Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Computer Science with Specialization in Software Engineering 30.0 credits

MIGRATING AND EVALUATING A TEST ENVIRONMENT FROM A DYNAMICALLY TO A STATICALLY TYPED LANGUAGE

Marija Djordjevic
mdc17001@student.mdh.se

Hamza Sabljakovic
hsc16001@student.mdh.se

Examiner: Jan Carlson
Mälardalen University, Västerås, Sweden

Supervisor: Wasif Afzal
Mälardalen University, Västerås, Sweden

Company supervisor: Thomas Sörensen,
Westermo Research and Development AB,
Västerås, Sweden

May 21, 2018
Abstract

Maintenance takes considerable effort in software development. Consequently, improving software maintainability can reduce costs and improve future maintenance of software. Previous research on the topic of software maintenance suggests that a type system might have impact on software maintainability. More precisely it indicates that statically-typed languages have positive impact on software maintenance in a long run. However, the previous work on the topic only takes bug fixing as an indicator of maintenance while ignoring others. Therefore, this thesis is interested in answering how the typing system affects refactoring and code navigation as two representative software maintenance activities. Furthermore, in consideration of positive impacts of static typing and increase in dynamic languages popularity in last two decades, the second aspect of this thesis is interested in software migration from dynamically typed to statically typed language. By following the process of migration, this thesis provides several contributions. The first one is state-of-the-art research on the topic of testing frameworks for embedded systems, which is used as an input for the migration process helping to produce a set of guidelines for a software migration. Finally, while it was previously shown that type system has a positive impact on bug fixing, the experiment done as a part of this thesis demonstrated that there is no difference in time it takes to refactor or navigate code except in one of the tests where statically-typed language showed better performance.
# Table of Contents

1 Introduction .................................................. 6  
2 Background .................................................. 7  
3 Problem formulation ......................................... 8  
4 Method .......................................................... 9  
5 State-of-the-art in implementing testing frameworks for embedded systems .......................................................... 10  
  5.1 Background .................................................. 10  
  5.1.1 Software testing life cycle ................................ 11  
  5.1.2 Black and White box testing methods ..................... 11  
  5.1.3 Clarifying terminology .................................. 12  
  5.2 Selection criteria ........................................... 13  
  5.3 XIL ......................................................... 14  
  5.3.1 Model-in-the-loop ....................................... 14  
  5.3.2 Software-in-the-loop .................................... 15  
  5.3.3 Hardware-in-the-loop .................................... 15  
  5.3.4 HIL Lab ................................................. 16  
  5.3.5 XIL integration ......................................... 16  
  5.4 Test case generation ....................................... 17  
  5.5 Limitations .................................................. 18  
6 Prototype development ........................................ 19  
  6.1 Introduction ................................................ 19  
  6.2 Defining testing context .................................... 19  
  6.3 The framework architecture and testing process ............. 20  
  6.3.1 Scope of the prototype .................................. 22  
  6.4 Challenges encountered during the prototype development . . . 22  
  6.5 Prototype implementation ................................... 23  
7 Software modernization ....................................... 26  
  7.1 Reusing components from a legacy software .................. 26  
  7.2 Migrating to a new programming language ..................... 27  
  7.3 White-box and black-box migration ........................ 27  
  7.4 Reverse engineering ........................................ 28  
  7.5 When to perform software migration? ........................ 29  
  7.6 Software modernization challenges ......................... 30  
  7.7 Proposed method ............................................ 31  
    7.7.1 Understanding of the current system .................... 32  
    7.7.2 Creating an abstract representation of the system ....... 33  
    7.7.3 Improving the abstract representation of the system ... 34  
    7.7.4 Implementation and testing .............................. 35  
    7.7.5 Final remarks .......................................... 35  
  7.8 Statically-typed to dynamically-typed language ............. 36  
  7.9 Related work on software modernization .................... 36  
  7.10 Applying the proposed method ............................. 37  
8 An experiment: the effects of the type system on software maintainability .................................................. 40  
  8.1 Background .................................................. 40  
    8.1.1 Statically-typed languages and dynamically-typed languages 40  
    8.1.2 Software maintainability ................................. 41  
  8.2 Related work on the impact of type system on software maintainability .................................................. 44  
  8.3 Experiment design ............................................ 45  
    8.3.1 Goal ................................................... 45
List of Figures

