Aalto University
School of Science

Master’s Programme in Information Networks

Automating an Integration Test Environment for a Wi-Fi Access Point Manager in enterprise wireless networks

Joel Juhela

Master’s Thesis
2025

Aalto University
School of Science

Author Joel Juhela

Title Automating an Integration Test Environment for a Wi-Fi Access Point Manager in enterprise wireless networks

Degree programme Information Networks

Major Software Engineering and Architectures

Supervisor D.Sc. (Tech.) Lassi Haaranen

Advisors M.Sc. Marko Teittinen, M.Sc. Hannu Heinilä

Collaborative partner Nokia Solutions and Networks Oy

Date 30.7.2025 **Number of pages** 44 **Language** English

Abstract

Software testing is a crucial part of the software development process to ensure quality. However, testing can be particularly challenging for systems that interact with physical hardware and external services. This thesis investigates how the testability of an existing software system can be improved through the introduction of an integration testing environment. Additionally, the thesis explores the perceptions software developers have on software testability.

This thesis adopted a design science research methodology. An integration testing environment and test cases were built for a Wi-Fi Access Point Manager used in enterprise wireless networks. Empirical data was gathered through semi-structured interviews and a survey with six developers in the case team.

The results align with the existing literature on software testability. The results emphasize the importance of testability factors such as controllability, observability, and amount of coupling between software components.

The thesis highlights practical considerations when testing systems that interact with physical hardware. The results of the study indicate that the proposed environment can improve the development workflow, make experimentation faster, and reduce the need for testing in production-like environments. The study highlights remaining limitations that make testing with near production-like environments necessary.

Keywords software testability, integration testing, software engineering, enterprise wireless networks, software testing, test automation

Aalto-yliopisto
Perustieteiden
korkeakoulu

Tekijä Joel Juhela

Työn nimi Integraatiotestausympäristön automatisointi Wi-Fi-tukiasemien hallintapalvelulle yritysverkoissa

Koulutusohjelma Information Networks

Pääaine Software Engineering and Architectures

Työn valvoja TkT Lassi Haaranen

Työn ohjaajat DI Marko Teittinen, DI Hannu Heinilä

Yhteistyötaho Nokia Solutions and Networks Oy

Päivämäärä 30.7.2025 **Sivumäärä** 44 **Kieli** englanti

Tiivistelmä

Ohjelmistotestaus on olennainen osa ohjelmistokehitysprosessia laadun varmistamiseksi. Testaus voi kuitenkin olla haastavaa järjestelmissä, jotka ovat vuorovaikuttavat laitteiston ja ulkoisten palveluiden kanssa. Tämä opinnäytetyö tarkastelee, miten olemassa olevan järjestelmän testattavuutta voidaan parantaa ottamalla käyttöön integraatiotestausympäristö. Lisäksi työssä tutkitaan ohjelmistokehittäjien käsityksiä ohjelmistojen testattavuudesta.

Opinnäytetyö toteutettiin hyödyntäen suunnittelutieteen tutkimusmetodologiaa. Integraatiotestausympäristö ja testitapaukset toteutettiin yritysverkkoihin tarkoitetulle Wi-Fi-tukiasemien hallintapalvelulle. Empiiristä aineistoa kerättiin haastatteluilla ja kyselyllä, johon osallistui kuusi kehittäjää kohdeorganisaatiosta.

Tulokset ovat linjassa aiemman ohjelmistojen testattavuutta koskevan tutkimuskirjallisuuden kanssa. Tuloksissa korostuu testattavuuden keskeiset tekijät, kuten hallittavuus, havaittavuus ja ohjelmistokomponenttien välinen riippuvuus.

Opinnäytetyö nostaa esiin käytännön näkökulmia järjestelmien testaamiseen, kun ne ovat vuorovaikutuksessa laitteiston kanssa. Tutkimuksen tulokset osoittavat, että ehdotettu testausympäristö voi parantaa ohjelmistokehitysprosessia, nopeuttaa testausta ja vähentää tarvetta testata järjestelmää tuotantoympäristöä muistuttavissa olosuhteissa. Vaikka integraatiotestausympäristö voi parantaa testattavuutta, tietyt jäljelle jäävät rajoitteet edellyttävät edelleen testauksen suorittamista tuotantomaisissa ympäristöissä.

Avainsanat ohjelmiston testattavuus, integraatiotestaus, ohjelmistotuotanto, langattomat yritysverkot, ohjelmistotestaus, testiautomaatio

Preface

Thank you to my advisors Marko Teittinen and Hannu Heinilä for the opportunity to work on an interesting project and for taking the time to advise on the thesis work. Thank you also to all the participants from Nokia I had the pleasure to interview. I enjoyed our discussions and I hope you did as well.

I would like to warmly thank my supervisor Lassi Haaranen for his excellent input and guidance throughout the whole thesis process from start to finish.

To my friends and family, I am deeply grateful for all the support and encouragement I have received throughout my studies. Special thanks to both my parents for taking the time to read the final drafts of this thesis and provide feedback.

Nurmijärvi, 30.7.2025 Joel Juhela

Contents

Abbreviations

ACS Auto-Configuration server.

AP Access Point.

API Application Programming Interface.

CI Continuous Integration.

CS Configuration Syncher.

SUT system under test.

TA thematic analysis.

TR-069 Technical report 069.

UDP User Datagram Protocol.

UI User Interface.

VCS Version Control System.

WSL Windows Subsystem for Linux.

1 Introduction

Software testing is a crucial part of the software development process to ensure quality. When software is easily testable, developers can ship new features with more confidence and higher velocity. Less time is spent diagnosing defects and bugs, thus reducing the cost of delivering quality software. However, not all systems are as easily testable as others, since they may, for example, depend on external services that might not be controllable or available during testing. In more complex systems and especially in systems that interact with actual hardware, testing can be even more expensive and time-consuming. The use of hardware in testing limits the speed with which testing can be performed and requires investment in hardware to be viable (Foxworthy 2025).

In addition to architectural challenges, testability can also easily be neglected when building new software products. Ghafari et al. (2019) found that poor testability and difficult-to-test components can lead to a situation where developers will not write tests for the software they build.

In the literature software testability is not clearly defined, and many authors have specified different aspects that could affect it. Most of the research is focused on the internal attributes of software, but testability can also be considered from a more holistic point of view, where the surrounding context, such as what stakeholders are taking part in testing, is considered.

These challenges serve as a framework and motivation for investigating testability and how it can be improved in the context of a practical industry problem faced by the case team at Nokia Solutions and Networks. There was a need to increase coverage and confidence in testing of their Wi-Fi Access Point Management software. The organizational goal of the project was to improve the efficiency of the development team, enabling faster feature development and issue diagnosis by introducing a new integration testing environment.

1.1 Research objective and questions

This thesis studies the perceptions developers have when it comes to software testability in general and explores how the testability of an existing software product can be improved. In particular, this study is concerned with whether a new integration test suite and a new testing environment can be used to improve testability.

Considering this objective, the following research questions were formed:

- **RQ1:** What aspects affect the testability of software?
- **RQ2:** What factors influence developers' willingness to adopt the integration test environment?

1.2 Thesis structure

The thesis is structured as follows. Chapter 2 describes key concepts that relate to testability and software testing in general. Chapter 3 presents the context in which the study was carried out. It describes the Wi-Fi Access Point Manager, the Technical report 069 (TR-069) protocol and the existing state of testing practices within the case team.

Chapter 4 describes and justifies the methodology chosen for this study. Chapter 5 details the implementation of the testing environment and associated tests. It also explains the design decisions made to support testing.

Chapter 6 presents the empirical results from the survey and semi-structured interviews. Chapter 7 provides answers to the research questions and discusses the implications and limitations of the study. Finally, Chapter 8 concludes the thesis.

2 Research Background

This chapter focuses on the topics and concepts that are relevant for this study. Chapter 2.1 describes different concepts related to software testing. Chapter 2.2 explores the software testability as a concept and the characteristics that affect it.

2.1 Software testing

Software testing is a crucial part of the software development lifecycle. The aim of software testing is to ensure quality of the software being built, validate it against its requirements, and ensure that software defects are caught as early in the development process as possible.

2.1.1 Levels of testing

To ensure software quality and reliability, testing is conducted at multiple levels. Each of these levels targets specific aspects of the software development process. According to Dennis et al. (2012), testing can be broadly divided into four levels: unit testing, integration testing, system testing and acceptance testing.

