Harri Erme

CHALLENGES OF QUALITY ASSURANCE OF SOFTWARE DEVELOPED BY A SUBCONTRACTOR : A CASE STUDY

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA

ABSTRACT

Erme, Harri
Challenges of quality assurance of software developed by a subcontractor : A case study
Jyväskylä: University of Jyväskylä, 2024, 72 pp.
Information Systems, Master's Thesis
Supervisor: Marttiin, Pentti

Outsourcing of software development to subcontractors has been a popular business strategy for a long time. Subcontracting at its best enables acquisition of missing competencies, dampening of demand fluctuations, and cost savings due to the subcontractor's benefits from economies of scale. However, it's important to carefully assess a subcontractor's capabilities, as a poorly managed subcontracting relationship can cause significant issues in quality and delays delivery timelines. Quality problems in particular can widely affect the cooperation between the client and subcontractor. The risks have materialized in the case company of this thesis, where unexpectedly significant quality issues have led to slow and difficult quality assurance processes. This thesis conducts a case study in an international IT company in order to map out the challenges the case company has experienced in the subcontracting relationship from the perspective of software quality assurance. Additionally, solutions to these problems are explored by interviewing employees of the case company. The empirical part of the thesis results in a comprehensive and categorized collection of challenges and solutions. Based on these findings, the target company, as well as other organizations in a similar situation, can create a strategy and take actions to start improving the subcontracting relationship. The study was conducted using qualitative research methods. Data was collected by interviewing employees of the case company in focus groups. Semi-structured interviews covered different phases of quality assurance, after which the transcribed data was analyzed. Content analysis revealed a total of 28 challenges and 20 solutions. Many of the identified problems can be traced back to the subcontractor. On the other hand, the target company itself has its own problems but also the capabilities to correct them. For example, by focusing particularly on requirements engineering, the target company can create better conditions for the subcontractor to produce higher quality software.

Keywords: software testing, software quality assurance, software outsourcing, subcontracting

TIIVISTELMÄ

Erme, Harri
Alihankkijan kehittämän ohjelmiston laadunvarmistuksen haasteet : Tapaustutkimus
Jyväskylä: Jyväskylän yliopisto, 2024, 72 s.
Tietojärjestelmätiede, pro gradu -tutkielma
Ohjaaja: Marttiin, Pentti

Ohjelmistokehityksen ulkoistaminen alihankkijalle on ollut jo pitkään suosittu liiketoimintastrategia. Alihankinta mahdollistaa parhaimmillaan mm. puuttuvien kyvykkyyksien hankinnan, kysynnän vaihteluiden vaimentamisen, sekä kustannussäästöt alihankkijan mittakaavaetujen myötä. Alihankkijan kyvykkyydet kannattaa kuitenkin arvioida tarkkaan, sillä pieleen mennyt alihankintasuhde voi aiheuttaa merkittäviä ongelmia laadussa ja toimitusaikatauluissa. Erityisesti laatuongelmat voivat hyvinkin laajasti heijastua ja vaikuttaa asiakkaan ja alihankkijan väliseen yhteistyöhön. Näin on käynyt tämän tutkielman kohdeyrityksessä, jossa odotettua suuremmat laatuongelmat ovat johtaneet laadunvarmistusprosessien merkittävään hidastumiseen ja vaikeutumiseen. Tämän tutkielman tarkoituksena on tuottaa tapaustutkimus kansainvälisessä IT-yrityksessä ja kartoittaa kohdeyrityksen haasteita alihankintaympäristössä ohjelmiston laadunvarmistuksen näkökulmasta. Tämän lisäksi näihin ongelmiin pyritään etsimään vastauksia haastattelemalla kohdeyrityksen työntekijöitä, sekä tieteellisen kirjallisuuden perusteella. Tutkielman empiirisen osuuden tuloksena on kattava ja kategorisoitu kokoelma haasteita ja ratkaisuja, jonka perusteella kohdeyritys, sekä muut vastaavassa tilanteessa olevat organisaatiot voivat suunnitella kehitystoimenpiteitä alihankintasuhteen kehittämiseksi. Tutkielma toteutettiin laadullisia tutkimusmetodeja hyödyntäen. Aineisto kerättiin haastattelemalla kohdeyrityksen työntekijöitä fokusryhmissä. Puolistruktroiduissa haastatteluissa käytiin läpi laadunvarmistuksen eri vaiheita, jonka jälkeen litteroitu aineisto analysointiin. Sisältöanalyysi paljasti yhteensä 28 haastetta, sekä 20 ratkaisua. Suuri osa havaituista ongelmista voidaan jäljittää alihankkijaan. Toisaalta kohdeyrityksellä itsellään on omat ongelmansa, mutta myös kyvykkyyksiä korjata niitä. Esimerkiksi keskittymällä erityisesti vaatimusmäärittelyiden kehittämiseen, kohdeyritys voi luoda alihankkijalle paremmat lähtökohdat tuottaa laadukkaampia ohjelmistoja.

Asiasanat: ohjelmistotestaus, ohjelmistojen laadunvarmistus, ohjelmistokehityksen ulkoistaminen, alihankinta

FIGURES

TABLES

TABLE OF CONTENTS

1 INTRODUCTION

Outsourcing of software development has been a popular business strategy already for decades. Despite the potential benefits, subcontracting relationships can prove to be complex and difficult to manage. Even so, software outsourcing is expected to grow as an industry as especially global software development is gaining popularity (Shah et al., 2014). This thesis aims to investigate difficulties of software subcontracting through the lens of quality assurance (QA). QA processes should be put in place to align the subcontractor's output with the company's standards and regulatory requirements, especially in industries where software reliability and safety are non-negotiable. However, aligning the operations between the client and the subcontractor can prove to be difficult and imposes risks to the overall success of the project. This thesis aims to investigate what issues subcontracting relationship can cause in terms of quality assurance and its activities. Additionally, the thesis tries to shine a light on possible solutions and improvements.

This thesis aims to fill the gap in academic literature by investigating and documenting the practical implications to QA activities when the subcontractor does not perform up to expectations. Prior research has addressed risk factors and hidden costs in outsourcing software development. Additionally, QA and software testing as a practice is a well researched area but combination of outsourcing and QA has been mostly a side note in prior literature. Shah et al. (2014) has investigated the topic of global software testing and revealed the vendor-side difficulties when software testing and development are conducted in separate locations internationally as different entities. However, existing literature has not comprehensively considered the perspective of the client in discussions about quality assurance activities for software developed by subcontractors.

To shine a light on what quality assurance is and intricacies of software outsourcing, literature review is conducted to first establish a theoretical basis for the thesis. Literature review is conducted using multiple databases which include ACM Digital Library, IEEE Xplore, AIS eLibrary and Google Scholar. An effort was made to include only peer-reviewed journal articles in the source

material. In certain special cases, field textbooks and standardized industry vocabulary are also used to illustrate the practical point of view. Multiple search queries were used to collect the articles depending on the topic. The core queries were "software subcontracting", "software development outsourcing" and "software quality assurance".

The goal of the empirical part of the study is to describe and categorize challenges in assuring the quality of software developed by a subcontractor experienced by the case company's employees, as well as their respective solutions. Furthermore, the solutions proposed by scientific literature are taken into consideration and discussed whether they are applicable to the case company's scenario and whether they support the solutions proposed by the study's participants. Empirical portion of the thesis is conducted by interviewing the case company's employees that are involved in quality assurance related tasks and either directly or indirectly with the subcontractor. To achieve this goal, the following research questions are established:

1. *What challenges and issues subcontracting relationship can cause for quality assurance?*
2. *How can these challenges and issues be solved or alleviated?*

The structure of this thesis is as follows. In chapters 2, 3 and 4, theoretical background for the thesis is layed out. The chapters 2 and 3 discuss key themes in software quality assurance and outsourcing of software development, while chapter 4 summarizes these findings. In chapter 5, the methodology of empirical portion of this thesis is described. Additionally, description about the scenario of the case company is presented. In chapter 6, results of the study are revealed. Chapter 7 discusses these results further by taking existing scientific literature into account. Chapter 8 concludes the thesis and presents the limitations of the study, as well as directions for further research.

2 QUALITY ASSURANCE IN THE SOFTWARE INDUSTRY

As software grows in complexity, the challenge of ensuring its reliability and robustness intensifies. This chapter delves into quality assurance (QA) within software development, defining it as:

> A set of activities that define and assess the adequacy of software process to provide evidence that establishes confidence that the software processes are appropriate for producing software products of suitable quality, for their intended purposes, or for their intended operation services and fulfils the requirements of schedule and budget keeping (Galin, 2018, p. 5)

From this definition, we can extract meaning for quality, which aligns with Axelsson & Skoglund (2016) and Nistala, Nori & Reddy (2019) definitions. Software quality can be thought of as the degree to which a software product satisfies stated and implied requirements in operational use (Axelson & Skoglund, 2016; Nistala et al., 2019).

This chapter discusses various key topics related to software quality assurance that are relevant to the context of the case study. First, the chapter explores why making high quality software is so difficult. Second, requirements engineering is discussed to highlight the meaning of well defined requirements to software quality. Thirdly, the general process of software testing is explained to elaborate the activities performed by test engineers. Fourth topic delves deeper into how software is tested in different abstraction levels and phases of development. Fifth subchapter explains a general process of how software defects, resulting from testing activities, are fixed and resolved. Sixth subchapter briefly explains different types of non-functional testing. Finally, test automation and it's ability to reduce test engineers' workload is discussed.

2.1 Factors influencing poor software quality

Software development is a human-centric activity, which means that it is influenced by various organizational, psychological, and behavioral factors (Ghanbari et al., 2019). This means that software is subject for mistakes. Utilizing recommended software development and quality assurance practices can help minimize the amount and severity of software defects. However, research suggests that these deficiencies often arise due to the omission of quality practices. Such omissions can stem from organizational decisions influenced by resource limitations or market pressures. Ghanbari et al. (2019).

Historical data indicates that these deficiencies are significant causes of software failures and vulnerabilities, leading to substantial financial impacts on both software stakeholders and the wider society (Fonseca & Vireira, (2008). Minor defects might be perceived by users as just technical hitches, while severe bugs in software can escalate into safety and security threats that cause far-reaching consequences for stakeholders and even society at large. (Ghanbari et al. 2019.)

Over the past four decades, significant amount of effort and research has been made to enhance the quality of software development processes by introducing various methods, best practices, and tools (Arcos-Medina & Mauricio, 2019). Though these tools and practices can aid developers in spotting and addressing mistakes, some might remain undiscovered until after product delivery. Addressing such deficiencies post-delivery is both costly and time-consuming, particularly for complex systems (Kuutila et al., 2020). Hence, the initial prevention of these issues is the key in maintaining time and budget goals (Ghanbari et al., 2019).

While there has been a concerted effort towards refining the technological and procedural aspects of software development in order to reduce bugs, the psychological and social dimensions have been relatively overlooked (Ghanbari et al., 2019). This oversight is concerning, given that many software deficiencies can arise from bypassing recommended practices or adopting expedient shortcuts. In situations driven by urgency or by other external factors, software professionals, such as requirement analysts, programmers, testers or project managers, might ignore best practices for quicker results, which Ghanbari et al. (2019) refer to as the omission of quality practices. This phrase essentially means that professionals knowingly ignore or bypass established processes and instead opt for practices that risk the overall software quality. Let's discuss this further and examine how and why software professionals decide to ignore quality practices.

Based on Ghanbari et al. (2019) analysis, market environments that firms are active in influence software development approaches and the extent to which they are followed by organizations. Ghanbari et al. (2019) suggest two

main factors: organization's business objectives and the requirements of customers within market environments.

From the business viewpoint, growing up sales and expanding market presence are the key for boosting revenue. On the other hand, reducing development timelines and costs not only increase profits but can be vital for a firm's survival in a competitive business landscape. Driven by these short-term business goals, there's an incentive for companies to accelerate development processes, ensuring they outpace competitors in introducing new or innovative features and products to the market. This proactive approach not only secures an early market entry but can also enhance revenue streams and attract external funding, as suggested by Lim et al. (2012), allowing further product enhancement.

However, moving too fast can lead to shortcuts. In order to reach business goals, development teams might choose to discard some activities related to quality assurance. Ahonen & Junttila (2003) give an example: when sales teams work on offers for clients, they might try to cut costs and time by not involving technical experts, seeing their input as an extra expense and delay. The importance of including tech savvy professionals in project planning has been proven many times and this kind of lack of understanding and aim for short term goals is really hurtful for the overall success of the project (Ghanbari et al., 2019; Kuutila et al., 2020).

In fact, time pressure has been widely reported to lower the quality of developed software (Kuutila et al., 2020). They mention that time pressure during the initial product design and development lowers quality of the code, thus leading to rework and redesign during later product development. Moreover, Kuutila et al. (2020) highlight the risk of unrealistic deadlines causing minimal effort on quality assurance. Regarding this issue, Jones (2006) raises an interesting fact, which is that project managers' might often have an realistic and conservative idea about the estimated development and testing time, but are unable to defend them or convince upper management about the realities of the project's situation.

Another reason for skipping quality steps comes from trying to meet customer needs. Customer requirements are often unclear at the start of software projects. Misunderstanding these requirements can lead to extra work later on as designs need changes. Sometimes, this lack of clarity is used as an excuse to make development quicker or easier. However, as the project goes on, everyone tends to get a clearer picture of what's needed, which again means changes to the initial plans. Therefore, development teams might decide to ignore certain quality practices to aid in being responsive to customers' needs.

Software defects are not just about human and business factors but from the inherent difficulty of software development (Corral et al., 2015). Advanced mobile applications are an example of this as they must operate in constrained environment (Corral et al., 2015). The constraints in this case would be twofold: evolving constraints and inherent constraints. Evolving constraints, such as bandwidth, signal coverage, and current technological limitations (eg., computing power and size of the battery), might eventually be overcome as

technology advances. For instance, a slow data network today may become faster in the future. Conversely, inherent constraints are intrinsic to mobile platforms and unlikely to change soon. Examples include navigating via a limited keyboard or interacting with a relatively small touchscreen. These intrinsic characteristics make designing intuitive and error-free applications a challenge. Additionally, the rapid evolution of terminal devices and the need to incorporate multiple development standards mean that mobile app developers must constantly adapt and innovate. They face the laborious task of ensuring compatibility across diverse protocols, network technologies, and multiple platforms, all while catering to the specific needs of mobile users. This complexity is further amplified by previously discussed strict time-to-market requirements. Software defects, then, can be seen as almost inevitable byproducts of this business. (Corral et al., 2015.)

2.2 Requirements engineering

As discussed already, the quality of a software system depends on how well it fulfills its requirements. To bridge the gap between developers and other stakeholders, such as end user and business partners, in understanding of requirements, requirements engineering (RE) is used to define, document and maintain stakeholder needs for software systems (Shah & Patel, 2014). The main activities of RE is requirements elicitation, requirements analysis, requirements representation, requirements validation and requirements management (Laplante & Kassab, 2022). Of course, different domains face their own intricacies when performing RE, thus the overview of RE by Laplante & Kassab (2022) is not complete. For example, machine learning (ML) systems have introduced a paradigm shift and thus Vogelsang & Borg (2019) argue that traditional RE methods are not applicable as is to ML systems and they require alternative and more extensive methods. Nevertheless, RE activities presented by Laplante & Kassab (2022) is very much valid for most system development, so let's discuss them in more detail.

Requirements elicitation means discovering what the customer wants and needs (Laplante & Kassab, 2022). This activity is not at all trivial, and as Laplante & Kassab (2022) puts it: "elicitation is not like harvesting low-hanging fruit from a tree". The reason for this metaphor is that many requirements are often hidden and require well-defined approaches to be discovered (Laplante & Kassab, 2022). Also, as an important sidenote, common mistake during requirements elicitation is overlooking non-functional requirements (Laplante & Kassab, 2022).

Requirements extracted during requirements elicitation usually include number of problems (Laplante & Kassab, 2022). Requirements analysis is therefore conducted to deal with these problems found from the "raw" requirements. However, the number of problems is often reduced if

requirements elicitation is conducted with good elicitation techniques (Laplante & Kassab, 2022). Possible issues discovered from the raw requirements may be incoherence, contradiction to each other, inconsistency, incompleteness, just plain wrong things or they might have problematic dependencies (Laplante & Kassab, 2022).