1. The thesis process .................................................. 9
2. V model: software development and testing activities [1] ................... 12
3. Test process activities .................................................. 13
4. Accuracy - cost and time trade off in different XIL levels (inspired by [2]) .................................................. 16
5. Block diagram of hardware in the loop lab [2] .......................... 17
6. Testing process on local test setup ..................................... 20
7. Testing process on the network hardware .............................. 21
8. Fawlty architecture ....................................................... 22
9. The scope of the prototype in comparison with the legacy system ......... 23
10. Fawlty class diagram ................................................... 24
11. White-box migration process ........................................... 28
12. Steps of the proposed method .......................................... 32
13. Sub-steps of understanding the current system ....................... 33
14. Sub-steps of creating an abstract representation of the system ........ 34
15. Sub-steps of improving the abstract representation of the system .... 35
17. Experiment participants division ...................................... 51
18. Results related to the Task 1 ......................................... 55
19. Results related to the Task 2 ......................................... 55
20. Results related to the Task 3 ......................................... 55
21. Results related to the Task 4 ......................................... 55
22. Combined results of the experiment .................................... 56
23. Combined results of the experiment .................................... 57

List of Tables

1. Testing models with respect to abstraction levels ....................... 15
2. Data gathered during the experiment (time in seconds) ............... 53
3. Experiment data with applied mean substitution (time in seconds) ... 55
4. Results of the Shapiro-Wilk normality test for each task .................. 58
5. Results of the hypotheses testing ...................................... 59
Acknowledgments

We would like to thank the company Westermo R&D for the possibility of doing the case study within the company and to conduct the experiment with Westermo employees as participants. Furthermore, we would like to thank our supervisors Thomas Sörensen and Wasif Afzal for the support and help throughout the whole thesis process.
1 Introduction

The research on the topic of software maintainability with respect to the typing system suggest that static typing increases software maintainability [4, 5, 6]. Yet, GitHub’s report shows that dynamically typed languages: JavaScript, Python, PHP and Ruby take four of the first five places when it comes to the popularity of programming languages [7]. In consideration of the popularity of the dynamic languages, their suitability for prototyping and fast development, and, on the other hand, benefits of statically-typed languages with the respect to the software maintainability, there is an interest among stakeholders of legacy systems, and fast-growing projects for migration to the statically-typed languages. However, software migration is very complex and time and resource consuming process. Furthermore, it is not only an engineering task, but also, an organizational and business challenge. Therefore, all stakeholders need to be familiar with possible drawbacks and benefits of a migration, as well as, possible challenges that may be encountered. Consequently, investigating the practices and possible difficulties when migrating a legacy software contributes to the body of knowledge.

The goal of this thesis is to identify and propose a method for migrating a legacy software from a dynamically typed to a statically typed language and measure type system effects on the aspects of software maintainability, refactoring and code navigation. As a subject of migration a concrete legacy, still in production software (a test framework) has been used. The study was a collaboration with a local company that provided the subject software and later helped in organizing the experiment for measuring the impact of a type system.

Successful migration does not only consist of porting a software from one language to another but also takes the leverage of a fresh start and opens up possibilities for taking advantage of already implemented solutions proposed by the research community. Consequently, the first output that this thesis provides is a state-of-the-art literature review on the topic of testing embedded systems.

Once the mentioned literature review was done, next step was to perform a migration. Together with the findings from the related literature on software migration and practical experience gained from the migration process itself, the second output of this thesis is a method on how to do a software migration.