- *Unit testing* verifies individual software components, modules, or functions of the program in isolation. Each unit is tested against its specification to ensure that it behaves correctly. Since the tests are derived from the specifications, errors can be discovered in later stages as the specifications can be misinterpreted during development.
- *Integration testing* focuses on interactions between software units. The individual software components should have passed their individual unit tests. At this stage, the interaction and combined behavior between the software units is tested.
- *System testing* is similar to integration testing but the scope is extended to include the entire software system. Thus, the focus is on how the whole system complies with its requirements. This involves testing both functional and non-functional requirements. It often includes performance, usability, and security testing, as well as verification of the supporting documentation.
- *Acceptance testing* is the final stage before software deployment. At this stage, testing is typically carried out by the end-users of the software, and the system is validated against customer requirements.

2.1.2 Automated testing

In the context of modern software development, where release cycles are rapid and continuous integration and deployment are prevalent, automated testing is essential. Automated tests are effective for detecting regressions and ensuring software stability when updates are frequent. Compared to manual testing, automated testing tends to have a higher upfront cost, but in return the tests are more repeatable, reliable, and faster to execute.

Cohn (2010) originally introduced the concept of the *automated testing pyramid*. Although the different levels outlined in the previous section address the scope of testing at different stages, the automation pyramid addresses the quantity and related cost of automated testing. The pyramid is structured with unit tests at the bottom, followed by integration tests and finally end-to-end (E2E) tests at the top. This model has later been adopted and adapted by many practitioners (Fowler n.d.; Ackerson and Fernandez 2022). All levels of software testing allow for automation except acceptance testing, which is usually manual as the testing is carried out by the end-users of the software.

The automated testing pyramid is presented in Figure 1. In the automated testing pyramid, the width of the pyramid corresponds to the quantity of tests present at each stage. The higher up in the pyramid, the less efficient and more expensive testing becomes. Thus, according to Cohn (2010), you should have many unit tests compared to more complex integration and end-to-end tests, as developing and running those tests is more time consuming.

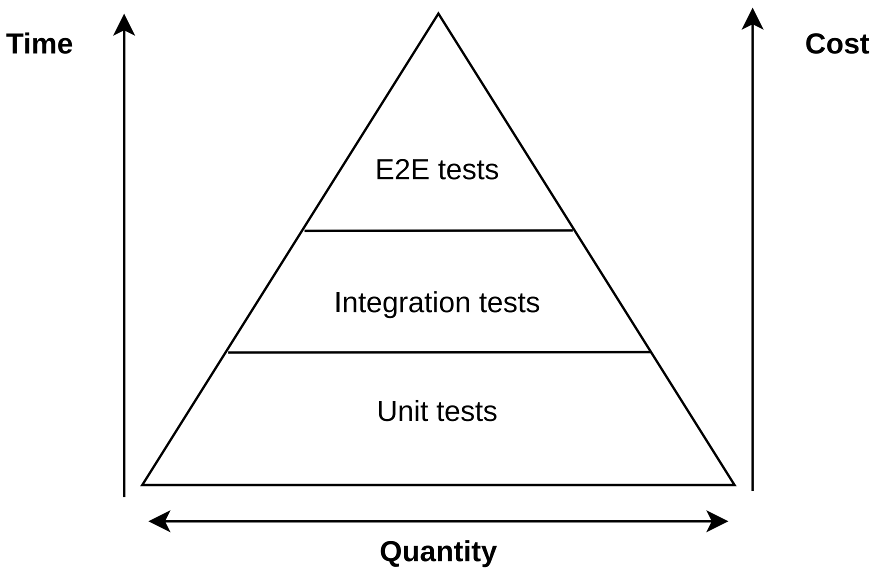

Figure 1: Automated testing pyramid adapted from Cohn (2010)

Cohn (2010) proposes that the reason for the higher cost associated with end-to-end tests comes from the following factors. Firstly, such tests can be brittle. If tests rely on, for example, a user interface, a small change can break many tests. Secondly, these tests are expensive to implement as they involve multiple components that interact together. Finally, end-to-end tests take longer to execute compared to unit or integration tests. (Cohn 2010)

2.1.3 Testing strategies

Software testing strategies can be classified based on the degree of knowledge about the internal workings of the system under test (SUT).

Black-box testing is a testing strategy, where testing is carried out without concern about the internal structure of the SUT. The test cases are designed only on the basis of the input and expected output of the SUT. This makes black-box testing useful for validating functional requirements. (Myers et al. 2011)

White-box testing on the other hand is the approach in which the internal structure and messaging of the SUT are known. This involves testing the internal logic, code paths and conditions to ensure that all statements and branches are correctly implemented and executed (Myers et al. 2011).

Gray-box testing is between white box and black box testing. In this strategy, some of the internal structures of the SUT might be known, or there might be probes to observe the internal state of the SUT. (M. E. Khan and F. Khan 2012)

Generally, the white-box testing strategy is more prevalent in unit testing, whereas black-box testing is found in integration and end-to-end testing.

2.1.4 Integration testing with hardware

When software systems communicate with physical devices, it is vital that actual hardware is included in the testing. Otherwise, it is difficult to validate that the system functions as expected in real-world conditions.

Testing with physical hardware can reveal defects that would not be visible when using purely software-based testing. Issues can rise from for example concurrency, different firmware versions on the networked devices or vendor-specific differences in implementation (Bures et al. 2020).

When working with hardware in testing, there are several limitations, that can make testing more cumbersome than with purely software-based systems. The hardware can require more sophisticated setup To achieve specific situations that are required by testing, the devices might need to reach a certain state. This can lead to a slower setup. (Bures et al. 2020)

2.2 Software testability

Software testability is vital in assessing the quality of a software system. According to Voas and Miller (1995), low software testability leads to a need for increased amount of testing, both automated and manual. Thus, this makes testing more time-consuming, expensive and more prone to errors. This in turn slows down development and increases technical debt.

Software testability is a widely discussed topic in software engineering. In the literature, there are various different definitions and contributing factors proposed by multiple different authors. Voas and Miller (1995) define software testability as the probability that a piece of software will fail on its next execution during testing if the software includes a fault.

There is no consensus among researchers about which factors affect software testability. One option is to consider software testability as a characteristic or property of software. Binder (1994) proposes that software testability consists of two parts. These parts are *controllability* and *observability*. Controllability is defined by the degree to which the input and internal state of a software component can be controlled. Observability, on the other hand, is defined by how well the output and internal state of the software component can be seen (Binder 1994). Freedman (1991) introduces a concept of *domain testability* which means that a program is both observable and controllable.

Some researchers have proposed specific attributes in software design that can negatively impact software testability. Binder (1994) states that the existence of global and local state variables reduces the observability of software component. This is due to the fact that the possible response of a component is not explicitly determined by the input (Binder 1994).

A system for which an oracle is infeasible is fundamentally untestable (Binder 1994). Test case development is in itself a process which, like software development itself, is prone to errors. Faulty test cases can be really problematic as they can lead to faulty programming code ending up in production or working code not making it through as the tests are broken. (Binder 1994)

Garousi et al. (2019) analyzed 208 articles related to software testability. They raise multiple properties of software in addition to controllability and observability that also affect software testability. In addition to observability and controllability, these factors include complexity, cohesion, coupling, understandability, inheritance, reliability of the test suite, availability, and maintainability. The factors and their effects on the testability of software are presented in Table 1. Of the factors presented, controllability and observability are the most frequently reported factors influencing testability (Garousi et al. 2019).

Although testability can be considered as a characteristic of the software itself, it can also be considered from a more holistic point of view. Kedemo (2015) describes how other dimensions, such as how communication is handled in the development team, can have an effect on testability. Similarly Garg and Singhal (2016) propose that the qualities of the software developers involved in the development process may also influence software testability. Developer attributes, such as years of experience and confidence in the programming language, affect how complex and understandable code a developer creates. This in turn has an effect on software testability (Garg and Singhal 2016).

For the purposes of this thesis, software testability is considered only as a property of the software itself, excluding team or process-related dimensions.