Requirements representation takes the analyzed requirements and converts them into some model, such as natural language, mathematics or visualizations (Laplante & Kassab, 2022). This ensures that the requirements are communicated and discussed accurately and are easily understandable. Additionally, proper representation helps the conversion of requirements into system architecture and design (Laplante & Kassab, 2022). Representing requirements can be informal (sketches or natural language), formal (mathematical models), or semiformal. Most often, a combination of these methods is used to achieve comprehensive documentation of requirements (Laplante & Kassab, 2022).

Requirements validation ensures that the gathered and represented requirements truly describe what the customers needs. Requirements validation basically answers the question: "Are we building the right product?" Techniques employed for validation range from inspections and visualizations to text-based tools and other formal methods. (Laplante & Kassab, 2022.)

Software development is a dynamic process, and requirements can change over time. Requirements management deals with the evolving nature of these changes, ensuring traceability and clear communication of any modifications. It's also about managing the broader aspects, such as combating scope creep by using tools that track the mentioned changes to maintain traceability. (Laplante & Kassab, 2022.)

2.3 Software testing lifecycle

Software testing is one of the most essential activities in software quality assurance. This phase within software development life cycle puts software under a series of systematic phases, from test analysis and planning to preparation, execution, and closure. The primary objective of software testing is to evaluate and execute the software to expose any differences between expected and actual functionalities. Software testing aims to validate if the system aligns with its specified requirements by manually testing the software, or by using test automation tools. Through the various phases of the software testing life cycle, errors are not only detected but also fixed.

Software testing lifecycle can be separated into four phases as presented by Hooda & Chhillar (2015). Hooda & Chillar (2015) present the order of the phases as follows: test analysis, test planning and preparation, test execution and test closure. Let’s discuss these phases individually in more detail.

In the software testing lifecycle, the test analysis phase marks the beginning of the testing activities. During this stage, the focus is on analysis of both functional and non-functional requirements, which includes everything from business needs to specific technical specifications. The test team communicates with stakeholders to clarify these requirements that describe the expected outcomes of the test cases. This is also the stage where the team should identify any gaps in both functional and non-functional requirements. Not all requirements, however, are feasible for testing due to certain constraints within the system or the testing environment itself. These untestable elements are important to be communicated by the testing team back to the business stakeholders to ensure transparency and to set realistic expectations. The test team reviews and analyzes all the gathered requirements, determining what tests need to be conducted and assigning priorities to them. (Hooda & Chhillar, 2015.)

Test planning and preparation phase stands as a set up stage that includes several key activities designed to prepare for the actual testing. This phase commences with the creation of a test plan. This document defines the testing scope, objectives, features that will undergo testing, those that won't, the various testing methods to be employed, the roles of each member of the testing team, and the criteria that will determine when testing begins and when it concludes. Central to this phase is the creation of test cases, which are structured documents detailing the steps necessary to test specific functionalities. Each test case includes an action and an expected result. Also, test data is prepared. This data, both valid and invalid, corresponds to each test case, ensuring comprehensive testing coverage. This data can be created by testers or generated through specialized algorithms and tools. The setting up of a test environment usually assigned to a specialized team dedicated to maintaining these testing environments. Once development of the software that is going under testing has reached a certain milestone, the code undergoes a review. After that, a test build is developed, signaling testers to initiate the test execution process. (Hooda & Chhillar, 2015.)

The test execution phase is where the actual testing is conducted. During this phase, testers run the software according to the test cases defined earlier. When differences arise between expected and actual results, testers mark the test case as failed and log these failures as defects and assign them to developers to be resolved. Each found defect goes through a comprehensive logging and tracking cycle which is discussed in more detail in the following subchapter. Just like software development teams, also testing teams use regular reports and meetings, usually daily or weekly, to monitor the project's velocity. These periodic reviews allow teams to discuss the testing progress, giving the testers and project manager overall picture of the project’s status. (Hooda & Chhillar, 2015.)

Finally, test closure includes a review of all test reports, ensuring that the software has been properly tested with an acceptable pass/fail ratio. A conclusive evaluation is made at this stage where it is determined whether all the requirements have been effectively tested and if any critical bugs remain

unresolved or their fixes unverified. Once the manager has reviewed and approved all these artifacts, the software is green-lit for release. A retrospective analysis is conducted with the testing team to reflect upon the entire testing process, identifying successes, setbacks and areas for improvement. (Hooda & Chhillar, 2015.)

In conclusion, through systematic phases including test analysis, planning, execution, and closure, software testing lifecycle ensures that software meets both functional and non-functional requirements. Each phase has its significance, from understanding business and technical requirements to executing test cases and ensuring the software's readiness for release. The software testing lifecycle's goal is to validate the system against its specified requirements, identify errors, and implement fixes. Any differences between expected and actual outcomes are reported and resolved.

2.4 Software testing levels

In order to effectively manage software development, it is abstracted into various levels like modules or components which are tailored specifically for the system or reused from previous systems or libraries (Baresi & Pezzé, 2006). These components are integrated together to form subsystems which in turn are compiled together to form the final system or application (Baresi & Pezzé, 2006). Software testing is conducted in all of these levels, often illustrated by the widely adopted V-model (figure 1). The V-model is a well known methodology that allows effective and systematic software verification and validation (Han et al., 2016). Different testing levels are distinguished as component, integration, system and acceptance testing (Baresi & Pezzé, 2006). As Han et al. (2016) points out, there are multiple variations of the V-model and different research articles use different terminology when discussing about the same concepts. Therefore, this thesis uses the terminology defined by the the International Software Testing Qualifications Board (ISTQB) to comply with industry standards while discussing the software testing V-model. ISTQB is a globally recognized organization responsible for defining standardized guidelines and practices for software testing standardization and certification.

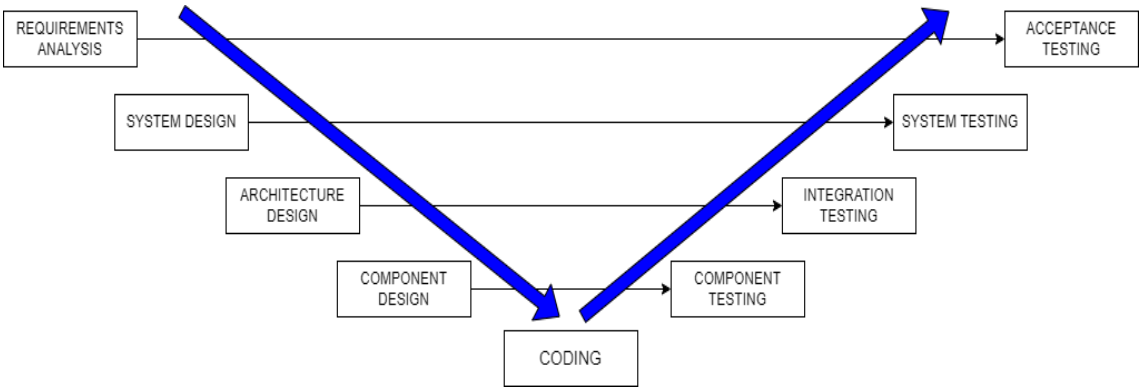

FIGURE 1 Software testing V-model (adapted from Han et al. (2016); ISTQB Glossary, 2023)

Component testing, often referred to as module or unit testing, is a test level that focuses on individual hardware or software components (ISTQB Glossary, 2023). Component testing is conducted in isolation from other software components by the developer (Baresi & Pezzé, 2006). In test driven development, automated component tests are written prior the actual code which yields superior external code quality according George & Williams (2004). In other methods, component tests are written during or after the code is being developed, if at all, as component testing is often viewed as a waste of time or uninteresting by developers (Ghanbari et al., 2019). The goal of component testing is to identify and resolve issues at the component level to achieve a level of confidence in that the new code will not introduce faults or hide existing faults in the current code base (George & Williams, 2004). When talking about component test coverage and adequacy, it implies how much the software is tested and how effective these tests are in covering different aspects of the component such as statements, branches and the ability to handle faulty inputs or other errors (Zhu et al., 1997). According to Baresi & Pezzé (2006), acceptable coverage should be around 85-90%, otherwise it might indicate bad design or neglect for quality.

Integration testing is a test level that focuses on interactions between components or systems (ISTQB Glossary, 2023). In software development, the whole is greater than the sum of it's parts, which means that testing single components is not enough to ensure the quality of the system (Baresi & Pezzé, 2006). Therefore integration testing is conducted to reveal faults when two or more components communicate with each other. It is performed at the component level rather than at the statement level (Leung & White, 1990; Baresi & Pezzé, 2006). As both Leung & White (1990) and Baresi & Pezzé (2006) highlight, all possible interactions cannot be tested as it might prove to be too difficult or cost-ineffective. Baresi & Pezzé (2006) speculate a reason for this, which is that it's very hard to predict interactions between seemingly unrelated components. There are different strategies related to integration testing. One of the least effective ones is the Big bang testing which waits until all components are integrated (Baresi & Pezzé, 2006). More Agile solution is testing the modules incrementally, like in feature-driven strategy where components are integrated

in an order that produces working executable systems as early as possible (Baresi & Pezzé, 2006).

System testing is a test level that focuses on verifying that a system as a whole meets specified requirements (ISTQB Glossary, 2023). System and acceptance testing evaluate the complete system's behavior, including both functional and non-functional aspects (Baresi & Pezzé, 2006). Unlike module and integration testing, which are internal and do not require user involvement, system and acceptance testing consider the end-user perspective (Baresi & Pezzé, 2006). Most module and integration tests focus on functional properties. Some non-functional properties, like modularity, maintainability, and testability, are ensured through design rules, static analysis tools, and manual inspection during development. However, properties like usability and performance require testing with the entire system in addition to user involvement for accurate assessment (Baresi & Pezzé, 2006). Performance and usability tests on early prototypes can guide critical design decisions and mitigate development errors, but final testing on the deployed system is essential to gain reliable data. Other non-functional properties, such as safety and security, are typically handled by specialized teams or individuals working in parallel with the testing team (Baresi & Pezzé, 2006; Camacho et al., 2016).

Final and highest level of the V-model is acceptance testing which focuses on determining whether the system is generally sufficient. Acceptance testing should involve validating the software in a real setting and by the intended audience (Otaduy & Diaz, 2017). Acceptance testing can also be conducted by test engineers or other stakeholders depending on the project. Acceptance testing can be therefore very similar to system testing depending on the situation, but generally the main aim is not anymore to validate the defined requirements (although should still be considered), but to ensure that the software satisfies the customer’s needs (Otaduy & Diaz, 2017). One common way to conduct acceptance testing throughout the development process is with software demos (Otaduy & Diaz, 2017). It is also important to point out, that acceptance testing is usually very critical phase from the contractual and business point of view as it often serves as a milestone where money changes hands and warranty and other legal aspects come into play (Atkins, 2005). These topics are discussed in more detail in later chapters.

In conclusion, the V-model illustrates a structured approach to testing at various levels to ensure comprehensive integration and validation. Component testing focuses on individual components, conducted in isolation from other aspects of the system. Integration testing examines interactions between these components to identify issues that are not evident when testing modules separately. System testing evaluates the entire system against specified requirements and takes into account both functional and non-functional aspects, which include user-focused properties like usability and performance. Finally, acceptance testing validates the software in real-world settings to ensure it meets customer needs and often marking a critical milestone in terms of contractual aspects. Each testing level, from component to acceptance, plays a

vital role in ensuring the overall quality and efficacy of the software and each discovering faults that other levels won't find (Leung & White, 1990).

2.5 Software defect lifecycle

A significant majority of software organizations and IT departments employ a defect management process to identify, track and resolve software defects found within their products (Rahman & Hasim, 2015). According to the model by Rahman & Hasim (2015), key phases of software defect management are defect identification, defect analysis, defect prevention, defect resolution, defect monitoring and defect process improvement. Defect management life cycle (DMLC) is illustrated in figure 2.

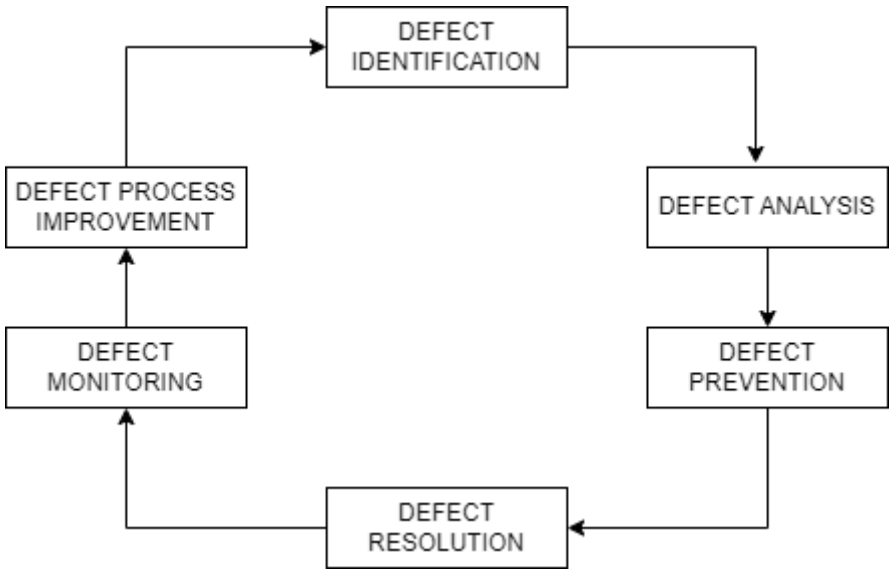

FIGURE 2 Software defect management life cycle (adapted from Rahman & Hasim, 2015)

First phase in the DMLC is defect identification (Rahman & Hasim, 2015). When a defect occurs, it needs to be identified and recorded to the defect management system (Rahman & Hasim, 2015). Discovery of a defect must be communicated to the development team as soon as possible (Lambdatest, 2023). Then, it is the development team's task to acknowledge or reject the defect and move forward with either fixing it or documenting why the defect was invalid (Lambdatest, 2023)

Second phase of DMLC is defect analysis (Rahman & Hasim, 2015). In this phase, Defects are categorized utilizing specific naming conventions for coherence. Such classification offers insights into the various stages and activities within the software development cycle where the defect might have originated. (Rahman & Hasim, 2015.)

Third phase of DMLC is defect prevention (Rahman & Hasim, 2015). Here, the focus shifts to understanding the root cause of the defect. By conducting a root cause analysis, the core issues of the defects are figured out. This knowledge makes it possible to implement preventive strategies to ensure that similar defects don't recur. (Rahman & Hasim, 2015.)

Fourth phase is defect resolution (Rahman & Hasim, 2015). At this point, the identified defects are fixed by the development team (Rahman & Hasim, 2015). The process of fixing the defect starts with assigning it to a developer, who then schedules it for fixing according to its priority (Lambdatest, 2023). Once the issue is fixed, the developer sends a resolution report to the test manager (Lambdatest, 2023).

Fifth phase is defect monitoring, which ensures that the defect management process is effectively and correctly performed at a project level (Rahman & Hasim, 2015). Also, this phase includes defect verification, where the fixes are validated by the test team to assure that the defect has actually been resolved (Rahman & Hasim, 2015).

Final phase is defect process improvement (Rahman & Hasim, 2015). All team members of the project should reflect on and identify the root causes of issues to enhance the process. While it's essential to address high-priority defects during the resolution phase, it doesn't mean that lower priority defects lack importance or don't impact the system's process significantly. Every detected defect is viewed as vital for process improvement. This perspective aids in refining foundational documents, reviewing methods, and adjusting validation processes. By doing so, defects can be identified earlier in the defect management process, making them less costly to address. (Lambdatest, 2023.)

Let’s now discuss in more practical level how newly discovered defect flows through a defect management process. Following overview (figure 3) presents a generalized defect management workflow based on industry recommendations. However, the actual workflow and terminology might vary across different organizations but the main concepts remain the same.

