Sebastian Öberg

Software Test Automation

A qualitative study on optimizing maintenance in test automation

Information Systems

Bachelor's thesis

Term:	AUTUMN-2022
Supervisor:	Ala Sarah Alaqra
Examiner:	John Sören Pettersson

Abstract

In IT the waterfall model is being replaced by agile development processes. When transitioning into agile processes software products are delivered in iterations, or cycles, of the same software. That software will be tested repeatedly, by the same tests. When the same test is executed several times test automation comes into consideration. With test automation organizations aim to cut costs but also have predictable and efficient test execution. Over time though, research suggests that maintenance of test automation could become a burden rather than a success.

This thesis aims to investigate and research problem areas that affect test automation maintenance and what measurements can be taken to manage maintenance. In total seven semi-structured interviews are conducted with software testing professionals. The gathered empirical data have been analyzed using a thematic approach, which is the result of this study. The outcome of the thematic analysis resulted in four themes *Maintenance Issues, Minimize Maintenance, Coding Guidelines,* and *Collaboration Guidelines*. Findings from those themes and underlying codes revealed several sources and problem areas in test automation maintenance. Respondents mentioned several challenges with Flaky tests, unreliable environments, and usage of incorrect tools. When it comes to reducing maintenance, respondents evaluated a list of guidelines but also mentioned a set of tools or approaches to use, to keep maintenance to a minimum.

Keywords: Test automation maintenance, test automation guidelines, tech debt

Acknowledgments

I would like to thank my supervisor Ala Sarah Alaqra for the support and guidance during this study. I am also grateful for the review help received from my study group.
Without the input from the conducted interviews this study would never have been completed, so I am expressing gratitude to the testing and QA (Quality Assurance) professionals that participated.

Karlstad 2023-01-23
Sebastian Öberg

Table of Contents

1 Introduction

1.1 Background

Some companies have been using test automation for decades, but it has become more predominantly used in recent years. One of the reasons for this is the transition from the waterfall approach to the agile approach (Axelrod, 2018, p.4). With a waterfall approach, a software project was considered more of a one-time thing. With proper planning and engineering the system was built and tested once. A test case might fail because of programming mistakes, and the test case might be executed a second time after a fix is deployed. With test cases executed once or twice, it is faster, and cheaper, to run them manually rather than automating them (Axelrod, 2018, p.4).

Nowadays it has become clear that the waterfall approach is having a hard time delivering its promises, projects are more and more complex which makes it hard to plan everything in advance. Technology and business evolve fast, and it is more important to listen to the customer rather than stick to an original plan. Projects and products are delivered in cycles of the same software (Axelrod, 2018, p.4).

Garousi & Felderer (2016, p.68) states that automated software testing today is mainstream within the software industry. Companies like Microsoft have been automating tests for many years back. More than a million automated test cases were written for Microsoft Office 2007. Using test automation organizations aim to utilize the benefits of automation such as predictability, repeatability, and efficient test execution (Garousi & Felderer, 2016, p.69).

When utilizing these benefits organizations can see a reduction in cost spent on automation (Moses et al., 2012, p. 38-39). According to a report published by Zion Market Research (2021), the global test automation market is expected to have an annual growth rate of approximately 12.3% between 2021 and 2028. In 2020 the market was estimated to be around USD 22 billion, and in 2028 it is expected to be 54 billion.

Money spent on test automation is increasing but previous research suggests that there are challenges. The initial cost to set up the automation framework, train people, and adapt test case design is a subset of issues. However, over time organizations can see that investment turn into cost savings. In the long run test automation maintenance is identified as a burden likely to increase over time (Moses et al., 2012, p.41-42). Collins et al. (2012, p.441) claim that minimizing maintenance costs is the most difficult obstacle in test automation.

Challenges associated with maintenance are highlighted in several other studies. A case study performed by (Wiklund et al., 2012, p.6) concluded that they couldn’t identify guidelines on how to design, implement and maintain an automated test system in a way that keeps technical debt on an acceptable level. In another study conducted by Borjensson & Feldt (2012, p. 358) their industrial partner raised concern over maintenance costs as the system grows. If the cost is too high that might limit the long-term applicability of visual GUI testing.

1.2 Purpose

The purpose of this thesis is to contribute with an analysis of problem areas related to test automation in agile development projects, but also provide a set of guidelines to follow to minimize maintenance of test automation.

This thesis addresses two Research Questions (RQ):
RQ1: What issues might affect the test automation maintenance?
RQ2: What measures can be taken to reduce test automation maintenance?

1.3 Target group

The target group of this thesis is individuals and professionals working on agile development projects. Professions or roles could be testers, automation engineers, and developers. That group includes stakeholders which might be in a position to make strategic decisions. Ultimately the results of this study could partly form the basis for decisions on test automation in ongoing projects.

1.4 Scope

The scope of this study is restricted to agile IT projects. Respondents will be software testing professionals with experience in test automation. Age, gender identity, or geographical location will not be taken into consideration. The study will be conducted online for flexibility.

2 Literature review

The investigation process of existing knowledge has been made through a variety of digital sources. Diva portal, both advanced search for research publications and advanced search for a student thesis, Scopus, Google Scholar, and Karlstad University library.
These sources have been explored using a combination of search terms and keywords such as: “test”, “test automation”, “regression testing”, “functional test”, “maintenance”, “tradeoffs” and “tech debt”. A non-digital source has been Fyrens library located in Kungsbacka, south of Gothenburg, Sweden.

This chapter aims to present previous findings and related literature connected to software testing, test automation, and test automation maintenance in agile projects. The chapter is finalized with a revisiting of the research questions.

2.1 Software development Lifecycle

2.1.1 Waterfall software development

Prior to 1970 computer projects were executed in a less formal way. Due to this several high-profile projects failed during the 1970s. In an attempt to address the lack of formality, the waterfall model was introduced in 1970. This linear model has been popular when it comes to large-scale development projects. Mainly because it makes project planning and control easier (Beynon-Davies, 2003, p. 318). System development following the model is split into separate activities which are executed in sequential order. The output from one activity act as input for the next activity thus creating a flow of the project. Before moving from one activity to the next both verification and validation are performed on the completed work. With verification, the aim is to determine if the product is built in the right way, and with validation, the aim is to determine if the product built is fit for its purpose (Cadle & Yates 2008, p. 69). Between the different activities, there is usually a small degree of iteration, the iteration mainly occurs inside an activity. That means that a done activity won’t be revisited once the project moves on. If a new requirement is identified after a requirement-related activity is done that won’t be incorporated into the product. This could result in product changes that are amended at a later stage (Cadle & Yates 2008, p. 69). According to Beynon (2003, p.318), this is also one of the major disadvantages of the waterfall approach. Changing early analysis and design late in the project is very difficult.

In modern development, the waterfall model is more of a generic approach that emphasizes a sequential execution of different stages. The naming and contents of those stages are not fixed and easily be adapted to every unique project (Cadle & Yates 2008, p. 69).

The waterfall model is most suitable for smaller projects in a context where business requirements are fixed and less likely to change as time pass. If the business requirements are hard to understand there is a risk that the system might undergo radical changes as the project progresses, and then another approach might be adopted (Cadle & Yates, 2008, p. 71).

2.1.2 Agile software development

As stated by Beynon-Davies (2003, p. 318) an iterative, or agile, model seems to reduce risks in computer projects and creates a stronger commitment from stakeholders. Factors that tend to mitigate some of the risks are a high degree of user involvement and prototyping. Beynon-Davies (2003, p. 318) points out that iterative models have been particularly popular in small and medium-sized organizations.

An agile approach is a set of techniques and frameworks which enabled a system to be built in iterations rather than building the whole system in one go (Cadle & Yates, 2008, p. 79). Unlike the linear waterfall approach, agile methods are suitable when customers are unable to deliver clear and detailed requirements. These customers might be moving into an unknown business area for example. To be able and respond to organizational change the system is built in iterations that include features or improvements to previous iterations (Cadle & Yates, 2008, p. 79). Feature is a keyword central in agile approaches. Other agile keywords from Agile Alliance (n.d.) are sprints, retrospectives, and stories.