Factor	Description and effect on testability
Observability	How easy it is to observe the state and behavior of software component. High observability improves testability.
Controllability	How easy it is to control the state of a software component during testing. High controllability improves testability.
Complexity / Simplicity	High complexity in software decreases testability, simplicity improves testability.
Cohesion	How well elements in a software module fit together. High cohesion improves testability.
Coupling	How interdependent software components are. High coupling decreases testability.
Understandability	How well a software component is documented or is self-explanatory.
Inheritance	High inheritance decreases testability in object-oriented systems. Is related to complexity as high level of inheritance makes code more complex.
Reliability of tests	Confidence in the outcome of tests. With low confidence, testability decreases.
Availability	The degree to which a software component allows for runtime testing without being extensively affected.
Reusability of the test suite	The ability to reuse test code across the software improves testability and reduces testing effort.
Maintainability	Software that is easy to maintain and modify enhances testability.

Table 1: Software testability factors adapted from Garousi et al. (2019)

3 Research context

In complex systems that interact with other services and hardware, testing becomes more challenging due to, for example, dependencies and limited control over the full execution environment. In the context of this thesis, the system of interest is the Wi-Fi Manager, which coordinates interactions between Wi-Fi APs and various supporting services. The Wi-Fi Manager communicates with the Wi-Fi APs via the TR-069 protocol, serving as a critical integration point between the APs and the rest of the system.

The following chapters describes the Wi-Fi Manager and surrounding development context in more detail. Chapter 3.1 describes the TR-069 protocol, which is the communication protocol used by Access Points (APs) for configuration management. Chapter 3.2 describes the system architecture for the Wi-Fi Manager, and finally Chapter 3.3 describes the testing practices in the case team at the time of starting the thesis.

3.1 Technical Report 069 (TR-069)

Although there are other possible protocols that can be used for the management of Wi-Fi Access Points such as TR-369, SNMP, and others, for the purposes of this study, we focus only on TR-069 as it is the one used by the Wi-Fi Manager. We cover only the functional components of TR-069. The following functional components are covered by the definition of TR-069 by Wich and Blackford (2020).

Auto-Configuration and Dynamic Service Provisioning: Enables automatic configuration and provisioning of APs.

Software/Firmware Image Management: New versions of firmware and software images for APs can be downloaded. The protocol provides mechanisms for version identification and file download initiations, and notifications on the possible success or failure of a file download.

Software module management: The protocol enables an Auto-Configuration server (ACS) to manage software and execution environments in a AP. This includes the ability to install, update, and uninstall software modules. The protocol also provides support for starting and stopping applications on the AP.

Status and performance monitoring: The protocol provides support for an ACS to monitor the status and performance metrics of a AP, if they're made available by the AP.

Diagnostics: The protocol provides support for APs in providing information for the diagnosis of connectivity and services issues.

3.2 The Wi-Fi Manager

The Wi-Fi Manager is a service developed in Nokia Solutions and Networks as part of a private wireless enterprise solution. The solution enables centralized management of Wi-Fi Access Points and the network. Wi-Fi Access Points are provided as part of the solution. The hardware and firmware of APs are not built or designed by the development team.

The Wi-Fi Manager is responsible for the configuration, firmware management, monitoring, and diagnostics of the Wi-Fi APs. The Wi-Fi Manager does not have a User Interface (UI), but instead it receives commands and fetches configurations from other services. The communication between the Wi-Fi Manager and the Access Points is based on the TR-069 protocol. Thus, the Wi-Fi Manager must support the full range of operations defined in the TR-069 specification.

3.2.1 System description

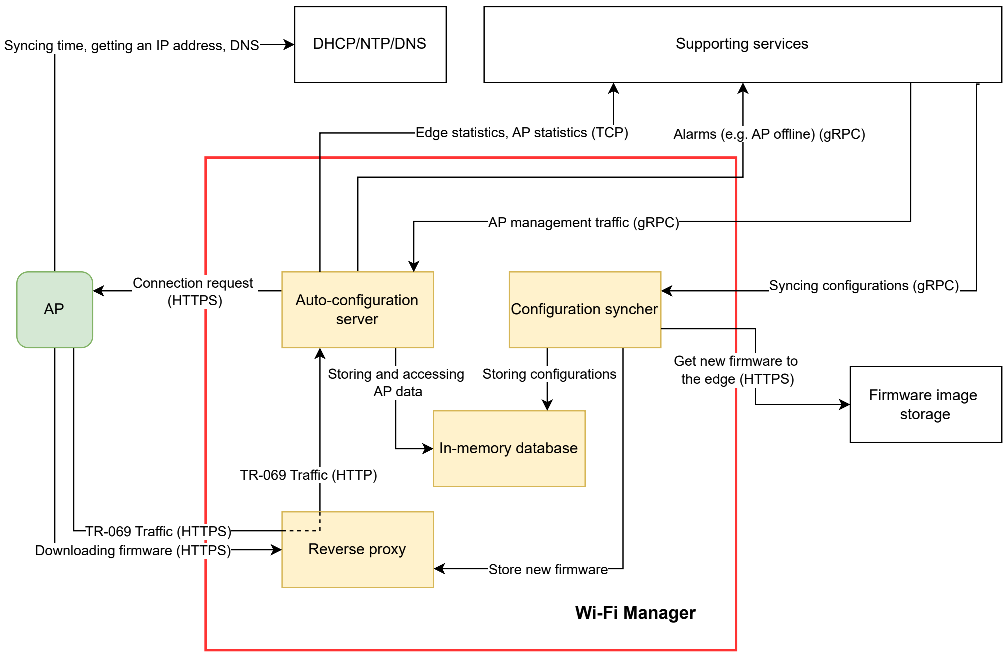

Figure 2: Simplified architecture for the Wi-Fi manager, with the Wi-Fi Manager components marked with red

The Wi-Fi Manager consists of multiple services which are depicted in Figure 2. The services that are developed and maintained by the case team are inside the red rectangle in the figure. All services are containerized using Docker[1] and deployed in a Kubernetes[2] cluster.

[1]https://www.docker.com/
[2]https://kubernetes.io/

The APs communicate with the Wi-Fi Manager by sending TR-069 messages over HTTPS. Communication occurs with the Wi-Fi Manager periodically. The Wi-Fi Manager can request the APs to initiate a connection when needed (for example, when an AP is ordered to restart) by sending a connection request to the AP.

The reverse proxy decrypts and forwards HTTPS traffic. It also offers the APs new firmware versions for download.

Auto-Configuration server (ACS) is responsible for handling the TR-069 traffic and forming responses that will be sent back to AP. In addition to this, ACS sends diagnostics to APs and raises alarms if necessary. Reasons for raising alarms include configuration failures and cases where an AP is disconnected from the network.

Configuration Syncher (CS) is responsible for syncing changes to configurations and AP firmware from outside the Wi-Fi Manager. CS fetches configurations from supporting services. Additionally, CS downloads firmware images from storage and stores them to be served by the HTTP proxy.

The in-memory database facilitates all communication between CS and ACS with a publish-subscribe pattern. When CS publishes a new configuration, ACS will be notified about the change and the new configuration will be applied to the APs that require it. The in-memory database is used to keep track of the status of configuration updates and other management jobs for APs.

3.3 State of testing and development

The application is tested using a Continuous Integration (CI) system, Jenkins[3]. The Jenkins pipeline is responsible for running tests and building images of the services in the Wi-Fi Manager.

There is a large existing suite of unit tests for the Wi-Fi Manager. The Wi-Fi Manager services have adequate levels of unit testing, which cover most of the important cases and requirements for the individual software units.

Unlike unit tests, there are no integration test cases built for the Wi-Fi Manager. There exists a large suite of component test cases that do not use the actual software components but instead rely on mocked interfaces and services in the tests.

In addition to automated test cases, the development team performs extensive exploratory testing to verify that the changes they introduce work as expected. This manual testing is performed on shared remote server environments. In this context, aforementioned shared environment means a testing setup, where multiple testers use the same server infrastructure to facilitate their testing. Before this study, it was not possible for developers to run the software locally.