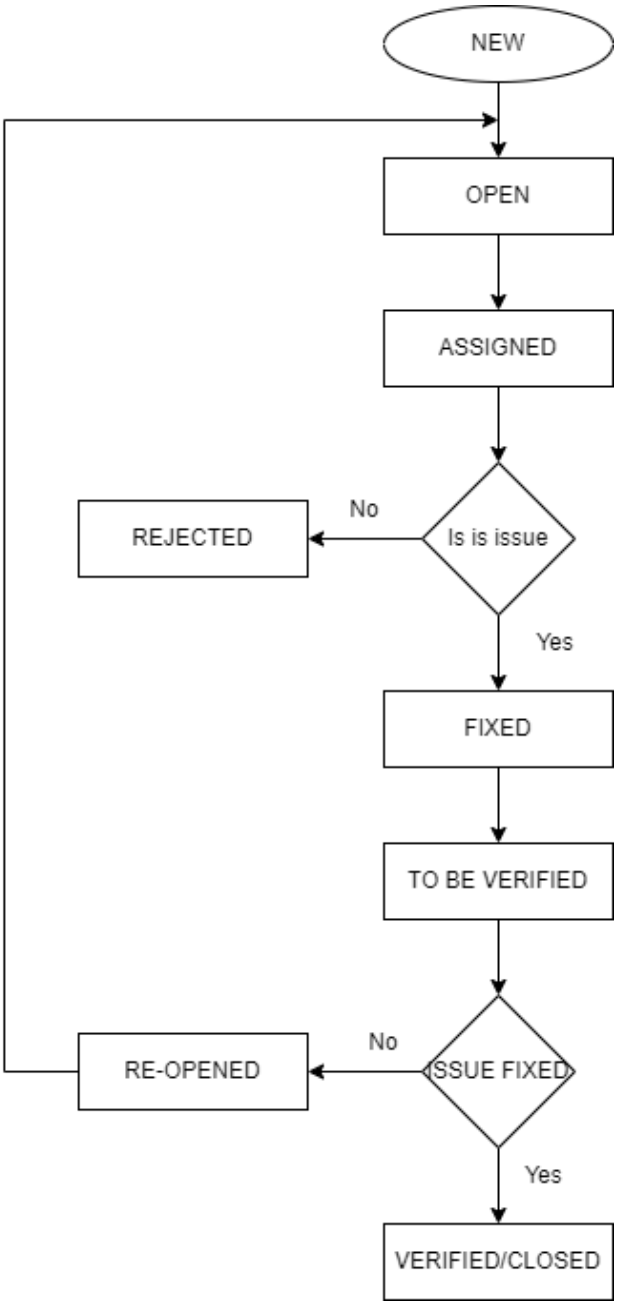

FIGURE 3 Example defect workflow (adapted from Lambdatest, 2023)

As the testing team identifies a defect, they label it as "New" in their chosen project or defect management tool (eg., Jira). Once assessed and deemed valid by a test manager, the bug's status transitions to "Open" and awaits assignment to a developer. The bug is marked "Assigned" when a specific developer is chosen to investigate and address it. Should the developer deem the bug as invalid or irrelevant, its status changes to "Rejected". If a defect is reported more than once or is identical to another, it's labeled "Duplicate". Some defects might be delayed in their resolution due to constraints, giving them a "Deferred" or "Postponed" status. When developers face difficulties replicating a reported defect, they might tag it as "Not Reproducible" or "Need Additional

Information," requesting clearer instructions or more information from the testing team, such as logs or other additional evidence. A known issue in the production environment is labeled a "Known Defect". After resolving a defect, a developer marks it "Fixed," and it is then set to "Ready for Retest." If testers confirm the fix, the defect becomes "Closed." However, if issues persist, the status reverts to "Reopened" waiting to be assigned to a developer once again. (Lambdatest, 2023.)

2.6 Non-functional testing

As discussed previously, non-functional requirements describe the general qualities and attributes of a software system that do not directly define the functions of the software system (Camacho et al. 2016). The nature of non-functional requirements makes them inherently more difficult to test, as they are more vague or less specific compared to functional requirements (Camacho et al. 2016). Let's discuss few key non-functional testing methods in more detail.

First, performance testing validates several characteristics of a given system, such as speed, velocity, scalability and stability (Meier et al., 2007). The key question to which performance testing aims to answer is "what is system performance if we change this software configuration option or if we increase the number of users?" (Jiang & Hassan, 2015). The idea is to gather information about the system in terms of response times, throughput and resource-utilization in order to meet the performance requirements, or if they are not defined (which often might be the case), a reasonable level of performance (Crispin & Gregory, 2014). Performance testing can evaluate the system as a whole, or focus on different components of the system, such as the graphical user interface or data storage solution (Jiang & Hassan, 2015). Performance testing can also be used to evaluate and compare design and architectural decisions, algorithms and system configurations (Jiang & Hassan, 2015).

Secondly, load and stress testing, which is related to performance testing, inspects the system's behavior under normal and heavy loads (Camacho et al. 2016). Load and stress testing is important when operating large scale software systems as the amount of concurrent requests might range from thousands to millions (Jiang & Hassan, 2015). The purpose is to verify that the system functions correctly and as expected under load and to identify possible load related problems such as deadlocks, crashes and memory leaks (Jiang & Hassan, 2015). The difference between load and stress testing is in the severity of the imposed load (Jiang & Hassan, 2015). Additionally, stress testing takes into account component failures (eg., severe database errors) in order to test overall resilience of the system. Scenarios for load and stress testing can be

created using tools, which automatically create requests to the system, or by reducing the amount of compute resources available (Jiang & Hassan, 2015).

Thirdly, software security testing is the process of assessing and evaluating a software application to discover vulnerabilities, weaknesses, and flaws that could be exploited by attackers. The primary objective is to identify potential threats and ensure that the software is resistant to malicious attacks, ensuring data protection and system integrity. Techniques include static application security testing (SAST), dynamic application security testing (DAST), and penetration testing, among others. The end goal is to ensure that software operates securely and can resist unauthorized attempts to disclose, modify, or deny access to data. (Takanen et al., 2018.)

Additionally, there are multiple other ways to evaluate the non-functional qualities of software systems. Usability testing can be conducted to evaluate user interfaces to identify design issues from the user's perspective to ensure comfortable and efficient user experience (Bandi & Heeler, 2013). Accessibility testing ensures that that the developed software is usable by as many people as possible, independent of their physical capabilities (eg., people with visual impairments) (Bai et al. 2017). Also, in the context of mobile applications and embedded systems, battery consumption should be considered and tested (Silva et al., 2017). Finally, if the system supports multiple languages, localization tests should be conducted to ensure that the system is translated fully and correctly for the end user (Camacho et al. 2016).

To summarize, non-functional requirements focus on the overall qualities and attributes of a software system, separate from its explicit functions. Testing these requirements can be challenging due to their less specific nature. Key methods to evaluate non-functional aspects include performance testing, which assesses system speed, scalability, and stability under various configurations and user loads. Another vital method is load and stress testing, which observes the system's behavior under normal and increased loads to detect potential issues like crashes or memory leaks. Software security testing scans the system for vulnerabilities and weaknesses, aiming to safeguard it against malicious threats. Moreover, usability testing examines user interface design from the user's viewpoint, while accessibility testing ensures the software is usable by those with physical limitations. For mobile apps and embedded systems, battery consumption is a critical factor, and systems supporting multiple languages require localization tests to guarantee accurate translation for users.

2.7 Test automation

Automated software testing involves using technology to perform test activities that are typically done manually. This automation is carried out using specialized software known as test tools. (Garousi & Mäntylä, 2016.) Study conducted by Rafi et al. (2012) presents main benefits of test automation, which

are reusability, repeatability and effort saved in test executions. These results highlight the fact that test automation is the superior decision when several regression testing rounds are needed (Rafi et al., 2012).

Nowadays, most prominent commercial or open-source software incorporates automated test suites to ensure its proper functionality. This is particularly true for software projects that undergo multiple iterations, as the benefits of automated testing are most evident during regression and repetitive testing. For numerous large scale systems, the size and complexity of automated test suites continue to grow. (Garousi & Mäntylä, 2016.) Therefore, making informed decisions about when and what to automate is crucial, as incorrect choices can result in significant waste of resources and efforts (Garousi & Mäntylä, 2016).

When determining which parts of the system should or should not be automated, it's essential to consider the following factors. First factor is the rate of change of the component under testing. If what is being tested is subject for major changes, automation maintenance costs can escalate. Another consideration is the frequency of test executions. It's important to assess how valuable each test result is and the associated costs of obtaining it. Lastly, the ongoing value of automation should be evaluated. Do automated tests consistently provide value by either identifying bugs or confirming significant attributes of the software, such as specific scenarios? (Garousi & Mäntylä, 2016.)

The theoretical ideal for automated testing is to achieve 100% automation test coverage. Yet, this goal has not been realized in actual practice and mostly likely won’t be in the near future. Reason for this according to Rafi et al. (2012) is that some tests require subjective evaluation of the system (eg., some non-functional tests such as usability) or extensive knowledge in the domain in which software is used to which modern testing tools are not yet capable of. However, by automating all the possible well-defined and repeatable test cases, the tester’s time is freed up for the types of testing from which manual testing gets the most value out (Rafi et al., 2012).

3 SOFTWARE SUBCONTRACTING

Outsourcing of software development has been a popular practice for decades now (Dey, Fan & Zhang, 2009). Outsourcing is a management strategy that allows organization to acquire technical expertise, benefits of economies of scale or labor to dampen demand fluctuations in order to focus on the core competencies and business objectives (Assmann & Punter, 2004; Dey et al., 2009). However, hiring a outside contractor can prove to be a challenging operation despite the benefits, as the incentives between the client and subcontractor may diverge in addition to information asymmetry and communication issues (Dey et al., 2009). Another popular theme in software outsourcing literature is international outsourcing into developing countries. Global Software Development (GSD) is widely adopted due to its ability to decrease software development costs as vendors in developing countries cost significantly less than in-house operations (Ó Conchúir et al, 2009; Deshpande et al, 2011; Niazi et al., 2016).

In this chapter, software subcontracting is discussed from various points of view in order to understand what sets subcontracting projects up for either success or failure. Topics discussed in this chapter include contracts, different subcontracting models, criterias for selecting a subcontractor, financial considerations, risk management and finally management of subcontractor relationships.

3.1 Software development contracts

The world of software contracts is interesting, as software engineering as a process is highly complex which introduces multiple unique challenges in terms of contracts that are not present in most industries (Dey et al., 2009). Two main characteristics of software development contracts are the need to support

incomplete requirements specification and difficulty of quality assessment (Dey et al., 2009). These are the two main reasons why fixed prices are rarely seen in software development contracts as they impose a high risk to budget overruns due to change requests and bug fixes (Dey et al., 2009). That being said, a typical software development contract between a client and a subcontractor (vendor) includes a variety of interrelated issues (Dey et al., 2009). The main issues that should be agreed in the contract include quality of the system, delivery timeline, effort and cost of the project, payment and maintenance of the system postdelivery (Dey et al., 2009). Let's discuss these characteristics of software development contracts further.

In Dey et al.'s (2009) research paper they analyzed 15 software outsourcing contracts to uncover factors that are important in the outsourcing process. Firstly, project type or more precisely project complexity is a major factor in contractual relationship (Dey et al., 2009). Mission critical projects require higher resource allocation and effort from the vendor (Dey et al., 2009). Additionally, the more complex the project, the detailed the customer specification and requirements become (Dey et al., 2009). As mentioned already, software development contracts need to deal with certain uncertainties common for software projects. In most cases, neither functional nor non-functional requirements are complete at the start of the project (Horkoff et al., 2019). Therefore software contracts require a certain degree of flexibility when it comes to project's scope. It is also important to note that the uncertainties may increase alongside increasing requirements (Dey et al., 2009).

Software projects along with their associated contracts are usually separated into milestones along a specified timeline. This timeline may usually be tied to payment terms (Dey et al., 2009). One of the common milestones in which payments are made is during acceptance testing where if the client approves the system, the vendor receives the agreed sum (Atkins, 2005). The importance of acceptance testing on the contract and on the subcontracting process in general is discussed later in this chapter.

Software support is another major element in software development contracts. Most contracts mandate a warranty period where the vendor is obligated to correct defects after the software's deployment (Dey et al., 2009). This period varies, but it's essential for ensuring that the software functions as intended post-delivery. The length and terms of this support are often a point of negotiation and reflect the project's complexity and the developer's confidence in their work (Dey et al., 2009).

The sophistication and knowledge of the client in IT and software development also play a significant role in the contracting process. Clients with greater understanding and experience in these areas tend to specify contracts in more detail (Dey et al., 2009). This detailed specification includes aspects like system functionalities, performance expectations, and the scope of post-delivery support. Their familiarity with the software development lifecycle enables them to set more precise expectations and effectively manage potential risks (Dey et al., 2009).

Final characteristic that Dey et al. (2009) mension is the measurability of project quality. This involves setting clear performance standards and metrics for the project. When companies can define these quality measures explicitly, they are more likely to draft detailed contracts incorporating these standards (Dey et al., 2009).

3.2 Subcontracting models

Subcontracting can take many forms in the software industry. Unfortunately, there is no definitive model describing all potential intricacies of different ways to outsource software development. There exists however multiple classification models for software subcontracting from different perspectives. Let's go over classifications by Kobitzsch et al. (2001), Penttinen & Mikkonen (2012) and Minetaki & Motohashi (2009), each taking a unique view to gain a basic understanding of different ways in which software subcontracting can take shape.

Kobitzsch et al. (2001) base their categorization model (Table 1) on legal relation between the participating companies and the setup of the teams as they argue they are the most distinguishing features based on their experiences. In Kobitzsch et al. (2001) paper, cooperation models are separated on the team setup axis by the amount of teams, and legal relationship where the companies involved are either legally independent or legally related. Model 1, which is separate teams in basically independent company is the standard contractor-subcontractor relationship (Kobitzsch et al., 2001). All legal, knowledge-transfer, development and project management, and quality management issues apply here (Kobitzsch et al., 2001). In case that the independent companies are situated on different sides of cultural borders, these issues are amplified and language, time, and infrastructure issues might emerge (Kobitzsch et al., 2001). Model 2 is separate teams in legally related companies which means a specialized contractor-subcontractor relationship between mother and daughter companies (Kobitzsch et al., 2001). Implications of this model are the same as model 1, but issues in legal, knowledge-transfer and project and quality management topics are easier to solve as the mother company owns the subcontractor which reduces conflicts of interest (Kobitzsch et al., 2001). According to Kobitzsch et al. (2001), many large companies (Lucent, Siemens and Nokia) have adopted this model and set up development sites in India that are legally related to their own company. Model 3 is one team distributed across multiple sites of legally related companies (Kobitzsch et al., 2001). Kobitzsch et al. (2001) highlight the importance of project and quality management due to the scattered nature of this setup. In case the sites are distributed in different nations, language, time, and infrastructure also become challenges (Kobitzsch et al., 2001). Model 4 is one team distributed across multiple sites of several basically independent companies (Kobitzsch et al.,

2001). This is the common mode for globally operating company according Kobitzsch et al. (2001). The implications of this model are the same as model 3's but legal issues might additionally become a challenge (Kobitzsch et al., 2001).

TABLE 1 Four cooperation models based on companies' legal relationship and team setup (adapted from Kobitzscgh et al., 2001)

	Independent companies	Legally related companies
Separate teams	Model 1	Model 2
Single team	Model 4	Model 3

Penttinen & Mikkonen (2012) on the other hand base their subcontracting models purely on the team setup. According to Penttinen & Mikkonen (2012), there are three different types of subcontracting model's in the software industry, which are called subcontractor team, mixed team and virtual team. Subcontractor team is the most straightforward to setup (Penttinen & Mikkonen, 2012). As the name implies, the team consists only of subcontractors, but subcontractors do not have to be from the same company but must have relevant experience (Penttinen & Mikkonen, 2012). Mixed team includes team members both from within the company and from the subcontractor(s) (Penttinen & Mikkonen, 2012). The selection of team members is based on criteria like skills, technical and personal character, suitability, and managerial fit (Penttinen & Mikkonen, 2012). However, it is important to not become overly dependent on the subcontracted team member's, as the nature of their contracts are not permanent and might change rapidly (Penttinen & Mikkonen, 2012). Therefore, competence transfer to internal team members is important to ensure continuity after the subcontractor leaves (Penttinen & Mikkonen, 2012). Third model, virtual team, involves team members who are located at different locations, often in different countries, working together (Penttinen & Mikkonen, 2012). This is most common in companies practicing global software development and is more difficult when it comes to communication and management topics compared to the other models (Penttinen & Mikkonen, 2012). Additionally, virtual teams require robust communication culture, infrastructure and tools to ensure effective collaboration (Penttinen & Mikkonen, 2012).

