MASTER'S THESIS 2022

Return on investment and maintenance in a mobile test automation implementation

Martin H Tomičić

ISSN 1650-2884
LU-CS-EX: 2022-53

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2022-53

Return on investment and maintenance in a mobile test automation implementation

Möjlig vinst och kostnader för underhåll av automatiserad testning i en mobil miljö

Martin H Tomičić

Return on investment and maintenance in a mobile test automation implementation

Martin H Tomičić
dat15mto@student.lu.se

September 22, 2022

Master's thesis work carried out at Robert Bosch Sverige AB.

Supervisors: Masoumeh Taromirad, masoumeh.taromirad@cs.lth.se
Niklas Kenéz, niklas.kenez@se.bosch.com

Examiner: Per Runeson, per.runeson@cs.lth.se

Abstract

Bosch SensorTec develops microelectromechanical system (MEMS) sensors and related software for many uses. The thesis investigates and develops a test automation solution for a mobile application that interacts with one such sensor. It is focused on the current state of testing and concerns regarding the implementation of an appropriate solution, and, in particular, the return on investment and the possible maintenance of the solution. This was carried out using the design science paradigm. The results include a list of requirements for the solution itself, as well as a working implementation that is integrated into the current development environment. The maintenance is presented via measurements of executions on several versions of the application as well as how the solution will handle different types of test cases available. The return on investment is defined for the particular project and is shown to be feasible both through improvements in the execution time and the possibility of integration within the current development pipeline as well as enabling the testing team to focus on system testing and other tasks.

Keywords: mobile test automation, maintenance, return on investment

Contents

Chapter 1
Introduction

Software engineering is a discipline that involves collaborative work on large projects, where the customer or end-user expects a product with no or few bugs. However, the code must be tested to ensure that the product works as intended and follows the agreed-upon requirements.

Testing is a strenuous task that requires both time and resources from the project. The testing can be automated in some way to reduce this cost and the overall cost of development. However, this is not feasible for all projects or products, depending on the item to be tested and the current testing regime. An automated solution grants the additional boons of increased quality during development, and the developers' increased confidence in the code. Test automation has become a staple in software engineering, not the least because of continuous integration and continuous deployment. CI/CD is a paradigm that aims to automate as many things as possible and keep them in a readily available pipeline, to reduce development costs and time to release.

However, implementing test automation in a project is challenging in itself. This includes the possibility of a return on investment, as well as the maintenance of the automated solution between new releases or with added functionality.

1.1 Problem description

1.1.1 Case company

Bosch SensorTec in Lund is developing solutions around their MEMS (microelectromechanical) devices, including accelerometers, gyroscopes, magnetometers, and pressure sensors. Application areas include e-bikes and fitness tracking. Those solutions are complete systems with embedded software, intelligent algorithms, and visualization layers such as Android ap-

plications.

No automated testing is implemented in one of the currently active projects at Bosch SensorTec. The end product is a fitness tracking system consisting of an Android application paired with one or more MEM sensors connected via Bluetooth. The scope of their project has grown larger than expected because of customer needs and desires, and with it, the verification and testing have become cumbersome. As of today, they rely solely on manual testing. A future goal for Bosch is to connect the possible solution together with a Bluetooth injection, to reduce the amount of fitness exercises to be carried out by the testers as well as and overall testing time. This means emulating Bluetooth data instead of relying on the sensor itself for said data input. The Bluetooth injection is not finished and is out of scope for this thesis.

The primary scope of this thesis is to investigate, implement, and evaluate a test automation solution for the team at Bosch. This solution should provide the benefits of lessening the load on the testers and other team members while providing increased software reliability, and greater confidence in the code, among others. Furthermore, the solution needs to be developed in a way so that it can be connected to the ecosystem available at Bosch and, in the future, be part of a CI/CD pipeline.

1.1.2 Problem instance

The project within which the solution will be implemented is mature, and no test automation has been planned for it before this thesis. The application developed by the project is in a working state and has updates regularly. The ultimate goal is to develop a test automation solution for the project, that in particular, considers maintenance, return on investment, and scalability.

These concerns result in the following focus areas and research questions:

RQ1: What is the current state of testing at Bosch SensorTec, and what are their concerns regarding the testing and a possible upcoming solution?

RQ2: What are the main functionalities and requirements of a potential solution?

RQ3: How can return on investment be defined and achieved in a solution?

RQ4: How can maintenance be defined and addressed in a solution?

Chapter 2
Background

This chapter aims to provide a top-down summary of software testing and go into specifics for automated testing, mobile application testing, and GUI testing.

2.1 Software testing

Software testing is an essential part of software engineering. It ensures that the system works as intended, with a basis on the requirements. This is not only important to users as they interact with a well-made product, but it provides quality assurance to the development team. However, it is a costly endeavor, in time, effort, and turn financially.

There are several ways of conducting development in a software project. The two most common models are the so-called waterfall and the agile software development. Waterfall is a sequential model where each phase depends on the completion of the phase before it. Here testing is its own separate phase and is carried out well into the project, towards the end. [22] One of the main issues with the waterfall model is that it can lead to unpredictable software quality, due to late testing [27]. The agile methodology, which companies and the software engineering community have gravitated towards over the last 20 years, is an iterative scheme where testing is carried out several times during the project's lifetime. If strictly following the agile methodology found in the Agile manifesto by Beck et al. [2], testing is built into the code via test-driven development and as such, precedes any coding. The goal is that all written code should be testable.

There are two primary types of testing, black box, and white box. White box testing is when the person in charge of testing has access to the inner workings and complex knowledge of the SUT (System under test), thus giving the possibility of testing the internal structure and workings of the system. On the other hand, black-box testing is akin to what the user would experience and tests the system itself and its responses to input. The tester does not

have access to anything except the SUT through the medium of interaction.

Below are some of the more common methods and terms in software testing, and in this thesis.

Unit testing
In unit testing, you test each program unit (method, functions, classes, etc.) to ensure the correct behavior of even the most basic parts of the program. Following the Agile methodology, one should develop code using TDD - test-driven development. In TDD, you write the unit tests themselves before you write the code that the tests will examine.

Integration testing
In integration testing, all the units created are used to form larger, more cohesive program blocks. After they have been assembled, the bonds of these units are being tested.

System testing
After integration testing, when you have created a somewhat stable system, this needs to be tested as well. System testing contains many tasks and is rigorous so as to find as many bugs as possible. This is often carried out close to a deadline, as a whole, working system is required before system testing can commence.

Acceptance testing
Acceptance testing happens when the customer or other stakeholder receives the project and tests it to ensure it follows all the requirements set at the beginning of the endeavor. This measures the quality of the product instead of looking for defects, as this is done in system testing [21].

Smoke testing
Smoke testing is a quick system test over the main features to ensure they work as intended. Making sure the SUT does not "smoke" (term originally from electrical hardware) when turned on.

Exploratory testing
In exploratory testing, the tester is exploring the application and can often reach states not interacted with by tests by following routes that are not expected.

Regression testing
Regression testing does not create any new tests; it does, however, use the ones already implemented. These chosen test cases are run regressively to ensure that nothing breaks when a new feature or version of the software is released. This can be done manually, but as it requires a human tester, it will take both time and effort. This is tightly connected with test automation, as one can see the whole endeavor of automating tests as an extreme version of regression testing.

Test oracle
One of the most essential terms in software testing is the test oracle. An oracle is a set of conditions or expected results that define when a test case has passed or failed [6]. This can be done in many different ways, of which some are more theoretical than others. A common practice is to explicitly include a boolean expression in the code (a

so-called assert), relying on image recognition of the graphical user interface (GUI) to ensure that the correct path has been followed, or user errors and failures caused by the test for determining whether a test has passed or failed.

As Bosch is looking to reduce the time spent on testing the application developed by the team, automating a part of the testing in some way would be of great benefit, as resources spent on manual testing can then be used elsewhere in the team. This will in turn be a main part of the return on investment possible by an automated solution.

2.2 Test automation

Test automation is the most common and basic solution to reduce the load and, in turn, cost of testing software. It is also a cornerstone of CI/CD. Here the goal is to automate as much as possible, to be able to regularly integrate and release the product with new features. Automation is considered and applied on different testing levels and activities, depending on the goal.

Automated test case generation
There are tools available that help to generate cases to provide not only unit tests but also system tests. This has been a research area for the last few years and is now starting to get adopted in the industry. There are several categories, but the most prominent ones currently are the input generation as either random-based, systematic, or model-based [20]. While the research is ongoing and promising, the more significant problems at hand persist such as flaky tests and fragmentation. Record & Replay tools are readily available in the market and incorporate both the generation of test cases and the execution of them. The one downside of it is that while the code granted is automatically generated; it still requires manual input from a user to be able to produce said code. Research in this area wants to move away from the manual part to have it fully automated.

Automated test case execution
One can automate the execution of these test cases with a set of already specified test cases. This is usually done as part of a CI/CD pipeline, where as soon as a pull request is made, many if not all tests are running and made sure not to fail before the code is accepted into the branch or repository. This is the choice of automation most prevalent in the industry, and industry-wise, test automation equals test case execution in most cases [14].

Test oracle automation
Test oracles are, as mentioned, what decide when a test has passed or failed [6]. Deciding what a test oracle should requires knowledge of the expected state, which is hard to automate. The most promising ongoing research into automating the creation of test oracles is into machine learning algorithms or search-based models.

The academic research regarding test oracles is an especially active area, and while solutions regarding automated test oracles exist, no new paradigms have yet been set.

As it is very time-consuming to create or interpret test oracles, it is a crucial part of the ongoing research in software testing. Oracle definition is generally considered the most challenging phase of the testing process to be automated [7].

As test automation and its related areas have become a staple for most agile, forward-thinking companies in the last few years, there are now many self-proclaimed test-automation frameworks to choose from.

2.3 Automated mobile application testing

Testing mobile applications can be different compared to software developed for dedicated workstations. At the same time, such software is often extensive when it comes to uses, settings, and tools. Mobile applications are often more limited and usually consist of different views being presented and interacted with by the user. Being handheld, their computation power, memory, and overall specifications are often magnitudes less than dedicated devices. Because of the facts above and the explicit use case for the application and how the users will interact with the application, most testing overall, especially automated testing, is of the black box variant.

2.3.1 GUI testing

Graphical Unit Interface testing is a kind of testing where you interact and examine the SUT (system-under-test) or AUT (application-under-test) via the graphical user interface that the users themselves interact with to traverse the program. As the application is interacted with by emulating a user, this approach does not have access to the source code or other inner workings of the system.

There are different ways to achieve GUI testing, described below.

Scriptable tests
One can manually script tests in a programming language that interacts with the testing library used. However, this is a time-consuming task, as is writing any tests.

Record and replay
These tests are created using a tool that records the user's actions and translates the movements into code in the chosen scripting language. Often these tests are kept black box, but with the possibility of downloading the generated code. The execution of these test cases can then be automated themselves [8]. According to Leotta et al. [19], the implementation cost of these kinds of test are lower than for scripted tests but, in turn, have a higher maintenance cost.

There are also different kinds of testing through the GUI, whether the tests are scripted or recorded.

Element-based GUI testing
Element-based GUI testing, also called DOM-testing (Document Object Model), interacts with the GUI through references to the elements available in the GUI and visible to the user. These references are simply called an element of the application for Android applications. These elements have several attributes and actions available to the tool used to access them, but the naked eye cannot see them.