The steps in how a developer deploys their code to a shared environment are described in Figure 3. After the developer has validated that their changes pass local tests, the

[3] https://www.jenkins.io/

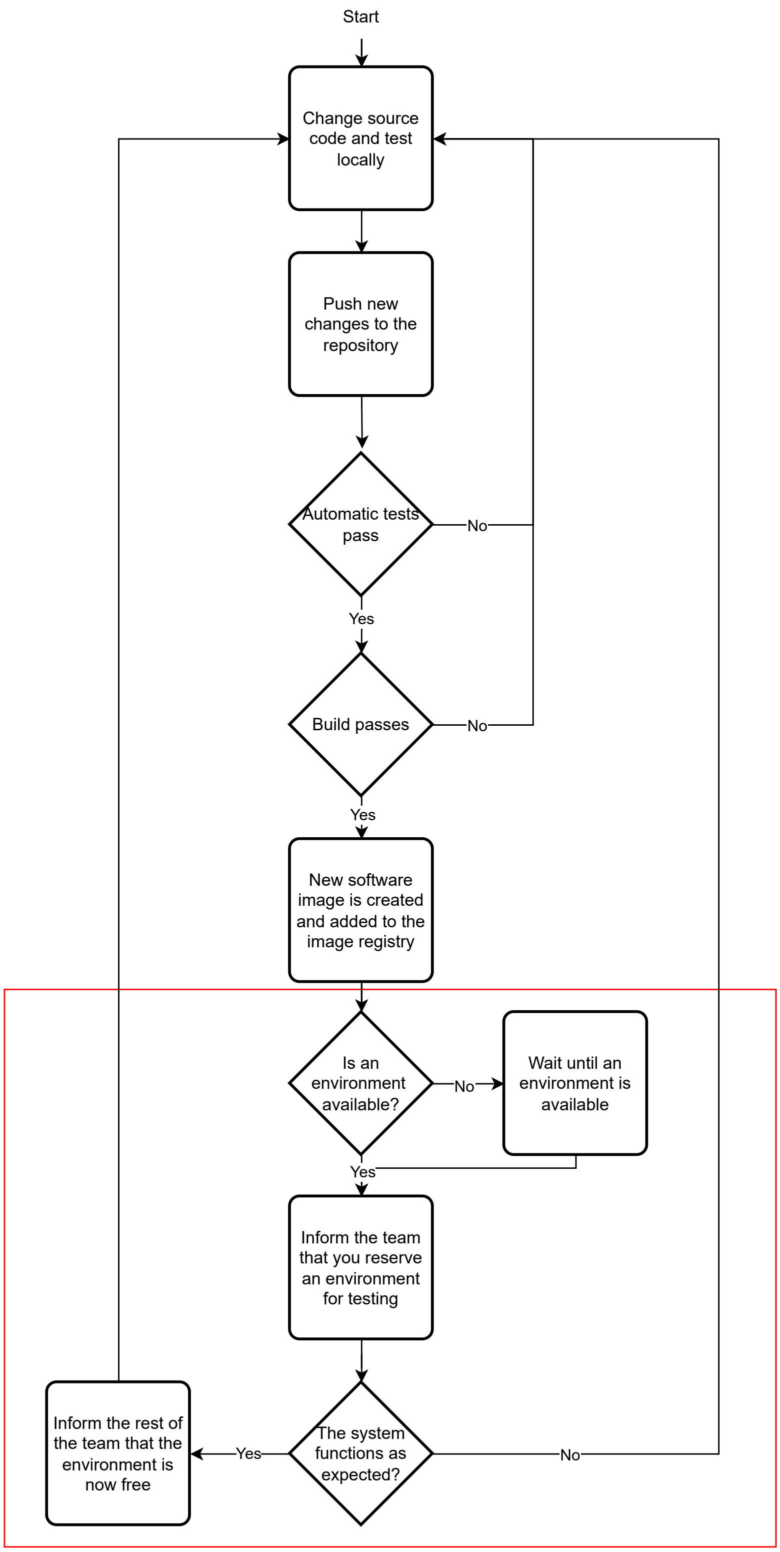

Figure 3: Flow of testing the Wi-Fi Manager in a shared remote environment

changes are pushed to a Version Control System (VCS). The changes are then put into a Jenkins CI pipeline that runs automatic unit and component test cases. A software image is created and pushed to the image registry. The developer can then pull this development image into a remote server environment for testing. After this point, the testing is done manually to ensure that the software functions as expected in a production-like environment with real hardware.

4 Methodology

This section explains the methodological approach of this study, including the research approach chosen, data collection methods, and how the data was analyzed.

4.1 Research approach

This study adopted design science research approach. This approach is particularly fitting for answering **RQ2** *What factors influence developers' willingness to adopt the integration test environment?* as the answers to this research questions might be highly specific to the specific context where the study was carried out in.

The empirical part of this thesis was concerned with evaluating the software artefact in addition to finding general testability aspects. The design of the artefact was done based on the initial set of requirements and studying literature. The initial set of requirements are discussed further in Chapter 4.3.

4.2 Research Design and process

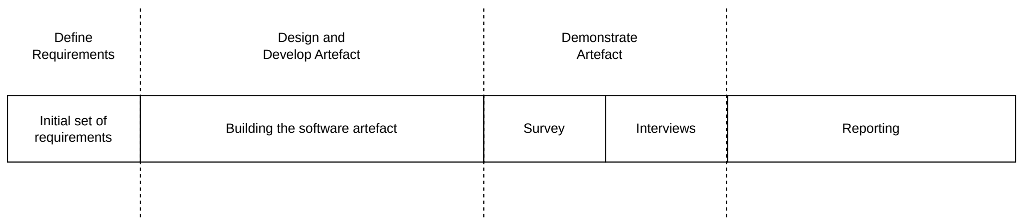

Figure 4: Research process timeline

The research process (seen in Figure 4) is roughly divided into four activities. The process loosely follows the design science research activities from Johannesson and Perjons (2021).

The research process (seen in Figure 4) began with informal discussions with the thesis advisors where the set of initial requirements for the test environment were elicited. The requirements are presented in Tables 2 and 3. These discussions were not recorded but instead notes were taken.

Based on the initial set of requirements, new test cases and a testing environment were built. This is considered as the phase for designing and development of the artefact. To demonstrate and evaluate the software artefact and the initial set of requirements, a survey was carried out, and semi-structured interviews were conducted with the developers of the Wi-Fi Manager.

4.3 Requirements for the test environment

The set of requirements was elicited and prioritized in a meeting between the researcher and the thesis advisors. The requirements were categorized into functional and non-functional requirements. Prioritization was based on the expertise of the thesis advisors as domain experts.

Table 2 presents the functional requirements for the testing environment and the test cases. These requirements are considered when building and evaluating the artefact with the prioritization helping in organizing the design and development of the testing environment.

Requirement	Description
FR1: No dependence on shared resources	The test environment should be able to run on an individual developer's laptop instead of the shared environment or other shared resources.
FR2: Support for production dependencies	The test cases should be run on the same dependencies as the production environment
FR3: TR-069 compliance	The test environment should support devices that operate on TR-069
FR4: Support plugging in physical hardware	The test environment should support testing integrations utilizing physical hardware
FR5: Support for AP firmware verification	The test environment should support interactions for testing the firmware on an AP
FR6: Support for User Equipment	The environment should support using user equipment such as phones.

Table 2: Functional requirements for the test environment

Table 3 presents the non-functional requirements for the testing environment and test cases.

Requirement	Description
QR1: Diagnosable feedback from failed tests	The test environment should provide clear feedback for the developer on possible reasons for the failed test cases
QR2: Performance	The test cases should be executed in a reasonable time frame. This is to ensure that we end up reducing the developer feedback loop during testing
QR3: Clarity and understandability	The test cases and environment should be well-documented, easy to understand, and follow clear logic.
QR4: Use of familiar technologies	The test environment should be implemented using tools that are already familiar for the developers in the team
QR5: Test code modularity	The test code should be extendable to suite new and different types of test cases

Table 3: Non-functional requirements for the test system

4.4 Data collection

Data collection for this study was done in two parts. The first part focused on requirements elicitation from informal discussions with the thesis advisors. The second part of data collection was conducted after the artefact was developed. The goal for this part of the data collection was two-fold. First, the goal was to explore what testability factors that the case team finds important in their daily work. Secondly, the goal was to evaluate the artefact and to see how well it meets the needs of the developers.

The second part of data collection involved both a survey and semi-structured interviews, targeting five developers from the team developing the Wi-Fi Manager. The same participants participated in both the survey and semi-structured interviews. The participants were selected as a convenience sample.

The survey aimed to assess the current state of testing and the level of confidence the developers had in the existing testing practices. The survey included open and closed questions, and the full content of the survey can be found in Appendix A.

Following the survey, semi-structured interviews were conducted with the same developers. These interviews were held in December 2024, each lasting approximately 60 minutes and conducted in Finnish. All interviews were held in person, recorded and machine-generated transcripts were produced. The purpose of these interviews was to evaluate how well the artefact fulfills its' requirements and to find out, which aspects of testability the developers find most important. The set of interview questions can be found in the Appendix B. The questions served only as a guide for the interviews. Clarifying and expanding questions were asked when necessary.