Minetaki & Motohashi (2009) take more economics based perspective on software subcontracting models (Table 2). Their classification is based on two factors. First is the outsourcing cost ratio, which is the proportion of total cost occupied by outsourcing cost (Minetaki & Motohashi, 2009). Second factor is the intra-industry sales ratio which is the proportion of sales contributing to the sales within the information service industry as a whole (Minetaki & Motohashi, 2009). This classification results in four different types of

contractors. First, prime contractors are large enterprises that are positioned at the top of the pyramid of industrial organizations. They tend to have a high proportion of sales outside the software industry and largely outsource to subcontractors (Minetaki & Motohashi, 2009). Second, intermediate subcontractors are enterprises that typically receive orders from prime contractors and also subcontract to other entities. They have both high outsourcing costs and high intra-industry sales ratios. Minetaki & Motohashi (2009) also report that intermediate subcontractors, despite being a significant part of the industry, have the lowest productivity. Third, end contractors are enterprises that do not place orders with enterprises beneath them, indicated by a low outsourcing cost ratio, but they have a high intra-industry sales ratio, which in turn indicates that their sales are largely for intermediate subcontractors (Minetaki & Motohashi, 2009). Finally, independent enterprises are businesses that function autonomously and do not fit into the subcontractor-prime contractor structure (Minetaki & Motohashi, 2009).

TABLE 2 Classification of enterprises in software industry (adapted from Minetaki & Motohashi, 2009)

	Above average intra-industry sales ratio	Below average intra-industry sales ratio
Above average outsourcing cost ratio	Intermediate subcontractors	Prime contractors
Below average outsourcing cost ratio	End-contractors	Independent enterprises

To summarize, Kobitzsch et al. (2001) categorize software subcontracting based on legal relations and team setups into four models: independent teams in separate companies, teams in legally related companies, a singular team across multiple sites of related companies, and a singular team across independent companies. Penttinen & Mikkonen (2012) offer a simpler classification based on team composition alone: subcontractor teams composed only of subcontractors, mixed teams combining company employees and subcontractors, and virtual teams spread across different locations. Minetaki & Motohashi (2009) approach the categorization economically, identifying four types of enterprises within the subcontracting hierarchy: prime contractors at the top, intermediate subcontractors who also outsource, end-contractors selling mainly to subcontractors, and independent enterprises operating autonomously.

3.3 Criteria for selecting software subcontractors

There are multiple aspects to consider while evaluating potential subcontractors. This subchapter aims to explore some of the most relevant topics in this subject in order for the reader to gain understanding what to look for in a subcontractor to avoid most common pitfalls. The assumption is that the reader has decided that taking in a subcontractor is the best way to execute or and complete the project. The topics are divided roughly into two categories: evaluating competence and technical expertise of the team and evaluating portfolio and past project success.

First of all, overall competence of team members should be one of the main things to consider. Both Khan et al. (2021) and Rahman et al. (2021) highly suggest to ensure that the team members are properly trained and skilled in relevant technologies and tools. On top of this, Seppänen (2002) and Rahman et al. (2021) recommend to inquire about subcontractors knowledge on the clients application and business domain. The client should also ensure that the subcontractor follows well-established and Agile architecture for their software development and maintenance processes in addition to maintaining thorough project documentation (Khan et al., 2021). In regards of subcontractor's physical infrastructure, internet connectivity, servers and data centers need to be deemed robust to enable high quality service delivery and collaboration in distributed project (Rahman et al., 2021). The client should not forget to also evaluate subcontractor's security measures and protocols (Khan et al., 2021).

On more general level, the client should look into the subcontractor's portfolio, past project success and reputation within previous clients. By looking into past projects, the client gets an idea whether their management is effective and if they are able to maintain collaborative relationships with their clients (Seppänen, 2002; Rahman et al., 2021). By going over client testimonials, it can be verified that wether subcontractors adhere to industry standards, have a history of reliable and ethical business practices, and possess certifications that validate their technical competencies (Khan et al., 2021)

3.4 Financial considerations

Switching to a subcontractor will financially impact the company both in short and long term. According to transaction cost theory (TCT) by Williamson (1991), organizations will choose to outsource functions when the estimated overall transaction costs of doing so are lower than the production cost of performing these functions internally (Dhar & Balakrishnan, 2006). Williamson (1991) argues that outsourced work translates to lower production costs due to economies of scale. However, this means that the transaction costs are high as

subcontractors need to be managed and monitored. Inversely, performing functions internally can indicate high production costs but low management cost (Williamson, 1991). In the context of software development outsourcing, transaction costs include the cost of searching potential subcontractors as well as negotiating and managing relationships with them (Dhar & Balakrishnan, 2006).

Second factor that affects both production and transaction cost is asset specificity (Dhar & Balakrishnan, 2006). Williamson (1991) defines asset specificity as the degree of customization of the transaction. Asset might refer to anything from physical or human assets to software assets (Dhar & Balakrishnan, 2006). In any case, high asset specificity indicates high transaction costs. High asset specificity also increases production costs because specific assets have limited utility in other markets (Hirschheim & Lacity, 1993).

Third consideration is exposure to opportunism threats (Williamson, 1991). Both subcontractors and internal employees may show opportunistic behavior but subcontractors are more likely to do this and in more bigger scale (Dhar & Balakrishnan, 2006). Managing subcontractors becomes more difficult if they are opportunistic and thus management costs can increase (Dhar & Balakrishnan, 2006). Risk for opportunistic behavior increases when there are only few vendors in the market (Hirschheim & Lacity, 1993). Study by Kobelsky & Robinson (2010) also confirms this as a major concern and thus should be thoroughly evaluated. In this case, organization may not save much by outsourcing because the vendor may charge excess or may not perform as promised (Dhar & Balakrishnan, 2006). This can be potentially combated in the contract by imposing penalties for non-performance, but this is also highly dependent on the clients bargaining position (Dhar & Balakrishnan, 2006).

Study conducted by Gopal & Goka (2010) reveals that the pricing model should also be considered carefully as they found out that it can impact the quality of the software either positively or negatively. Gopal & Goka (2010) differentiates between fixed price (FP) and time and materials (T&M) contracts. They suggest that FP contracts incentivize vendors to optimize efficiency and effectiveness due to fixed revenue gap, which leads to higher quality and profit margins. The incentive structures in FP contracts leads to higher quality because according to Gopal & Goka (2010), it motivates project managers to allocate resources more strategically, prioritizing projects with better-trained personnel.

T&M are less riskier for the vendor because it allows overflowing costs to be passed on to the client (Gopal & Goka, 2010). From the client's point of view, this worse model because it offers weaker incentives for efficiency (Gopal & Goka, 2010). However, in T&M contracts, the link between quality and profitability is less straightforward, with potential for both higher and lower quality outcomes depending on the project management and design processes. (Gopal & Koka, 2010).

3.5 Risk management

Outsourcing software development does not come without it's risks. Shah et al. (2014) reports that there are many challenges within the industry, such as cultural differences, information security threats and lacking educational policies, that clients wanting to outsource their software development need to take into account. Furthermore, these risks are increased when considering international outsourcing (Kobitzsch et al., 2001), especially into developing countries that bring their own intricacies like corruption and legislation issues into the discussion (Wang & Shi, 2009). In this subchapter, potential risks in subcontracting in academic literature is discussed. Secondly, viable risk mitigation strategies are explored and finally, possible exit strategies are briefly presented.

In global software development there are risks associated with the stability, infrastructure and government policies of the vendor's country (Smith et al., 1996; Kobitzsch et al., 2001; Shah et al., 2014). Intellectual property (IP) rights and safety of sensitive information should be a major concern when selecting the country and the vendor, but also when drafting the subcontracting agreements (Smith et al., 1996). For example, Wang & Shi (2009) report that outsourcing operations are a widespread phenomenon in China, but Li & Alon (2020) point out the generally known fact that China is known for gross intellectual property rights violations in a systematic scale. Li & Alon (2020) elaborate further that there are no systematic, open and fair channels for the IP owners to demand independent, unbiased court on such matters as the the only political party has full control of the courts and may choose to protect or not to protect international IP owners according to its own agenda. These types of issues are present all over developing countries on varying levels. Potential risk for corruption in other forms, such as exploitation or poor working conditions must also be recognized and evaluated according to the country (Smith et al., 1996). Unfortunately, legal issues are not limited to information security and corruption issues, but also compliance issues. Differences in legislation and legal systems across national borders can make the enforceability of contracts more difficult when problems or disagreements arise (Smith et al., 1996). Therefore, matters like cost overruns and quality issues need to be rigorously defined in order to minimize the risk for legal actions in foreign courtrooms (Smith et al., 1996). Operating internationally introduces also geopolitical risks, such as significant currency fluctuations, international sanctions or impact of armed conflicts (Smith et al., 1996; Wang & Shi, 2009).

As outsourcing software development often means that the client and vendor operate from different locations and different ways, it increases the difficulty of effective communication which in turn imposes risks for poor collaboration, flow of information and knowledge sharing (Smith et al., 1996; Shah et al., 2014). Subcontracting structures may introduce long and rigid communication chains which often leads to delayed decision-making and

unresolved issues (Shah et al., 2014). Shah et al. (2014) mentions that risks of long communication chains are often realized during time sensitive phases such as during test executions where testers might struggle to obtain necessary information in time. This includes difficulties in understanding bug fixes or identifying appropriate contacts for clarification (Shah et al., 2014). Other communication issues might arise from language barriers, cultural differences and time zone variations (Smith et al., 1996).

Subcontracted software projects add a layer of difficulty also to managerial and business topics. When outsourcing their software development, companies also outsource portion of their reputation to vendors. Therefore, potential delays or PR issues on the vendor's side are reflected upon the client (Smith et al., 1996). Quality of the software developed by the vendor should be also a major concern (Smith et al., 1996). Unfortunately, Shah et al. (2014) report that clients often have poor visibility into the vendor's testing activities and quality expectations between the client and vendor might differ significantly. This lack of clarity often resulted in unaddressed issues and hindered smooth testing operations (Shah et al., 2014). Finally, depending on the project, there is a risk of becoming overly dependent on the vendor for critical business functions or proprietary knowledge of the system which might result in a difficult vendor lock-in (Shawosh & Berente, 2019). Companies may become overly dependent on a particular vendor due to the specificity of software being developed which has multiple business implications. Higher asset specificity diminishes client's bargaining power in further negotiations for maintenance, change requests etc. (Shawosh & Berente, 2019). Moreover, the uniqueness of the software indicates higher vendor switching cost especially in cases where proprietary technologies or specialized knowledge is required. (Shawosh & Berente, 2019). Also, vendor locked companies are subject for opportunistic vendor behavior such as price increases or taking more control over the software development process (Shawosh & Berente, 2019). To mitigate these risks beforehand, Shawosh & Berente (2019) propose using popular technologies wherever possible to maximize the amount of other potential vendors in the market. They also suggest insisting the vendor to comply with open or widely accepted software development standards in order for the software to not be tied to proprietary platforms. Finally, Shawosh & Berente (2019) recommend acquiring and maintaining key personnel with critical knowledge in-house whenever possible to not rely entirely on the vendor in domain expertise.

If however, some of these mentioned risks are realized in serious ways, the project might need to be abandoned. In contrast to total abandonment, Pan (2008) suggest partial abandonment as a strategic exit option. Partial abandonment is defined as termination of some, but not all project activities before the system is fully implemented (Pan, 2008). This approach can help extract any existing value out of the work already done and mitigate further losses (Pan, 2008). This strategy is especially relevant in scenarios where continuing with the original scope is no longer feasible or desirable due to various constraints or challenges (Pan, 2008).

3.6 Building and managing relationships

Establishing strong and fair relationships between the client and subcontractor is a critical success factor for effective and efficient outsourcing (Assmann & Punter, 2004). In order to form these tight relationships, strong and honest communication is required from both sides. To achieve this, Khan et al. (2019) propose building personal relationships with the subcontractors. Employees should be encouraged to form these relationships by promoting open communication channels between stakeholders like face-to-face meetings, direct instant messages, video conferences and organizing frequent onsite visits (Kobitzsch et al., 2001; Khan et al., 2019). Khan et al. (2019) also recommend organizing activities that include knowledge and information sharing between teams and team members in order to better understand each other. Also, recruiters should pay attention to candidates cross-cultural skills if the subcontracting relationship is international in nature (Khan et al., 2019).

Assmann & Punter (2004) writes that outsourcing in the software industry has evolved from customer-vendor relationship more into partnership. Hence trust and fair relationship between the customer and subcontractor has become increasingly important (Assmann & Punter, 2004). Despite open and freely flowing communication, it can take years to build enough trust to form smoothly operating software development processes and quality management systems between the contractor and subcontractor as different sites can view sensitivity of data very differently (Kobitzsch, 2001). Organizations should strive for mutual trust because the importance of exchange of development knowledge cannot be exaggerated (Assmann & Punter, 2004).

Another important aspect of building inter-stakeholder relationships is team stability. Study by Narayanan et al. (2011) found that team stability positively influences project management and customer satisfaction and also interacts with project size, planning capabilities, and communication effectiveness. Team stability in this case means the consistency and continuity of team members working on a project (Narayanan et al.,2011). Team stability manifests itself as low employee turnover, consistent collaboration and in cumulative knowledge, skills, reliability and predictability, improved efficiency (Narayanan et al.,2011). This means that in either client or subcontractors side there are minimal amount of employees leaving or being replaced, which helps in maintaining a consistent work rhythm and understanding among team members (Narayanan et al.,2011). A stable team allows for the accumulation and retention of project-specific knowledge and skills. Team members become more familiar with the project's requirements, objectives, and challenges. Familiarity among team members with each other's working styles can lead to better coordination and fewer misunderstandings. Over time, stable teams often develop more efficient ways of working together, leading to improved productivity and better quality (Narayanan et al.,2011).

4 SUMMARY OF THEORETICAL BACKGROUND

Chapter 2 presented key concepts and terminology for this thesis based on prior literature from academia and software industry. Core terms like software quality, quality assurance and software testing were explored to form a sufficient understanding to then discuss key processes for this thesis. It was found that software testing is conducted at various levels, all with their own methods and tools. Software testing V-model was presented to help conceptualize these levels. Non-functional testing was discussed in more detail in order to show that there is a lot more to software testing than verifying inputs and outputs. Finally in regards of software testing, test automation was briefly discussed in order to understand how it can help reduce repetitive testing so that test engineer's time is freed up for testing types that require skills that only humans currently posses.

Second half of literature review discussed subcontracting software development. First, software development contracts were investigated to figure out what makes them different from other industries. The main finding was that software contracts need to support ever changing requirements, because all requirements are rarely known at the beginning of the projects, which makes the project's scope volatile. Next, different ways to categorize software subcontracting was presented. Three different models were found, each taking a different point of view, ranging from team composition into more economical stance. Lastly, the thesis presents different considerations that should be made when choosing a subcontractor and wether to choose one at all and do things in-house. Discussed topics ranged from validating competence of the subcontracting team to financial aspects like calculating in detail wether subcontracting is the cost effective option. Heavily related to this, risk management was another major topic as failed subcontracting projects and relationships can become very expensive for the client. Therefore different risks were mapped, some of them mainly affecting off-shore outsourcing to developing countries and some affecting all scenarios, like long and rigid communication chains.

The thesis will return to these concepts as the discussion chapter presented in chapter 7 will rely on works presented in the literature review. Pieces of key literature are used evaluate the findings from the empirical portion of the study in order explain their implications, strengthen their validity and to put them into context of existing research. Collection of central literature consists of scientific papers written about software outsourcing contracts (Dey et al., 2009), vendor-side experiences of global software testing (Shah et al., 2014), managing outsourced software projects (2011), vendor lock-ins in software development (Shawosh & Berente, 2019) and requirements engineering (Laplante & Kassab, 2022) among others. These contain highly relevant insights to the findings of the case study in terms of both challenges and solutions. Before examining these challenges and solutions, the thesis will explain the selected methodology used to conduct the study.