- Sprint is a set amount of time a team spends working on one increment. It can be one month or less. The outcome should be a potentially shippable product.
- A retrospective is a review being arranged after each sprint. The purpose is to identify adjustments that can be done going forward.
- Story is work split into functional increments.

In 2001 the Agile Alliance (n.d), a global non-profit membership organization, created the Manifesto for Agile Software Development. This document contains values and 12 principles that are good to live by when executing agile projects in a given context. Agile Alliance (n.d.) describes the principles:

1. Our highest priority is to satisfy the customer through the early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for a shorter timescale.
4. Businesspeople and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

According to recent reports, more and more companies are adopting agile rather than waterfall. Data summarized in the 15th State of Agile Report (Digital.ai, 2021) points out that 94% of 1380 global respondents in their annual survey have adopted agile and 65% said that they have more than three years of practicing agile. Another report presented in 2020 by The Standish Group data (2020, as referenced in the blog Mersino, 2021) suggests that waterfall projects are two times more likely to fail.

In the book *Agile Testing: A practical guide for testers and agile teams*, Crispin & Gregory (2008, p.13) point out that testing in agile projects is different from traditional projects. Agile projects are iterative and incremental. Agile teams develop and test small pieces of code, and when it works, they move on to the next small piece.

2.2 Software Testing

In the process of constructing an IT system, various types of tests must be performed to ensure that the system is working effectively. Effectiveness can be expressed in two terms, functional requirements, and non-functional requirements (Beynon-Davies, 2003, p.366) Functional requirements describe functionality that the system should have to fulfil business requirements and non-functional requirements are characteristics a specific functionality cannot fulfil, like response time (Schotanus, 2009, p.53). The activity of testing software is both time-consuming and expensive, it is often accounting for 50% – 60% of the development cost (Memon, 2002, p.87). Before the end of each coding phase, there is a variety of activities that needs to be performed. These activities should be done in parallel with the development work, testers collaborate with both developers and business analysts to produce a set of test deliverables. Using this approach testers are involved, in different test levels, from the start to the end of a development phase and this is illustrated in the V-model (Black et al., 2012, p.28).

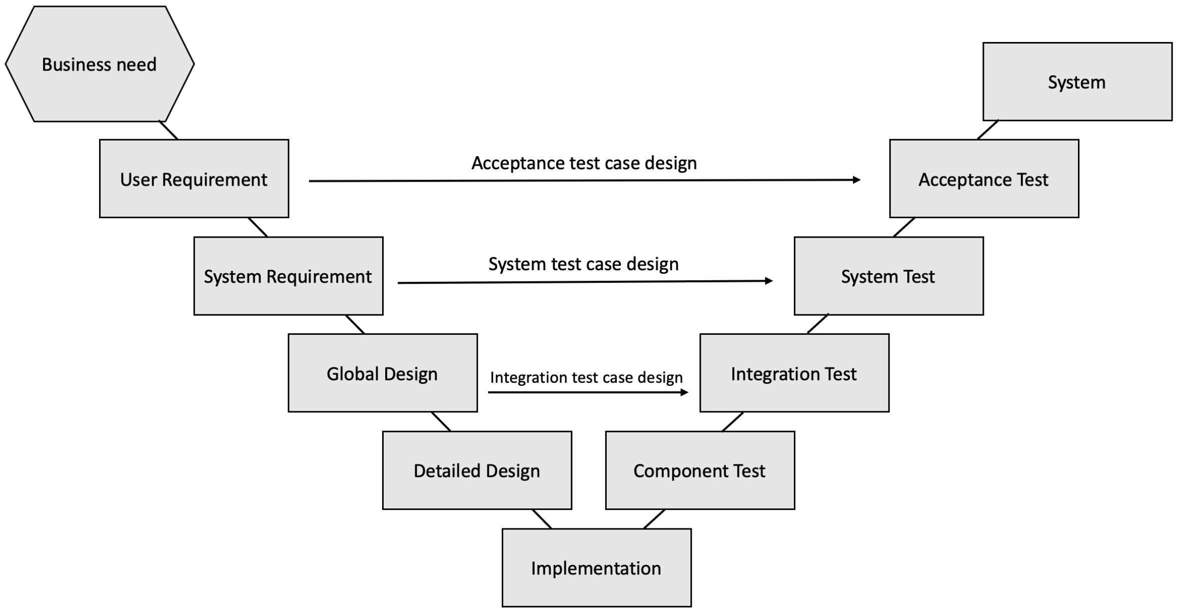

Figure 1: V-model (based on medium.com, 2019)

As seen in figure 1, the V-model contains different levels of testing activities (verification and validation) that can be integrated into each phase of the life cycle (Black et al., 2012, p.29). Black et al. (2012) state that there exist different variants of the V-model, but it is common that four test levels are used, each with its own objectives.

- Component testing – search for defects in software components that are separately testable (Black et al., 2012, p.29).
- Integration testing – aims to test interfaces between components or systems (Black et al., 2012, p.29).
- System testing – testing related to the behavior system of the system as a whole. The system is tested against requirements (Black et al., 2012, p. 29).
- Acceptance testing – validation testing verifying the system from a user or business perspective. This level aims to determine whether to accept the system or not (Black et al., 2012, p.29).

According to Black et al. (2012, p.43), the final target of testing is the testing of changes, this is referred to as regression testing. The purpose of regression testing is to verify modifications of the system or environment haven't impacted the functionality of the system. To perform regression testing organizations usually use a regression suite or regression test pack. These sets or packs contain high-level test cases specifically selected for regression testing with the purpose of exercising most functions of a system. Test cases added to a regression pack are natural candidates for automation since the tests should be executed every time there is a change to the system (Black et al., 2012, p.44).
In the book *Agile testing: The agile way to quality,* Baumgartner et al. (2021, p.17) write that test automation plays a crucial role in an agile environment, especially for regression testing. Software developed and delivered in shorter cycles must be tested by automated regression tests.

2.3 Software Test Automation

In the book *Implementing Automated Software Testing,* Dustin et al. (2009, p.23) write, if implemented correctly, the purpose of test automation is to time and cost of software testing, improve software quality, help manual testers by replacing cumbersome tasks but also execute tests that manual testers have a hard time running (like memory leaks, concurrency, and performance testing).
Another point declared by Garousi & Felderer (2016, p.68) is that test automation is most common in software projects of evolving nature, where the system is released in many

versions. The reason behind this is that test automation payoff is higher for regression- and repetitive testing. Baumgartner et al. (2021, p.157) confirm that automation is more suitable for projects of evolving nature and states that automation is key to ensuring the quality of deliverables at the end of each sprint.

Dustin et al. (2009, p.3–6) state that all manual test cases can be automated, no matter what type of test it is. Further, he distinguishes between the differences between manual and automated testing. Automation can help manual testers by automation of test cases that are hard to test manually, and test automation is not replacing the need for manual testers analytical thinking and testing skills. Ultimately test automation and manual testing complement each other.

In 2005 Berner et al. (p.571) published a paper on observations and lessons learned from automated testing. The observations (see section 2.5) were based on dozens of projects related to test automation. These observations were analyzed and resulted in four recommendations (see section 2.4.1). In summary, one of the observations confirms that test automation and manual testing complement each other. Berner et al. (p.575 - 576) highlight that automated testing cannot replace manual testing. Manual testers find most new defects because automated testing is re-testing existing software. Manual testers use their knowledge of fins to find weak spots in the system and development process.