Finally, an experiment is conducted in order to evaluate the work done and investigate whether the migration has introduced improvements with respect to software maintainability. Moreover, another reason for conducting an experiment is to contribute to the research done on this topic. All research we are aware of on the topic of comparing the statically-typed languages with the dynamically-typed languages, have used Java and Groovy languages as representatives for the research. Therefore, use of another programming languages could bring diversity and therefore be beneficial to the body of knowledge. Furthermore, the experiment conducted in this thesis is performed using real-world software, in contrast to the existing research where the subject software was developed for the purposes of studies. Finally, the previous work has been measuring maintainability with bug fixing related tasks, while this experiment examines refactoring and code navigation tasks as two important tools for maintainability. Additionally, an informal qualitative questionnaire is conducted in order to provide additional information that could be beneficial in understanding experiment results and developers’ preferences.

This thesis report is organized as follows: it starts with the Background section 2, followed by the next two sections, Problem Formulation 3 and Method 4 where the research questions are presented along with detailed steps that will be taken in order to answer them. Next, section 5 is outlining findings from the literature review. Section 6 addresses the prototype development followed by section 7 were the migration method is proposed. The next two sections are experiment presented in the section 8 and informal quantitative questionnaire written in the section 9. Finally, the conclusions are drawn in the section 10.
2 Background

The thesis can, in essence, be divided into three separate units due to the fact that it is composed of three separate research goals. Even though these units are highly connected, they are not very much alike. For that reason, it is decided that every unit have its own background section focusing only on presenting information needed for understanding the remaining text in the section. We believe that the report will be easier to read and comprehend if the text is organized in such a way that each section tells its own story. However, we are aware that this can have a negative impact on the reader when trying to understand problem formulation or method sections and therefore suggest reading background sections on the topic the reader does not feel comfortable with. The first background section is on the topic of testing and testing frameworks and can be found in the 5.1, next, software migration process is introduced in first six subsections of the section 7. Finally, a background on statically-typed and dynamically-typed languages as well as software maintainability is provided in section 8.1. The same organization decision is applied to the related work sections.
3 Problem formulation

This thesis is interested in tackling multiple related problems. The overall problem is how to perform a successful migration. However, the migration process further benefits from preparation and evaluation of its effects. The preparation for the migration is a question for itself and therefore a separate research question presented later in the text. In this context, the preparation assumes exploring the related literature with the goal of finding relevant information that can serve as an input for the next step. Next, once the literature is reviewed and applicable information is collected, it is time for the migration itself. Aspects of the migration that are subjects of interest to this thesis are determining whether the migration is worth doing. If the answer is yes it raises the question how to perform it. Finally, after the migration is completed it is of interest to evaluate the of migration. In this particular case analyzed and developed for the purpose of this thesis, the migration is from a dynamically-typed to statically-typed language with the assumption of improving maintainability. There are numerous researchers indicating that the statically-typed source code has better maintainability over the dynamically-typed source code. However, the maintainability is a broad term and the previously mentioned papers evaluate only the bug fixing aspect of it. This fact leaves a research gap to be filled. Therefore, this thesis aims to address the missing aspects of the effects of typing system on maintainability while testing refactoring and code navigation as two essential actions performed by developers during software maintenance.

In the interest of addressing challenges mentioned in the paragraph above, we propose the following three questions:

**RQ 1**: What is the existing state-of-the-art in implementing testing frameworks for embedded systems?
**RQ 2**: What are the practices and possible difficulties when migrating a legacy test environment application from a dynamically-typed to a statically-typed language?
**RQ 3**: What is the difference in the time needed for refactoring and navigating between the new system written in a statically-typed language in comparison with the legacy implementation in dynamically-typed language?