5 METHODOLOGY

The structure of this chapter is the following. First, explanation of the chosen methodology to answer research questions is presented. Next, the chapter shifts to an in-depth look at the case company and it's subcontracting scenario. Report of data collection and analysis is then presented, and the chapter is concluded with a discussion on research ethics of this thesis.

5.1 Selected methodology

Qualitative research methodology was chosen for this thesis. Qualitative research aims to investigate the subject comprehensively in order to understand it as thoroughly as possible (Hirsjärvi et al., 2009). Qualitative research methods include observation, interviews and the use of written sources among others (Hirsjärvi et al., 2009). The results that these methods produce are in-depth when comparing to quantitative methods (Hirsjärvi et al., 2009). However, the trade-off is that qualitative methods are not as generalizable as quantitative methods. For the purposes of this thesis it was important that the participants are able to thoroughly describe the issues that they personally have faced in the subcontracting scenario. According Hirsjärvi et al. (2009) this can be achieved through interviews.

The chosen interview method was semi-structured interviews. It allows for flexibility which was important, because the researcher did not want to limit the participants potential answers, but to enable the participants express their views, opinions and experiences freely. This also gives room for the researcher to ask additional questions if something is unclear.

The interviews were decided to be conducted in focus groups. Focus groups are semi-structured, unofficial meetings where the participants discuss together about the selected topics, moderated by the researcher (Carey &

Ashbury, 2016). Focus groups are used to collect comprehensive and detailed information, which was required for this thesis. The goal is to arrange the meetings so that the participants feel at ease and comfortable (Carey & Ashbury, 2016). The choice was made over individual interviews, because focus groups allow for interaction between participants which can lead to more profound observations (Carey & Ashbury, 2016).

5.2 Case description

Company in question is a major IT company with offices all over the world and diverse product portfolio. The case study examines a team responsible of one of those products by interviewing employees, such as test engineers and managers. The project utilizes a subcontractor situated in another country who is largely responsible for the development of the software system. Unfortunately, the subcontracting relationship has proven to be difficult largely due to software quality issues which has significantly affected the company's activities.

The subcontracting relationship is slightly complicated and requires further elaboration. The case company has hired the subcontractor to develop a system, which is based on the subcontractor's own product. It is important to note that the subcontractor also sells the product independently to their own customers. The case company sells the subcontractor's software to their own clients but with additional features and changes that they have requested to be developed. This means that the case company and the subcontractor compete in the same markets and with similar products. When the case company's clients request changes, bug fixes or new features, the case company handles them and orders them from the subcontractor. Therefore, there is no contact between the case company's clients and the subcontractor. Furthermore, the case company has also own development activities which are based on the SDK provided by the subcontractor. The case company also uses separate subcontractor teams in their own development.

Employees in the case company have found this model difficult. The main issue has been poor quality which has resulted in slow delivery times and impacted various processes, especially on the quality assurance side. The motivation for the case company to participate in this study was to gather the experienced problems and proposed solutions in one place so that the cooperation with the subcontractor and internal processes can be improved.

The thesis project was started in summer 2023 by inquiring representatives of the case company about suitable topics for a master's thesis. The issues in quality assurance came up quickly and the idea was refined further to investigate what unique or interesting aspects and difficulties in quality assurance might emerge when the software is developed by a

subcontractor. The representatives in the case company agreed that this was a good topic and the writing process could be started in fall of 2023.

5.3 Data collection

For data collection method, semi-structured group interviews were chosen. The data was collected during December 2023 in four sessions, which three included three participants and one included only single participant due to scheduling related reasons. To be noted that all interviewees participated only to one interview session. The small group size allowed the gathering of more in-depth information, as everyone had time to speak about their own experiences and views on the topics.

The chosen topic automatically defines who could be interviewed for the study. The participant had to be involved in quality assurance and with the subcontractor either directly or indirectly. 14 individuals were identified to be suitable for the study. Invitations were sent to all 14 individuals, of which 10 participated. No rewards were provided to the participants either by the researcher or the case company. Due to relatively small sample size, the gathering of personal information was left to minimum to protect the participants' identities. The participants were only asked define themselves either as a manager or as a test engineer. Participants consisted of 3 test engineers and 7 managers.

The interview was divided into five topics based on the software testing lifecycle presented in chapter 2.1.4. The topics included the four phases in software testing lifecycle (test analysis, test planning and preparation, test execution and test closure) and additionally general cooperation. The researcher decided to move creation of test cases from test planning phase to test analysis phase, so that during test analysis participants would mainly discuss requirements, documentation and making the test cases based on those. Participants could then focus on scheduling and environment topics in the test planning phase. The interview questions and structure can be found in appendixes of this thesis. In the first four topics, the participants were asked to first describe how the process currently is practiced in the case company and evaluate the maturity on a 1-5 scale to give the interviewer general idea about the state of the process. The participants were then asked speak about the problems and possible solutions about the respective testing phase. The phases and questions served more as a rough guideline to make sure intuitive flow of the interview and to achieve good coverage on various topics. The participants answers often intertwined problems and solutions, overlapping different phases which was expected and encouraged. If needed, the interviewer asked additional questions.

The participants were informed in advance by e-mail about the progression of the interview. They were also asked to provide written consent to

data collection and data use and anonymization. The participants were asked in advance to familiarize themselves with the given information and ask the questions before the interview. All questions were answered before the start of the interview session. For every interview, the researcher had reserved 60 minutes. The interviews themselves lasted approximately 55 minutes, as the first 5 minutes were used to explain the topics one more time and to answer any remaining questions.

The interviews were recorded using tools provided by the case company, in the case that the recordings included sensitive information. The recordings were then transcribed and responses anonymized, while also removing any sensitive information, such as names of stakeholders, descriptions about products and technical information.

5.4 Data analysis

Content analysis is a widely adopted systematic way to describe qualitative research data (Schreier, 2012). The data for this thesis consisted of 47 pages of transcribed text collected from interviews. Content analysis as a process consists of building the coding framework, coding the text according to the framework, testing the coding framework, evaluating and modifying, main analysis and finally interpreting and presenting the results (Schreier, 2012).

The coding framework contains categories which can be assigned either deductively or inductively. Deductive means defining categories based on existing theories, and inductive on the other hand based on the research data (Schreier, 2012). For this thesis, inductive approach was chosen based on the purposes of the study.

Transcripted text was first read through multiple times by the researcher, before starting the coding process. First, identified issues and solutions stated by the participants were highlighted and extracted from the text. The statements were then reduced to their core ideas to reveal repeating or similar answers. Those statements were then grouped together. Based on these core ideas, the categorization was made. The categories that emerged reflected the root causes of the issues and who or what has the most influence over them. The identified categories were subcontractor's internal issues, case company's internal issues, issues in partnership structure and cooperation issues.

5.5 Research ethics

The decision to investigate issues in quality assurance of software developed by a subcontractor only from the client's (case company's) perspective might have resulted in some limitations for this study. The identified issues which root causes lay in the subcontractor's side may have been a subject for speculation or slight inaccuracies as the participants don't have all the information of what happens within the subcontractor's organization, which was to be expected. Additionally, the subcontracting relationship has caused some frustrations within the case company, which might have resulted in emotional charges within the statements. The participants however often recognized their biases and presented self criticism as well.

Some sensitive information was left out as instructed by the case company, but their effect on the results can be argued to be minimal. Information left out included mainly names of different shareholders, descriptions about various products and technical information.

The participants engaged voluntarily to this study, without persuasion from the interviewer or the case company's upper management. Although unlikely, it is possible that some might have felt obligated to participate to the study, for example as a consequence of peer pressure. The participants were given information about the study's aims, methods and potential implications in written form beforehand. Some raised concerns about data protection, but they were assured that the data would be handled anonymously and according to the university's and case company's guidelines. In the interview invitation, the participants were also offered possibility to see their quotes that would end up in the thesis, as well as the possibility to remove them for any reason to ensure open and comfortable interview session. However, no participant chose to withdraw their quotes.

6 RESULTS

In this chapter, results found from the conducted qualitative study are presented. The following results include both issues in quality assurance of software developed by the subcontractor and their respective solutions that that the participants proposed. The found issues are divided into four categories that emerged from evaluating their possible root causes. The categories are following: subcontractor's internal issues, case company's internal issues, partnership structure and cooperation issues. In no particular order, table 3 presents the issues distributed under their respective categories and in table 4, the solutions and ways of improvement proposed by case study's participants are presented with the same categorization.

TABLE 3 Experienced quality assurance issues

Subcontractor's internal issues	**Case company's internal issues**	**Issues in partnership structure**	**Cooperation issues**
Subcontractor's attitude towards solving problems and improving the system	Lack of requirements engineering	Unclear partnership model	Slow communication process
Poor feature documentation	Technical individuals do not validate requirements	Conflict of interest	Long communication chain from tester to developer
Lack of internal testing	Complicated acceptance process	Vendor lock	Multiple project/ticket management systems
Subcontractor's resources	Lack of knowledge about the system	Opponents instead of partners	Retest requests

(continues)

TABLE 3 (continues)

Unexpected software changes	No test analysis	High cost of small changes	Test engineers are disconnected from the development process
		Disconnection between end clients and the subcontractor	No visibility into the subcontractor's development processes and progress
			Non-Agile operating frameworks
			Lack of trust
			Cultural differences
			Language barrier
			No face-to-face meetings with the subcontractor
			Defect lifecycle involves a lot of bureaucracy

TABLE 4 Proposed solutions for the case company

Subcontractor's internal solutions	**Case company's internal solutions**	**Solutions in partnership structure**	**Cooperation solutions**
Subcontractor to improve their testing	Replan the complete acceptance process	Buy the subcontractor	Reduce the amount of project/ticket management systems
Move towards continuous delivery	Improvement of feature requirements specification	Initiate own full development gradually	Direct communication channel from test engineer to developer
Improve installation tools	Test engineers to validate new feature requirements	Aligning common goals	Move to Agile frameworks
	Perform test analysis		Focus on common planning and goals

(continues)

TABLE 4 (continues)

	Investigate and communicate how the end clients use the system		Improve partnership culture
	Competence for installation activities to test team		Knowledge sharing
	Global defect priority list		Various process improvements

6.1 Subcontractor's internal issues

Subcontractor's internal issues means that the subcontractor is mainly responsible of them, or the issues are most prevalent in their internal context. A considerable portion of the identified issues were discovered to depend greatly on the subcontractor, an entity over which the case company has little to no control. Therefore subcontractor's internal issues emerged as one of the categories.

The most often identified theme was *subcontractor's attitude towards solving problems and improving the system*. In every interview, it was recognized that the subcontractor appears to often prioritize resolving tickets with minimal effort over actually solving the underlying problems.

> When we ask questions from [the subcontractor] and try to quickly solve things, [the subcontractor's] answers appear as they haven't thought of our problem as how they can solve it, but how they can resolve the ticket.

It was mentioned that the same thing applies when something might not be considered as a defect based on the argument that something works as specified, but causes significant issues in user friendliness.

> I often feel like they do not have interest in developing their product on the basis of what we find. They just want to get the ticket ignored or refused based on the argument that it works as specified even though it is very user unfriendly.

When the subcontractor does acknowledge that there is a problem they are obligated to solve, their solutions are made adhoc and not thoroughly.

> They do easy or light solutions. They do not do things as they should be done to improve the product.

It was also reported that the subcontractor has various ways that they can try to play time to avoid fixing something. These included a "gatekeeper" for incoming tickets and arbitrarily changing requirements for defect logs.

> They have hired persons to just play time and protect developers from direct questions.

> Suddenly we were supposed to provide these [additional] logs or they [subcontractor] weren't able to do anything even though we had done things the same way for years.

However, the need or reason for a gatekeeper was understood so not to waste the developer's time but the downside is that it's a very bureaucratic step where the use of common sense is not possible. The lack of common sense refers to a scenario where there might be a simple UI bug that can be identified and understood fully from a screenshot, but the subcontractor anyway requires various logs that are unnecessary and take a long time to take.

> The subcontractor has a gatekeeper that meticulously inspects that the defects have good descriptions, all the required logs etc. I understand that this is to make sure that the developer's time isn't wasted but this is very bureaucratic step and doesn't allow the use of common sense.

Poor feature documentation was another common theme. Feature documentation was also recognized on the case company's side, but the issues are slightly different in the subcontractor's context. The features that were designed from the beginning by the subcontractor, were recognized to be badly documented on a general level. This has also lead to a situation where case company has to process the documentation and requirements after they have received them.

> Documentation of features originating from the needs of either the subcontractor or their own clients can be very poor.

> We get maybe two requirements for it [feature] from our partner, which our architect needs split into ten requirements. This means that we need to process [the documentation] quite a lot once we have received it.

Some more precise issues were identified. Firstly, the documentation frequently contains errors, including typos, incorrect words, and improper terminology, which can significantly alter the intended meaning of a feature, making writing of test cases more difficult. Additionally, deficiency in configuration guidance was observed. While descriptions of functionalities are provided, instructions on how to configure the system to enable these new features are often missing. This was recognized as particularly challenging due to the system's complexity and the requirement for the configuration to be done from multiple places to enable new features.

> There are mistakes in the documentation like typos or even wrong terminology and words which can make the description of the feature to something different, or it might not make sense.
>
> We see only the feature, but usually there is configuration involved. Configurations might need to be made from multiple places. We do not initially know where each configuration is done and how, as well as to what they have affect on.

The issues in feature documentation are amplified by the chosen way of working with project management tools. All the information about the feature need to be included verbally under one ticket, such as user stories, acceptance criterias and support for different platforms etc. As the system is very complex, this is not the most effective way. One of the participants hinted that if the whole system was developed internally, they would for example create a Jira epic for a new feature, under which dozens of user stories etc are collected as separate tickets, making their management easier. This could potentially work in the subcontractor context as well.

> Usually converting a big new feature into verbal form to user stories and acceptance criterias where every aspect is noted, like support for [different platforms] etc. is very challenging because they are so vast systems.
>
> If this was done internally, we might have an epic, under which we would have dozens of user stories. Now we just try to include everything under one ticket which is challenging.

From the test engineer's perspective, it was reported that using exclusively user stories is not a good place to start writing test cases. It was wished that user stories and acceptance criterias would be kept as separate things.

> Requirements are often replaced by user stories and I would like to see them as two separate things. It is very constricted to make test cases based on just user stories.

Lack of internal testing is the fundamental root cause for many of the issues in this subcontracting relationship. This was reported in all interviews and the common understanding was that reason behind poor software quality was the apparent lack of testing on the subcontractor's side. This means that when the software releases arrive to the case company for acceptance testing, the software has significant amount of defects. This is not even limited to major releases, as regression, sometimes very severe, emerges in fix versions as well. This can lead to a never ending loop in the acceptance process, until some concession agreements are made.

> Testing on the subcontractor side has been minimal which has lead to a situation where regression emerges in fix versions that only includes bug fixes and not any new features. This forms a never-ending loop until management decides that it is good enough as long as long as some concessions are made.

> During acceptance we and our clients find a lot of defects that shouldn't be found in such a late stage

> The subcontractor does not think about what is sufficient testing and how to test sufficiently during early stages of development.

One participant mentioned that the subcontractor has a different understanding of where software quality comes from compared to modern software development principles. They seem to be fixated to processes instead of quality.

> It seems that they think quality is a by-product of their processes. If they just handle the tickets correctly and handle changes in some meeting the quality will come. In my understanding this is against the current principles of software development where the number one goal is quality. If the current processes do not enable quality, the processes must be changed, but in this case current processes seem to be the most important thing.

Subcontractor's resources do not match the project's needs. Combined with the suboptimal software quality, the subcontractor only has resources to fix the high priority defects. Many low and medium priority defects are left without a fix, which decreases the overall quality of the system over time.

> Resources, whether skills or personnel, are insufficient on the subcontractor's side.