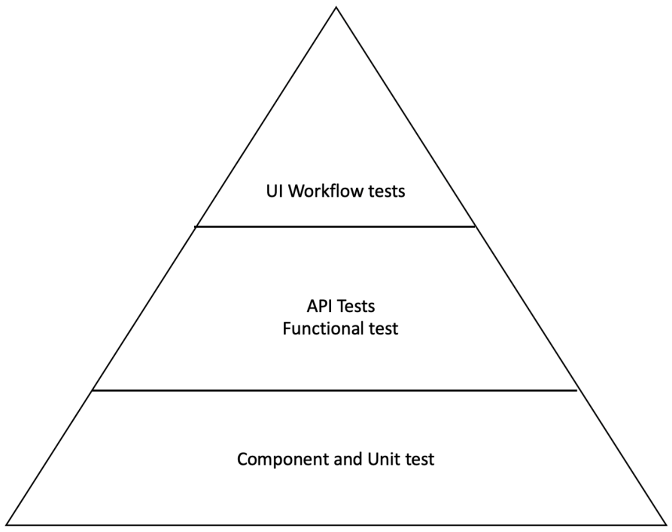

Figure 2: Test automation pyramid (based on luxoft-training.com, 2022)

Figure 2 is an example of the automation pyramid. In the book *More agile testing: Learning journeys for the whole team,* Gregory & Crispin (2014, p.223) explains that the test automation pyramid is commonly used when teams decide on what to automate. Starting from the bottom of the pyramid there are unit tests. These tests should be executed when new code is added and provides fast feedback and the highest ROI (Return on investment) (Gregory & Crispin, 2014, p.115). In the middle of the pyramid, integration tests are running against APIs (Application programming interface) (Gregory & Crispin, 2014, p.223). Most applications are controlled using UI (User Interface), but some applications and components are designed to be controlled by other applications. For an application to be controlled by another application it should expose an API (Axelrod, 2018, p.49). According to Gregory & Crispin (2014, p.223), integration tests against APIs also provide fast feedback due to the lack of UI.
On the top of the pyramid, tests are running on UI. UI tests tend to be more complex and return slow feedback (Gregory & Crispin, 2014, p.223). As an extension to the top of the pyramid exploratory tests could be added, Gregory & Crispin (2014, p.223) highlight that many teams require manual tests to complement their automated checks.
The purpose of exploratory testing is to give testers the possibility to explore the system in an unscripted manner to find bugs (Axelrod, 2018, p.19).

2.4 Maintenance in test automation

Zhan (2021) states in an article that 'Automated test scripts require ongoing care and maintenance. Unmaintainable or hard-to-maintain automated test scripts are in vain.'. Just like source code, test code should be properly verified and improved if needed (Garousi & Felderer, 2016, p.4).
A case study conducted by Wiklund et al. (2012, p.6) highlights that technical debt is a challenge as a project grows. Observations from that paper state that the researchers found no clear and detailed guidelines on how to implement and maintain an automated test system. Furthermore, the researchers believe that there is a need for such guidelines (Wiklund et al., 2012, p.6).
Their chosen method that led them to that conclusion was a series of interviews combined with the inspection of several documents, design rules for test scripts, quality guidelines for the studied application, and improvements identified within the studied teams (Wiklund et al., 2012, p.6).

In more recent years Alégroth et al. (2021, p.11) conducted a study including guidelines that suggest that the issue still exists. The study objective was to identify and evaluate coding guidelines for VGT (Visual GUI testing) scripts and measure their impact on maintenance costs in industrial practice (Alégroth et al., 2021, p.3). The coding guidelines were identified from an unstructured literature review. One part of the literature review identified guidelines derived from the Software Improvement Group (SIG). Software improvement group (2022) is providing a set of ten guidelines for delivering software that is easy to maintain and adapt.
“

1. Write short units of code: limit the length of methods and constructors
2. Write simple units of code: limit the number of branch points per method
3. Write code once, rather than risk copying buggy code
4. Keep unit interfaces small by extracting parameters into objects
5. Separate concerns to avoid building large classes
6. Couple architectural components loosely
7. Balance the number and size of top-level components in your code
8. Keep your codebase as small as possible
9. Automate tests for your codebase
10. Write clean code, avoiding “code smells” that indicate deeper problems. (Software improvement group, 2022)"

As a complement to the SIG guidelines Alégroth et al. (2021, p.5) interviewed a VGT expert with more than 15 years of experience. To enable the evaluation of the effects on maintenance cost a set of manual test cases were selected, these tests were then developed with and without the guidelines. Finally, a series of interviews were conducted with individuals who had maintained the scripts. Each guideline was a subject for discussion (Alégroth et al., 2021, p.3-6). The Guidelines are presented in table 1 (Alégroth et al. 2021, p. 6-7).

Table 1: Coding Guidelines

Guideline (G)	Description
G1 Write short units of code	With short units of code reuse and analysis is made easier. Units should have one single responsibility.
G2 Write simple units of code	By keeping branch conditions to a minimum, the code will be easier to analyze, understand and change.
G3 Write code once	Avoid duplication and maximize reusability. This helps improving maintenance by restricting changes to one location, rather than several.
G4 Keep unit interfaces small	Units accepting several parameters are usually a sign that the unit has multiple responsibilities. This will also make the unit harder to reuse.
G5 Separate test cases accordingly to manual steps	Test cases should not be linked or dependent on each other. One test case should have one responsibility. Increased modularity and loose coupling will make maintenance easier by improving readability and analyzability.
G6 Refer to images once	Images used with image recognition should only be stored once and referred to via variables.
G7 Keep The codebase as small as possible	Increase maintainability by avoiding unnecessary growth of the code base.
G8 Use descriptive and uniform naming conventions	Scripts and code will be easier to analyze and understand with descriptive and uniform naming conventions. It will also be easier to search in the codebase.
G9 Use synchronous checks	Instead of having static waits, e.g., wait for a set amount of time, dynamic wait should be used. As an example, a dynamic wait could wait for a visual state.
G10 Take a screenshot when tests fail	Take a screen shot when the test fails. This will help when trying to reproduce the bug.
G11 Take case to choose suitable imagery	Image recognition is dependent on images of suitable size. Too large an image can disrupt the test since it contains

	too much information. If the image is too small, it might contain to less data to be reliable.
G12 Write clean code	Avoid TODO's and "dead" code in the codebase. Keeping the codebase clean makes it easier to read and understand.

From the interviews and analysis, Alégroth et al (2021, p.10) found that the proposed guidelines had a negative effect on maintenance costs. There were two identified reasons for these findings. The first reason is that the researchers assumed that test automation was closely linked to traditional software, but that assumption was partly false. In contradiction, as part of the definition, Dustin et al. (2009, p.4) write that test automation is software development. Since it is software development the implementation of automaton requires a development lifecycle, and therefore best practices of software development should be followed (Dustin et al., 2009, p. 6). The second reason was that the number of guidelines was too many. Essentially, it is hard for a developer to consider 12 guidelines and apply all of them.

2.4.1 Managing maintenance

Teams often make mistakes in their initial attempts at functional test automation partly because today's frameworks and drivers make it easy to automate tests quickly (Gregory & Crispin, 2014, p.211). To address this Gregory & Crispin (2014, p.221) propose a set of ways to manage test technical debt and maintenance, especially related to test automation:

- Make the time spent debugging test failures and maintaining automated tests visible by writing task cards for these activities or reflecting the extra time in story estimates.
- Making bugs and the resulting blocked stories visible reminds the team to fix defects quickly, focus on defect prevention, and avoid incurring technical debt in the form of defect queues.
- The whole team needs to take responsibility for managing technical debt - both code and test.
- Identify the biggest source of technical debt and try experiments to reduce that debt.
- Have team members with different specialties work together and apply those different skill sets to reduce technical debt.

In Berner et al. (2005)'s paper on observations and lessons learned from test automation, four consequences, or recommendations, which help implement sustainable test automation were concluded (Berner et al., 2005, p.579).