The interviews began with a brief description of the study topic and background questions. The interviewee was then asked general questions about software testability and the current state of testing of the Wi-Fi Manager.

After this, the researcher demonstrated the new test cases and the testing environment that was created. After the demonstration, the interviewee was asked clarifying questions regarding the differences between the existing and newly created test cases and the perceived usefulness of the newly created testing environment.

4.5 Data analysis

To ensure the quality of analysis, the interviews were listened to manually and inaccuracies in the machine-generated transcripts were corrected.

For analyzing the open questions presented in the survey and interview transcripts, thematic analysis (TA) was used. TA was chosen as the method for data analysis as it is a flexible and accessible method that can be used for qualitative data analysis (Braun and Clarke 2012).

In total, 214 quotations were obtained from the material with a total of 85 codes that were identified. Table 4 presents the most common codes found from the interview material.

Code	Occurrences
Use of remote server environments	18
State management	12
Controllability	9
Feedback loop	5

Table 4: Most common codes

Related codes were grouped into higher level themes, which are described in more detail in Chapter 6.

5 Implementation

This chapter explains the details and choices made during the implementation of the artefact.

The artefact consists of two main components. The first component, described in Chapter 5.1, is the virtual machine environment to run the Wi-Fi Manager with mocks of the surrounding services. The second component, described in Chapter 5.2, are the automated test cases. There are two main ways the artefact is used. The first is running the automated test cases. The second is using the virtualized environment for development and exploratory testing with an AP, described further in Chapter 5.3.

5.1 The Virtual Machine Environment

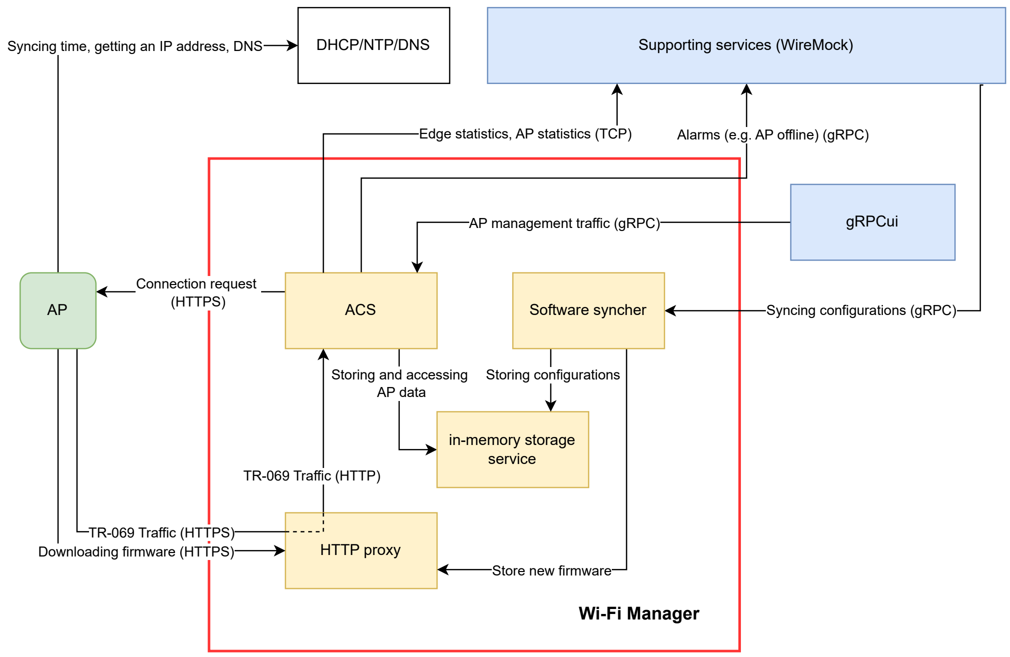

Figure 5: Structure of the virtualized testing environment, showing the Wi-Fi Manager components (red) and mocked services (blue).

Developers use Windows 10 laptops with Windows Subsystem for Linux (WSL) for development work. Initially, an attempt was made to build the new environment using only WSL and Docker. However, the limits of WSL were quickly noticed as it did not provide the necessary control over the environment.

A significant challenge was the network configuration, which is essential for communication between the AP and the Wi-Fi Manager. As an example, configuring User Datagram Protocol (UDP) port forwarding between the Windows host and WSL required running multiple commands as an administrator in PowerShell. However, the configuration was still prone to failure every time WSL was restarted.

As a result, the environment was moved to a virtual machine. Using a Linux virtual machine provided significantly better control over the environment. To make the environment as developer-friendly as possible, great care was taken in how the environment is set up. The virtual machine is configured using Vagrant[4] to allow the configuration of the virtual machine to be stored in version control and to provide a consistent experience between multiple developers.

The Wi-Fi Manager relies on surrounding services for its operation. These surrounding services are depicted in Chapter 3.2. Including these services would not have been practical, so a decision was made to mock the services. Figure 5 depicts the services present in the virtual machine. The boxes marked with blue are the services that are not present in a production environment.

WireMock[5] was chosen to mock up the services. WireMock allows developers to simulate the behavior of Application Programming Interfaces (APIs). This makes it possible for developers to programmatically set responses to specific requests. By default, WireMock is only able to work with HTTP-based APIs, but with an extension, WireMock is able to support gRPC (WireMock 2025). This makes it suitable for mocking services around the Wi-Fi Manager.

In addition to WireMock, a gRPCui[6] server is added. GRPCui makes it possible to interact with gRPC servers via a Web browser. In the context of the Wi-Fi Manager, this allows the developer to interact with the gRPC endpoints of the ACS to send management commands to the connected AP, such as rebooting the AP.

5.2 Test cases

Ten test cases were produced to evaluate the artefact. The test cases were taken from the existing component test cases that were already present in the Wi-Fi Manager. This choice was made in part because of time constraints related to the thesis project and in part because using test cases that were already familiar to the developers of the Wi-Fi Manager would make it easier to show differences in approach between the previously built test cases and the new ones.

The test cases were built using the same programming language and framework as the existing component test cases to allow easy adoption by developers. Additionally, this meant that existing testing utilities and other tools already present in the source code could be used for the development of the new test cases.

In contrast with the existing component test cases, the new test cases use production dependencies and a black-box approach to testing. In the test cases, the Wi-Fi Manager is tested only using the inputs and outputs of its interfaces. Although this means that there is a reduced level of visibility in the reason for software defects (such as the

[4] https://www.vagrantup.com/
[5] https://github.com/wiremock/wiremock
[6] https://github.com/fullstorydev/grpcui

internal state of a software component), we can gain more certainty that the different services within the Wi-Fi Manager integrate as expected.

5.3 Exploratory testing and development

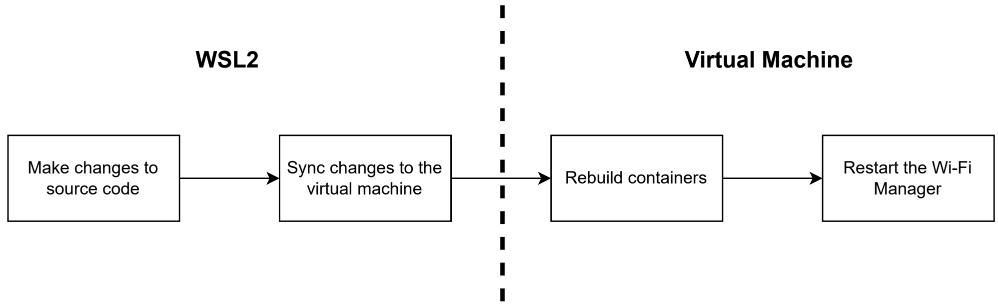

Figure 6: Getting changes to the environment

One goal of the artefact was to reduce the need for shared server environments and reduce the feedback loop for developers when testing hardware. The flow of how developers can have their changes tested in the local environment is depicted in Figure 6.

With the new environment, the development workflow goes as follows. The developer first makes changes to the environment, after which the changes are synced to the virtual machine. The developer then rebuilds the services on the virtual machine and restarts the Wi-Fi Manager. At this point, the developer can test their new changes against an AP which is connected to the laptop.

6 Results

This chapter presents the findings from the interviews and the survey. Some quotes have been altered for improved readability while maintaining the original tone and content. The quotations are translated from the original Finnish into English. Quotes are not connected to individual participants to protect the anonymity of the participants. This precaution is taken because all participants work in the same team and are familiar with each other.