> The subcontractor is only able to fix escalated defects. [...] The outcome of this is that there is increasing amount of medium and low priority defects that are not fixed and the overall quality of the system decreases even though it should get better by every update.

Unexpected software changes reduce testability and make installation of test environments more difficult. Significant changes should be included only in major versions, but sometimes they are introduced without a notice in fix versions.

> The subcontractor sometimes includes new content or major changes in fix versions which reduces testability and makes test runs more challenging.

> There might be changes that can cause the installation of test environments to take undefined amount of time from one week to three months.

Lack of automation was also recognized as a problem. The subcontractor has minimal test automation and it isn't developed in cooperation with the case company. Lack of automation is also apparent in the way subcontractor delivers their software, which doesn't enable end-to-end installation and test automation pipelines so these things must be kept separate.

> The subcontractor only has little test automation and it is not developed together with us.

> At the moment there is no possibility to build automation pipelines due to the way the subcontractor delivers their releases.

The identified issues in the coding process above are primarily or entirely the responsibility of the subcontractor, and only they have the ability to address and rectify them. Let's now discuss what actions participants suggested that the subcontractor could potentially take.

Subcontractor to improve their testing is the core solution which benefits would cascade over the entire project. If the releases only included minor defects, the acceptance process would be quickly streamlined and would release pressure also on the case company's side. It would also mean a return to the intended partnership model where subcontractor develops and tests the software and the case company only validates that the functionalities are what they hoped for.

> The subcontractor must improve their testing so that the software would only include minor defects. Then we would easily be able to pass the acceptance in a week or two.

> Our role should change from testing to assuring quality. They are philosophically different things even though the same actions are taken.

> Try to get back to a model where the subcontractor tests their own code and we only validate that the functionalities are what we wanted.

One concrete action that the subcontractor could make is hire a test lead as currently there isn't one.

> No test lead or equivalent on the subcontractor's side.

Move towards continuous delivery would require fundamental changes to the way the organization operates but would solve a lot of problems with quality as well as with delivery and installation times and the case company wouldn't need to test in big bangs. Unfortunately this would be very difficult to realize in practice currently.

> If their model was closer to continuous delivery, we would avoid a lot of these problems.

> If we would get versions more often we could practice the installation more and become better quickly.

Improve installation tools is a solution that is already being implemented by the subcontractor. They are moving from context dependent scripts to a single general one which is expected to help a lot. Alongside this subcontractor should also be required to focus on creating detailed and easily understandable documentation for the installation process.

> The subcontractor has started to improve their installation tool.

> Demand and get the subcontractor to improve quality of their installation documentation.

> The subcontractor is changing the installation scripts from customer specific scripts to standardized scripts, which should hopefully help a lot.

6.2 Case company's internal issues

Not all issues can be attributed to the subcontractor, as there are numerous challenges originating from the case company's side, over which the subcontractor has no control over. Therefore company's internal issues were selected as the second category.

Lack of requirements engineering in terms of the features that originate from the needs of case company's clients or from the case company itself stands out as one of the main problems. There seems to be missing a process where the given requirements are scrutinized and thought about systematically. Instead, someone just creates them without proper analysis. Also, knowledge about what the feature benefits actually are for the client in the testing team seems to be sometimes missing.

> It feels like there is no real process for creating requirements, instead someone makes them arbitrarily.

> We rarely talk about what is the expected benefit for the customer. What the feature is really? Who uses this and what is the problem we are trying to solve?

Related to this, *technical individuals do not validate requirements* was another concern in the case company. For example, customer's requirements go straight from the product business managers to the subcontractor, which might result in technically vague requirements and thus makes the feature vulnerable for unwanted qualities.

> Technical individuals do not audit or verify the requirements before development.

> There is no test engineer involved in feature design.

Complicated acceptance process is another issue that needs to solved. There has been efforts to speed up the acceptance process but greater efforts need to made. Currently, the software needs to be accepted four times which puts the releases at risk if they are rejected late in the process.

> There is an effort to speed up the acceptance process but as long as the software quality is bad when it undergoes acceptance testing, acceptance will take time.

> The complete acceptance process is complicated and difficult. The software needs to be accepted four times. First by the subcontractor internally, secondly by us, thirdly by the end client and finally by the end-user team.

> When the client finds defects that they think are blockers, we wind up changing the software half a year after we had accepted the software from the subcontractor. The result is that we get one major release through per year when we should get them constantly accepted.

Solution for this is to *replan the complete acceptance process*. Obviously this is a large and difficult change to initiate at such a late stage.

> Complete acceptance process should be planned again.

Lack of knowledge about the system is an occasional issue within the case company. This is a natural consequence from using an outside entity to develop the system.

> We do not know the system that well because we operate a system made by somebody else.

No test analysis phase exists as it is defined in literature. Test analysis is replaced by an order process in which test engineer's are not involved. However, there is some feature analysis made by test engineers late in the development process. Nevertheless, when test analysis is not done or done late in the process, it can lead to delays or barriers for testing. A lot of these issues can be avoided if test analysis is done early to ensure that all required elements are in place.

> The requirements come from the client and they are prioritized by PBM's which isn't always that easy. So we basically we haven't had this kind of test analysis phase. In our case it's more like an order process that we go through where test engineers are not involved.

> Because test analysis phase is missing, we sometimes might be in a situation where testing the new feature is not possible, or at least not on a short timeframe because some platform or configuration is missing or it's otherwise impossible.

> We have ordered a test environment and noticed during acceptance testing that some components are missing which then takes time. It depends on situation whether this is our or the subcontractor's problem.

The coding process revealed few internal solutions that the case company could do on their behalf. *Improvement of feature requirements specification* seems to be one of the core solutions. The requirements need to be made more accurate before development in order to minimize problems that can be sensed beforehand.

> If we put more effort towards requirements specification then we would know what we get. We know that if we wish something with low specificity we get something with low specificity. The only solution is to make the specification so accurate that we can avoid a lot of problems that we sense beforehand.

A kind of sub-solution for this was identified, which is *test engineers to validate new feature requirements*. This was mentioned during all interview sessions by both test engineers and managers. In other words requirements engineering should be done early on by test engineers or other technical individuals alongside PBM's and other relevant business stakeholders. Collaboration and discussion is important to make sure that there is a collective understanding of the feature and it's benefits to the customer. This could be done using a benefit hypothesis introduced by the SAFe model.

> When new feature is under design, test engineer should be involved in it and give input.

> As many technical people with understanding about the system as possible should be included in the requirements specification.

> Involve the test team in addition to PBM's and architects before submitting the feature request to the subcontractor.

> Collectively discuss about the new features to improve collective understanding about the feature what the benefit is to the user.

> SAFe model suggest that when designing a feature, do a benefit hypothesis.

Perform test analysis is a solution that has already been partially implemented in the case company, which was time allocation for test engineers to familiarize with the features before they are installed. It was reported that this has improved test planning and preparation, but to collect all it's benefits, it should be done already when requesting new features.

> Partial solution has been time allocation for the test engineer to familiarize with the features before they are installed or released by analyzing the documentation. This has improved test planning and preparation as it has improved overall understanding of the feature before testing.

> Perform test analysis when we request new features.

As there is not a direct access from the subcontractor to the case company's clients, the case company needs to more effectively *investigate and communicate how the end clients uses the system*. Currently there is no way for the subcontractor to understand how the case company's clients use the system because it isn't communicated to them. This can be problematic to solve as some confidentiality aspects might come into play. However, if this is managed to be solved, the subcontractor could in some cases explain that some new features

might not be necessary if the end user's would for example use the system in some other way to achieve the same results and thus saving money.

> We should better explain how our clients use the system and their problems for the subcontractor if we do not want to give the subcontractor communication access to our clients.

> The subcontractor owns the product and knows the system better than we or the clients do so there is no point implementing the customer requirements when the subcontractor could tell the customer to use the system this other way and achieve the same result.

Inorder to alleviate some setup and configuration issues, transferring *competence for installation activities to test team* could be a solution so that they could independently solve at least minor issues. This would reduce the devops team's workload and they could focus on more challenging tasks while test engineers get their problems solved faster so they can focus on the testing itself.

> Teach the testing team in installation abilities.

Finally, it would be beneficial to create and maintain a *global defect priority list*, meaning a list where all the defects between different versions and teams would be listed and prioritized based on their importance and how fast the case company wants them fixed. This list would then be communicated to the subcontractor, which would make the case company more in control of what is being fixed.

> I would absolutely like a global list through which we would prioritize everything to our partner. Then we wouldn't have multiple priority lists and it would be clearly in our control.

6.3 Issues in partnership structure

Some issues were identified to be the consequence of the partnership structure or model. Over time, the partnership structure has deviated from its original model, leading to alterations in the nature of the relationship and resulting in ambiguity regarding the roles of the parties involved.

Unclear partnership model has caused confusion within the case company. In theory, the case company and the subcontractor are considered as partners, but there has been suspicion that the subcontractor sees the case company as just one of their clients. Additionally, due to lack of testing inside the subcontractor, the case company has evolved to be an unofficial test organization for the subcontractor.

> We talk about our relationship with the subcontractor as a partnership but I think the subcontractor does not see it the same way. We probably are just one client with the rest of them who is a bit more difficult than others. We create more tickets and ask more questions.

> The overall subcontracting model has skewed in practice. We have basically become the subcontractor's test organization over the years.

Conflict of interest also has raised concerns within the case company. As explained in the case description, the subcontractor competes in the same market with a similar product, which introduces a major conflict of interest to the relationship.

> The subcontractor is at the same time our competitor which brings it's own tensions.

> We have different goals with the subcontractor. Our goal is to provide high quality software to our customers but for the subcontractor we are just one of their clients, that try to do the same thing that they do so we necessarily aren't their number one priority.

On top of this, the case company is highly dependant on the subcontractor's system which has caused a *vendor lock*. It has made negotiations challenging as well as limited the ability to influence the subcontractor's actions, as the subcontractor has other customers that are happy with their products.

> It is very hard for us to dictate any terms because we are so badly vendor-locked.

> We have clients that use the system in larger scale but if subcontractor's other customers use it less and do not complain, then we can only do so much.

Disconnection between end clients and the subcontractor imposes inefficiencies to the product development and makes agreeing on defect's severities difficult as the subcontractor does not understand how the end client uses the system. Instead, the subcontractor has their own idea how the system is intended to be used, but this information does not reach end clients as the case company wants restrict the subcontractor's access to the clients.

> It is hard to agree on severity of defects as the subcontractor does not understand how our clients use the system because the subcontractor does not have access to our clients.

> The challenge is probably also that we have those customer requirements that can be very precise from one of our customers which we need to forcefully implement. [...] [the subcontractor] owns the product, they know how it is intended to be used better than we do, they know the big picture of it in a way. They could just tell the end clients to use the system in this other way and you can do the same thing.

Some participants felt like they are *opponents instead of partners* in the relationship. Due to constant conflict, the relationship has become sour and repairing it can be difficult.

> I feel like there is a lot of conflict between us and the subcontractor. Almost like we are on opposite sides instead of partners. It's a constant struggle and surely difficult for both sides.

> Over the years we have become sort of opponents to each other. It's quite difficult to get back to good cooperation.

High cost of small changes, meaning improving aspects like usability and other things that might not be considered as defects, are expensive and time consuming to make. This is mainly due to agreements made between the case company and the subcontractor. Therefore the agreements themselves impose barriers to quality.

> If according to subcontractor something isn't a defect it means that we have to pay extra. Then I as a test engineer have to think whether I want to make a change request because this button is in slightly different place than on the other page and then pay. It lowers quite a bit my threshold to pick on these little problems because I know how much paper work and money it takes to get it fixed so it lowers quality in that way also.

As mentioned by the participants, these kind of issues are hard to solve. Most of the proposed solutions would require major investments on the behalf of the case company, so they were mentioned as mainly theoretical, but potentially highly effective. An example of theoretical solution was to *buy the subcontractor*. Although a radical strategy, it would put the case company completely in charge of resources and how things are done.

> Let's buy the subcontractor. It is difficult to solve this [quality issues] with money. [...] As long as their management does not prioritize quality we won't get it.

Another theoretical solution would be to *initiate own full development gradually* to over time move away from the vendor-lock.

> If I was a wealthy stakeholder in this case, I would think about initiating our own full development and try to little by little get rid of the subcontractors system.

One more practically applicable solution came up in terms of the minimizing the issues of the current subcontracting structure. The case company needs to convince the subcontractor about of their quantifiable benefits they would gain from focusing on quality, and thus *aligning common goals*. Luckily this is slowly happening and the results will hopefully amplify the subcontractors testing efforts.

> We need to better align our goals. The subcontractor needs to understand that the better the quality is, the better it is for them. Luckily they are beginning to understand this and are increasing their testing. We get better quality and they get their money faster.

6.4 Cooperation issues

Large proportion of issues that the coding process revealed are related to the ways in which the case company and the subcontractor cooperate with each other. These issues are most prevalent in the interfaces between the parties and thus cooperation issues were chosen as the last category.

Three most significant issues are related to the communication processes, chains and tools. Firstly, *slow communication process* was identified to be one one of the main hindering factors for efficient collaboration. The process was also found as unbalanced. The current processes were build in the early days of the subcontracting relationship, when the eventual issues hadn't manifested themselves yet. Therefore, the current processes does not serve the needs of the current state of the project.

> Slow flow of information from side to side.

> Enforcing ticket response times is not equal. We are required to update the ticket within two weeks or the ticket will be closed. The subcontractor is not enforced to handle tickets in reasonable time.

> The communication processes were build on top of the assumption that we would only do light acceptance testing that results in small amount of defects.

> Communication is done in the comments of the relevant ticket. Between each comment there might be one to two day delay. The comments get bounced from side to side and during this there isn't any progress made on the actual issue.

Relating to this, especially *long communication chain from tester to developer* was seen as a definitive inhibitor for solving problems quickly and having enough information to perform testing. Both the process and tools were criticized.

> Long communication chain from tester to developer.

> No fast or direct channel to ask questions about new features during analysis phase.

> No direct way to ask anything from the developer and it can take months to get an answer through the middlemen.

> The problem is that we are using a tool which doesn't allow the test engineer to ask questions directly from the developer. The gap is very large between the test engineer and developer.

Third identified theme in regards of communication topics was the problem of *multiple project/ticket management systems*. They are used depending on whether the system is under SLA or not. They each have their own reporting workflows creating additional complexity. Multiple systems also means duplication of data which makes information vulnerable for mistakes as information needs to be updated to multiple places manually. Multiple management systems makes tracking and managing priorities difficult as well.

> There was an attempt to fix the high number of defects by introducing multiple ticket management systems, but this didn't fix the problem and only resulted in more complexity and increased workload on personnel.
>
> Multiple ticketing systems which are chosen based on SLA creates complexity.
>
> There is no common project management system between the client and subcontractor.
>
> Only way to report some defects is to tag a certain person in the defect's comments and ask them to duplicate it forward. This is bad if this person for some reason misses the comment.
>
> A lot of manual input and bureaucracy is required when there are multiple project management systems and projects within those systems.
>
> We have three places where we can report defects but we don't have a general control in how we prioritize what are the most important defects to fix.
>
> Multiple Jira's introduce barriers to information flow because there is duplicated data in two places which causes the information to not be up to date in both places.

The efforts have already started to *reduce the amount of project/ticket management systems*. New common Jira between the case company and the subcontractor is coming, but it doesn't necessarily solve all the issues and thus further efforts can be made by renegotiating with all stakeholders to solve this issue completely.

> We are getting a new common Jira which we hope will help a lot.
>
> The use of different ticket management systems could be solved by better negotiation and agreement.
>
> Agree with [shareholders] to give our test engineer's access to their system.
>
> Common Jira despite of the nature of the partner.

One of the most time-consuming and labor-intensive aspect for the test engineers is handling *retest requests* after a defect has been reported. Subcontractors frequently ask for additional logs, videos, or testing on different

software versions, which complicates the process. Testers find that setting up complex test cases for retesting is particularly time-consuming. A significant issue arises when the subcontractor inspects the content of logs and videos late in the process, often leading to extra work for testers if the provided information is insufficient. There are also instances where retest requests seem unfounded, especially when no relevant changes have occurred that could affect the reported defect. This entire retesting process disrupts the workflow of test engineers.