- Adopt a sound test automation strategy to increase the chances of successful test automation. To be able to proactively estimate cost-effectiveness the test strategy should be documented. The test strategy should contain quality objectives and activities to refine and maintain test cases (Berner et al., 2005, p.579). Goals for test automation should also be defined and test automation approach should be diverse, meaning combining test levels rather than focusing on one. Finally, the test automaton approach should be frequently evaluated and improved.
- Design for testability should be taken into consideration as early as possible. If a system architecture is not designed for test automation it can be impossible to change.
- To improve software quality and avoid degrading test cases it is advised to execute the test suite as often as possible. This will help identify issues with both product and testware.
- Treat test automation as any other software project. They are prone to the same issues. Most of the observed organizations ended up with a major maintenance burden due to inappropriate software design. To avoid this Berner et al. (2005, p.579) advise applying and following software development guidelines.

Observations and mistakes often made, leading to the consequences mentioned above, are *inappropriate test automation strategy, tests are more often repeated than expected, run tests frequently to avoid degradation, automated testing is not replacing manual testing, testability is forgotten as non-functional requirements,* and *maintenance of testware is hard* (Berner et al., 2005, p.572-577).

To increase the chances of successful test automation a sound testing process and appropriate test (automation) strategy should be adopted. Typically, four mistakes are made concerning the automation strategy (Berner et al., 2005, p. 572).

Misplaced or forgotten test types – Tests that are hard to perform manually are usually hard to automate as well. As a result, many organizations rely on system tests and ignore integration tests. This has an impact on test robustness (Berner et al., 2005, p. 572).

Wrong expectations – Many organizations expect a short ROI for their test automation investment, if that expectation is not met there is a risk of test automation being canceled (Berner et al., 2005, p. 573).

Missing diversification – Goals of test automation are usually clear and that is to save money and time. Organizations new to test automation tend to automate GUI tests, and that is not an efficient test strategy. To create a good test strategy a combination of test levels is advised. E.g., system test, integration test, and unit test (Berner et al., 2005, p. 573).

Tool usage is restricted to test execution – Test automation tools are often restricted to test case execution. There might be other areas where test tools could add value. It could be installation or configuration processes (Berner et al., 2005, p. 573).

The next observation is that tests are more often repeated than expected. Organizations tend to use a number of expected test executions when selecting automation candidates. Almost all test cases are executed 5 – 10 times, and 25% of the test cases are executed more than 20 times. If automation is not cost-effective, it is a sign of a weak test automation strategy. Focus on running component and integration tests rather than GUI. GUI-based tests require more effort to implement and are difficult to maintain (Berner et al., 2005, p. 574). The following observation is related in terms of test runs. Tests should be run frequently to avoid degradation. When a test suit is not frequently executed it will end up in a state of inconsistency and will be difficult to understand (Berner et al., 2005, p. 574-575).

Berner et al. (2005, p.575-576)'s next observation was that automated testing is not replacing manual testing. Automaton is the revalidation of a unit under test and manual testing is more of a destructive type. 60% – 80% of bugs were found when developing the tests.

Another observation was that testability is forgotten as a non-functional requirement. This observation is split into two parts, the design of the software itself and the design of the environment for test automation. The system could be hard to test if testability is not being taken into consideration. Obstacles could be technical limitations such as missing error messages. For the test automation environment, common problems are manual intervention like installations, lack of basic infrastructure, or test automation not being observable to the tester (Berner et al., 2005, p. 576-577).

The last observation is that testware maintenance is hard. Testware is everything that is needed for automated testing. That includes test scripts, simulators, input data, utilities, and test reports. Testware is usually built without proper planning, design, documentation, or architecture, unlike ordinary software projects. With the lack of testware architecture, there are consequences to the overall test automation maintenance. It impacts both test scripts and automation infrastructure (Berner et al., 2005, p. 577). From the observations, four pitfalls were identified.

Undocumented architecture that is not engineered and built in the same manner as "real" software projects. Standard software development is often neglected (Berner et al., 2005, p. 577).

Missed opportunities for reuse results in high maintenance costs because repeatable tests and repeatable functions are not reused. GUI automation is more prone to cause maintenance. A small change in the GUI can lead to changes in many scripts. This is a common problem in organizations (Berner et al., 2005, p. 577-578).

Poorly structured testware is causing duplication of code. Testware is usually poorly structured which makes it hard to analyze.

Untested testware – test code also needs some amount of testing.

2.5 Test automation maintenance in practice

As an evidence-based experience, Garousi & Felderer (2016, p.4-5) published a paper on how to develop and maintain test scripts. They present a case where researchers used an IBM automation tool to create a set of tests that was executed against two versions of an application. In the second run against the updated version two maintenance activities were identified. The first activity was to update test cases because of changes between the two versions. The second activity was related to updating, or "repairing" existing test cases. Out of 71 test cases, 34% needed re-recording.

In the same year, 2016, Alégroth et al. conducted a study related to the maintenance of automated GUI tests and in the introduction concluded that empirical data is limited (Alégroth et al., 2016, p.1). In 2014 Christophe et al. (p.141) conducted a study on the maintenance of functional web tests. In the 'Related work' section, Christophe et al. (2014, p.146) state that "Little is known about how automated functional tests are used in practice" and then elaborate on the importance of maintenance using Berner et al. (2005)'s observations (see section 2.4.1)

Telerik is a public company offering software solutions for developers and testers. Amongst all products, there are tools focused on test automation. With Telerik's experience and knowledge, they share what they think are tips to avoid test automation maintenance (Telerik, n.d.).

Those tips are focused on automating the right things, don't automate low-value or stable features, cutting your configuration matrix, avoiding complex test scenarios, keeping tests granular and independent, learning your system's locators, learning your system's asynchronous actions, leveraging code where needed, configure your system for easier testing and go beyond the UI for your functional tests (Telerik Test Studio, n.d.).

2.6 Revisiting the research questions

The literature review exposes issues with test automation and test automation maintenance, it also provides suggestions on how to approach maintenance. Using guidelines identified from the literature review we aim to answer the below research questions. This will be done via semi-structured interviews speaking to testing professionals with experience in test automation:

RQ1: What issues might affect the test automation maintenance?

RQ2: What measures can be taken to reduce test automation maintenance?

3 Method

This chapter explains the research approach which has been taken to conduct this study, this includes both data collected but also how it is used. To answer the stated research questions a qualitative method has been adopted.

3.1 Research method

The chosen research method for this study is of deductive qualitative approach. The deductive way of gathering information can be described as “from theory to empiricism”, in theory, acquire expectations of what the world looks like and then gather data to support or reject that view (Dag, 2011, p. 34). This method is preferably used if the aim is to bring more clarity to a concept or phenomenon (Dag, 2011, p. 145). With an exploratory way of looking at the problem, a qualitative approach will gather nuanced data, in more depth rather than spread, and with an openness to contextual aspects. Data can be gathered in several different ways such as interviews, observations, or group interviews (Dag, 2011, p.56-57).
Qualitative studies are, almost always, of an unstructured nature because the informants can speak freely and answer the interview questions in their own words (Patel & Davidson, 2011, p.104). For conducting interviews in this study, a semi-structured approach is taken. A semi-structured type of interview means that the researcher predefines a list of specific domains or themes to be touched upon, but the informants are free to shape their answers. (Patel & Davidson, 2011, p.105). The location for the interviews is online using the tool Zoom, with a Karlstad University linked account. Respondents will be given the guidelines sent to them before the interview (see Appendix B). With interviews being conducted online there will be higher flexibility due to the lack, or need, of travelling.

3.2 Analysis method

Qualitative data collected during this study will be analyzed using thematic analysis. The data will be categized into themes, which are split into codes.
From data, thematic analysis is a method applied to identify, analyze, and report patterns or themes. It describes and organizes data into details (Braun & Clarke, 2006, p.80).

3.3 Recruitment and sampling

In the process of identifying informants for this study a set of selection criteria, or preconditions, must be fulfilled. The criteria are:

- Profession or role within the agile project.
- At least 1 year of experience working with test automation.

To identify, evaluate and recruit respondents an existing professional network is used.