The purpose of the first question’s answer is to provide the supporting knowledge needed for a successful migration. In consideration of the software migration’s complexity and expensiveness as a process lays the benefit of answering the question of the migration’s benefits and drawbacks as well as its challenges. Next, the purpose of providing guidelines is to contribute to the field of white-box migration which lacks the guidance on how to perform a migration. Finally, the third research question has two purposes. The first one is to evaluate the migration performed and the second one is the contribution to the research filed of effects that typing system has on software maintainability.
4 Method

This section presents methods used for answering the questions defined in the problem formulation section 3. The first question is interested in the state-of-the-art in the field of testing environments and therefore the method for answering this question is straightforward: scientific literature review.

The next research question asks how to perform a software migration from a dynamically to a statically-typed system. In order to do so an industrial case study was conducted. The case study was a collaboration with Westermo, a Swedish company specialized in industrial network solutions. Westermo has an in-house built automation testing framework written in Python 2, a dynamically typed language and it is interested in the process of migrating it to the Go programming language, a statically-typed language. The objective of the case study is to identify challenges and best practices when migrating a test environment software from a dynamically-typed language to a statically-typed language from the point of view of a software developer. During the above-mentioned process, report write up will be continuously done according to the guidelines of structuring case study results in software engineering [8]. Therefore, a partial migration of the testing framework to the Go programming language is going to be performed. This process will be inspired by the legacy implementation and guided by discoveries from the research question one.

Next, in the interest of measuring aspects of software maintainability, the third research question, an experiment is conducted where the new implementation is tested and compared with the legacy one. One reason for choosing an experiment is its statistical significance. The second reason being that it overcomes the problem of difference in size between the two systems. Put in other words, it is hard to perform a reliable static analysis of the codebase due to the size difference. Therefore, the experiment is more suitable where the impact of the software size can be mitigated. Furthermore, in a case of an experiment, it is possible to adjust the measurements to show the dependence on the typing system.

Finally, an informal qualitative questionnaire is conducted in order to complement the insights gained during the experiment. Even though it is not a formal scientific method, it helps understanding the empirical results and produces further information on developers preferences. In order to further clarify, the thesis process is depicted in figure 1.

![Figure 1: The thesis process](image-url)
5 State-of-the-art in implementing testing frameworks for embedded systems

The aim of this section is answering the first research question and getting insights in “what is the existing state-of-the-art in implementing testing frameworks for embedded systems?”

In order to perform a migration of a test environment to another programming language, it is important to consider the state-of-the-art in the field in order to take advantage of modern approaches to similar problems. Therefore, the first thing that is covered in this thesis is investigation of the state-of-the-art in the field of test environments with a special focus on the embedded systems.

Seen that the literature study with the goal of perceiving related work within the topic, and the one with the intention of presenting state-of-the-art overlap to some extent, this section will cover and fulfill both objectives on the topic of the testing frameworks for embedded systems.

There are numerous definitions of embedded system with the goal of representing it from different perspectives. One way of comprehending embedded system is to see it as a combination of software and hardware designed for a specific purpose as a standalone or as a part of a larger system. Furthermore, a common feature of all embedded systems is the ability to interact with the physical world via sensors and actuators [9]. Having in mind that embedded systems are usually developed for a specific purpose, testing them requires approaches tailored for specific cases. However, embedded software faces problems that are common for testing in general. Therefore, in the following sections, we have addressed both, testing specific for embedded systems but also techniques applicable in testing in general.

5.1 Background

One of the famous clarifications of the purpose of testing is given a long time ago by Dijkstra: “Testing shows the presence, not the absence of bugs”. Therefore, it is impossible to have a complete confidence in the software. However, testing increases the level of software reliability, and consequently, it is an important factor in software engineering. The definition of software testing as given in glossary produced by International Software Testing Qualifications Board (ISTQB) [10] is the following:

“...The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.”