6.1 Perceptions on testability in general

During the interview, participants were asked what in their experience makes software testable. In addition, they were asked what are parts that make software hard to test. The participants raised multiple aspects also present in Garousi et al. (2019). The different testability factors mentioned by the participants are presented in Table 5. The aspects mentioned most frequently by the developers were controllability and observability.

> *“ If the component has clear and broad APIs that allows me to programmatically set how the component functions. That helps a lot in testing. ”*

Another aspect often mentioned was coupling. The reduced amount of coupling between software components was seen as enabling better testability of a software component. Highly coupled components, on the other hand, were seen as more difficult to test.

> *“ If different parts of the software are tied up together then it is really difficult to define the test scenario without testing the whole system at once. It is just not practical. ”*

> *“ If the dependencies are tightly coupled. It makes testing very difficult. In practice, this usually means that the entire component needs to be rewritten. ”*

Some participants also linked coupling to the controllability of a software system. Highly coupled components are hard to separate for testing purposes, making tests more complex and brittle.

Complexity was mentioned as a factor that was seen as an aspect that can easily make testing more difficult. According to multiple participants, the software components should have clear roles and responsibilities, otherwise they become difficult to test. One participant raised this in the case of APIs. According to the participant, having APIs that are too broad can lead to unusable and difficult to test components.

An aspect that was often brought up in interviews was *understandability*. This was discussed in the form of how well the test cases document the functionality. In particular, the messaging structure of the TR-069 protocol was deemed important in the context of Wi-Fi management software.

> *“ The test case should document well, what is the expected behavior and what messages are expected. From the documentation [for the TR-069 protocol] we can see then how it expected to work and we can quickly find the point of failure. ”*

Aspects	**Mentioned by the developers**
Observability	Yes
Controllability	Yes
Dependency Factors (Cohesion/Coupling)	Yes
Complexity and Simplicity	Yes
Understandability	Yes
Inheritance	-
Reliability	-
Availability	-
Reusability of Test Suites	Yes
Maintainability	-

Table 5: Testability factors presented by Garousi et al. (2019) and if they were mentioned by developers in the case team

6.2 Testability of the Wi-Fi Manager

6.2.1 Existing test cases

The testability and testing practices as they relate to the Wi-Fi Manager were measured both via the survey and semi-structured interviews. The developers in the case team trust current automated testing practices, with most participants rating their confidence in current testing as Moderate. Only a single participant rated that they did not trust the current level of automated testing. Their reasoning was tied to the lack of automated testing done with AP hardware. Another participant also mentioned that the most common reason for testing in shared remote server environments is the need to validate changes on actual AP hardware.

> *“ The interoperability of the system with the APs is the usual issue. We can simulate the AP but in cases where the behavior is non-deterministic, it becomes difficult. This is usually the main reason for extensive testing to verify a fix or a feature. ”*

As the new test cases are based on the existing component cases of the Wi-Fi Manager, the participants were asked for input regarding the existing test cases.

With the current system, the most important issue highlighted by multiple participants is the handling of state in the service. As the Wi-Fi Manager is a reactive software system by its nature, there are inherent limitations in how state is used. According to multiple participants, setting the state can be complex, and there can be a lot of different configuration options that need to be set correctly to be able to run tests that accurately reflect situations that are faced in production.

> *“ You need to work around the state. The setting of the state can be inconvenient. So, you need to take that into account when designing the tests. But it brings itself with its own challenges, like you need to take into account in which order test cases are run. ”*

Observability was also another issue highlighted by multiple participants.

> *“ As the usual issues relate to timings, so the fact, that you can get the AP to get to do a certain action at a certain moment when the system is in a particular state. This can be really difficult if there is no way for knowing in what state the system is in. ”*

6.2.2 Challenges in using shared environments

As mentioned in Chapter 3.2, the testing of the Wi-Fi manager was performed in shared remote server environments. All participants used the shared environments as a part of their development and testing, but their approaches varied. Some participants used environments in multiple parts of their development process, and some used them only as a final check that their changes were working as expected.

> *“ First you create a Proof of Concept for your solution. Then, you check that it works correctly in the environment with real APs. If this is the case, you continue the work . . . I always do a final check in the shared environment, ensure for the last time that my changes work as expected. ”*

When asked about possible hindrances to their work, multiple developers mentioned that the shared use of the environments can lead to a lack of available resources as there are not enough available environments to serve all developers. This was also present in the survey data, where most developers had faced a situation where the lack of an available environment had been a blocker of their work. Although the development team is still relatively small, there are still situations where the work of a single developer can be interrupted by not having an available environment to be used for testing.

One of the participants even highlighted that the team should move away from using shared environments altogether. They see that the goal is to eliminate manual testing entirely as it is slow, tedious, and reduces the velocity of delivering new features.

Another grievance the participants highlighted was that the shared environments are not suitable for accurate testing.

> *“ There is no way of catching issues, it is more like finding out if there is something fundamentally wrong, that it doesn’t work at all. ”*

The time to deploy to the environment was also another aspect that made the use of shared environments more cumbersome.

> *" There are multiple steps involved when you have written your changes, and you want to test it in the environment. You need to create a build in the pipeline. You need to change the deployment on the environment, scale down the operator, and wait until the system is in the state that is required for your testing. And after you're done you need to remember to change it all back. "*

As the environments are shared by multiple developers, it is essential, that each developer reverts the system back to a known starting state after their testing session. If the environment was not reverted to its original state between test sessions, remaining configurations, software versions, or states left from previous tests can have an effect. This can lead to situations where testing can produce false positives or negatives.

In addition, the developers need to be certain that they're the only ones performing testing in the given environment. Multiple testers using a shared environment at the same time can create unexpected results.

To address this challenge, the developers used a group chat for coordination. This allowed developers to communicate about which environment they were performing testing in. However, this approach was not foolproof, as it depended entirely on manual communication for reserving the environment. Because it depended on individuals remembering to notify others, it was seen as a potential risk to coordination and the reliable use of the shared environments.

> *" There is always the risk, that people accidentally work at the same time in the environment, and somebody makes changes, and then you can't always be sure, whose changes broke the environment. "*

Despite the challenges in testing with shared environments, participants saw the use of shared environments as the only practical option. This was largely due to the nature of the solution. The Wi-Fi Manager interacts with multiple Wi-Fi APs and surrounding services that cannot be practically be replicated or installed on local development environments. Additionally, certain technical limitations further restricted the ability to run the software outside of the shared environments. Thus, the use of shared environments was vital in ensuring that the Wi-Fi Manager worked as expected on real hardware.

6.2.3 AP Firmware verification

One of the key findings found during the interviews was the need for supporting various different Access Point (AP) firmware versions with the Wi-Fi Manager. Although the TR-069 protocol is standardized, there is the possibility of firmware versions having different implementations. Thus, ensuring consistent and correct AP behavior was a significant challenge. The developers had to identify which firmware versions were fully supported and compatible with the Wi-Fi Manager.

The Wi-Fi APs belongs to the complete end-to-end Wi-Fi solution which is delivered to the customers. Different AP firmware versions can lead to unexpected behavior and possible software defects. There is also the possibility of regressions between firmware versions.

> *“ We have a lot of different supported firmware versions, and some features work on some versions and some on others. ”*

The firmware on the APs is not developed by the Wi-Fi Manager development team. However, the development team is still responsible for supporting and validating various versions of the firmware to work together with the Wi-Fi Manager. This required testing against multiple different firmware versions, increasing testing effort and complexity.

6.3 Benefits of the localized testing environment and test cases

This section describes the benefits identified related to the new test cases and the testing environment during the interviews.

6.3.1 Use of production dependencies

The existing test cases of the Wi-Fi Manager did not use actual production dependencies for services in the component test cases but instead relied on mocking them or using separate test versions of the dependencies.

Multiple participants expressed that the use of actual production dependencies increases their confidence in integration tests. Previously, there had been cases where the mocked components, that were used during testing, did not behave exactly the same as the components that are used in production.

> *“ There are two major things here [in the new test cases] that I like. Firstly, these are ’actual’ integration tests, using the actual containers. Secondly, these tests showcase better which outside services we are connecting to. ”*

6.3.2 Reduced feedback loop

A key benefit for the environment being on the developer laptops according to the participants was the reduced feedback loop. One participant stated that they often work on timing-related issues which often require multiple manual repetitions in the shared environment to verify. Working on a local environment would allow them to iterate faster and do changes that are needed in development.