3.4 Reliability and validity

When speaking of qualitative studies, the term's reliability and validity are intertwined. Because of that is reliability rarely spoken about in these kinds of studies (Patel & Davidson, 2011, p.134). For qualitative studies, the validity is denoted that we study the right things (Patel & Davidson, 2011, p.133). The concept of generalization is often perceived as problematic in qualitative studies (Patel & Davidson, 2011, p.137). To help improve generalization respondents from different companies have been selected. These respondents have been identified from an existing professional network, but all the respondents have been unknown to the interviewer.

3.5 Ethical considerations and GDPR

When conducting research, we need to consider ethical aspects. The goal of all research is to generate knowledge that is as trustworthy as possible, knowledge that is of importance to both individuals and for the development of society. This means there is a balance between the social benefit of the research and protection against inappropriate insight into people's lives (Patel & Davidson, 2011, p.104).

Dag (2002, p. 483) mentions three fundamental requirements that a study should try to fulfill:

1. Informed consent
2. Privacy requirements
3. The requirement to be correctly reproduced.

This study is fulfilling all aspects mentioned above. All informants have been given the background and purpose of this study (see Appendix D), and all informants also have signed a consent form (see Appendix C). All data collected, stored, and analyzed are GDPR compliant. Respondents are referred to as characters like R1, R2, and R3 (see chapter 3.3). Company name and geographical location have been left out.

4 Results

4.1 Demographics

Totally seven interviews were conducted with different participants. All the participants fulfilled the criteria of sampling found under section 3.2. Note that company B has clients indicated by numbers. Company B is a consulting company, followed by a number which is the client.

Respondent 1 (R1)

Work as a QA (Quality Assurance) consultant for company A. Comfortable coding, quite new to the IT industry. Testing is mainly done in an application that lacks a UI. Most of the tests are automated.

Respondent 2 (R2)

Work as a front-end automation consultant for Company B1. Comfortable writing tests, but the respondent is not a developer.

Respondent 3 (R3)

Work as a test and QA consultant for Company B2 and is comfortable writing code. Currently, the respondent is hired to improve the test process, specifically the automation levels.

Respondent 4 (R4)

Work as a QA consultant, or resource, for Company B3 and is comfortable writing code. Mainly working with automation of UI tests.

Respondent 5 (R5)

Work as a QA consultant with a focus on test automation and test coaching for Company B4. Respondent is comfortable writing code. Experience working with test automation at different levels.

Respondent 6 (R6)

Work as a test automation consultant for Company B5 and is quite comfortable writing code. Primarily focusing on UI automation.

Respondent 7 (R7)

Work as a quality assurance engineer for Company C, not comfortable writing code. Automated tests are not on the UI level. Respondent support developers by writing new scripts.

4.2 Thematic analysis results

The qualitative data gathered from the interviews have been transcribed and used as the foundation for the thematic analysis. From the analysis four themes were identified:

1. Maintenance issues
2. Minimize maintenance
3. Coding guidelines
4. Collaboration guidelines

4.2.1 Maintenance issues

Problems, issues, and impacts related to test automation maintenance were widely mentioned in all the conducted interviews. From the transcribed interviews several codes could be identified. First, the issue of *flaky tests* was brought up in five of the seven interviews as a source of issues. According to R2 flaky tests are usual regarding maintenance "*Other.. test flakiness, like, where a test will fail intermittently and that's pretty usual issue*". R4 speaks of flakiness as a challenge in test automation "*Flaky tests. Uh, is a big challenge, or can be a big challenge.*". Five of the respondents brought up *environment* and system under test as a source of issues, especially connected to flaky tests. R2 mentioned the environment as a source of unexpected behavior and stated, "*It could also be that the APIs are responding slowly or not at all, and you need to just rerun it and then everything works*". R5 also elaborates around that the environment could be a source of unexpected behavior stating that the environment should be possible to reset "*maybe you test in a environment that you cannot even reset*", and "*Then you need to be even more careful what you do there and we'll have a lot of test data failures*". R3 described a situation where they ended up with problems because they expected the system to be smarter and faster.

The next code identified is connected to the selection or usage of *incorrect tools.* R2 stated, "*struggle with myself is usually if you tried to use your tools in a way they weren't meant to*", R5 elaborated on the statement "*sometimes they choose the wrong tools*". Test automation tools are often supposed to work in a certain context and when using the tool outside that context limitations are more exposed. The respondent continued "*you know, you cannot work in different domains easily*".

The last two codes identified was *number of tests* and *domain changes*. When answering Q7 connected to test automation problems two respondents spoke about the number of automated test cases. Besides the number affecting maintenance, it also impacts the overall strategy of the team, "*To rather have one test that I can count on than 10 tests that I can't count on*". If trust in the tests is compromised "*The whole quality activity isn't really relevant anymore because you're not taking it seriously. And if something actually goes wrong, you may, I mean, it's the boy who cried wolf.*". R1 mentioned that they have an unstable pipeline with between 100 and 200 test cases. These test cases are being tracked in a backlog and should be made stable "*They haven't been working for a while, and they're on the backlog to fix. And there's still, maybe, uh, like.. Between 100 and 200 scripts.*".

Domain changes were brought up as a maintenance issue and affected existing tests. Both R1, R6, and R7 spoke about changes to the software requirements which made the test cases fail. When being asked Q7 R1 answered, "*There's have been an update for the requirements, and it doesn't really work the same as it did when the test script was written originally.*". R7 stated a similar description "*We might have to refactor some of the existing scenarios. Implementations because they don't work with our new scenario*". R6 comments were more related to UI changes "*Changes has made be made to the UI*". R7 continued in the same pattern "*Another maintenance is if a new, changes has made be made to the UI*".

Table 2: Theme 1, codes and, respondents overview

Main theme	Codes / Number of codes / Respondents	Respondents
1. Maintenance Issues 1.1 Test 1.2 Tools 1.3 Environment 1.4 Domain	Test – Flaky test (5 – R2, R3, R4, R5, R6) Test – Number of tests (2 – R1, R4) Tools – Incorrect tools (3 – R2, R5, R6) Environment – Environment (5 – R2, R3, R4, R5, R6) Domain changes (3 - R1, R6, R7)	R1, R2, R3, R4, R5, R6, R7

4.2.2 Minimize maintenance

The second theme found was *Enhance maintenance,* and a set of five codes was identified. These codes will represent a measurement to be taken when minimizing maintenance. From the list of codes, pipelines were mentioned in five out of the seven interviews. Rather than running the test on their local computer pipelines were used to execute automated tests. For Q4 R6 stated, "*Next phase and then we have another pipeline that tests the UI test.. so..*

everything basically runs seamlessly or.. Through pipelines.". Pipelines were also used to track maintenance by identifying and keeping track of broken tests. For Q5 R1 answered "*Let's say our regression tests. They're running in a pipeline. So, we keep track of which ones are failing*", after further elaboration R1 continued "*then fixing the test cases, if it is the test cases that are wrong. Or point to developer if they need to fix something.*". For less stable tests a second pipeline was used "*we have basically 2 pipelines, one with stable test cases that should be working. And if a test case in that pipeline starts to fail, it's our priority to start fix.".* R5 speaks of pipeline in the same manner as R1, using the pipelines for to identify broken tests. When elaborating on Q4 the response was "*We have with them in the pipelines. Then you can have a, short it will be, a very quick response back if it doesn't work, that's good*".

Testing pyramid was mentioned by four respondents. For Q4 R3 spoke about the testing pyramid when planning for testing in different levels and phases. To minimize maintenance R3 mentioned that stable tests, in this particular project, were on a low level according to the pyramid "*Uh, most of the tests that worked, uh, fairly stable, were mostly on a rather low level, kind of API-endpoints type tests*". R4 when speaking of the pyramid suggested moving tests down the pyramid "*And maybe we can move corner cases and configurations further down the testing pyramid and try to validate that on component level or maybe integration level*". R6's approach to the pyramid was an overall suggestion to try adopting the model "*You should include all these three (levels) and that's in my.. our goal as well*".