> We get a lot of retest requests.
>
> The most time consuming and labor intensive part during test execution are requests for retesting after a defect has been reported. The subcontractor can ask for additional logs, videos or to test in another software version.
>
> Retesting takes time because test cases usually require complex test setups.
>
> The actual content of logs and videos are inspected by the subcontractor late in the process which creates additional work for testers if it proves to be insufficient.
>
> We get unfounded requests for retesting certain defects even if nothing has changed that could affect the problem.
>
> Retesting breaks the workflow of the test engineer.

Test engineers are disconnected from the development process. Testers typically only encounter a feature after it has been implemented and installed in an environment which indicates a clear disconnection from the development process. This is against the principles of quality assurance, which ideally involve testing as an integral part of the development from the beginning.

> Currently testers see the feature once it is implemented and installed to some environment and therefore testing is clearly disconnected from the development process event hough it should be included from the beginning when talking about quality assurance.
>
> Keeping testing and development separate is not a starting point where we could succeed.

Test engineers are not only disconnected from the development process, but also lack *visibility into the subcontractor's development processes and progress*. A key concern is that the "In progress" status of a defect does not necessarily indicate that it is being actively fixed; it only reflects that the defect has been logged into the subcontractor's internal system. This ambiguity prevents the effective utilization of test engineers' experience-based knowledge to efficiently focus testing efforts, as there is no clear understanding of what has changed at the code level due to the nature of blackbox deliveries. Furthermore, defect tickets

can sometimes remain in the "In progress" state for extended periods, ranging from months to even years.

> In progress state does not mean the defect is under fixing, it just means that it has been duplicated to subcontractor's internal system.
>
> Defect tickets can sometimes spent months or years in "In progress" state.
>
> We cannot utilize our test engineer's experience based knowledge to focus testing more effectively because we do not know what has changed on a code level due to blackbox deliveries.

The issue of *non-Agile operating frameworks* was another commonly reported, very fundamental issue. The subcontractor adheres to a waterfall model with major deliveries, leading to the testing of large updates at once. This traditional approach includes big bang releases that include numerous updates, such as bug fixes and new features.

> The subcontractor uses a waterfall model. They use major releases which include large amount of updates like bug fixes and new features.

Even though the case company tries their best to be Agile, the subcontractor's waterfall model restricts themselves to operate similarly. However, there are also internal non-Agile components in the way that the case company operates which haven't proven helpful.

> We get the new features in big bangs which naturally dictates that we cannot test the new feature once it's ready. This restricts our ability to be Agile and confines us to this kind of waterfall model.
>
> Our way of working with milestones and increments to reach certain increment targets does not benefit the development and deliveries. It's just an operating model.

The non-Agile way of working manifests itself in practice in many ways. For example, if there are major quality issues with new features, they cannot be taken out from the release. Instead, the case company has to make a decision whether to block the entire release and delaying deliveries, or to mention in the release notes to the customers to not use the feature.

> In this partnership we are not in control if there is a feature that is unusable and not fit for delivery. We cannot get it out of the release so our options are either to block the release or mention in release notes for the end client to not use this feature.

As both parties work in more or less non-Agile ways, common planning and scheduling can prove to be troublesome. If something goes wrong, both parties plans inevitably clash, causing delays and other issues. One common issue is that if a defect is found in a certain version, the subcontractor might not have that version installed anymore to their environments as they have moved on

with their development, they then need to rollback to the older version to reproduce the issues.

> We do not plan and schedule testing together with the subcontractor. We also use outdated milestone release train model which isn't very good, because sometimes we skip releases and test in large chunks couple times a year. And because both parties have their own plans and schedules, they are likely to clash at some point.
>
> As we lack common planning and direction, the defects we find do not always match the subcontractor's development process so it takes time for them to think about how they can reproduce defects or issues.
>
> Because we do acceptance testing in different phases and the defects are reported to the subcontractor in batches, it is up to chance if they fit in the subcontractor's development process and if they can react to them or not.
>
> We need to be more strict about delivery schedules when asking new features. When we ask something we know it will come in some big bang next year but it's not enough because we have promised our client something else. Then we ask subcontractor to bring it in some earlier version which messes up their schedule and that's why they are so busy and cannot deliver new releases in time.

L*ack of trust* was also reported between the case company's and subcontractor's representatives. From test engineer's perspective, this can manifest itself for example in the comments of defects.

> The subcontractor representatives do not always trust the test engineers and ask for additional proof even though the defect is obvious.

There are *cultural differences* that reduce consensus on certain topics. Major difference between the case company and the subcontractor is that the case company operates more in the mission critical domain and therefore has adopted stricter mindset and approach to testing and acceptance. This has caused conflicts with the subcontractor as they do not understand the needs of mission critical environments as well.

> The subcontractor does not understand testing needs for our mission critical systems and their other clients might not be as demanding.
>
> Our culture is more strict in terms acceptance [due to mission critical environment] than the subcontractor's which leads to conflict.

In addition to cultural differences, *language barrier* occasionally presents some challenges for collaboration in quality assurance activities.

> There is a language barrier. It is sometimes difficult to understand what the subcontractor means or it could be understood in many ways.
>
> Communication is difficult due to different languages and countries.

No face-to-face meetings with the subcontractor was also mentioned as a problem. This makes it hard to form personal connections and to foster deep collaboration. There has been some meetings and visits but they haven't included test engineers.

> We rarely meet the subcontractor in person.

The current *defect lifecycle involves a lot of bureaucracy*. As the time spent on moving tickets around, adding additional logs, retesting and having meetings about it all costs money. According to one manager, the sums could get quite high if they were calculated.

> If we would calculate the costs of the defect's lifecycle, including all the bureaucracy, the numbers would very high. It's very expensive and inefficient to work like that.

One of the most commonly mentioned improvement would be a *direct communication channel from test engineer to developer*. It could take form as a questions and answers channel between test engineers and developers. One participant highlighted that whatever the method would be, the test engineers need to be able to access it from their development laptops and not just from their email laptops. This further streamlines the process as the test engineers could send screenshots, logs etc. to the developer conveniently and not circle all attachments through the email laptop.

> Better communication tools so that we don't rely on email and tickets.

> Some channel where test engineer's questions about the features would be answered in a timely manner.

> Maybe some questions & answers channel between the testers and the developers.

> We need to avoid a scenario where once we get the communication tools, our test engineer's are able to access them from their development devices and not just from their email laptops.

One effective, but mainly theoretical solution is that both parties *move to Agile frameworks*. It's another question whether this is a realistic and executable solution.

> Move to some Agile framework. We could react faster, pass the acceptance faster and everything would be more under control.

What the parties can do is *focus on common planning and goals*.

> What I've read about Agile, all parties should focus together on improving the processes and not as separate entities and to keep the common goal in mind which is getting the software quality as high as possible as fast as possible.

> We should be more involved in the early stages of testing alongside the subcontractor. We should get their test results and together perform test analysis and planning.
>
> Planning should be done together.

In order to *improve partnership culture*, a lot of effort is required. All stakeholders should start striving to a culture where they are once again on the same side. This is not easy, but one way that the coding process revealed was to start organizing visits at each other premises. Only concern is that the subcontractor is scattered internally as well to different locations.

> We need to start building a culture where we are on the same sides.
>
> It would be good to meet them in person but they are located in different countries.
>
> It really helps when you actually meet the persons.

Knowledge sharing would improve various aspects. For example, sharing best practices between parties has already proven helpful in feature documentation.

> We imposed on the subcontractor our feature documentation templates. We used to request new features basically with one-liners and the results were not necessarily what we wanted. The template forces to describe the feature's needs clearly in a certain format. On the latest release the subcontractor even used it themselves to describe their own features.

Various process improvements could be made. One particularly helpful addition would be visibility to the subcontractor's development. As mentioned earlier, tickets can spend a long time in "in progress" which doesn't actually indicate much about the defect's actual status within the subcontractor's side.

> It could be a solution to get visibility to subcontractor's development processes and what they have done on a code level.

Another process improvement that was mentioned is to streamline the defect reporting process to get the tickets faster into progress or alternatively to get the subcontractor explain quickly why it isn't a defect.

> Quick and straightforward process to get the ticket under progress or to get the subcontractor representatives to explain why it isn't a defect.

7 DISCUSSION

As further highlighted by this case study, software subcontracting is not a risk free operation (Dey et al., 2009). Some of these risks have been realized in the case company and caused issues on quality assurance. Fortunately, there are solutions proposed by scientific literature, as well as the case study participants.

Day et al. (2009) discuss about project type's meaning to subcontracting project. They mention that mission critical projects require higher resources and effort to reach the quality and stability requirements set by mission critical environment. During the interviews, it was highlighted that the subcontractor does not understand the testing needs of mission critical systems because it seems that they have been used to operating in less stricter environments with their other clients. This has led to some conflicts with the case company, and it might have played a part in insufficient resource allocation because the subcontractor couldn't anticipate the greatly vaster testing and quality needs of the case company which is more involved in the mission critical domain.

During literature review, Williamson's (1991) transaction cost theory was explained to measure whether it's beneficial to outsource some activity. The theory states that outsourcing should result in lower production costs and higher management costs for the client, which in this case is the case company. This thesis cannot comment on the production costs, but the management costs can be seen to be compliant with the theory. Like one of the participants mentioned, if the cost of all the bureaucracy would be calculated for just defect management alone, the numbers would be very high. The transaction (or management) costs also include searching, creating, negotiating, monitoring and enforcing the service contract between the case company and subcontractor (Dhar & Balakrishnan, 2006). The key question then is, have the transaction costs of using a subcontractor surpassed the potential savings made on production costs?

Shah et al. (2014) report similar findings that this thesis made, but from the point of view of the subcontractor. They and Kobitzsch et al. (2001) found that working from different locations and with different operating models

causes difficulties in effective communication, which consequently can lead to poor collaboration. Ineffective communication can manifest in many ways. Both the thesis and Shah et al. (2014) found that long communication chains, especially from tester to developer, leads to delayed decision making and slowly resolved issues. The thesis reported that some technical questions might take days or even months to get answered as they must be routed through multiple middlemen and communication tools.

In order to minimize future issues, current professionals should be retained in-house as long as possible. Study by Narayanan et al. (2011) found that stability of the team is tied to project success and effectiveness. Retention of employees was also wished in the interviews for it's effects on competence generation. Narayanan et al. (2011) found this to be true in their study, because a stable team allows for cumulation and retention of project specific knowledge and skills. Shawosh & Berente (2019) also recommend maintaining key talent with deep knowledge about the system in the building, as it in-part reduces dependency to the subcontractor and alleviates the symptoms of vendor-lock. The loss of key competence should not only be avoided, but increasing everybody's skills and knowledge about the system should be another goal. Both Shah et al. (2014) and participants reported lack of visibility to the subcontractor's development processes to be a problem. One participant mentioned that as there is no visibility to what the subcontractor has done or changed on the code level, it limits the in-depth knowledge accumulation, which in turn limits the test engineer's ability to use experience based knowledge to focus testing efforts. If visibility to subcontractor's development activities is combined with direct communication channel between test engineers and developers to quickly attain specific information, the gained knowledge and competence could be used to solve problems quicker and to better focus testing efforts more efficiently, especially during major releases which take a long time to test.

Systematic effort to requirements engineering has been scientifically proven many times to reduce the amount of rework needed to software products (Laplante & Kassab, 2022). Laplante & Kassab (2022) also mention that prior research suggests that it is not done well in the industry. The same thing seems to apply in the case company. The participants mentioned that the requirements are not accurate enough and the requirements are not scrutinized by test engineer's or other technical individuals to poke holes in them and reveal discrepancies or other insufficiencies. During all interviews, participants recommended that test engineers should become involved in creating and reviewing new customer requirements to prevent predictable issues and improve overall software and documentation quality. This would potentially improve things for the subcontractor in many ways. Test engineers input could help the developer identify and understand potential corner cases so they can be found and handled before the software even reaches the test engineers. Additionally, if the communication channel is established between test engineers and developers, test engineers could then help the developers already during development, if they have questions about the requirements. The article

by Shah et al. (2014) reports that test engineers helping developers has sped up defect resolution times as well, because test engineers understood the requirements better as they had more time to familiarize with them. For this reason, test engineers being part of requirements engineering would greatly benefit the subcontractor in building quality software and therefore the case company.

Strong and fair relationships between the client and the subcontractor is the backbone of effective and efficient outsourcing (Assmann & Punter, 2004). The interviews conducted in this thesis unfortunately reveal that the current status of the relationship is not as good as it could be and that the cooperation hasn't been strong enough, thus supporting the statement made by Assmann & Punter (2004). Employees on both sides should be encouraged to form strong personal relationships between each other according Kobitzsch et al. (2001) and Khan et al. (2019), but currently it seems that they are actively limited by the established communication processes and tools. The participants therefore wished open communication channels to allow free collaboration. The participants were also very open for visiting the subcontractor or vice versa. These are recommended practices by both Kobitzsch et al. (2001) and Khan et al. (2019). If these are not possible to organize Khan et al. (2019) also recommend organizing knowledge and information sharing events between the teams in order to better understand each other, although active and continuous communication would be the best solution. This is further supported by Shah et al. (2014), where subcontractor's engineers reported that they spend extra effort establishing good communication relationships with their clients which helped in reducing information gaps and communication breakdowns. Additionally, they got their needed additional information quickly. However, even if site visits are organized and open communication channels established, the results might not be immediate because it can take years to rebuild trust and rigid collaboration (Kobitzsch, 2001).

8 CONCLUSIONS

This thesis goal was to understand what the intricate dynamics of software subcontracting can cause for quality assurance. Prior research on software subcontracting suggested that there are many risks involved in such arrangements and consequently some of them have been realized in the case company. This thesis shed further light into this topic by investigating the practical difficulties and various solutions proposed by scientific literature and the study's participants.

The thesis set out to answer two research questions: what challenges and issues subcontracting relationship can cause for quality assurance and how can these challenges and issues be solved or alleviated. In chapter 6, The thesis proposed two corresponding tables that included the identified issues and challenges (table 3) and the proposed solutions (table 4) with in depth descriptions later in the chapter to answer these research questions. In total, 28 issues and 20 solutions were identified.

The case study was conducted using qualitative research methods. Semi-structured focus group interviews were used to collect the case company's employees' experiences about the issues that they had faced when working with the subcontractor in quality assurance related activities. After analyzing the transcripts, plethora of issues and solutions emerged. These findings were divided into four distinctive categories depending whether they were related to exclusively to the subcontractor, the case company, the subcontracting structure or whether they are caused by the cooperation interfaces.

Many of the discovered issues can be tracked down to the quality of work done by the subcontractor. Many participants highlighted the evident insufficient testing done internally by the subcontractor. The result of this has been that the case company has had to perform the bulk of the testing which has resulted in large amounts of defects found late in the development process. Quality assurance research suggests that the later a defect is found, the harder and more expensive it becomes to fix. This is evident in the case company as

these quality issues has cascaded down causing various issues along the way to many processes and deliveries.

However, not all the issues can be attributed to the subcontractor as highlighted many times by the participants. The case company has many areas for improvement especially in terms of producing more comprehensive feature requirements. Test engineers and other technical individual possess valuable knowledge and expertise which should be used to prevent predictable issues already during the design phase of the feature.

Moreover, both research and this study highlights the importance of collaboration and communication. With a combination of strategic communication, skill retention and development, and the nurturing of strong client-subcontractor relationships can lead to more successful and efficient software subcontracting arrangements.

This thesis contributed to the software development outsourcing literature by addressing the often overlooked aspect of client perspectives in quality assurance activities for software developed by subcontractors. This study fills the gap in research by providing an in-depth analysis of the challenges and solutions encountered in QA when subcontractors do not meet quality expectations, which is a scenario not thoroughly addressed in existing literature. The goal of both the literature review and empirical portion of this thesis was to provide description of core concepts, issues and solutions so that this information could be as practically applicable as possible. Based on this thesis, the case company among other organizations in similar situation can create strategies to recognize, solve or prevent many of the issues found in the study.