An issue with using element-based GUI testing is that the test can try to perform actions or a chain of actions before the element has been rendered, making the test fail. This is one large provider of flaky tests, as the render time can change depending on several conditions, such as the operating system and underlying hardware.

Visual GUI testing
Contrary to accessing the actual inner workings of the program or application, one can use visual GUI testing. This works by using images of SUT/AUT paired with image recognition. One can construct tests that use said recognition to find a specific element, e.g., a button, and then press it with regular input. As the only resources available for visual testing are the same as a human user would interact with, it is a true black-box testing endeavor. These tests are generally slower than element-based tests, more akin to a human tester, but require less implementation time [18]. Furthermore, as it is image-based and not based on inputting a program or application, one can readily switch contexts or programs to look for other images. Hence it is a very flexible approach and can have a chain of actions beyond a singular program or context.

2.4 Related work

There has been an increase of work in the area in the last couple of years, not in the least because CI/CD is quickly becoming a must in most software projects, as customers get accustomed to releases being frequent and consistent.

Haar & Michaëlsson [16] compare the approaches of visual GUI testing and element-based, focusing on maintenance of the solution. The solution is based on a problem instance containing a web-based application. They found that while visual GUI testing was faster to implement, the maintenance cost was worse than element-based, where the latter required 32% less time spent on maintenance for their thesis project. They also found that getting a return on the investment of either scheme was unfeasible if kept at one release per year. Automated testing has been shown to grant higher ROI and usefulness if implemented in an iterative scheme such as agile or SCRUM. With the case company of their thesis, CANEA, running manual tests every three months, a question could be whether or not automation of this testing provided the best solution; considering that most testers and other industry professionals accept that some testing must remain manual, as the human input and perception are invaluable.

Regarding automated test case generation, Brunetto et al. [8] have proposed and built a GUI testing solution, AutoBlackTest (ABT). ABT generates test cases by sampling possible GUI interaction sequences while relying on machine learning algorithms to steer the gener-

ation process. The tool is implemented in Java and uses the IBM Functional Tester, a record and replay tool to interact with the GUI. Some of their findings include that while automation is a key to reducing development costs and effort, it is not alone sufficient to address the needs of complex projects in larger organizations. This is where a experienced developer with domain knowledge of the area and product must step in to make sure that the product is working as intended manually. Other findings show that adding manually-specified test oracles can significantly increase the effectiveness of automated test cases while also improving the failure detection capability of the automated tests.

Coppola et al. [10] investigate scripted GUI testing of Android applications, focusing on diffusion, evolution, and fragility. They researched this using six open-source tools (Espresso, UIAutomator, Selendroid, Robotium, Roboelectric, Appium) for scripted GUI testing of mobile applications. From sniffing repositories for hooks or other code regarding the six mentioned tools, they could carry out their research on the resulting repositories. They found through their study that diffusion was low, less than 4,5% for all tools considered. Hence, at this point,< there was a low diffusion of test automation in Android application projects. They also found that 14,8% of all test classes and 3,6% of all test methods were modified after a new release. They conclude that the fragility changes require an effort comparable to implementing new tests for added features. This shows that fragility is quite a significant issue and could impact maintenance severely.

Linares-Vásquez et al. [20] want to introduce a new paradigm for mobile application testing, and that it should have the qualities of being Continuous, Evolutionary, and Large-scale. They propose to take mobile application testing as seriously as any testing and build a pipeline containing many virtual and real mobile devices. This vision is probably out of scope for most companies but still provides valuable insight. They ascertain that one of the primary reasons mobile application testing has been lacking is that the available automated testing tools have limitations, which has driven a preference for using manual testing in the industry. Fragmentation and test flakiness (test cases that sometimes pass, sometimes fail, with no changes to the code) are also mentioned as issues for all currently available automated approaches, regardless of whether they are GUI-bound. They also discuss test oracles, precisely the absence of mobile-specific test oracles. While there are different approaches towards this, such as manually coded oracles, exceptions-as-oracle, and GUI-state-as-oracle, they all fail somehow.

Tramontana et al. [28] conducted a systematic mapping study regarding automated functional testing. They have a plethora of research questions, which are not all relevant to this thesis. Amongst their findings are that the most popular approach is test case execution and oracle definition, with a close second testing case generation and execution. The lion's share (122 out of 131 papers) regarding test levels presents system testing approaches. They theorize that there are few documents regarding unit or integration testing because these approaches can easily be used for mobile applications, even if designed for desktop applications. As such, there is no need for a specific unit or integration testing for mobile applications.

Regarding return on investment in the endeavor of test automation, Amannejad et al. [5] carried out a search-based approach for cost-effective software test automation decision support, as they so eloquently phrase it. Several equations are set up to calculate ROI for the

three test activities: test design, execution, and evaluation. A search-based genetic algorithm is then used to find the most beneficial activity and its possible return on investment, based on 15 rounds. Findings include that test execution automation grants the highest return on investment by a wide margin, as its ROI after 10 runs land at a 675% return. The second is test case design at a 307% return. Finally, test evaluation automation had no feasible return before seven executions but started to be worthwhile, sitting at a 41% return. According to their equations and algorithm, it is more than a magnitude between implementation and evaluation. This paper is from 2014, and much has happened in the research regarding software testing since then. However, from the literature, it can be seen that automating test evaluation is still one of the main problems and no feasible solutions are available.

Coppola et al. [9] have interviewed members of the Italian software industry regarding automated mobile testing. They find that automated testing is not widely adopted, and manual testing is still the most popular choice in the industry. The interviewees cite test flakiness and evolution of the user interface as issues faced by developers and to a cost of 30% of all maintenance performed on test suites. They mention that one of the issues with test flakiness is the absence of reliable ways to fix and refactor test cases automatically. They say that this hinders mobile testing and is as severe as the fragmentation of devices, which might hold automated mobile testing back in favor of manual testing.

There is, of course, a downside to automated testing. It is costly to implement, and when the solution is available, it must be maintained and updated for upcoming versions. Most studies estimate that around 20-60% of the time spent on testing after it is automated is regarding maintenance and managing flaky tests [9]. This concerns mobile application testing even more because of the fragmentation amongst the devices as well as its rapidly evolving platforms.

There seems to be a disjunction between academia and the industry regarding software testing [14]. Academia promotes and is active research-wise in areas that would be seen as fringe from the industry's viewpoint. The industry focuses heavily on test case execution as a means in CI/CD and reduce overall cost and time spent in development [8]. Meanwhile, academia sees most industrial problems as lacking scientific novelty.

Chapter 3
Methodology

This thesis uses the design science paradigm [24]. The methodology consists of problem conceptualization, solution design, and validation. The chapter begins with the path to the research and its motivation and then follows further details on each of the aforementioned main activities explaining how they are carried out in the context of this thesis.

3.1 Research approach

The initial thesis suggestion, containing a problem description as well as the focus areas, was presented by Bosch. The method chosen for the thesis is critical, as it is vital to fit the technique after the task to carry out the best possible research given the objectives.

The thesis intends to research and implement a test automation solution for an application that has outgrown its original environment and now requires much more time and resources than initially thought. Amongst others, the maintenance cost and possible return on investment of the potential solutions should also be considered compared to the current manual testing. The proposed solution (if applicable) shall then be integrated into an expanding CI/CD environment but is considered out of scope for the thesis itself.

There are three major research paradigms, as stated by van Aken [31]. These are

- formal sciences
- explanatory sciences
- design sciences

Engineering science, as well as computer science, often fall under the design science paradigm [30]. In design science, the academic research objectives are more pragmatic, solution-oriented, and can be seen as a pursuit to improve human performance [31]. Research problems

are formulated and assessed by studying specific problem instances in practice, where the research activities consist of problem conceptualization, solution design, and validation [24].

Runeson et al. [24] convey how to use the design science paradigm as a frame for empirical software engineering research. Software, its tools as well as the organizations surrounding them are all human-made constructs, and as such, this suggests design science be a feasible research paradigm to adopt. They build further on the design science model by Engström et al., [13] as seen below in Figure 3.1. The model displays two dimensions of problem-solution and theory-practice through a design science lens; the arrows describe different processes of generating knowledge or information. It can also be seen that the overall process is iterative, and as such, one can revisit the problem instance (or any other state) after additional knowledge has been gained.

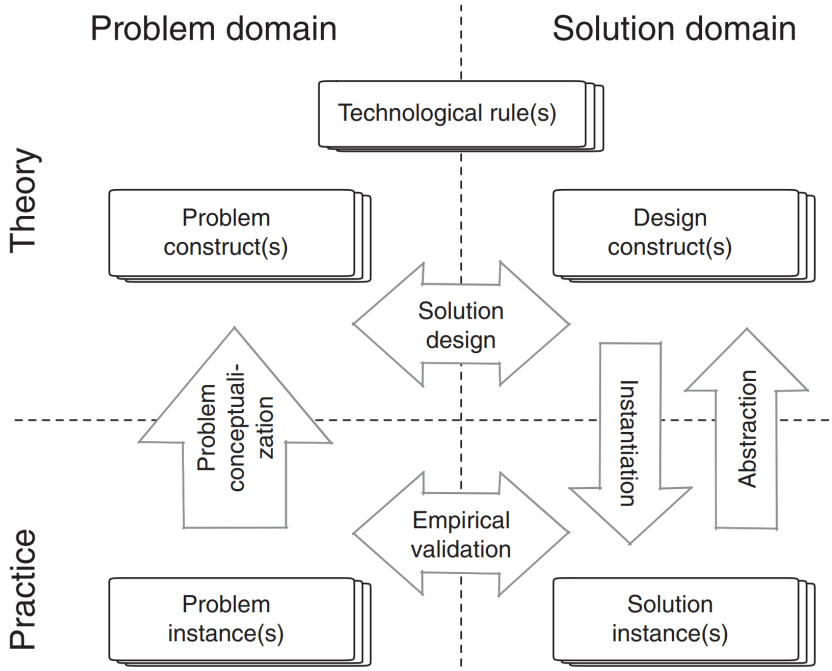

Figure 3.1: Design science model according to Engström et al.[13]

This thesis researches mobile test automation. Test automation is heavily researched, but test automation aimed toward handheld devices is often seen as not as necessary [20] as toward applications geared toward desktops or the like. The design science paradigm fits well, as research problems are formulated and assessed by studying specific problem instances in practice. The research activities consist of problem conceptualization, solution design, and validation [24]. Regarding the number of iterations, as the thesis is quite limited in time and scope, the most probable scenario is that only a single iteration of the cycle will be possible. After the solution design has been validated, the data and discoveries can be passed along to Bosch, who can modify the problem instance if needed or continue the work.

The research has been divided into specific actions, where some actions depend on others before being able to be carried out. The problem conceptualization will consist of a literature study, domain research containing an interview study with Bosch employees, and a case study

of the mobile application mentioned in the problem instance. The solution design is the activity phase and is carried out with a firm basis in the problem conceptualization. The actions considered for this phase are designing and implementing a proof of concept solution for the problem instance. The validation step will contain the assessment of the solution design compared to the current way of solving the problem instance with regard to the focus areas. These actions and their corresponding states in the design science paradigm are stated below in table 3.1.