> *“ I do a lot of manual repetitions for [debugging] timing related issues either in our software or in the communication with the APs. You just need to repeat it multiple times to recreate the problematic situations. Now I will not have to reserve an environment for the whole day. ”*

One participant estimated that the "time to beat" in the current situation is close to 15 minutes from making a change to the software to getting the shared environment to be in the correct state for testing. Therefore, they believed that the local environment could significantly streamline their development workflow.

6.3.3 Extendibility and flexibility

One of the participants had the possibility to use the new environment for their own development work. A benefit of using the environment for them was that the new environment was easily extendable for new use cases, such as adding new services that were not yet present in the shared environments.

> *“Firstly, you can easily simulate different situations, different configurations. Secondly, you can easily modify the environment. [Feature] I am working on can not be tested in the server environments as the features needed are not yet in there.”*

6.3.4 Onboarding new developers

Multiple participants felt that the existence of a localized environment could be really helpful for new developers who are not yet familiar with the Wi-Fi Manager. One participant stated that since the new environment does not require the use of Kubernetes, the learning curve for the environment might be less daunting than for shared environments with actual hardware.

> *“ For someone who is new to this project, this environment is a ’lottery win’. I myself am not that comfortable using Kubernetes, for example, and when using this environment, there is no need for me to use it. ”*

Another participant also stated that there would be a lower barrier to entry for new developers in the team, as messing up their local environment does not have an effect outside their own work.

On the other hand, another participant was concerned that the lack of an UI would make the test environment less approachable to new developers. According to the participant, new developers might like to see a UI when making changes, as UI can be better at communicating those changes to the user. Thus, the use of shared environments might be easier for new developers who are familiarizing themselves with the Wi-Fi Manager.

6.4 Limitations of the new environment and test cases

As the newly created test cases were based on the existing component test cases present in the Wi-Fi Manager, they had some of the same drawbacks as the original component tests.

One participant noted that it would be really important for debugging to be able to easily see which TR-069 messages are sent between the tester and the SUT. This would make it easier to debug possible problems and better document the purpose of the test case. This was not possible in the old test cases, and the same issue was carried out in the new test cases.

An existing problem with the old test cases that persists with the new integration test cases is the difficulty of handling test cases with multiple Wi-Fi APs with different firmware versions. The difficulty in testing different firmware versions of APs is that the message sequences of TR-069 and the content of the messages can be different. This means that some test cases may fail with certain AP firmware versions due to differences in message sequences and the way the protocol is implemented.

The localized environment and test cases do not allow for testing with multiple APs. Instead, this would still need to be done in a shared remote server environment running the Wi-Fi Manager.

Multiple participants raised concerns about the use of mock services as supporting services for the Wi-Fi Manager in the localized testing environment. According to one participant, the environment cannot provide them with full confidence that their changes work, as there can be situations where the mocks do not match the way a production environment works. This means that there would still be a clear need for testing using the shared remote server environments.

> *" If we can trust [the mocked services], then this testing would be adequate. But as I think that is not the case, it is still better to do the final verification in a real environment. "*

Another limitation was that there were parts of the Wi-Fi Manager that were not present in the localized test cases. The main component was the HTTP proxy, which was used in the localized environment but was not tested in the new test cases.

7 Discussion

This chapter summarizes the results of the thesis and how they relate to the research questions that were established at the beginning of the thesis. The results align with the existing literature on software testability with some notable factors missing from the developers' responses. The results emphasize the importance of testability factors such as controllability, observability, and amount of coupling between software components. These results are further expanded in chapter 7.1.

There are further considerations that are important when designing testing environments in similar domains as the one this study was carried out in. These considerations are further discussed in chapter 7.2.

Finally, chapter 7.3 discusses the limitations of the study to provide appropriate context for interpreting the results and their applicability.

7.1 RQ1: Testability aspects

The testability aspects that developers raised during interviews align mostly with those presented in the literature. From the factors presented in chapter 2.2 most of the factors were also deemed important by the developers in the case team. An important finding for the thesis is that some theoretically important factors from the literature were not explicitly mentioned by any of the participants. These were *inheritance*, *reliability*, *availability* and *maintainability*.

The findings support the definitions of testability by Binder (1994), particularly in terms of *controllability* and *observability*. The concerns of the participants about coupling and complexity further reflect Garousi et al. (2019) expanded list of testability factors. This does indicate that the foundational concepts of testability apply also in the context of systems such as the Wi-Fi Manager.

Inheritance was not mentioned by the participants. As inheritance is highly related to object-oriented programming, it is logical that it was not mentioned since the code in the Wi-Fi Manager is not object-oriented. This does not mean that inheritance is not an important factor to consider when developing software with testability in mind, but this highlights that the existing literature related to software testability seems to be focused on object-oriented programming. Inheritance is related to complexity according to Garousi et al. (2019), which is applicable to most software systems.

Availability and *reliability* are by their nature more interesting due to the fact that the participants did not explicitly mention them when asked what makes software testable. It could be that the participants understood that software testability meant explicitly the attributes of the software itself.

The way questions were asked could also have been a factor. Participants could have been asked to consider testability more holistically and this could have affected the responses. From the responses it may be seen that participants interpreted software testability to mean the intrinsic properties of software, and not the surrounding practices or testing infrastructure. However, both availability and reliability can be seen as aspects that affect the testing environment and not the software itself, so it might not necessarily come to mind when a developer is asked what makes the software testable.

Availability and reliability were mentioned by multiple participants when discussing the limitations that using shared environments had on testing. Thus, it can be interpreted that availability and reliability were still important testability factors for the developers at the case team, but were not necessarily considered as a factor that makes software testable or untestable.

Similarly, *maintainability* was not explicitly mentioned by developers in any part of the interviews or the survey. It may be the case that maintainability is seen as a fundamental aspect of good software development practice and thus is not explicitly mentioned by the participants. The participants mention many related factors such as complexity and coupling. This may be due to them being more concrete factors than maintainability and thus more present in their day-to-day activities. Thus, it can be still seen as an important factor in software testability while not explicitly mentioned by the participants.

The findings suggest that, while there are a wide range of factors that affect software testability, not all of them are equally visible or prioritized in day-to-day development work. This highlights an important difference in perspective worth further exploration and research.

7.2 RQ2: Factors influencing testing environment adoption

It is clear from the findings that for a software system like the Wi-Fi Manager, using production hardware in testing is vital. As there are multiple different firmware versions that need to be supported and there might be regressions between the firmware versions, automated testing becomes more challenging. Thus, there are instances, as the participants mentioned, where manual testing in a shared environment can not be avoided. This is also the case for the services that the Wi-Fi Manager depends on. It does not make sense to include full versions of the services that the software depends on in the local environment, but it would be interesting to see, what benefits could be gained from including some of the surrounding services.

A localized testing environment can still provide developers with better controllability and observability over the software they are developing. This indicates that the introduction of a localized testing environment can have a positive impact on the testability of an existing software system as a whole.

The localized environment is easier and safer to modify than the remote environments as there is no risk of affecting the work of other members in the team by your own testing. However, the actual impact of this type of environment can only be seen after extensive use by the development team.

However, testing in production-like environments still remains needed as the localized environment cannot fully replicate the behavior of a production environment. There might be potential missing edge cases or integration issues that only appear in real deployments.

7.3 Limitations

An inherent limitation on the way the study was conducted was the use of a prototype in interviews. It is a known phenomenon that interviewees often want to please the interviewer. In the case of this study, the researcher was a colleague of the interviewees. This can also affect the level of criticism that was expressed in the course of the interview.

In addition, the use of a prototype can have its own set of limitations. Although the feedback from the participants was positive, the actual limitations of the software artefact can only be seen after extensive use. As a result, the findings gathered from initial reactions may not fully capture long-term usability, stability, or real-world adoption challenges. This was also highlighted by one of the participants during an interview. Despite that, the use of the software artefact was the only possibility considering the time limitation on carrying out a Master's thesis project.

Another limiting factor is the participant selection for the interviews and the sample size. There were only five participants, all colleagues from the same team. This may limit the diversity of perspectives in the responses given by the participants. Thus this can limit how well the results of this study are applicable to other teams and domains. However, the results from the interviews mostly align with existing literature on software testability.

8 Conclusions

This thesis contributes practical insight into how testability can be improved in a software system that interacts with hardware by introducing a localized testing environment.

We had two research questions. Firstly **RQ1** *What aspects affect the testability of software*, and secondly **RQ2** *What factors influence developers' willingness to adopt the integration test environment*.