However, the presented definition does not say anything about the actual techniques and approaches when it comes to software testing. Testing can be manual, performed by a human where the tester or developer checks if the system behaves as it is expected. Besides the manual approach, testing can be automated in which case a specialized software is used to perform actions of running tests, comparison between test results and expected values and generation of reports. Automated testing requires preparation of tests, however, once it is done, they can be reused over and over again with occasional modifications. In short time, manual testing can have advantages over the automated testing since it usually does not require as much as initial setup like automated tests do. However, if observed in a long run when tests cases have to be repeated multiple times it is shown that automated testing saves time and increases software quality [11].

The rest of this report is focused on automated testing; more precisely it goes into details of...
testing frameworks, the supportive element of automated testing.

5.1.1 Software testing life cycle

In the last two decades, software product stakeholders started to recognize and appreciate the role of automated testing and testing in general in the software development process. Software testing is no longer an activity that starts once the development phase is completed. As SWEBOK guidelines recommend “planning for software testing should start with the early stages of the software requirements process, and test plans and procedures should be systematically and continuously developed—and possibly refined—as software development proceeds”.

During its development, software passes through different phases [12]. For every development phase there are corresponding tests. Tests are not intended to focus only on the source code, they can be derived from requirements, specifications and design artifact as well. Therefore, different levels of testing, based on software development process and activities, have been defined, as presented in the book Introduction to Software Testing [1] and SWEBOK [13] guidelines:

- **Unit testing** - assess the software building elements, with respect to the implementation, in a complete isolation. Usually, unit testing is done by the developers who were responsible for developing particular functionality or unit.

- **Module testing** - the next phase of the testing process that comes after the unit testing is module testing. As the name suggests, module testing focuses on verification of system module as a whole.

- **Integration testing** - due to the complexity a large portion of software is developed in a component fashion and later integrated into a system. However, software component developed and tested in isolation can result in new unexpected behaviors once integrated with the other components. In order to discover and prevent this type of behavior, integration testing is performed. Integration testing is usually done by incremental steps as the software components are integrated rather than performing it at the end when everything is integrated at once.

- **System testing** - is focused on testing the entire system. It is not expected to identify many defects by performing system testing for the reason that many of them will be identified by unit and integration tests. Yet, system testing is usually conducted in order to evaluate the system with respect to the nonfunctional system requirements (e.g. security, speed, reliability). Furthermore, it is an appropriate level of testing for validating hardware devices and external interfaces to other applications.

- **Acceptance testing** - the aim of conducting an acceptance test is to determine if the system fulfills the acceptance criteria. In the other words, the acceptance test is responsible for comparing the software results with its requirements. This stage of the testing process is usually done by the system customers.

The figure 2 depicts previously outlined testing levels with respect to the software development phases. It is usually addressed as a “V Model” of software development. Additionally, it supports previously elaborated importance of early preparation of tests, even when their implementation and execution is not feasible.

5.1.2 Black and White box testing methods

To wrap-up the background section of software testing introduction, this subsection focuses on discussing the difference between white and black box testing. Black and white-box concepts are
quite straightforward where black-box testing can be seen as an action where test developers are not aware of internal mechanics (source code, system architecture etc.) of the test subject. Put in other words, test developers do not have any knowledge of the code implementation and treat the software as a black box. On the other side, when performing a white-box testing, knowledge of the internal structure and details of the test subject is required. Therefore, tests can be designed with the goal of putting the system in a particular state. In such a way, the main intent is to test as many as possible system states [11].

Aside from the testing that is presented so far, the dynamic testing, it is important to note the role of static analysis. In contrast to dynamic testing, it does not require a running code, therefore, program analysis can be performed earlier. Static program analysis and testing are complementary in a way that an error found by testing is evidently present in the system, however, testing cannot guarantee the absence of error. On the other hand, an error discovered by static analysis may be a false positive.

5.1.3 Clarifying terminology

Before getting into details of testing software it is important to explain the terminology used. Terms such as framework, test framework and test environment can be ambiguous and represent different things to different readers. Therefore, in order to avoid confusion, the next paragraph addresses all of the previously mentioned terms.