Reference	Activity	Design science paradigm
A1.1	Literature study	Problem conceptualization
A1.2	Interview study	Problem conceptualization
A1.3	Platform exploration	Problem conceptualization
A2.1	Designing proof of concept	Solution design
A2.2	Implement proof of concept	Solution design
A3	Follow-up analysis	Validation

Table 3.1: Activities in the thesis

3.2 Problem conceptualization

This is where the research for the thesis begins and the first foray into the design science paradigm. It is the initial phase and one of the most crucial. Given problem instances from a sort of stakeholder, this is where the knowledge is built about the subject and the surrounding areas through different kinds of activities, which are used as the basis for the later solution design. These activities include, as mentioned above: a literature study, an interview study, and a platform exploration of the application itself.

The literature study is where the lion's share of the theoretical support for the research is gained. It is also vital, as it provides a better understanding of the area and all its intricacies. This, in turn, helps not only to construct valid, fruitful questions for the interviews, but to be able to interact with the interview subjects in a meaningful way, and lastly, to be able to analyze and extract research material procured from said interviews. As mentioned, this is a basis for exploring the given problem instance and constructing a solution design.

The second activity of the problem conceptualization was to investigate further and clarify the problem instance more practically, carried out through semi-structured interviews at the case company. It is also a way to gain theoretical support outside the literature study; by collecting knowledge and insights from the team.

Outside of the literature study and interviews, a platform exploration was carried out to gain technical and practical knowledge of the application in which the solution design will be applied. It results in increased confidence in the AUT (application under test), its functionality and qualities, and leads to a suitable solution design.

3.2.1 Literature study

A literature study is essential to a thesis to gain a deeper understanding of the subject at hand [17]. It must be carried out thoroughly to avoid missing any vital research on the subject. The

study should be well-structured and iterative on new findings found either in the problem instance, supplied by the case company, or in any other way that influences one's interpretation of the area.

The following criteria have been imposed to ensure that the papers collected during the literature study are of at least some quality research-wise.

- Literature shall be in English or Swedish
- Literature shall be accessible through LUBSearch or Google Scholar.
- Literature shall be peer-reviewed.
- Literature shall be available as full documents.

The literature study follows the pattern staked out by Höst et al. [17].

Search wide
To get a sense of the research available on the subject's background at the beginning of the thesis, including related areas, and search wide. This includes many keywords and phrases identified from the problem instance, as well as from both supervisors.

Selection
From the vast array of papers collected in the first step, discard papers depending on their quality or possible use for the thesis. The title and abstract were first reviewed for applicability. Then, if seemingly relevant, the central part of the paper was skimmed, and the results and discussion were analyzed. The trimmed set of papers is now a starting set [32], which can be used to deepen the research in the next step.

Precise search
The trimmed set of papers from the previous activity is used to find further papers. This is done using the technique called "snowballing" [32]. By checking the references of your already collected papers, it is possible to head even deeper into the area at hand and, in turn, be able to find more well-founded articles.

The literature study resulted in 44 initial papers collected, with a rigid selection process regarding relevancy. These papers were further analyzed, and using the snowball method described above [32], five articles were found to be of good quality and would contribute to the thesis' theoretical basis. Besides the initial literature search and triage of the found papers, finding literature is an iterative task. While writing the thesis, more papers and articles were found that were useful to the thesis and its research. Furthermore, one author was found to be active in the research on automated GUI testing, Emil Alégroth [4] at Blekinge Tekniska Högskola.

3.2.2 Domain research

After the literature study, having gained an extensive research basis and a greater understanding of the problem and surrounding areas, further research was carried out to deepen

the theoretical and practical knowledge of the problem instance. This is achieved by carrying out an interview study, a platform exploration of the application under investigation, which will play a substantial role in the solution design. The interviews are carried out with the developers and testers in the team at hand and with other parties within Bosch, where a test automation journey has already been accomplished.

Interview study

Interviews can either be open, structured, or semi-structured. As semi-structured, the interview follows a pre-defined pattern (closed), but new areas can be discovered and investigated during the interview (open) [29]. Semi-structured interviews were chosen as the medium, as the primary purpose of the interviews is to collect data regarding the problem instance and area but not to limit the possible exploration into neighboring areas that industry professionals might veer into and provide helpful insight.

Interviews were conducted with several parties, mainly within the team surrounding the application. As the team developing Activity Hub is relatively small, the number of interview participants is four. Contact with the possible participants was carried out via e-mail. The e-mails contained an explanation of the thesis, the area, the purpose of the interview, and if they were willing to participate.

The one-to-one, face-to-face interviews are planned according to the hourglass method [25]. First general, broad questions to start with, then narrow down and go deeper and return to more broad questions. Finally, a draft of questions is prepared and discussed with the academic supervisor, which results in sound questions, and in turn, the data gained would lead the thesis forward. The interviewees are presented in the below Table 3.2.

ID	Role	Team
P1	Android developer	Bosch SensorTec - Klio
P2	Android developer	Bosch SensorTec - Klio
P3	Test lead	Bosch SensorTec - Klio
P4	Principal software architect	Bosch eBike

Table 3.2: A presentation of the interview subject's roles and team-standing

Interviews were conducted according to the following schema:

1. Participant was welcomed and presented with a quick explanation of what the thesis entails

2. Participant was asked if they had any thoughts or questions regarding the background, the purpose of the interview, etc.

3. Verbal agreement obtained regarding the audio recording of the interview.

4. Interview performed following the hourglass principle explained above.

During the interviews, limited notes were taken on a laptop. While the answers to the stated questions gave great insight and data on the current project after each interview was completed, a private discussion regarding software engineering and testing granted valuable insight. This gave a great perception of the industry and how different companies have taken different approaches when it comes to testing. After all, participants had been interviewed; the emergent material was transcribed.

Platform exploration

A platform exploration will be carried out to get better acquainted with the application under study, as well as the sensor, that will be integral in the solution design. This is to get a sense of confidence in the application and, in turn, a higher sense of confidence in the solution design. The goal was to explore the platform, that is the application itself and its paired sensor, as well as the current way of testing.

Knowledge of the application and the problem instance itself is key to successfully implementing the best possible solution to the problem instance. As a result, part of the platform exploration followed the current manual testing regime. The current testing follows test flows set up to create as small leaps between different tests as possible and save time. These are available online when connected to a Bosch-approved VPN and show when and how a test has been completed. It gives not only a sense of the application itself but also regarding the testing and what could be improved. Following the already set-up test flows gave greater confidence in the application and served as exploring the application itself.

Outside of following the test flows, the application and its paired Bluetooth sensor were explored using the application as intended. This mainly consisted of using the application while exercising (which is the primary goal of the application) and using its contained "workouts" (different exercises). This was done after following the test flow set up to avoid getting startled by error messages or other communication and, in turn, missing some vital features. Besides these two structured testing, exploratory testing was also carried out. This goal was not to find bugs but to get even more comfortable with the application. This last kind of testing can be classified as "freestyle" testing, as mentioned by Ghazi et al.[15]; Only the test object is provided, and the tester is free to explore the system without any constraints.

3.3 Solution design

For the solution design, the work consists of two activities; *Designing proof of concept* and *Implementing proof of concept*.

3.3.1 Designing proof of concept

The first activity intends to use all the theoretical and practical knowledge from the problem conceptualization to motivate and design a proof of concept (PoC).

This is achieved using a research synthesis method, as explained by Cruzes et al. [11]. They propose three methods of synthesizing in software engineering research: thematic, cross-case, and narrative. The goal of the synthesis is to analyze the research findings and to be able to make a correct and supported decision on a test automation solution to implement. Narrative synthesis is chosen for this endeavor, as the input consists of one literature study and the domain research. A narrative synthesis is an approach that relies on words and text to condense and explain the findings of the synthesis [11]. This textual approach to the synthesis can be seen as "telling a story" of the conclusions of the studies [23].

3.3.2 Implementing proof of concept

After the PoC has been designed and communicated to Bosch, it is implemented as a solution design to the given problem instance. After the implementation is complete, comparisons are made between the proposed solution and the current manual way of testing, especially when it comes to the time taken for running tests together with their implementation time and what sort of maintenance is required on the test code between different versions of the application.

The implementation of the PoC is carried out on several versions behind the current, active version. Therefore, being implemented while having access to several "new" already existing versions will allow running the solution on several different versions of the application. This will, in turn, make it possible to collect data on the solution's maintenance effort and suggest what the possible maintenance can resemble going forward in the project. In addition, while implementing the PoC, data regarding the time spent on the implementation of the solution is recorded to give the possibility to calculate a return on investment.

3.4 Validation

The final activity of the design science paradigm and the research is validation. This is where the evaluation and analysis of the solution are carried out, based on the criteria either found in the problem instance or collected during the problem conceptualization. The main aim of the validation is to decide if the solution would be feasible for use in the practical context of the project, in the surrounding ecosystem, and if the solution meets the requirements, and in the process answer **RQ3** and **RQ4**.

As both return on investment and maintenance are fickle terms, they need to be defined for this thesis.

For **RQ3** and return on investment, it will be defined as

Improvements made to execution time compared to manual testing

Project-specific advantages from the solution

Resources released by implementing the solution

As for **RQ4**, maintenance will be defined as

Possible maintenance of the solution

Types of test cases and their maintenance

Possible preventative measures

Outside of the research questions, the scalability of the solution is also taken into account, as the one of the end goals for Bosch is to integrate the solution with Jenkins to form a pipeline together with a version control tool.

Chapter 4
Test automation at Bosch

This chapter presents the development team, its methods, and the current testing regime. The data is gained from the interviews as well as the domain exploration.

4.1 Developers and their practice

The following description is limited to members related to the application's software development.

Developers

P1: P1 is an Android developer working on test & demo applications that have been with Bosch for 12 months and have been in the project for just as long. He has an extensive career and has worked at many different large companies.

P2: P2 is an Android developer who has been with Bosch for one year and two months, working on the project full time. He also has an extensive career spanning decades.

The code is written in Kotlin, both developers use IDE, and some of the included tools such as refactoring and code suggestions. P1 mentioned that while their main occupation is being an Android developer, they also deal with various tasks in and around the team. There is no TDD in practice from any of them, as they see the practice as beneficial in general but at the same time be too time-consuming. P2 also mentioned that while the agile methodology works, he does not believe it is better than other methods. P2 said he has worked at a company where metrics, especially code coverage, were

virtually holy, and many resources were spent on achieving high numbers. He says that in the end, because of the focus on metrics, the team spent more time implementing tests than implementing actual new features. Thanks to the regression testing, both developers look forward to increased confidence in the code. At the moment, only smoke-testing is done before handing off the application to the testers. They state that they have no more significant concerns regarding the implementation of test automation. P2 mentions that for testing to be carried out successfully, there needs to be a developmental flow defined, which could have a negligible effect until it is fully set up.

Testers

P3: P3 is the test lead of the case project. Has worked at Bosch for six years and on the project for more than two years. He has so far been manually testing the code via test flows set up by himself.

There are not that many application releases at the moment, but it will steadily increase. About a year ago, there was a monthly release, if not more often. Regarding why he wants the team to implement test automation, he says it will help reach greater efficiency in the developer flow. It will also be able to find faults or errors as early as possible after they have been created.

Regarding why there is no test automation yet, and why it wasn't thought of in the beginning, he reports that the knowledge required to implement a solution was not present in the team at the time. As a result, the application development was ad hoc, and it was much later that the scope grew large enough to warrant such an implementation. Although there was also no time or resources for the implementation, the development was so quick that there was enough work just by running the manual testing and making sure the application worked as required.