The next set of codes, mentioned by three respondents each, are *treat as feature code and stable environment*. In a response to Q5 R5 mentioned one success factor to reduce maintenance over time. The test code should be treated as production code, or feature code "*handle it like if it's a production code that you will develop for the products*". R5 explained that the code should be placed together with production code and should be constantly reviewed "*we always had reviews of our code and so on*". Further mentions of test code came up when asking R4 if maintenance is problematic, Q7, R4 responded that it depends on how one looks at test code "*I like to look at test code and try, I mean elevate test code to the same level as feature code*". R4 continued stating that maintenance is ongoing in the feature code as well "*Maintenance that occurs during development in the feature code as well.*".

Another success factor stated by R5 is a stable environment. R5 highlighted that the most important thing to have under control is a stable environment. R5 said, "*The most important* *thing is to have a stable.. A stable test system under test*". R2 and R3 also spoke about the importance of a stable environment to avoid tests failing intermittently.

Lasts code identified was test data which two respondents brought up. R5 and R6 spoke extensively about the importance of test data and R5 stated "*They don't have a possibility to handle the test data correctly, so that's almost the hardest part to test automation, the test data.*". R5 continued "*And maybe you test in a environment that you cannot even reset the the data or the databases*", "*then you need to be even more careful*". Both respondents emphasize test data state and explain that test data should be unique, and not reused. R6 stated, "*To isolate it as much as possible so we can minimize the maintenance, so like the test data should be unique the state*".

Table 3: Theme 2, codes and, respondents overview

Main theme	Codes / Number of codes / Respondents	Respondents
Minimize Maintenance	Pipelines (5 – R1, R2, R4, R5, R6) Testing pyramid (4 – R2, R3, R4, R6) Treat as feature code (3 - R5, R4, R7) Stable environment (3 – R2, R3, R5) Test data (2 – R5, R6)	R1, R2, R3, R4, R5, R5, R6, R7

4.2.3 Coding guidelines

The theme *Coding guidelines* comes from the guidelines discussed in the interviews, see Appendix B. Respondents were asked if the guidelines provided are important/useful or not, and the first subset of guidelines was connected to software development in general. The codes for this theme are the guidelines.

- G1 – All respondents stated this guideline to be useful. R4's answer was both yes and no. Short units of code are important, but readability is also important "*Yes and no. I mean, I think that I have, the recent year, moved more and more away from for instance page objects and levels of abstraction.".*
- G2 - All respondents stated this guideline to be useful. R4 referred to the same answer as for G1.
- G3 – This guideline was categorized as useful by all respondents. However, several of the respondents mentioned that the guideline might be hard to live by. As an example,

R7 stated, “*If it can be done, absolutely.*”. R2 expressed concerns that one might end up “overengineering” the solution “*Uhm, it's a really good practice, but it can also be premature if your if your codebase is pretty simple. Uhm, it might be more maintainable to not break out stuff*’.

- G4 – All respondents stated this guideline to be useful.
- G5 – All respondents stated this guideline to be useful.
- G6 – Three out of seven respondents categorized this guideline as useful. The other respondents lacked experience in tools used for this type of UI testing.
- G7 – Six out of the seven respondents thought this guideline was useful. Among the respondents answering yes there were a couple of remarks mentioning that the guideline is dependent on context. R2 stated, “*Sometimes it makes sense to have a monolith. Sometimes it makes sense to break out into microservices*”. R4 elaborated that the guideline is kind of useful and wanted to highlight that there is a difference between tests and framework code “*You should make a difference between the framework and the tests*”. R4 continued “*Priority one shouldn't be to keep the codebase small, it should be keep it readable and accessible*”.
- G8 – All respondents stated this guideline to be useful.
- G9 – All respondents stated this guideline to be useful.
- G10 – All respondents stated this guideline to be useful. As a complement to print screens two respondents mentioned that log files also could be of help. R3 was clear that a print screen is a good start, but more information is better “*I want constant logging of everything, so a screenshot is not enough for me*”. R6 elaboration was on the same path as R3 “*I take screenshots, I take videos and of course I take something called traces like let's say trace (logs)*”.
- G11 – Out of seven respondents three responded that the guideline is useful. Three respondents lacked experience with UI tools where the guideline is applicable. One respondent told this guideline is not useful “*I would say that I disagree with that.*“.
- G12 – Respondents all agreed that this is useful.

Table 4: Theme 3, codes and, respondents overview

Main theme	Codes / Number of codes / Respondents	Respondents
1. Coding Guidelines 1.1 Useful	G1 – Useful (6) G2 – Useful (7) G3 – Useful (7) G4 – Useful (7) G5 – Useful (7) G6 – Useful (3) G7 – Useful (6) G8 – Useful (7) G9 – Useful (7) G10 – Useful (7) G11 – Useful (3) G12 – Useful (7)	R1, R2, R3, R4, R5, R5, R6, R7

4.2.4 Collaboration guidelines

The last theme, *Collaboration guidelines*, is the second subset of guidelines presented to the respondents. See Appendix B. These guidelines are related to collaboration in teams, rather than software development in the sense of coding. The respondents were asked if the guideline is useful or not. Out of the six guidelines four were categorized as useful, and all respondents answered yes when given the question of useful or not. For two of the guidelines, there was not a unified yes:

- G13 – R2 stated the idea behind this makes sense but it conflicts with business delivery and as a team, you should not end up in this state "*if your code or your components or your tests are hard to maintain. That feels like something that the team should talk about and discuss during retro's*". Furthermore, R2 continued "*And kind of address those and then create tech debt tickets for them*" then stated that this should not be a continuous activity "*something that the team discusses and then maybe does something about rather than and continuously think about.*".
- G16 – R3 made remarks saying that experimenting is too costly, and instead of analyzing the problem for too long just try something new and see what is found "*let's*

roll with a new test and try a new thing and see what we find instead of analyzing the problem for three weeks before we write one test.".

Table 5: Theme 4, codes and, respondents overview

Main theme	Codes / Number of codes / Respondents	Respondents
1. Collaboration Guidelines 1.1 Useful	G13 – Useful (6) G14 – Useful (7) G15 – Useful (7) G16 – Useful (6) G17 – Useful (7) G18 – Useful (7)	R1, R2, R3, R4, R5, R5, R6, R7

5 Discussion

5.1 Test automation maintenance issues

With the use of thematic analysis, four themes were distinguished. The first theme, *Maintenance Issues*, connects, and answers the first research question on issues that impacts test automation maintenance. The result lists five different problem areas, those were flaky test, environment, incorrect tools, number of tests, and domain changes.

Given the results of this study, Flaky tests and Environment should be considered common issues affecting maintenance. In the interviews R2, R3, R4, R5, and R6 all mention both flaky tests and environment. Further analysis indicates that the code environment is a root cause of flaky tests. The environment should be possible to control, R5 says that an uncontrolled environment that cannot be reset is a cause of test data-related failures. R2 says that tests failing intermittently (flaky test) could be blamed on for example slow APIs. A slow API could be related to performance which is a non-functional requirement (Schotanus, 2009, p.53). In literature, Berner et al. (2005, p. 576-577) mention environment in the observation regarding that testability is a non-functional requirement not considered.

Working with incorrect tools was concluded (R2, R5, and R6) to have an impact on maintenance. The interviews showed that test automation tools often are supposed to operate in a certain context and using these tools outside that context exposes limitations. This finding could speak against observations made by Berner et al. (2005, p. 573) who suggest that test tools are often restricted to test execution but should be expanded into more areas where applicable. It is important to keep in mind that context or tools are not explicitly elaborated in this study, the same applies for Berner et al. (2005)' study. That might explain why the results could be different.

The Number of test cases (R1 and R4) and Domain changes (R1, R6, and R7) are to some extent related. Analysis of the results shows that changes to requirements and software do affect maintenance. Often when there is a change to existing functionality, one or more tests need to be maintained, R1 and R7 said. R6 didn't mention requirements changes but UI changes. Garousi & Felderer (2016, p. 4-5) confirms this scenario when identifying maintenance activities for test scripts. One of the activities included updating test cases to match a new software version. If there are big changes many test cases might be affected, which also should have an impact on maintenance.