As stated in the Oxford dictionary, a framework is “An essential supporting structure of a building, vehicle, or object”. If we apply this definition in the context of software, a software framework is a backing structure that provides skeleton architecture and common functionalities for simpler and faster development of a specific software. In contrast to a library which is a set of related functionalities, software framework enforces its architecture and enables reuse of design patterns which are provided. A specialized type of a software framework is a testing framework. Testing frameworks usually provide supporting architecture for automated testing by facilitating
a wide range of activities from writing test scripts to their execution and result processing. Next significant term, *test environment*, can be seen as a software and/or hardware support needed for a successful test execution. In most of the cases, it is possible to configure it via the corresponding test framework to satisfy test case requirements. Also, another important concept in testing embedded software is *Device Under Test* (DUT). As the name suggests, it is a device that can be a physical or virtual machine that is subject to testing. This adds complexity to testing embedded software since the test framework has to know how to interact with the DUT.

Having all these commonly used, yet rarely precisely defined, terms briefly explained, the next section goes into more details regarding state of the art in the topic of testing frameworks for embedded systems.

Figure 3 shows the relation of test process activities. As it can be seen from the figure, the first step is for test developers to write a test script. Once a test script is written its execution and evaluation is delegated to the testing framework. Besides execution, the testing framework is responsible providing test scripts with common functionalities that can be reused and for controlling the test environment. Next, when a single test case or group of test cases (test suite) is executed the results are returned back to the subject who invoked it and/or stored to some type of permanent storage, depending on the test framework configuration.

### 5.2 Selection criteria

When it comes to the paper selection criteria for doing this state-of-the-art research, the main focus is put on the papers from last 10 years starting from 2008. Due to the previously explained
specificity of embedded software, aside to the term of testing frameworks, the search is extended to the topics different from frameworks for embedded systems but still related to the testing process. However, the findings from related topics covered can be applied to the topic of testing framework for embedded systems. The search included terms of embedded software testing environment as well as testing embedded software in general with the goal of finding techniques for enhancing testing frameworks and/or environments. The mentioned search terms were queried against the major scientific databases, IEEE Xplore, ACM and Springer. After analyzing the search results the three most frequent papers’ focus subfields have been identified. Identified fields are model-based testing, test case generation (as a possible feature of a testing framework) and X-in-the-loop where X can stand for software, hardware or model. Following subsections are dedicated to each one of them.

5.3 XIL

One of the challenges when testing an embedded system software is that the physical component or device might not be developed at the early stages of software development. Furthermore, hardware might be too expensive to be given to the test engineers due to the possible failures or destruction of equipment. Yet, testing has a huge impact on the software quality, and therefore experts advocate testing from the beginning of the development. Having this in mind, the simulation-based testing is one of the possible approaches for solving this problem by replacing the hardware with equivalent software simulation. Additionally, a simulation-based testing is not limited only to the early stages of the development/testing lifecycle, it rather can cover a broader range of testing environments with a different level of abstraction to fit the project needs. In the other words, different development stages require different testing environments, therefore, there are three simulation types that got the most attention from the scientific community: model-in-the-loop, software-in-the-loop and hardware-in-the-loop. Different definitions of the previously mentioned terms are discussed by many authors, accordingly, a brief overview of them and the summarized view is described in the following sections.

5.3.1 Model-in-the-loop

Model-in-the-loop (MIL) is a part of a wider field of software testing: model-based testing. It is one of the most frequent types of testing addressed in scientific papers, as it is shown by recent mapping study research [14]. Its main benefit is based on the fact that it can be run, and thus discover faults in software, in early phases of the development. However, this paper is going to focus on the model-in-the-loop testing, as a significant level of XIL testing, without addressing model-based-testing in details because it does not align with the goals of this thesis.

