could be achieved by naming the branch according to the functionality category that will be developed. The branch naming could, for example, be "MoaB/**Automation**/X". By utilizing GitHub action, the tests to run can be selected according to the branch name. The relevant subsets will need to be previously set up and linked to the right category in the workflow file by assigning test filters. Below it is shown how the selection can be formulated.
* This section is removed due to company data restrictions *

In the workflow definition, it can be decided if the tests should be run on every push to the branch or upon creating a pull request. We see pros and cons for both options.
Tests on every push:

+ Continuously verification of the code throughout the development
+ Quick response
+ Not needed to have open PR's to be able to get continuous feedback (we see some do this today)
- Computational heavy - GitHub have different price plans for how many actions can be run simultaneously and they share resources

Tests on every pull request:

+ Not as resource-demanding as push-option
- No continuous verification; can lead to needing to "go back to coding" when you believed you were done

For customer-specific development, branches could be named "CustomerSpecific" and then be set up not to run any specific test subset, if desired. In this case, the other recommendations given in this document could be relevant, for example, choosing to only run the most likely-to-fail tests.
If this alternative is adopted, the tests covering the chosen categories can be removed from TeamCity since they will be already tested on GitHub. We recommend that the tests covering critical functionality of the system remain in TeamCity as a final check before the changes reach the customer. Ideally, the code should be built and most likely-to-fail and category tests should be run upon every PR.

Threats

One drawback of this solution is that the subset selected in GitHub action needs to be maintained; if new test files are created they could be included in the .yml. However, this task should be assigned to the person responsible for the specific category. Or if assigning a person responsible for the whole test maintenance.
Another threat is that the solution relies on the correct naming of branches - which is not optimal since spelling mistakes etc occur. We have unfortunately not found a way to work around this. However, the "risk" of implementing this is not big; you can always let all tests run on the branches without a category in the name (or no tests as it is today).

#6.2 Run a subset based on changed files

Since it is most likely that tests associated with the changed files will fail, it is recommended to run only the corresponding tests when performing test optimization. The changed files in the commit can be found by running the GitHub Action "Changed Files". Then, it is possible to find the test files that cover the changed files by building the path to the test file. This requires that the naming and file organization recommendation #1 is followed. We have not investigated customizing TeamCity's test execution according to the changed files, but if you run this in GitHub before the code reaches TeamCity, TeamCity is less likely to fail. In case this alternative is adopted, the current test suite in TeamCity should not be changed. This is because it is not possible to know which test did run before on GitHub.
Below is an example of how a GitHub action workflow that runs tests on changed files can be formulated.
* This section is removed due to company data restrictions *

Recommendation #7: Prioritization of test cases

A technique to optimize a test suite is to prioritize the order the test cases are executed. This recommendation is mainly beneficial when the set suite is large, otherwise, the effort might not be worth the benefits. Therefore, this recommendation is mainly applicable for the company if recommendation #4 is performed. By prioritizing test cases, there are two major advantages:

1. Effective use of test execution time - less time spent waiting on test execution if failure(s) occur
2. If the testing activities are abrupted: the most valuable test will have higher chance of have been executed

We suggest you to base the prioritization on a combination of which test cases are most likely to fail and which tests are most relevant. If you prioritize tests most likely to fail, you are likely to get errors sooner than if not utilizing prioritization, which means that you can start to address these errors faster and thereby decrease the time before fix. If you prioritize relevant tests, you get confirmation of the code faster.

We suggest applying prioritization by assigning test cases a category, for example, “Prio1” or “Prio2”, and then prioritizing the execution order of the categories. This means that the test cases within the categories are not explicitly ordered. This is advantageous since ordering every test case would require revising the order every time a new test case is created. It is important to not have too many categories, since it can confuse the differences between the categories.

The easiest approach would be to define one category, “Prio1”, where the test cases most likely to fail (identified by TeamCity logs, see recommendation #5) and the most important tests are assigned. This requires someone to examine the existing test suite and perform the categorization. This process must also be performed iterative, for example every half a year, to include new test cases and to update the prioritization according to the system updates. Alternatively, test categories could also be assigned immediately when test cases are created, leaving the decision to the developer creating the test. The drawback of assigning the priority category immediately is that the log records can not indicate whether this is a test that will fail a lot, and it might be difficult for the test designer to know if it is a critical test or not.

Test prioritization can be achieved in both GitHub and TeamCity. To begin with, choosing which tests to run on GitHub and which on TeamCity is a prioritization itself, but this recommendation is focused on test case prioritization within these two environments. Here is an example on how to achieve it:

* This section is removed due to company data restrictions *

We advise you to include the tests that have not been prioritized last in the workflow script. This makes it risk-free to apply this strategy since all tests will run even if no one adheres to the new process of assigning a test case a priority.

This can also be combined with categorizing subset, as mentioned in recommendation #6. Then, both the selection of test files to include and test case priorities can be combined into the filter.