He agrees that setting up a skeletonized version of a test automation framework at the beginning of a project would benefit the team. Even though test automation has quickly become somewhat of a must for software engineering, he believes that only a tiny part of testers/developers have the knowledge to set up a test automation solution.

Regarding the testing overall, he sees a need for cross-functional testing. While test automation provides excellent qualities and the ability to test the project regressively, he believes that manual testing, especially system testing, will still be needed. He says it is dangerous to think that the application works as intended if tests do not fail. A human would have noticed many things that an automated solution could miss.

P4: P4 has been a part of Bosch since the start of the Lund office, as did P3. His role is Principal Software Architect, and he has been involved in the transition from having outsourced manual testing teams to implementing test automation and all that it entails. For example, the change was made in a unit at Bosch named eBike, which develops and tests software for electric bikes. One of their main testing points is the touchscreen which is mounted on the handlebars where the user interacts with.

He explains that before implementing automation, Bosch had employed a manual testing team from India to carry out system testing. After that, developers were responsible for unit tests.

To make the change toward automated testing, members of the manual testing group were educated to become developers and could, in turn, program the coded tests that would be automated. This took time and resources, but in the end, it was a very successful endeavor, and there was a 60-70% reduction in time spent on testing. Especially the time to get feedback, which with the implemented solution was 20 minutes compared to what could be up to 3 weeks beforehand. This allowed the developers to quickly find any issues and correct them, which made the whole development flow quicker. Overall he mentions that test automation was very beneficial for the project and Bosch, even though the work to implement it was not easy and demanding in time and resources.

4.2 Development methodology

The team develops the application and its surrounding systems using SCRUM as a general method [26]. This consists of setting goals for so-called sprints, two-week periods where specific goals are set and followed up. A KANBAN [12] board is used to handle the assignment of tasks and act as a visual aid. Git and Bitbucket are used as version control. According to the interviews with P1 and P2, they use Android Studio or IntelliJ as integrated development environments (IDE). In addition, they use built-in tools, for example refactoring and code suggestions. The Figure 4.1 shows an overview of the development team, its infrastructure, and its flow.

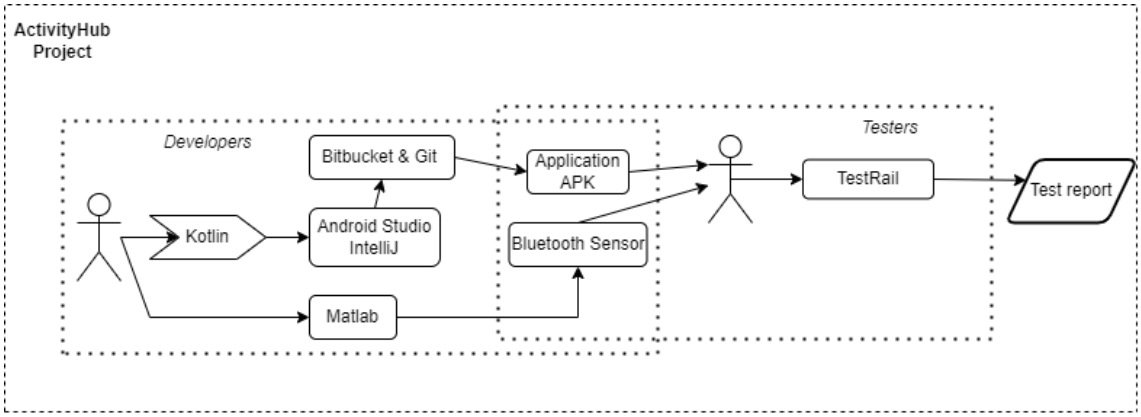

Figure 4.1: Overview of the development team

4.3 Testing regime

The testing done by the developers before they hand it over to the testing team is scant. They do not engage in unit testing but do a so-called smoke test before handing it over to the testing team. This can be seen as a quick system test of the most vital application parts to ensure

they work as intended.

When a new application version is ready for testing, it is delivered as an APK or compiled by the testers via Git. All of the tests are currently carried out manually. The test lead P3 has constructed test flows from the test cases available in TestRail, a test management platform. All test cases for a particular application part are passed following these flows. It is up to the tester following the flow to know the application to ascertain when a test has failed. When testing the Bluetooth sensor, the testing is more extensive as it requires the execution of physical exercises such as jumping jacks, and carried out most of the time in a gym. An example of a test flow is included in the appendix.

4.3.1 Platform exploration

To get acquainted with the testing regime currently in place and to give a higher confidence when using the application as well as general know-how in the handling of the application together with the sensor, a platform exploration consisting of the below activites was carried out.

- **Manual test executions**
  To get acquainted with the current testing procedure and what the automated solution will be compared to, manual test runs were run.

  The results are a greater understanding of the application and the sensor and how the testing procedure is executed. This is useful as the automated solution will be compared to manual test runs during the validation.

- **Actively using the application and sensor**
  Training sessions were carried out to experience the application and the sensor in the intended environment for which it is developed. This provided the sought qualities of understanding the application better and intrinsics that are hard to learn otherwise.

  Other things found were that the application crashed randomly at times, and the sensor lost connection. The latter was discussed with Bosch and determined to be because of interference from other signals in the environment since the prototype of the sensor is set in an unshielded case.

- **Exploratory testing**
  Carrying out exploratory testing improved the overall knowledge, but a bug was also found. The application has a maximum amount of patterns a user can input, and when that threshold is reached, the application should not be able to start a workout. However, when importing already defined patterns, it was possible to go over the limit of 25 total patterns. As a result, it made it possible to start a workout session where the sensor did not respond, but the application showed no error.

4.4 Current state of testing and concerns with the upcoming solution

The current state of testing at Bosch SensorTec in the team that develops the application is quite limited in scope. The limitation is explained by the application growing out of its original scope; as such, no test automation had been planned for initially. The developers who are interviewed and currently with the project are not the original developers of the application and have been working on it for a relatively short time. However, from the interviews, it is gained that they both do sparse unit and integration testing and mostly does smoke-testing before sending it over to the testers.

The testers use several tools to test the application. The testing carried out by the testers is so called system testing. Using the test management software TestRail, the testers have specified test cases from the requirements for the application. These are text-based and briefly explain the test case and the expected outcome. From knowledge of the application and TestRail, the test lead P3 has created manual test flows. Following these test flows, all the tests in a specific area (settings, etc.) are completed correctly. The tester following the flow is responsible for noting any faults or errors. These flows are followed after each new application version to ensure the expected functionality. If there are new features added, the flows are edited to include them.

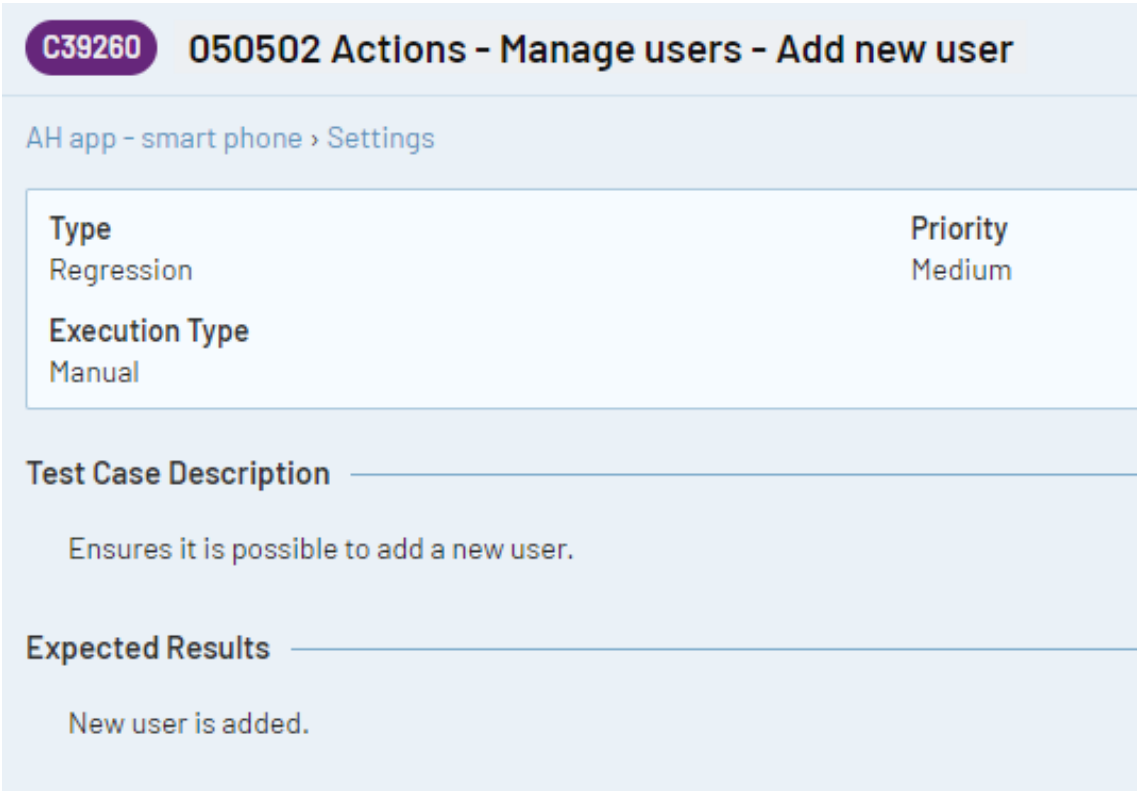

Figure 4.2: Example of a Test Rail item for a test case

The main concerns are regarding the solution itself, how much maintenance is required, and if there is a possibility of return on investment of the whole endeavor. These are stated in the problem description of the thesis and are the top focus areas. The developers in the team had few concerns regarding the solution, and are looking forward to the increased quality that the automated testing would provide. However, one developer, P2, noted that for testing to be carried out successfully, a flow for the testing has to be defined, which could affect the development cycle. Regarding concerns with the implementation of the solution, the test lead P3 believes that it is dangerous to only rely on test automation for the complete testing. He believes cross-functional testing is needed (including exploratory and manual testing) in addition to automated testing. He mentions that automated testing does its task well but is unnuanced in how it tests. He believes it is dangerous to think that an application works as intended simply because of automated tests passing.

Interviewee P4, from a team that has transitioned from manual testing to automated testing, noted that the transition was very costly, both monetary and time-wise, and he believed the implementation for this team would be no different.

Chapter 5
The solution

After consolidating the research and experience gained from the problem conceptualization, a decision was made on what kind of test automation would benefit Bosch and the project the most. This was achieved by a narrative synthesis of everything learned in the problem conceptualization.

The outcome of the synthesis, as well as otherwise gained from the problem instance and the interviews, is a set of requirements and required functionalities for the solution, which in turn answers **RQ2**.

1. A released, fully working Android Package (APK) of the application is available.
2. The solution must make use of Python as the primary scripting language.
3. The solution must be able to run on both Windows and Linux.
4. The solution must be able to deliver extensive reports in an easily interpretable fashion.
5. The solution must be able to make use of the hardware available at the Bosch office.
6. The solution must be able to use Jenkins as part of a CI/CD pipeline.
7. The solution must be able to be used by a team distributed worldwide.
8. The solution can be updated and maintained in-house.
9. The solution must make use of assert statements where needed.
10. The solution must provide automatically generated reports.
11. The solution must be implemented so as to reduce the possible maintenance as much as possible.