8.1 Limitations

A significant limitation of this thesis is that it did not take into account the perspectives of the subcontractors. This exclusion means that the findings represent a one-sided view by exclusively including the experiences and opinions of the case company under study. The absence of the subcontractor's viewpoints could have lead to an incomplete understanding of the dynamics and challenges in the relationship.

The study is based on experiences and data from only one company. This focus raises questions about the generalizability of the findings. Some of the experiences and challenges identified may be unique to the case company and might not necessarily apply to other organizations. On the other hand, certain issues present in other organizations might not have been observed or addressed in this study due to its limited scope.

The research was conducted within a relatively short timeframe, which applied constraints on the scope and depth of the study. Due to these time limitations as well as the intended tightly framed nature of a master’s thesis, the

research had to be more focused and selective. Therefore, some relevant but non-central topics were left out from the literature review and empirical investigation. This constraint means that while the study covers key areas, it may not provide a comprehensive overview of all relevant aspects related to software subcontracting or quality assurance.

8.2 Future research

Based on the limitations of the thesis, there are several pathways for future research. First, future research should include the viewpoints of both subcontractors and clients. This approach would provide a more balanced and comprehensive understanding of the dynamics between companies and their subcontractors. Research that captures both sides of the relationship can offer deeper and more objective insights into the challenges and opportunities in such collaborative ventures.

Second, conducting research on the topic with broader company sampling could reveal fully generalizable insight. Research project involving multiple companies across different industries or sectors could reveal a wider range of experiences and challenges while also highlighting common themes as well as issues specific for certain industries. Moreover, a framework for issues and solutions could be constructed. This framework could then be applied to a case company and by monitoring their improvement, the frameworks effectiveness could be evaluated. Such study would be particularly useful in understanding the long-term impacts of proposed practices and strategies.

REFERENCES

A Survey on Load Testing of Large-Scale Software Systems | *IEEE Journals & Magazine* | IEEE Xplore. (n.d.). Retrieved October 20, 2023, from https://ieeexplore.ieee.org/abstract/document/7123673

Abd Rahman, A., & Hasim, N. (2015). Defect Management Life Cycle Process for Software Quality Improvement. *2015 3rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS)*, 241–244. https://doi.org/10.1109/AIMS.2015.47

Arcos-Medina, G., & Mauricio, D. (2019). Aspects of software quality applied to the process of agile software development: A systematic literature review. *International Journal of System Assurance Engineering and Management, 10(5)*, 867–897. https://doi.org/10.1007/s13198-019-00840-7

Assmann, D., & Punter, T. (2004). Towards partnership in software subcontracting. *Computers in Industry, 54(2)*, 137–150. https://doi.org/10.1016/j.compind.2003.09.005

Atkins, R. (2005). Software contracts and the acceptance testing procedure. *Computer Law & Security Review, 21(1)*, 51–55. https://doi.org/10.1016/j.clsr.2004.11.010

Axelsson, J., & Skoglund, M. (2016). Quality assurance in software ecosystems: A systematic literature mapping and research agenda. *Journal of Systems and Software, 114*, 69–81. https://doi.org/10.1016/j.jss.2015.12.020

Bai, A., Mork, H., & Stray, V. (2017). A Cost-Benefit Analysis of Accessibility Testing in Agile Software Development—Results from a Multiple Case Study.

Baresi, L., & Pezzè, M. (2006). An Introduction to Software Testing. *Electronic Notes in Theoretical Computer Science, 148(1)*, 89–111. https://doi.org/10.1016/j.entcs.2005.12.014

Camacho, C. R., Marczak, S., & Cruzes, D. S. (2016). Agile Team Members Perceptions on Non-functional Testing: Influencing Factors from an Empirical Study. *2016 11th International Conference on Availability, Reliability and Security (ARES)*, 582–589. https://doi.org/10.1109/ARES.2016.98

Carey, M. A., & Asbury, J. (2016). Focus group research. Routledge.

Corral, L., Sillitti, A., & Succi, G. (2015). Software assurance practices for mobile applications. *Computing, 97(10)*, 1001–1022. https://doi.org/10.1007/s00607-014-0395-8

Critical Success Factors of Component-Based Software Outsourcing Development From Vendors' Perspective: A Systematic Literature Review | *IEEE Journals & Magazin*e | IEEE Xplore. (n.d.). Retrieved October 3, 2023, from https://ieeexplore.ieee.org/abstract/document/9663301

Dey, D., Fan, M., & Zhang, C. (2009). Design and Analysis of Contracts for Software Outsourcing. *Information Systems Research*. https://doi.org/10.1287/isre.1080.0223

Dhar, S., & Balakrishnan, B. (2006). Risks, Benefits, and Challenges in Global IT Outsourcing: Perspectives and Practices. *Journal of Global Information Management (JGIM), 14(3)*, 59–89. https://doi.org/10.4018/jgim.2006070104

Fonseca, J., & Vieira, M. (2008). Mapping software faults with web security vulnerabilities. *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 257–266. https://doi.org/10.1109/DSN.2008.4630094

Galin, D. (2009). Software quality assurance: From theory to implementation (Nachdr.). Pearson.

Galin, D. (2018). Software Quality: Concepts and Practice. John Wiley & Sons.

Garousi, V., & Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology, 76*, 92–117. https://doi.org/10.1016/j.infsof.2016.04.015

George, B., & Williams, L. (2004). A structured experiment of test-driven development. *Information and Software Technology, 46(5)*, 337–342. https://doi.org/10.1016/j.infsof.2003.09.011

Ghanbari, H., Vartiainen, T., & Siponen, M. (2019). Omission of Quality Software Development Practices: A Systematic Literature Review. *ACM Computing Surveys, 51(2)*, 1–27. https://doi.org/10.1145/3177746

Gopal, A., & Koka, B. R. (2010). The Role of Contracts on Quality and Returns to Quality in Offshore Software Development Outsourcing. *Decision Sciences, 41(3)*, 491–516. https://doi.org/10.1111/j.1540-5915.2010.00278.x

Han, Y., Lee, D., Choi, B., Hinchey, M., & In, H. P. (2016). Value-Driven V-Model: From Requirements Analysis to Acceptance Testing. *IEICE Transactions on Information and Systems, E99.D(7)*, 1776–1785. https://doi.org/10.1587/transinf.2015EDP7451

Hirsjärvi, S., Remes, P., Sajavaara, P., & Sinivuori, E. (2009). Tutki ja kirjoita (15. uud. p.). Tammi

Hirschheim, M. C. L. and R. (1993, October 15). The Information Systems Outsourcing Bandwagon. *MIT Sloan Management Review*. https://sloanreview.mit.edu/article/the-information-systems-outsourcing-bandwagon/

Hooda, I., & Chhillar, R. S. (2015). Software Test Process, Testing Types and Techniques. *International Journal of Computer Applications*, 111(13), 10–14.

Horkoff, J., Aydemir, F. B., Cardoso, E., Li, T., Maté, A., Paja, E., Salnitri, M., Piras, L., Mylopoulos, J., & Giorgini, P. (2019). Goal-oriented requirements

engineering: An extended systematic mapping study. *Requirements Engineering, 24(2)*, 133–160. https://doi.org/10.1007/s00766-017-0280-z

Kassab, P. A. L., Mohamad. (2022). Requirements Engineering for Software and Systems (4th ed.). *Auerbach Publications*. https://doi.org/10.1201/9781003129509

Khan, H. (2013). Establishing a Defect Management Process Model for Software Quality Improvement. *International Journal of Future Computer and Communication*, 585–589. https://doi.org/10.7763/IJFCC.2013.V2.232

Khan, R. A., Idris, M. Y., Khan, S. U., Ilyas, M., Ali, S., Ud Din, A., Murtaza, G., Khan, A. W., & Jan, S. U. (2019). An Evaluation Framework for Communication and Coordination Processes in Offshore Software Development Outsourcing Relationship: Using Fuzzy Methods. *IEEE Access, 7,* 112879–112906. https://doi.org/10.1109/ACCESS.2019.2924404

Kobelsky, K. W., & Robinson, M. A. (2010). The impact of outsourcing on information technology spending. *International Journal of Accounting Information Systems, 11(2)*, 105–119. https://doi.org/10.1016/j.accinf.2009.12.002

Kobitzsch, W., Rombach, D., & Feldmann, R. L. (2001). Outsourcing in India [software development]. *IEEE Software, 18(2)*, 78–86. https://doi.org/10.1109/52.914751

Kuutila, M., Mäntylä, M., Farooq, U., & Claes, M. (2020). Time pressure in software engineering: A systematic review. *Information and Software Technology, 121*, 106257. https://doi.org/10.1016/j.infsof.2020.106257

Leung, H. K. N., & White, L. (1990). A study of integration testing and software regression at the integration level. Proceedings. *Conference on Software Maintenance 1990*, 290–301. https://doi.org/10.1109/ICSM.1990.131377

Li, S., & Alon, I. (2020). China's intellectual property rights provocation: A political economy view. *Journal of International Business Policy, 3(1)*, 60–72. https://doi.org/10.1057/s42214-019-00032-x

Majchrzak, T. A. (2010). Best Practices for the Organizational Implementation of Software Testing. *2010 43rd Hawaii International Conference on System Sciences, 1–10*. https://doi.org/10.1109/HICSS.2010.83

Miguel, J. P., Mauricio, D., & Rodriguez, G. (2014). A Review of Software Quality Models for the Evaluation of Software Products. *International Journal of Software Engineering & Applications, 5(6)*, 31–53. https://doi.org/10.5121/ijsea.2014.5603

Minetaki, K., & Motohashi, K. (2009). Subcontracting Structure and Productivity in the Japanese Software Industry. *The Review of Socionetwork Strategies, 3(2)*, 51–65. https://doi.org/10.1007/s12626-009-0008-8

Narayanan, S., Balasubramanian, S., & Swaminathan, J. M. (2011). Managing Outsourced Software Projects: An Analysis of Project Performance and

Customer Satisfaction. *Production and Operations Management, 20(4),* 508–521. https://doi.org/10.1111/j.1937-5956.2010.01162.x

Niazi, M., Mahmood, S., Alshayeb, M., Riaz, M. R., Faisal, K., Cerpa, N., Khan, S. U., & Richardson, I. (2016). Challenges of project management in global software development: A client-vendor analysis. *Information and Software Technology, 80*, 1–19. https://doi.org/10.1016/j.infsof.2016.08.002

Nistala, P., Nori, K. V., & Reddy, R. (2019). Software Quality Models: A Systematic Mapping Study. *2019 IEEE/ACM International Conference on Software and System Processes (ICSSP)*, 125–134. https://doi.org/10.1109/ICSSP.2019.00025

Otaduy, I., & Diaz, O. (2017). User acceptance testing for Agile-developed web-based applications: Empowering customers through wikis and mind maps. *Journal of Systems and Software*, 133, 212–229. https://doi.org/10.1016/j.jss.2017.01.002

Pan, G. (2008). Partial abandonment as an exit strategy for troubled IT projects: Lessons from a small-and-medium enterprise. *Journal of Enterprise Information Management, 21(6)*, 559–570. https://doi.org/10.1108/17410390810911177

Penttinen, M., & Mikkonen, T. (2012). Subcontracting for Scrum Teams: Experiences and Guidelines from a Large Development Organization. *2012 IEEE Seventh International Conference on Global Software Engineering*, 195–199. https://doi.org/10.1109/ICGSE.2012.16

Rafi, D. M., Moses, K. R. K., Petersen, K., & Mäntylä, M. V. (2012). Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. *2012 7th International Workshop on Automation of Software Test (AST)*, 36–42. https://doi.org/10.1109/IWAST.2012.6228988

Rahman, H. U., Raza, M., Afsar, P., & Khan, H. U. (2021). Empirical Investigation of Influencing Factors Regarding Offshore Outsourcing Decision of Application Maintenance. *IEEE Access*, 9, 58589–58608. https://doi.org/10.1109/ACCESS.2021.3073315

Schreier, M. (2012). Qualitative content analysis in practice. Sage Publications.

Search—ISTQB Glossary. (n.d.). Retrieved November 11, 2023, from https://glossary.istqb.org/en_US/search

Seppänen, V. (2002). Evolution of competence in software subcontracting projects. *International Journal of Project Management, 20(2*), 155–164. https://doi.org/10.1016/S0263-7863(00)00043-0

Shah, H., Harrold, M. J., & Sinha, S. (2014). Global software testing under deadline pressure: Vendor-side experiences. *Information and Software Technology, 56(1)*, 6–19. https://doi.org/10.1016/j.infsof.2013.04.005

Shah, T., & Patel, S. (2014). A Review of Requirement Engineering Issues and Challenges in Various Software Development Methods. *International*

Journal of Computer Applications, 99, 36–45. https://doi.org/10.5120/17451-8370

Shawosh, M., & Berente, N. (2019). Software Development Outsourcing, Asset Specificity, and Vendor Lock-in. *AMCIS 2019 Proceedings*. https://aisel.aisnet.org/amcis2019/strategic_uses_it/strategic_uses_it/17

Silva, A., Araújo, T., Nunes, J., Perkusich, M., Dilorenzo, E., Almeida, H., & Perkusich, A. (2017). A systematic review on the use of Definition of Done on agile software development projects. *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 364–373. https://doi.org/10.1145/3084226.3084262

Smith, M. A., Mitra, S., & Narasimhan, S. (1996). Offshore outsourcing of software development and maintenance: A framework for issues. *Information & Management, 31(3)*, 165–175. https://doi.org/10.1016/S0378-7206(96)01077-4

Takanen, A., Demott, J. D., Miller, C., & Kettunen, A. (2018). Fuzzing for Software Security Testing and Quality Assurance, Second Edition. Artech House.

Transaction Costs Theory—An overview | ScienceDirect Topics. (n.d.). Retrieved December 6, 2023, from https://www.sciencedirect.com/topics/social-sciences/transaction-costs-theory

Turk, D., France, R., & Rumpe, B. (2014). Limitations of Agile Software Processes (arXiv:1409.6600). arXiv. https://doi.org/10.48550/arXiv.1409.6600

Vogelsang, A., & Borg, M. (2019). Requirements Engineering for Machine Learning: Perspectives from Data Scientists. *2019 IEEE 27th International Requirements Engineering Conference Workshops (REW)*, 245–251. https://doi.org/10.1109/REW.2019.00050

Wang, Y., & Shi, H. (2009). Software Outsourcing Subcontracting and Its Impacts: An Exploratory Investigation. *2009 33rd Annual IEEE International Computer Software and Applications Conference*, 1, 263–270. https://doi.org/10.1109/COMPSAC.2009.42

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys, 29(4)*, 366–427. https://doi.org/10.1145/267580.267590

APPENDIX 1 INTERVIEW STRUCTURE

Introduction

- Study's purpose
- Reminder of recording, data privacy and data processing
- Overview of upcoming interview questions

Test analysis phase

- Test analysis phase includes reviewing requirements and designing test cases. Describe the current process in which test analysis phase is conducted. Evaluate the maturity of the process on a 1-5 scale.
- What kind of problems caused by subcontracting relationship have you encountered during this phase?
- What solutions would you propose to solve the mentioned problems?

Test planning and preparation phase

- Test planning and preparation phase includes test scheduling, resource allocation and setup and maintenance of test environments. Describe the current process in which test planning and preparation is conducted. Evaluate the maturity of the process on a 1-5 scale.
- What kind of problems caused by subcontracting relationship have you encountered during this phase?
- What solutions would you propose to solve the mentioned problems?

Test execution phase

- Test execution phase includes running the functional and non-functional test cases as well as defect reporting and management. Describe the current process in which test execution phase is conducted. Evaluate the maturity of the process on a 1-5 scale.
- What kind of problems caused by subcontracting relationship have you encountered during this phase?
- What solutions would you propose to solve the mentioned problems?

Test closure

- Test closure includes reviewing test reports and artifacts by test manager and making a decision whether the software is approved. Describe the

current process in which test closure is conducted. Evaluate the maturity of the process on a 1-5 scale.
- What kind of problems caused by subcontracting relationship have you encountered during this phase?
- What solutions would you propose to solve the mentioned problems?

General cooperation

- What problems have you encountered in communication, mutual understanding of information and trust between the client and the subcontractor?
- What problems have you encountered in the general cooperation model? This means consolidating client's and subcontractor's goals and software development processes together.