5.2 Optimize test automation maintenance

Themes *Minimize Maintenance*, *Coding Guidelines,* and *Collaboration guidelines* all connect and answer the second research question. The theme *Minimize Maintenance* summarizes measures to be taken to reduce test automation maintenance. These measurements are pipelines, testing pyramid, treat as feature code, stable environment, and test data.

Pipelines, mentioned by R1, R2, R4, R5, and R6, are used to track and fix broken tests, they are also used to receive quick feedback if a test fails. R1 is using pipelines to keep track of flaky or broken tests. A subset of unstable tests is added to a backlog and should be prioritized to fix. To create robust and stable tests R2 and R4, execute the tests locally and then execute them in the pipelines. Test executed in a pipeline might behave differently from a local run (R2). R3 Using pipelines R5 and R6 get quick feedback from changes made by the developers.

Results show that the Testing pyramid should be adopted, and automated tests should be pushed down as far as possible (R2, R3, R4, R6). By pushing tests down the pyramid test are more stable and requires less maintenance R3 and R5 said. This is confirmed by Gregory & Crispin (2014, p. 223) who state that tests at the bottom of the pyramid provide fast feedback and faster ROI. Berner et al. (2005, p. 574)'s observations also confirm this and state "extreme caution is needed if script-based automation of GUI-based tests is attempted". Berner et al. (2005, p. 574) advise a proper automation approach with unit and component tests.

Treat as feature code is a good measure to prevent test automation maintenance (R4, R5, and R7). R4 is actively trying to elevate test code to the same level as feature code. R5 advised keeping the code structured and performing constant reviews. Reviews could be activities like code review on pull requests. This can further be confirmed by Berner et al. (2005, p. 579) who recommend treating test automation as any other software project, and if not followed organizations might end up with a major maintenance burden. Berner et al. (2005, p. 577) observed that testware maintenance is hard and is often built with poor structure, unlike regular software. On the contrary, Alégroth et al (2021, p. 10) saw a negative impact on maintenance costs because the assumption was that test automation was closely linked to traditional software, but that assumption was partly false. It could be because these studies are of two different types. Alégroth et al (2021) conducted a case study and according to (Patel & Davidson, 2011, p.3), the results might be hard to generalize depending on the population.

In the previous section (5.1 Test automation maintenance issues), Environment was considered an issue that could impact maintenance but from the results, it is also an enabler for minimizing maintenance (R2, R3, R6). A stable environment is important to create good preconditions for test automation (R5). The test environment should be in your control according to R5 and R6. You should be able to reset test data and database, and test data should be handled carefully and should be unique (R5, R6). You might be getting test data-related failures with poorly managed test data (R6).

One of the purposes of this study was to provide a set of guidelines to minimize maintenance in test automation. These guidelines can be found under the themes *Coding guidelines* and *Collaboration guidelines*. From the results, it is evident that the guidelines are useful and are recommended to follow.

Results show that the *Coding guidelines* and *Collaboration guidelines* are categorized as useful. In the interviews close to all respondent's considered all guidelines to be useful, with a few exceptions. For G1, short units of code, R4 remarked by stating that readability is also important. As for the guideline (G3) to write code once several respondents stated that it might be hard to live by (R1 and R7).

For G7, keep the codebase small, R4 elaborated that the guideline is kind of useful and wanted to highlight that there is a difference between tests and framework code.

The guideline findings are to some extent contradictive to what Alégroth et al (2021, p. 10) observed when evaluating the guidelines. The result of their paper suggests that the number of guidelines was too many. The developers had a hard time taking all guidelines into consideration. The reason for a different result could be related to the target group. Alégroth et al (2021, p. 10) are referring to developers, but this study's target group is test professionals with experience in test automation.

5.3 Limitations and future work

This study is limited to agile IT projects and from a test automation point of view. Perspectives of other roles such as developers or business analysts are not intended to be covered as part of this study. The study is also limited in size. To gain more knowledge and create a more detailed view of guidelines and measurements against increasing test automation maintenance suggested research could focus on environment and flaky tests. How can environment setup enable more stable tests. Other areas to expand on could be how test data and treat as feature code impact test automation maintenance.

6 Conclusion

This study has answered the two research questions related to test automation maintenance. Several issues affecting maintenance were identified, and the result also provided measurements to optimize maintenance. Speaking to seven practitioners with experience in test automation has provided good empirical findings and the outcome of this study is relevant to the specified target group. Interviewing the testing professionals, transcribing, and analyzing their answers resulted in four patterns: *Maintenance Issues*, *Minimize Maintenance*, *Coding guidelines,* and *Collaboration guidelines*. These patterns form the result of this research and summarize the findings.

- *Maintenance Issues* showed problem areas that impact test automation maintenance.
- *Minimize Maintenance* showed different approaches practitioners take to keep maintenance to a minimum.
- *Coding guidelines* provide practitioners with useful coding guidelines.
- *Collaboration guidelines* provide practitioners with useful collaboration guidelines.

References

Agile Alliance (n.d). The 12 Principles behind the Agile Manifesto https://www.agilealliance.org/agile101/12-principles-behind-the-agile-manifesto/ [2022-10-30]

Agile Alliance (n.d). User Stories https://www.agilealliance.org/glossary/user-stories [2023-01-16]

Agile Alliance (n.d). What is Scrum? https://www.agilealliance.org/glossary/scrum/ [2023-01-16]

Alégroth, E., Feldt, R., & Kolström, P. (2016). Maintenance of automated test suites in industry: An empirical study on Visual GUI Testing. *Information and Software Technology*, 73, 66–80. https://doi.org/10.1016/j.infsof.2016.01.012

Alegroth, E., Petersen, E., & Tinnerholm, J. (2021). A Failed attempt at creating Guidelines for Visual GUI Testing: An industrial case study. *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. https://doi.org/10.1109/ICST49551.2021.00046

Axelrod, A. (2018). *Complete guide to test automation: Techniques, practices, and patterns for building and maintaining effective software projects* (1st ed.). APress.

Baumgartner, M., Klonk, M., Mastnak, C., Pichler, H., Seidl, R., & Tanczos, S. (2021). Agile testing: The agile way to quality. Springer Nature. https://doi.org/10.1007/978-3-030-73209-7

Berner, S., Weber, R., & Keller, R. K. (2005). Observations and lessons learned from automated testing. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005*. https://doi.org/10.1109/ICSE.2005.1553603

Beynon-Davies, P. (2002). *Information Systems: An Introduction to Informatics in Organisations* (2002nd ed.). Palgrave Macmillan.

Borjesson, E., & Feldt, R. (2012). Automated system testing using visual GUI testing tools: A comparative study in industry. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. https://doi.org/ 10.1109/ICST.2012.115

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology, 3(2), 77–101*. https://doi.org/10.1191/1478088706qp063oa

Cadle, J., & Yeates, D. (2008). *Project Management for Information Systems* (5th ed.). Prentice Hall.

Christophe, L., Stevens, R., De Roover, C., & De Meuter, W. (2014). Prevalence and maintenance of automated functional tests for web applications. *2014 IEEE International Conference on Software Maintenance and Evolution*. https://doi.org/10.1109/ICSME.2014.36

Collins, E., Dias-Neto, A., & Lucena, V. F. de, Jr. (2012). Strategies for agile software testing automation: An industrial experience. *2012 IEEE 36th Annual Computer Software and Applications Conference Workshops*. https://doi.org/10.1109/COMPSACW.2012.84.

Crispin, L., & Gregory, J. (2008). *Agile testing: A practical guide for testers and agile teams*. Addison-Wesley Educational.

Digital.ai (2021) 15th Annual State of Agile Report. https://digital.ai/resource-center/analyst-reports/state-of-agile-report [2022-10-16]

Dustin, E., Garrett, T., & Gauf, B. (2009). *Implementing automated software testing: How to save time and lower costs while raising quality*. Addison-Wesley Educational.