Given the synthesis, it was decided that a mix of automated test case generation and automated test case execution would be the shade of test automation that would benefit Bosch the most. This test automation solution can then be expanded should the scope of the application change.

Given that the primary purpose of the thesis is to help address a testing need that has outgrown the current capabilities and to release time and resources for other parts of the project, automating the existing manual testing would serve the project in the best way. Many tests are regarding the interaction between the independently developed application and the sensor, also produced by Bosch. As such, the testing should preferably mimic human behavior. A test framework that works in a black-box way by interacting with the GUI would be able to replicate this. As for element-based versus visual GUI testing, the study by Haar & Micaëlsson [16] has shown that the former has lower maintenance when it comes to resources. Therefore, an element-based framework will be selected as maintenance is essential to Bosch and the project.

Several test automation frameworks were evaluated based on the requirements to make this decision. As being able to handle the Android operating system as well as being able to script in Python are two of the core requirements, all of the following frameworks have said capabilities.

Appium

EyeAutomate

Robot Framework

Katalon

TestProject

All test frameworks investigated ,except for EyeAutomate, are built upon Appium, and are therefore element-based. Appium is an option, though it is more barebones than the other frameworks, which usually wrap up the actual functionality to provide a smoother experience. Seeing as Bosch has access to its hardware and wants to keep as much as possible in-house, software-as-a-service (SaaS) solutions are not warranted. Given the requirements and the synthesis of the problem conceptualization and an investigation into different kinds of test automation frameworks, TestProject was chosen.

TestProject [3] is a test automation framework developed by Tricentis. It is open-source, free of charge, and comes with the possibility to both record and replay tests, as well as automate the execution of tests. Reports of the executions are automatically generated and saved locally and available online. As the company behind TestProject is well-established in the industry, the chance that it will fall out of maintenance is low.

5.1 Proof of concept

With the test automation framework decided on, it was time to implement the proof of concept itself.

TestProject is built on top of Appium and Selenium. It executes its automation through the TestProject Agent, a small, local program that starts and manages its own Appium server. This communicates with the project's landing page at www.testproject.io, where executions in progress are shown, and reports are generated after the batch job is completed.

Set up
Creating an account is the first step toward using TestProject for automation. When an account has been completed, the TestProject Agent is downloaded and registered to the chosen account. Now all the necessary software is in place.

Infrastructure
One of the caveats of the problem instance presented by Bosch is that the code if any, should be written in Python. To make TestProject interpret Python, it requires the TestProject OpenSDK. To collect all the tests and execute them, PyTest will be used as well. PyTest is a small, open-source testing framework for Python code [1]. It extends the built-in testing capabilities of Python itself.

Test case generation
The generation follows the flowchart in Figure 5.1. First, choose a test case from the test management system TestRail. As mentioned before, the test cases are stated with (most times) a stated expected outcome and a brief description. Next, investigate whether it is possible to implement the test cases given the current architecture. For example, many test cases require interaction with the Bluetooth sensor. Unfortunately, at the time of the thesis, the Bluetooth data injection feature necessary for these test cases was ongoing, and as such, no such test cases could be implemented. Implement the test case, and depending on what the expected end value of the test is, add an assert statement. Finally, add the newly implemented test case to the automated solution, and run it to ensure it passes.

To create test cases, the choice was to use the built-in record and replay tool in TestProject. This records the user's actions, and you can manually change or add the steps in any way. Afterward, the test case is saved and can be executed. This can either be done through the TestProject website, or the recorded test cases can be downloaded as generated code for the chosen language and run the actual code.

To record the tests, the application itself is required. This can be done by emulating a phone with Android Studio or granting access to an actual mobile phone via USB. In the end, 20 test cases were produced. An example of the process of generating a test case using the Android Studio emulator and converting it to Python code is shown in Figure 5.2. An example of the automatically generated Python code for the same test case is available in the Appendix.

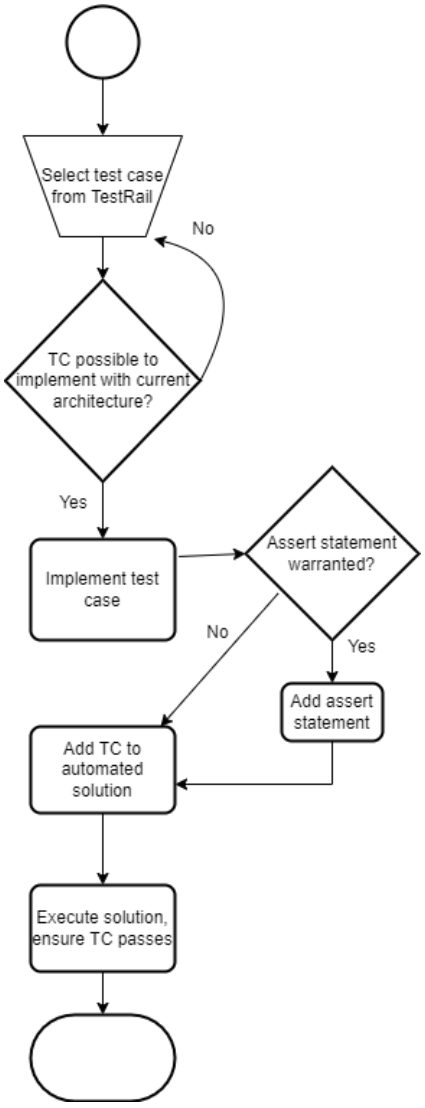

Figure 5.1: Test case generation process followed

.

Test case execution

For the collection and execute the test cases via TestProject, PyTest was used. Running tests with PyTest requires a configuration file where several fixtures and capabilities are stated. These include variables such as device, operating system, what application, and other application-specific items are required for Appium to be able to install the application. In addition, TestProject has extended the capabilities to include step settings for actions in the tests and other required variables.

PyTest also requires a particular structure, namely keeping the configuration file in a top-level category, the application below, and one category further down the tests. Running the command 'pytest' in a Python environment collects all the files whose names contain "test_" as the first characters and runs all methods named similarly inside them.

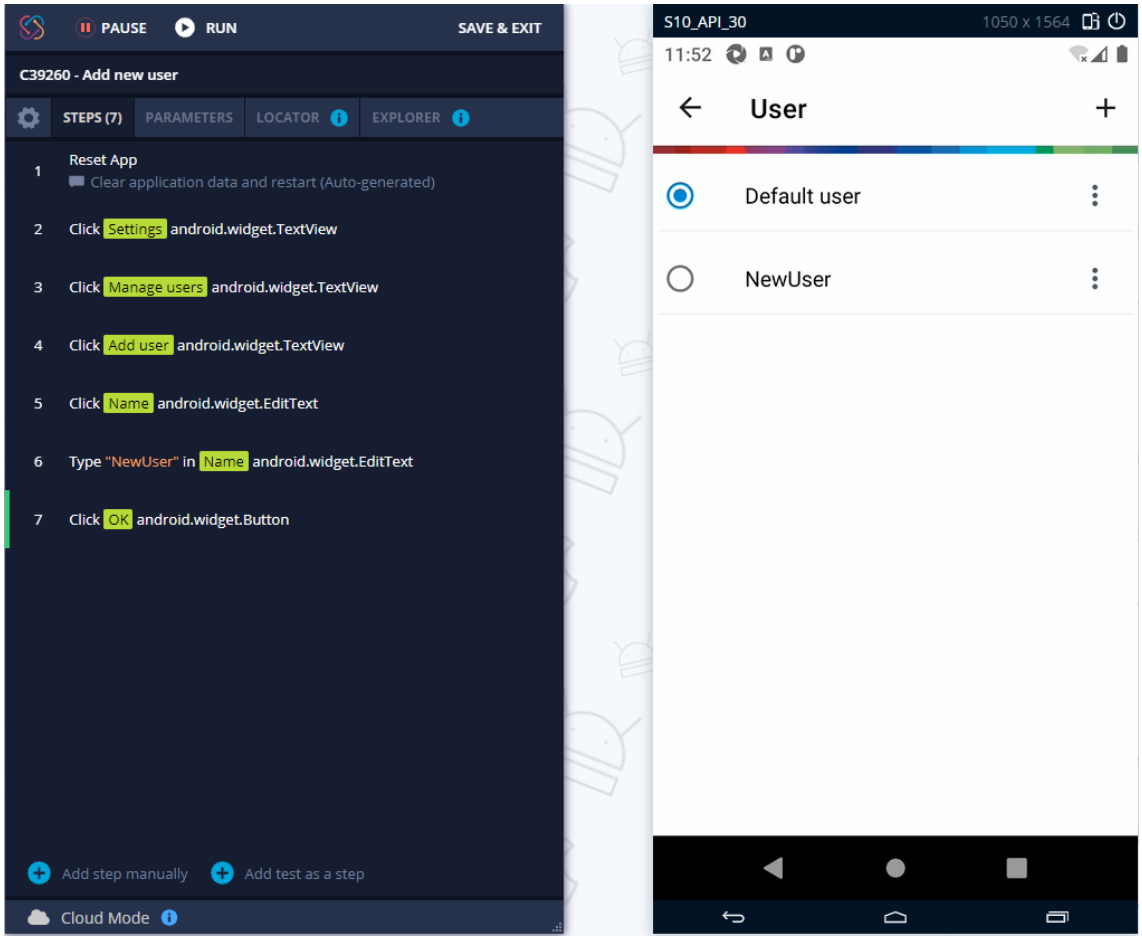

Figure 5.2: TestProject test case generation of the test C39260 - Add a new user.

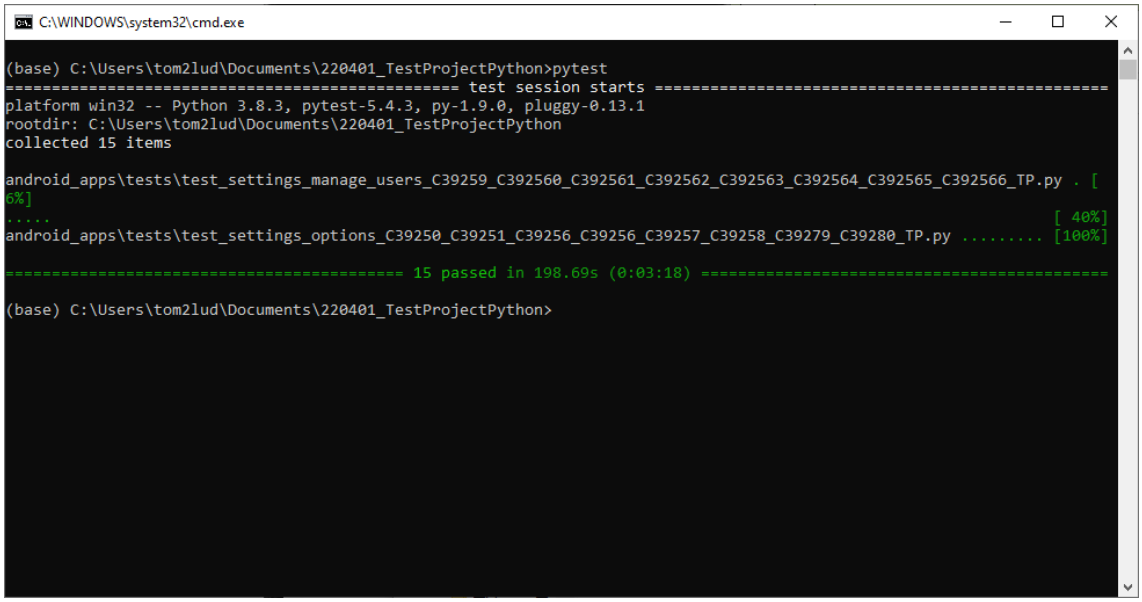

Figure 5.3: Executing the automated testing on an Android Studio emulator via PyTest

.