To address these questions, a localized integration testing environment and associated tests were designed and implemented. The environment enabled developers to test the Wi-Fi Manager with production-like configurations on their local machines.

In terms of **RQ1**, this study identified relevant testability factors from the literature and highlighted that some factors might not be present in the day-to-day work of a developer. Some factors, such as inheritance, are highly specific to object-oriented systems, and this might not be applicable to all development environments. However, the findings are mainly in line with existing literature on software testability. For example, factors such as *controllability* and *observability* are still vital in the context where this study was carried out.

In terms of **RQ2**, there were identified benefits for using localized testing environments for the developers in the case team. These included faster feedback loops, better support for modification and experimentation and reduced dependency on shared remote server environments. However, there were clearly identified limitations, such as the lack of support for multiple APs and reliance on mocked services. These issues limited the developers' confidence in using the localized environment for validation. Thus, there is still a need to use shared remote environments for testing with physical hardware.

References

Ackerson, Tomas and Dan Fernandez (Mar. 2022). The Testing Pyramid: How to Structure Your Test Suite. URL: https://semaphore.io/blog/testing-pyramid.

Binder, Robert V. (1994). Design for testability in object-oriented systems. *Communications of the ACM* 37.9, 87–101. URL: https://dl.acm.org/doi/pdf/10.1145/182987.184077.

Braun, Virginia and Victoria Clarke (2012). Thematic analysis. *APA handbook of research methods in psychology, Vol 2: Research designs: Quantitative, qualitative, neuropsychological, and biological.* Ed. by Harris Cooper, Paul M. Camic, Debra L. Long, A. T. Panter, David Rindskopf, and Kenneth J. Sher. Washington: American Psychological Association, 57–71. ISBN: 978-1-4338-1005-3. DOI: 10.1037/13620-004. URL: https://content.apa.org/books/13620-004.

Bures, Miroslav, Matej Klima, Vaclav Rechtberger, Xavier Bellekens, Christos Tachtatzis, Robert Atkinson, and Bestoun S. Ahmed (July 2020). Interoperability and Integration Testing Methods for IoT Systems: a Systematic Mapping Study. DOI: 10.48550/arXiv.2007.11308. URL: http://arxiv.org/abs/2007.11308.

Cohn, Mike (2010). *Succeeding with agile : software development using Scrum*. 1st edition. The Addison-Wesley signature series. Upper Saddle River, N.J: Addison-Wesley. ISBN: 978-0-321-66053-4.

Dennis, Alan, Roberta M Roth, and Barbara Haley Wixom (2012). *Systems analysis and design*. 5th ed. Place of publication not identified: John Wiley. ISBN: 978-1-118-05762-9.

Fowler, Martin (n.d.). Test Pyramid. URL: https://martinfowler.com/bliki/TestPyramid.html.

Foxworthy, Jared (May 2025). A Guide to Hardware-in-the-Loop (HIL) Testing in 2025. URL: https://www.opal-rt.com/a-guide-to-hardware-in-the-loop-testing-in-2025/.

Freedman, R.S. (June 1991). Testability of software components. *IEEE Transactions on Software Engineering* 17.6, 553–564. ISSN: 00985589. DOI: 10.1109/32.87281. URL: http://ieeexplore.ieee.org/document/87281/.

Garg, Umang and Abhishek Singhal (Jan. 2016). An estimation of software testability using fuzzy logic. *2016 6th International Conference - Cloud System and Big Data Engineering (Confluence)*, 95–100. DOI: 10.1109/CONFLUENCE.2016.7508094. URL: https://ieeexplore.ieee.org/document/7508094/?arnumber=7508094.

Garousi, Vahid, Michael Felderer, and Feyza Nur Kılıçaslan (Apr. 2019). A survey on software testability. *Information and Software Technology* 108, 35–64. ISSN: 09505849. DOI: 10.1016/j.infsof.2018.12.003. URL: https://linkinghub.elsevier.com/retrieve/pii/S0950584918302490.

Ghafari, Mohammad, Markus Eggiman, and Oscar Nierstrasz (Aug. 2019). Testability First! URL: http://arxiv.org/abs/1908.01476.

Johannesson, Paul and Erik Perjons (2021). *An Introduction to Design Science*. Cham: Springer International Publishing. ISBN: 978-3-030-78131-6 978-3-030-78132-3. DOI: 10.1007/978-3-030-78132-3. URL: https://link.springer.com/10.1007/978-3-030-78132-3.

Kedemo, Maria (2015). Testability Awakens: Moving Testability into New Dimensions. *Testing Trapeze magazine*. URL: https://mkedemo.wordpress.com/wp-content/uploads/2015/12/testability-awakenstestingtrapeze-2015-december.pdf.

Khan, Mohd Ehmer and Farmeena Khan (2012). A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications* 3.6.

Myers, Glenford J, Tom Badgett, and Corey Sandler (2011). *The art of software testing*. 3rd. John Wiley & Sons.

Voas, Jeffrey M. and Keith W. Miller (1995). Software Testability: The New Verification. *IEEE software* 12.3, 17–28. URL: https://web.archive.org/web/20040825124151id_/http://www.cigital.com:80/papers/download/ieeesoftware95.pdf.

Wich, Klaus and John Blackford (2020). TR-069 Amendment 6 Corrigendum 1 CPE WAN Management Protocol. URL: https://www.broadband-forum.org/pdfs/tr-069-1-6-1.pdf.

WireMock (Feb. 2025). Mocking gRPC services. URL: https://wiremock.org/docs/grpc/.

A Survey questions

1. How many years of experience do you have in software development?
   - 0 - 2 years
   - 3 - 5 years
   - 6 - 10 years
   - 10+ years
2. How long have you been involved in the development of the Wi-Fi Manager?
   - Less than a year
   - 1 - 2 years
   - over 2 years
3. How often do the features or bug fixes you work on require multiple rounds of testing in a lab environment?
   - Frequently
   - Occasionally
   - Rarely
   - Never
4. How often has the lack of an available lab environment been a blocker for your work?
   - Frequently
   - Occasionally
   - Rarely
   - Never
5. Are there specific areas of the Wi-Fi Manager that are difficult to address with automated testing? If so, please describe them. **(Open-ended)**
6. What resources or changes would make it easier to test the Wi-Fi Manager? **(Open-ended)**
7. What types of issues are difficult to diagnose with our current testing approach? **(Open-ended)**
8. How confident are you that our current automated testing will catch major issues related to the Wi-Fi Manager?
   - Extremely confident
   - Very confident

- Moderately confident
- Slightly confident
- Not at all confident

9. What contributes most to how confident you feel about our automated testing? **(Open-ended)**
10. How often do you encounter false positives (tests that fail without a real issue) or false negatives (tests that pass despite an issue)?
11. Anything else you would like to add? **(Open-ended)**

B Interview guide

Interviews were semi-structured in nature, so the following questions serve as a guideline to allow for more nuanced discussion. Additional questions were asked when deemed necessary. The interviews were held in Finnish.

Instructions for the interviewer are marked with *cursive*.

Introduction

Begin the interview with recap of informed consent and the research topic

- What is your current role?
- How long have you been working in your current role?
- How long have you been working in the software industry and in which type of roles?

Testability and testing of the Wi-Fi Manager

- How would you define software testability?
- What steps are involved when you develop a new feature for the Wi-Fi Manager?
  - What type of testing do you do?
  - Do you use the lab environment?
  - Are there any challenges regarding the use of the lab environments?
- What areas of our software do you think is the most prone for defects?
- In your work, what type of testing takes the most amount of time?
- Are there areas in our development where testing could be improved?
- Have you specifically worked with our integration tests?
  - What are the specific issues that you have encountered when using or writing integration tests utilizing our current suite?
  - What works well? What makes a good integration test case?

Evaluating the new test cases

The interviewer shows new test suite that has been built utilizing the new localized environment. 10 test cases are run in succession and after the interviewee is given an option to look at the test cases and produced logs.

- Is there something you would like to change about these test cases?

- Are there issues our current testing does not catch that would be caught by this testing?
- Can you think of issues that can not be addressed with this type of testing?

Evaluating the new test environment

The interviewer shows the local test environment

- Could the testing environment reduce the need for using the lab environment for you? Why/Why not?
- How would you use this type of environment?
- What would make you not want to use this environment?

Ending the interview

- Is there anything else you would like to add regarding the topics we covered?