Garousi, V., & Felderer, M. (2016). Developing, verifying, and maintaining high-quality automated test scripts. *IEEE Software*, 33(3), 68–75. https://doi.org/10.1109/ms.2016.30

Graham, D., Van Veenendaal, E., & Black, R. (2012). *Foundations of software testing ISTQB certification* (3rd ed.). Cengage Learning EMEA.

Gregory, J., & Crispin, L. (2014). *More agile testing: Learning journeys for the whole team*. Addison-Wesley Educational.

Luxoft-training (2019). [image] Available at: https://www.luxoft-training.com/news/the-test-automation-pyramid/ [2022-10-03].

Medium (2019). [image] Available at: https://medium.com/@tiff.sage/connecting-test-activities-with-software-development-activities-1e5bce61666c [2022-10-03].

Memon, A. M. (2002). GUI testing: pitfalls and process. *Computer*, 35(8), 87–88. https://doi.org/10.1109/mc.2002.1023795

Mersino, A. (2021 November 1). Why Agile is Better than Waterfall (Based on Standish Group Chaos Report 2020). *vitalitychicago.com*. https://vitalitychicago.com/blog/agile-projects-are-more-successful-traditional-projects/ [2022-10-16]

Rafi, D. M., Moses, K. R. K., Petersen, K., & Mantyla, 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)*. https://doi.org/10.1109/IWAST.2012.6228988

Schotanus, C. C. (2009). *TestFrame: An approach to structured testing* (2009th ed.). Springer. https://doi.org/10.1007/978-3-642-00822-1

Software Improvement Group (2022) https://www.softwareimprovementgroup.com/resources/ebook-building-maintainable-software/ [2022-11-14]

Telerik (n.d.). Automated Testing That Just Works. https://www.telerik.com/teststudio [2022-12-27]

Telerik Test Studio (n.d.). 10 tips on how to dramatically reduce test maintenance. Available: https://www.telerik.com/docs/default-source/Test-Studio/10-tips_maintainable_tests.pdf?sfvrsn=2 [2022-12-27]

Wiklund, K., Eldh, S., Sundmark, D., & Lundqvist, K. (2012). Technical Debt in Test Automation. 2012 *IEEE Fifth International Conference on Software Testing, Verification and Validation*. https://doi.org/10.1109/ICST.2012.192

Zhan, Z. (2021 July 20). Is Your Test Automation on Track? Maintenance is the key. *Medium.com*. https://medium.com/geekculture/is-your-test-automation-on-track-maintenance-is-the-key-267ddb94b525 [2023-10-16].

Zion Market Research. (2021). "*Global Test Automation Tracker to Witness Impressive Growth, Revenue to Surge to USD 6728.5 Million By 2028*": Zion Market Research. 2021-09-20 Available at: https://www.zionmarketresearch.com/report/test-automation-market [2022-10-05].

Appendix

Appendix A - Interview Guide

<u>Purpose of the study and consent.</u>

Introduction: 5 mins

1. What is your profession or role in your organization / in the project?
2. Are you comfortable with developing software and coding?

Testing in general: 5 – 10 mins

3. What is your opinion on the amount of time spent on testing in your project? How much time is spent testing?
    a. Too much? too little? Why not if no?
4. How many test cases are usually executed in different test levels and phases?
    a. How many are automated?
    b. UI / API?
    c. Regression?

Maintaining automated testing 5 – 10 mins

5. How much time is spent maintaining the automated tests compared to writing new tests?
6. How are you maintaining test automation?
7. Is maintenance of test automation problematic?
    a. Have you experienced problems with test automation?
    b. Can you elaborate, and give examples?
8. What issues can be related to maintenance of automated tests?

Guidelines: 30 mins

9. Guidelines (combine “coding” guidelines and “soft” guidelines in a list).
    a. Present the list to participants and tell them to read all of them.
    b. Which of the guidelines (check boxes?) do you consider useful/important for automation maintenance?
        i. Why/why not?
    c. What else do you think can help minimize automation maintenance?

<u>Comments and THANK YOU.</u>

Appendix B – Guidelines

Guideline	Description
G1 Description:	**Write short units of code** Avoid units with mixed responsibilities to maximize code reuse and understandability of the code.
G2 Description:	**Write simple units of code** Avoid branch conditions to maximize readability.
G3	**Write code once**
G4 Description:	**Keep unit interfaces small** Avoid more than 4 method parameters
G5 Description:	**Separate test cases accordingly to manual steps** Avoid tight coupling and “chaining test cases”
Condition: G6 Description:	**IF you are using a recording tool:** **Refer to images once** Avoid multiple instances of the same image to minimize repeated maintenance.
G7	**Keep The codebase as small as possible**
G8	**Use descriptive and uniform naming conventions**
G9 Description:	**Use synchronous checks** Avoid static synchronization (pause execution for X seconds) to reduce potential need for performance related maintenance.
G10	**Take a screenshot when tests fail**
Condition: G11 Description:	**IF you are using a recording tool:** **Take case to choose suitable imagery** Avoid development cost associated with image refinement by taking unique, information rich, screenshots of suitable size.
G12 Description:	**Write clean code** Avoid decreased productivity of team members by removing code smells, dead code, bad comments, commented code, long identifier names, magic constants, and badly handled exceptions.
G13	**Make maintenance visible to team – Failures**

Description:	Make the time spent debugging test failures and maintain automated tests visible by writing task cards for these activities or reflecting the extra time in story estimates.
G14 Description:	**Make blocking bugs visible to team** Making bugs and the resulting blocked stories visible reminds the team to fix defects quickly, focus on defect prevention, and avoid incurring technical debt in the form of defect queues.
G15 Description:	**Shared responsibility** The whole team needs to take responsibility for managing technical debt - both code and test.
G16	Identify the biggest source of technical debt and try experiments to reduce that debt.
G17	**Collaboration:** Have team members with different specialties work together and apply those different skill sets to reduce technical debt.
G18 Description:	**Adopt a sound test strategy** Test levels, test tools, what is the goal with automation?

Appendix C - Consent form

Working Title: *Software testing – Optimizing maintenance in test automation*

A. I consent to participate in the study.
B. I consent to the recording of this study.

I have been informed in writing about the study and agree to participate.

I am aware that my participation is completely voluntary and that I can cancel my participation in the study without giving any reason. By replying "Yes" when being asked about your participation in the interview, you agree to be part of the study and that you have understood the purpose of the research.

Contact information student:

Sebastian Öberg

sebaober101@student.kau.se

Contact information supervisor:

Ala Sarah Alaqra

Assistant professor in Information Systems

As.alaqra@kau.se

Appendix D - Information letter

Information on the thesis (working title: Software testing – Optimizing maintenance in test automation)

The purpose of this thesis is to conduct a qualitative study on how to enhance maintenance in the test automation of software. Data gathering will be done using online interviews which will be recorded using a web camera (only sound will be stored). The answers will be transcribed, analyzed, and anonymously presented in the final report.

Online interviews will be arranged using Zoom, logged in with a Karlstad University linked account. With that account, the video and sounds are encrypted and stored within the EU.

Gathered data will be kept until the thesis is completed, with a passing grade. Once the grade is registered and accessible in Ladok the data will be destroyed.

Karlstad University is responsible for personal data. According to the Personal Data Act (data protection regulation as of 25 May 2018), you have the right to access all information about you that is handled free of charge and, if necessary, have any errors corrected. You also have the right to request deletion, restriction, or to object to the processing of personal data, and there is the possibility of filing a complaint with the Data Inspectorate (Datainspektionen). The contact information for the data protection officer at Karlstad University is dpo@kau.se.

Contact information student:
Sebastian Öberg
sebaober101@student.kau.se

Contact information supervisor:
Ala Sarah Alaqra
Assistant professor in Information Systems
As.alaqra@kau.se