Reporting
As seen in the above Figure 5.3, running 'protest' collects methods (test cases) in the code contained in two different Python files (ending in .py). The reporting done by PyTest is scant if all pass. If an error is found, a large amount of information that is hard to investigate is produced. To improve upon this, TestProject has its own reporting. HTML overview reports are automatically generated and can be stored locally and on the project landing page. The overview is quick and well-structured. Meanwhile, a summary or a full report can be downloaded as a PDF. An example of the report presented on the landing page is provided in the appendix.

5.2 Validation

The implemented test automation is compared to the manual testing currently in place to validate the solution design. In addition, the return on investment for the whole endeavor and the possible cost of the code will also be evaluated and presented.

To gather the data, the following steps were iterated over.

Requires: Implemented solution design, Android emulator or mobile phone, several sequential versions of the application, and time tracker tool.

1. Execute the implemented solution on the selected version, using either an emulator or mobile phone. Record the time taken, bugs found, or other issues/comments.

2. Carry out manual testing of the same span as the test cases. Record time taken, bugs found, and other issues/comments.

3. Change the application version to the next one in chronological order.

4. Go to 1.

To investigate the execution time and maintenance of the application, five versions of the application were collected and used. The versions are labeled after their release date; **21-06-14**, **21-07-15**, **21-10-14**, **21-12-23**, **22-04-27**.

5.2.1 Return on investment

As mentioned earlier, return on investment is a fickle term and had to be defined for this thesis. The definition is defined as the amalgamation of:

Improvements made to execution time compared to manual testing

Project-specific advantages from the solution

Resources released by implementing the solution

Improvements made to execution time compared to manual testing

The execution time was calculated by taking the average of three runs of the solution on said version. The runs were executed on an emulator running Android 12, via Android Studio. The manual run time was estimated in the same way. The results are presented in the table 5.1 below. Each run contains the same 20 test cases. From the table, it is possible to ascertain that the latest version (**22-04-27**) does not have a run time, either automated or not. This is because a fault was found in the tests impeding most tests from running. Somehow the "Add new user" function had been decoupled from the GUI button, and as such, one could not add any additional users.

One of the features of using TestProject as a framework is that it allows users to change the step settings of the automation, without calling any methods in the test code itself. This is specified as milliseconds and is how long the Agent will wait or sleep until it attempts the next step of the test case. This ensures that the application has been appropriately rendered and the available element is searched for. By changing this from the default value of 500 ms to 150 ms, the proof of concept's run time was able to be halved.

The table 5.1 shows that after reducing the step settings, that the automated solution is on average 40% faster than the manual testing, for the proof of concept solution currently implemented. The variation across the automated executions were at most 5%.

Version	Execution time (seconds)	Manual execution time (seconds)
21-06-14	114	180
21-07-15	113	180
21-10-14	107	180
21-12-23	111	180
22-04-27	∞	∞

Table 5.1: Execution time of the different versions

Project-specific advantages

While reducing the actual time when executing something is substantial in itself, the solution brings many benefits that are not as easily made into statistics but can have a major effect on the team and its work going forward.

Significantly reduced time for feedback
The chief difference between the current way of manual testing and the implemented proof of concept is that the proof of concept can be run remotely and regressively. The plan going forward for Bosch is to integrate the automated testing with Jenkins, which will make it possible for the solution to be executed any time the application is updated.

When this is integrated, every time a change is made to the version control system, the solution will be executed. As such, feedback to the developers or other team members

will be significantly reduced compared to relying on that no issues have been introduced and waiting for the testers to manually examine the application. As seen with the version **22-04-07**, a fault was found in the application that had made it all the way to the internal release stage. If the solution had been implemented during its release, the team would have been notified immediately.

Objectiveness in testing
Manual testing is by its nature subjective. This is great for the most part, as a human tester can interpret feedback from the SUT in ways that an automated solution can not. However, for the project at hand, several test cases require objectiveness and are nigh impossible to carry out manually. These include tests that ensure that a changed setting has the correct qualitative output (such as changing a threshold of when the sensor should determine when a certain movement has been detected) and others that are reliant on being on carrying out repetitive tasks a vast amount of times, with no changes introduced in the movement.

As of now, these test cases have only been sparsely tested by the team. This is made easy by the solution as it (if not modified) does not have the possibility of being subjective or getting tired, and as such will provide ways of testing such cases to their full extent.

Increased confidence in the code
As mentioned above, the solution provides swift feedback to the team whenever something is edited in the version control tool. This also has the added functionality of denying the change if a test case has failed. This will grant the developers and the rest of the team increased confidence in the application and in turn the code, as only code that passes the tests are allowed to be updated.

Improved reporting
Before the solution, all testing was manual. This also meant manually taking notes and creating a report of the test run. Thanks to the automated solution, automatically generated reports are now made for each test run, and shows the exact steps taken in the program amongst other information.

Resources released by the implementation

Having the solution in place, and regularly testing the application releases resources for the team to otherwise further the project. For the developers, this can mean that unless there are major changes or functionality added, the smoke testing otherwise can be skipped and the application instead delivered straight to the testers. For the testers, this means that they do not have to test the basics of the application but can instead focus on exploratory testing as well as the experience of the actual end user. All in all, resources are released, and can be used to further the project in other ways.

5.2.2 Maintenance

Just as the term return on investment, maintenance can mean several different things. It was defined earlier as a combination of

Possible maintenance of the solution

Types of test cases and their maintenance

Possible preventative measures

Possible maintenance for the implemented solution

Using the five versions mentioned before, the solution was implemented for the oldest version, **21-06-14**. After that, the version was iterated forward. The time spent on getting the solution to work again on the new version of the application was recorded, and then the iteration continued. This is presented below Table 5.2. The need for maintenance was not that prominent and regarded issues with differences between versions of the application, not the coded tests breaking. Nonetheless, getting the solution to run again is considered maintenance.

The maintenance required between versions **21-06-14** and **21-07-15** consisted of the connection the Agent having been interrupted and the Agent having to be restarted. After that, the execution ran flawlessly.

The most challenging maintenance of 45 minutes was regarding so-called "application packages," which is the specification regarding where the application can be started and found while on the mobile device. Between these two versions, one is a so-called "debug"-version, and the other is an updated release for customers. As such, the application package string was different for the two versions, and in turn, the automation failed. The issue at hand was quite challenging as the error reports did not indicate what had happened; the only insight into the issue was that the application itself was not being started. It took 45 minutes to find the issue and rectify it until the solution could be executed again.

The last "maintenance," going from version **21-11-23** to **22-04-27**, consisted of 60% of the tests failing. This was astonishing as it seemed nigh impossible for a single update to make such a large portion of the tests fail. Indeed there was something wrong, as the application's core functionality had been broken. As such, it was impossible to estimate the actual maintenance until the application released a new version where the functionality was restored.

Version	Maintenance required?	Time spent on maintenance (minutes)
21-06-14		
21-07-15	•	5
21-10-14	•	45
21-11-23		
22-04-27	•	∞

Table 5.2: Different versions of the application as well as possible maintenance

Types of test cases and their maintenance

Below is a generalization of the currently available test cases, and what their possible maintenance could be, going forward in the project.

Comparing text content
Many test cases require the comparison of the contents of a text box to ensure e.g. legal compliance through licenses or to make sure that a name change of a user has been carried out. Possible maintenance for these types of test cases is when the "correct" text has been changed or is unavailable, and as such much be manually edited.

Asserting a state
Asserting that the application has reached or changed a state is a requirement for many test cases. Possible maintenance is when the state is not correct anymore, or not reachable through the application GUI.

Interacting with Android OS GUI
Several test cases are regarding the input and output into the application, and as such will interact with the GUI of the mobile phone itself. Possible maintenance here could be if the phones are switched or otherwise given another Android version, which would require manual editing to ensure that the scripts are able to find their way.

Lengthy test cases
Some test cases require getting a large amount of input from the sensor (e.g. testing that an exercise can be done 100 times), and as such could be prone to timeout. TestProject has a method for a so-called adaptive wait, where it is possible to also state the maximum waiting time for the test to fail, otherwise, it will wait until the sought-after state is detected. Possible maintenance includes having to change the max timeout range for the adaptive wait, depending on the number of repetitions that need to be recorded.

However, as seen with the investigation into maintenance for the AUT, the maintenance had nothing to do with the actual test cases, but with the testing infrastructure. This that issues with the test cases themselves is not the only source of maintenance.

The most significant issue for the solution is if there were to be a large update to the GUI where elements and sections of the application are shuffled around. As the solution follows a flow throughout the application to in the end ensure the success of the test case, this would break most tests and require a great deal of interaction from the team.

Preventative measures

Many issues with element-based automated testing are based on not being able to locate an element. This is often because that certain element has not been given its own ID; instead, it relies on the so-called 'XPath' to find it. An XPath is a string that provides the address to the element, oftentimes by only a position in a vector or other data structure. However if there has been even a slight change in the application, the element might have moved, and the test will fail. Therefore it is a good practice to provide elements with an ID, which increases the

robustness of the application and in turn the tests.

As mentioned before, the majority of the maintenance for the solution was not because of the tests themselves failing, but because of the infrastructure around it. An example of this would be to merge versions being tested to always be in the "debug" package, so as to not have to edit the application packages between versions.

However almost all software is prone to change, and as such maintenance of the automated solution will always be required.

Chapter 6
Discussion

The discussion in this chapter is divided into different parts. First, there is a discussion regarding the given results, then the limitations of the implementation.

6.1 Results discussion

Regarding RQ1, the current state of testing in the project is strictly manual and relatively sparse. As a result, the developers have few concerns, but the testers realize that the implementation is a substantial undertaking that will require a lot of resources and care. Testers on the other hand had more concerns, and one of the main ones was that it is dangerous to rely solely on the automated solution for testing the application. They believe that manual testing will always be required to make sure the SUT works as intended, but the process can be greatly helped by an automated solution.

As for RQ2, the list of requirements produced for the solution is specific to Bosch, and the singular application is currently being developed. The use case for the application is relatively narrow, and therefore the solution is tailored to its requirements, especially the possibility of having an objective "tester" carrying out executions. An element-based approach was chosen as it has been shown by Haar & Michaëlsson [16] to have less maintenance than a visual approach. The market for visual GUI testing frameworks is also scarce. Since the product relies on the interaction between an independently developed application and a sensor that is reporting gyroscopic data, among other things, some test cases require mimicry of human interaction and movements. A test framework such as TestProject, which interacts and carries out testing through the GUI, means that implementing such cases is possible for Bosch.

Concerning RQ3, the defined return on investment does not only include the aspect of time. However, implementing the rest of the test cases could negatively affect the solution's execution time compared to manual testing, but doing so would still provide an objectiveness

that is sorely needed. Nevertheless, as shown in table 6.2, the run time as it stands now is substantially shorter than manually executing the tests, which would most likely still hold with more test cases. The solution also releases resources from both the developers and the testers, which is the most wanted quality of a test automation solution. The execution time of the solution itself is also able to reduced further. As of now, all test cases have an explicit reset of the application in the beginning of the case, as wanted by Bosch. This can be removed, and the execution order of the tests themselves can be streamlined to further improve the execution time.

As for RQ4, maintenance is defined as the time spent on getting the solution to run again, how the solution handles different types of test cases, their possible maintenance, and what preventative measures can be taken. As the investigation into the current maintenance of the solution did not provide any larger issues it is hard to predict what the severity of the maintenance can be going forward. It is however possible for Bosch to further streamline the testing by making sure all future versions of the application have the same attributes, which removes the need to change variables in the solution between versions. However, one cannot run from maintenance. In the end, for most test cases, correcting them requires manual interaction.

Given the research questions and the resulting solution, my recommendation for Bosch is to keep implementing test cases and updating the solution, as it provides many benefits for the team and for the application itself.

6.2 Limitations

The work in this thesis is limited to the application developed by Bosch and its associated sensor. Currently, it is only limited to the Android operating system, and many mobile phones are removed from the focus. Bosch SensorTec and Bosch, in general, implement extensive security practices. These are great for the company's protection but also affect the choice of framework to implement. Integrating a framework that requires many different dependencies, accesses, and installations into its existing ecosystem is much harder to integrate and might not even work.

The test cases themselves were chosen from the test cases provided through TestRail. TestRail is a test management platform where it can specify test cases and their ID:s, expected outcomes, and other qualities. Unfortunately, many of the test cases of the ones available to implement required the use of a sensor connected via Bluetooth. This would require either injection of Bluetooth data or some kind of manual robotic arm that would carry out the movements required. Unfortunately, this was not available at the time of the thesis, and as such, these test cases are not included in the thesis.

Some of these tests would be lengthy time-wise. It is possible these tests would require additions to the code such as a wait or sleep, however, this should not increase the complexity of the code in a substantial way and should in turn not make the maintenance more extensive for these kinds of tests.

Many of the tests rely on a Bluetooth-connected sensor that inputs data to the application. A way to inject Bluetooth data is being developed at Bosch, but at the moment the use of the sensors could lead to failing tests, as it is prone to rebooting and crashing. If this happens during the automated execution, then many tests would fail and the execution would need to be restarted.

Regarding the validation and in turn data collection, every execution was carried out solely by myself. As such, it is possible that the data is influenced by the knowledge gained from carrying out several executions of the manual testing sequentially. All executions were carried out on a singular emulator. In the future, should the project grow even larger, there might be several dedicated devices running the tests simultaneously, which in turn could possibly influence the effectiveness of the solution.

Chapter 7
Conclusion

In this thesis, the possibilities and qualities of test automation for mobile phones have been explored, implemented, and evaluated. The focus areas have been regarding return on investment for implementing as well as maintaining the solution, and what that maintenance could be defined as. The results include a set of requirements for the solution (**RQ2**) gained both through interviews regarding concerns with the solution within the team (**RQ1**) and the literature study, as well as a working proof of concept where both the test automation framework and test cases have been implemented. A connection to Jenkins and the emerging pipeline at Bosch SensorTec was considered out of scope. Still, there was enough time for the implementation that, in the end, was achieved together with a colleague at Bosch. The proof of concept is now being used in production, and more test cases are being developed.

Return on investment is defined and is able to both be able to save time and free up resources, among other benefits for the team. In general, automated tests are 40% faster than the current manual testing. On the other hand, increased confidence in the code thanks to the tests pointing out bugs and improved release quality are boons of the implementation that are not of time-wise and, in turn, monetary value. (**RQ3**)

Regarding maintenance of the implemented solution, it does not seem to be that large of an issue with the test cases currently implemented. Five versions of the application were tested, and the maintenance required between them was investigated. Analysis of the types of test cases available and their possible maintenance going forward is also presented. However, the software is almost always changing in one way or another, and applying maintenance to the code will always be required. Preventative measures that are possible to use to reduce maintenance are also presented. (**RQ4**)

Connecting the solution to the existing technology ecosystem at Bosch (Jenkins etc.) was not part of the scope. Still, as there was some space in the time staked out for the implementation, this was also achieved together with a colleague at Bosch. As such, the solution

is actively used in production and is being run regressively on any updates to the repository containing the application.

7.1 Future work

This master thesis has been researching mobile testing, especially the automated kind. As it stands now, almost all the approaches to this are somewhat code-related, using element-based objects to interact with the application. Further research could head into visual GUI testing. Unfortunately, it has not yet made (seemingly) any impact in the industry. Still, it is promising as it does not require the person constructing the tests to possess any larger technical or programming knowledge. At the same time, the implementation time is shorter than the DOM-based equivalent.

Many of the test frameworks on the market, such as TestProject, are advertising themselves as containing artificial intelligence and machine learning to make the testing faster, smoother, and (according to them) reduce maintenance on the code. However, as seen in the research, one of the most significant problems with test automation is still maintenance. Some future research into these could be comparing the frameworks themselves and how much of a difference there is in maintenance when the AI/ML is disabled compared to being active. Does it affect the actual outcome, or if it is primarily a marketing gimmick using buzzwords?

References

[1] pytest. https://docs.pytest.org/en/7.1.x/index.html Collected: 2022-08-09.

[2] https://agilemanifesto.org. Collected 2022-08-09.

[3] www.testproject.io. Collected 2022-08-10.

[4] Alégroth, Emil. https://www.bth.se/staff/emil-alegroth-eal/ Accessed: 2022-08-09.

[5] Amannejad, Y., Garousi, V., Irving, R., Sahaf, Z. A search-based approach for cost-effective software test automation decision support and an industrial case study. *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pages 302–311, 2014.

[6] Ammann, P., Offutt, J. *Introduction to Software Testing, second edition.* Cambridge University Press, 2017.

[7] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S. The oracle problem in software testing: a survey. *IEEE Transactions on Software Engineering*, 2015. https://doi.org/10.1109/TSE.2014.2372785.

[8] Brunetto, M., Denaro, G., Mariani, L., Pezzé, M. On introducing automatic test case generation in practice: A success story and lessons learned. 2021. http://arxiv.org/abs/2103.00465.

[9] Coppola, R., Ardito, L., Morisio, M., Torchiano, M. Mobile testing: New challenges and perceived difficulties from developers of the italian industry. 2020.

[10] Coppola, R., Miorisio, M., Torchiano, M. Scripted gui testing of android apps: A study on diffusion, evolution and fragility. *PROMISE'17 Conference*, 2017.

[11] Cruzes, D.S, Dybå, T., Runeson, P., Höst. M. Case studies synthesis: a thematic, cross-case, and narrative synthesis worked example. *Empirical Software Engineering, Vol 20 issue 6*, pages 1634–1665, 2015.

[12] Random House Dictionary. Kanban. 2012. Accessed: 2022-08-14.

[13] Engström, E., Storey, M-A., Runeson, P., Höst, M., Baldassarre, M T. How software engineering research aligns with design science: a review. *Empirical Software Engineering (2020)*, 2020. https://link-springer-com.ludwig.lub.lu.se/content/pdf/10.1007/s10664-020-09818-7.pdf.

[14] Garousi, V., Felderer, M. Worlds apart - industrial and academic focus areas in software testing. *IEEE Software, Vol 34, Issue 5*, pages 38–45, 2017.

[15] Ghazo, A.N., Petersen, K., Bjarnason, E., Runeson, P. Exploratory testing: One size doesn't fit all. 2017. https://arxiv.org/ftp/arxiv/papers/1704/1704.00537.pdf.

[16] Haar, P., Michaëlsson, D. Automated gui testing: A comparison study with a maintenance focus. 2018. https://odr.chalmers.se/bitstream/20.500.12380/255666/1/255666.pdf.

[17] Höst, M., Regnell, B., Runeson, P. *Att genomföra examensarbete*. Studentlitteratur AB, 2006.

[18] Clerissi D. Ricca F. Tonella P. Leotta, M. Visual vs. dom-based web locators: An empirical study. *International Conference on Web Engineering*, pages 322–340, 2014.

[19] Leotta, M., Clerissi, D., Ricca, F., Tonella, P. Capture- replay vs. programmable web testing: An empirical assessment during test case evolution. 2013.

[20] Linares-Vásquez, M., Moran, K., Poshyvanyk, D. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. *IEEE International Conference on Software Maintenance and Evolution*, pages 399–410, 2017.

[21] Naik, K., Tripathy, P. *Software Testing and Quality Assurance - Theory and Practice*. John Wiley Sons, Inc., Hoboken, New Jersey, 2008.

[22] Petersen, K., Wohlin, C., Baca, D. The waterfall model in large-scale development. 2009. http://bth.diva-portal.org/smash/get/diva2:835760/FULLTEXT01.pdf.

[23] Sowden A. Petticrew M. Britten N. Arai L. Roen K. Rodgers M. Popay J., Roberts H. Developing methods for the narrative synthesis of quantitative and qualitative data in systematic reviews of effects. 2006.

[24] Runeson, P., Engström, E. and Storey, M-A. The design science paradigm as a frame for empirical software engineering. *Contemporary Empirical Methods in Software Engineering*, 2020.

[25] Runeson, P., Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 2009. 10.1007/s10664-008-9102-8.

[26] Sutherland J Schwaber, K. The scrum guide: The definitive guide to scrum: The rules of the game. 2017. https://scrumguides.org/docs/scrumguide/v2020/2020-Scrum-Guide-US.pdf Accessed: 2022-08-14.

[27] Sommerville, I. *Software Engineering, 7th edition.* Pearson Education Ltd., London, 2004.

[28] Tramontana, P., Amalfitano, D., Amatucci, N. Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal, 15th of March*, pages 149–201, 2019. https://doi.org/10.1007/s11219-018-9418-6.

[29] Turner III, D.W. Qualitative interview design: A practical guide for novice investigators. *The Qualitative Report*, 2010. https://nsuworks.nova.edu/cgi/viewcontent.cgi?article=1178&context=tqr.

[30] Vaishnavi, V., Kuechler, B. Design science research in information systems. 2004. http://desrist.org/design-research-in-information-systems/.

[31] van Aken, J.E. Management research as a design science: Articulating the research products of mode 2 knowledge production in management. *British Journal of Management*, 2005.

[32] Wohlin, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. 2014. https://www.wohlin.eu/ease14.pdf.

Appendices

```
"""
This pytest test was automatically generated by TestProject
    Project: ThesisTest
    Package: TestProject.Generated.Tests.ThesisTest
    Test: C39260 - Add new user
    Generated by: Martin H Tomicic (tomicic @hotmail.com)
    Generated on 05/24/2022, 09:54:01
"""


@pytest.fixture()
def driver():
    capabilities = {
        "platformName": "Android",
        "udid": "emulator-5554",
    }
    driver = webdriver.Remote(token="gOWzFSQXRXfxkYs1eKRH8a9rxkAif-p34OG_THWZTTQ",
                              project_name="ThesisTest",
                              job_name="C39260 - Add new user",
                              desired_capabilities=capabilities)
    step_settings = StepSettings(timeout=15000,
                                 sleep_time=500,
                                 sleep_timing_type=SleepTimingType.Before)
    with DriverStepSettings(driver, step_settings):
        yield driver
    driver.quit()


@report_assertion_errors
def test_main(driver):

    # 1. Reset App
    # Clear application data and restart (Auto-generated)
    driver.reset()

    # 2. Click 'Settings'
    settings = driver.find_element(By.ID,
                                   "com.bosch.bst.aha.debug:id/action_settings")
    settings.click()

    # 3. Click 'Manage users'
    manage_users = driver.find_element(By.ID,
                                       "com.bosch.bst.aha.debug:id/settingsSelectActiveUser")
    manage_users.click()

    # 4. Click 'Add user'
    add_user = driver.find_element(By.ID,
                                   "com.bosch.bst.aha.debug:id/action_add_user")
    add_user.click()

    # 5. Click 'Name'
    name = driver.find_element(By.ID,
                               "com.bosch.bst.aha.debug:id/username")
    name.click()

    # 6. Type 'NewUser' in 'Name'
    name = driver.find_element(By.ID,
                               "com.bosch.bst.aha.debug:id/username")
    name.send_keys("NewUser")

    # 7. Click 'OK'
    ok = driver.find_element(By.ID,
                             "android:id/button1")
    ok.click()
```

Figure 1: Generated Python code from the test case in Figure 4.1

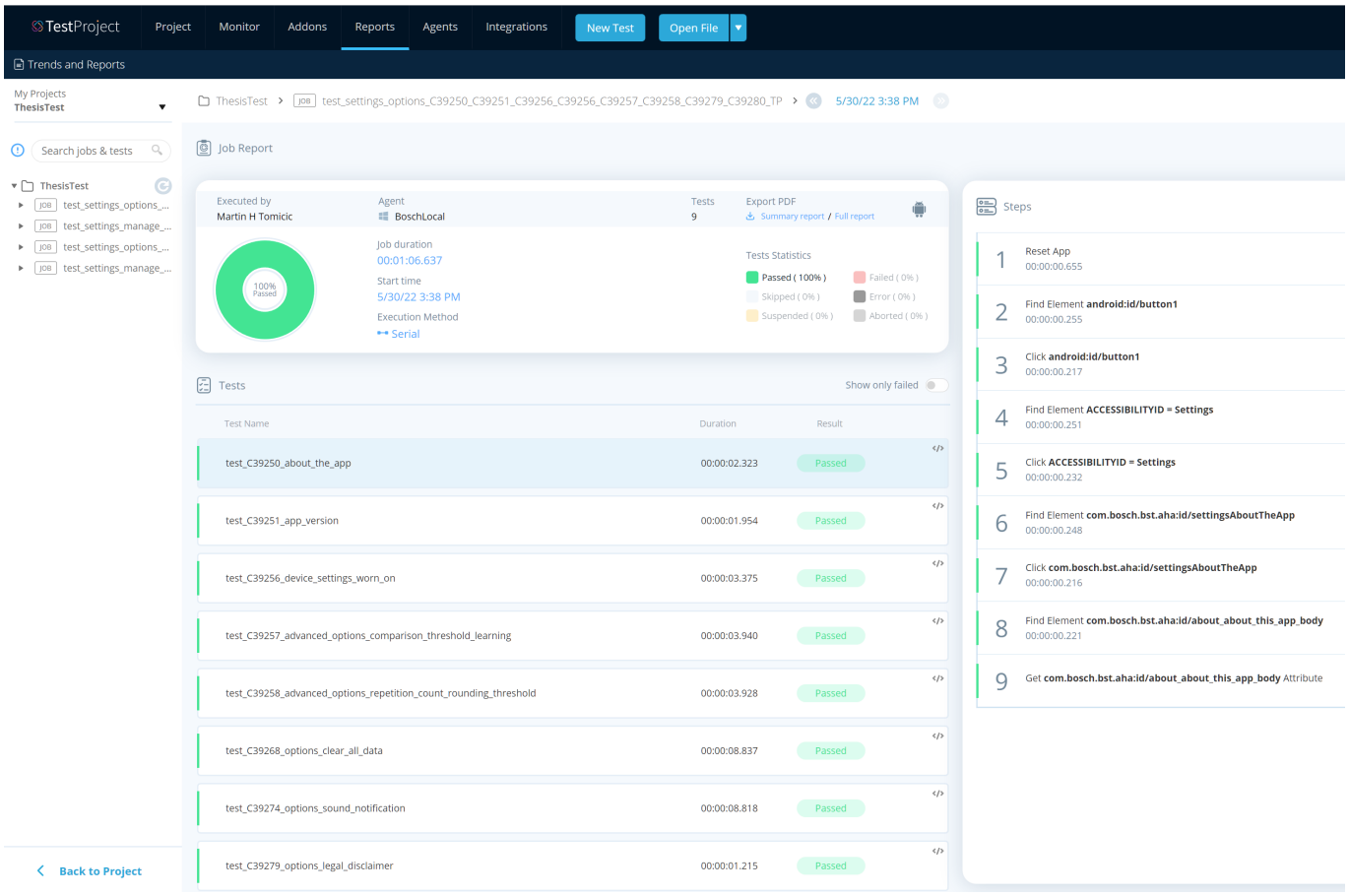

Figure 2: Example of a report of test execution using TestProject

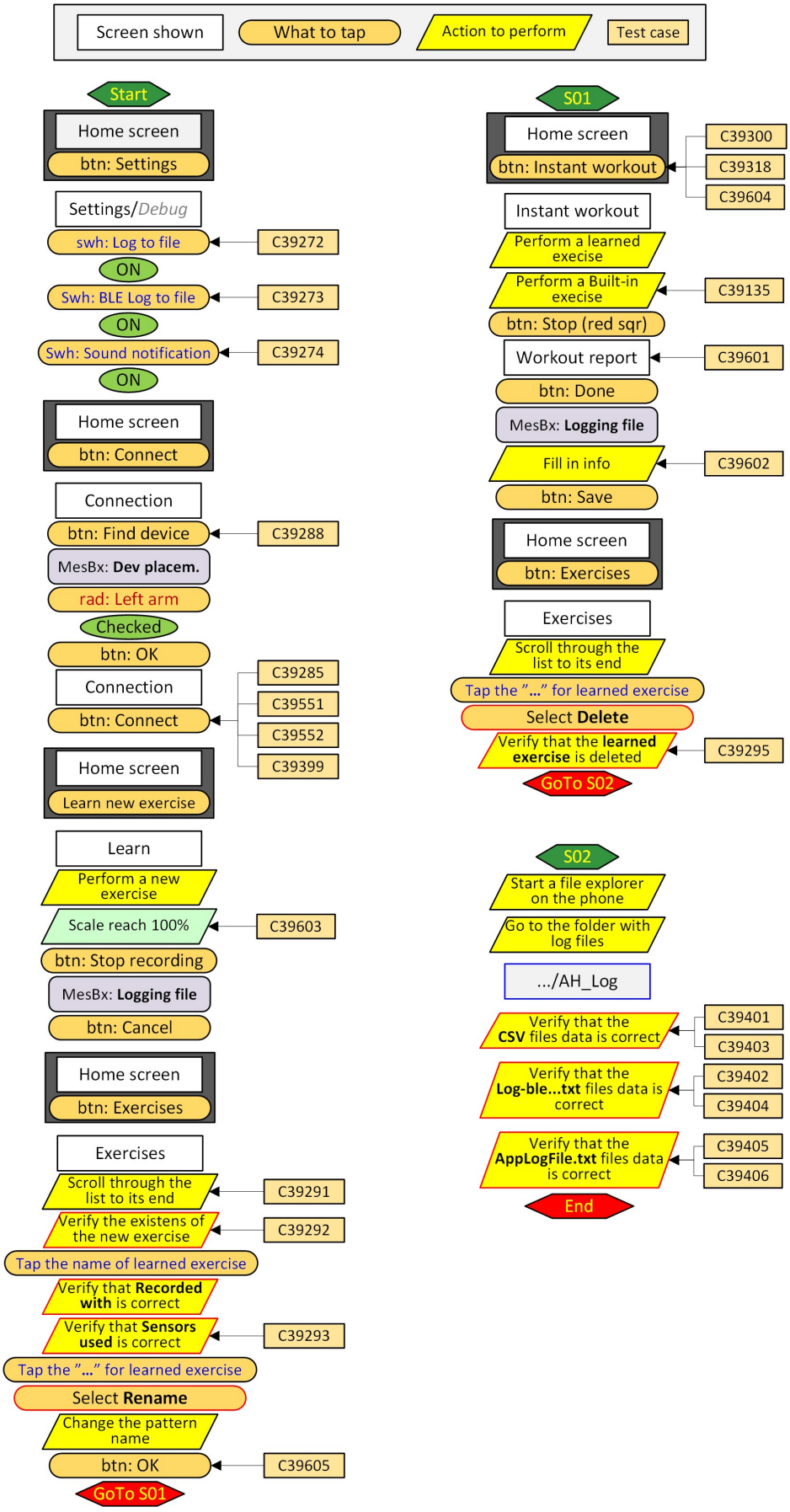

Figure 3: Example of test flows followed for manual testing

.

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2022-08-25

EXAMENSARBETE Return on investment and maintenance in a mobile test automation implementation
STUDENT Martin H Tomičić
HANDLEDARE Masoumeh Taromirad (LTH), Niklas Kenéz (Bosch SensorTec),
EXAMINATOR Per Runeson (LTH)

Test automation benefits and concerns in a mobile setting

POPULÄRVETENSKAPLIG SAMMANFATTNING **Martin H Tomičić**

Software testing is a way to check that a program works as intended; it is, however, time-consuming. So is it worth investing in automating the tests for a mobile application? My findings say yes! Some benefits include a 40% decrease in time spent carrying out tests, released resources, and increased confidence in the code.

Software testing is a challenging task no matter the project and can take up considerable resources. Automating the testing is a solution many turns toward in the industry. While automating the testing has its boons, the implementation can be costly, and the solution needs to be updated whenever new features are added. At Bosch SensorTec in Lund, they develop physical sensors for different uses, such as fitness tracking, and mobile applications to pair them with. One of these applications has outgrown its scope, and the current manual testing of the application has grown cumbersome and resource-draining. Bosch wanted an automated solution to lessen the testers' load to address this. However, they had concerns regarding the maintenance of the implementation of the solution and the possibility of a return on investment.

Results of this thesis, carried out at Bosch SensorTec and on one of their applications, as well as research into mobile automation testing, include:

- An analysis of the current state of testing in the project and what concerns the team members have regarding automated tests. Developers saw few issues, while the testers were wary about the costs, possible maintenance, and relying too much on only the automated tests to say that the application works as it should.
- A list of requirements the solution needs to follow in granting as much benefit to Bosch and the team as possible.
- A validation of the solution presents an (on average) 40% reduced execution time of the testing, resources being released within the team, and increased confidence in the code.
- An analysis of the maintenance of the automated tests, a presentation on different types of test cases and their possible maintenance, and describing some preventative measures to lessen the time spent on correcting tests.

By collecting data on different versions of the application for the validation, a critical fault was found in the latest version that inhibited a main feature of the application. This had not been detected yet and shows one of the considerable benefits of using automated tests for regression testing: test your already existing code base whenever something is changed to ensure nothing already existing has been broken.