In the model-in-the-loop environment, where embedded system under test is viewed as a composition of a controller and a plant (environment of an embedded system, including sensors and/or actuators), the entire system, both the controller and the plant are simulated. The main purpose of the simulation is testing just the software functional requirements without the physical devices [15], [16]. The most straightforward definition is stated by [17]: “the implementation model of the system is examined in with the modeling software running on development hardware (local developer machine) and a simulated environment”.

Since embedded software is becoming more and more complex, its development and testing life cycles are also becoming particularly extensive. Therefore, the benefit gained from early testing is increasing as well. Accordingly, the main advantage of MIL is its feasibility as early as the modeling phase is finished. In that way, costly corrections can be moved to the early stages when they are less expensive and easier to perform. Furthermore, test cases made in this phase can be reused in later phases of development [16].
5.3.2 Software-in-the-loop

The term Software-in-the-loop is the one with the highest diversity among its definitions. Several definitions are proposed by [16], however, all of them agree that the actual code is not running on the target Electronic Control Unit (ECU) hardware, but on the PC in the simulated ECU. The advantage of this approach is the removal of the need for any target hardware components.

This results in numerous benefits, the first benefit is that it can speed up development process since software development and software testing do not have to wait for hardware to be developed and tested. Also, by eliminating the hardware component, faults discovered in the testing phase are only result of the erroneous software behavior. Next, it affects the development process since developers can run it on the local machine removing the need for deploying on the target hardware while still in the development phase. Besides deployment, running software on a local development machine allows faster debugging [2].

Furthermore, by reaching such high level of division and completely removing dependence on concrete hardware, it allows testing the system under development in safe conditions without introducing any risk to the test engineers as well as the testing environment. Additionally, some system functions or system test can require a very expensive equipment or destructive behavior on the equipment, which in this case can be thoroughly tested before the testing with actual hardware, when testing developers will have enough confidence in the system.

5.3.3 Hardware-in-the-loop

HIL gained a lot of interest in the last couple of years, nevertheless it is being utilized by flight simulation systems since mid twenty century. Besides aircraft and aerospace, other industries started taking advantage of this approach. Today, HIL is applied in Vehicle systems, power system, robotics, marine systems etc. In contrast to the previously discussed x-in-the-loop approaches, by including the target hardware as well as the software, Hardware-in-the-loop tests are more accurate since the only part that is simulated is the environment. The clearest definition provided by [17]: “the software integrated into the target hardware (e.g. an embedded controller) is examined in a simulation environment”.

To summarize and clarify the difference between these testing models and their abstraction levels, the table 1 below outlines which parts of the embedded system under test are simulated and which ones are the actual ones (the same as in the final product).

<table>
<thead>
<tr>
<th>Testing environment</th>
<th>Software</th>
<th>Controller (hardware)</th>
<th>Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIL</td>
<td>/</td>
<td>Simulated</td>
<td>Simulated</td>
</tr>
<tr>
<td>SIL</td>
<td>Actual</td>
<td>Simulated</td>
<td>Simulated</td>
</tr>
<tr>
<td>HIL</td>
<td>Actual</td>
<td>Actual</td>
<td>Simulated</td>
</tr>
</tbody>
</table>

Table 1: Testing models with respect to abstraction levels

In comparison with previously mentioned testing environments (MIL and SIL) HIL is more accurate by reason of including the target hardware as well as the software. Differently, HIL is more time consuming and it costs more. On the other hand, in comparison with testing which does not include a simulation, but performs tests on the actual embedded system, HIL brings worthwhile benefits. The two main benefits that keep appearing are reduced costs and shorter development time. The way HIL simulation can affect development cost is by removing the need for expensive component necessary for the system to developed and/or tested by simulating them. This well-balanced tradeoff is depicted in figure 4.
As already mentioned, with the HIL, an actual physical component (a component outside of the system under the test) can be replaced with a simulation, making it possible for a developer to test the other part the system before the actual component is developed. However, HIL has its downsides, depending on the system complexity, it can be costly to set-up a HIL lab. Also, additional simulation requires maintenance work which can result in increased costs in the long run.

5.3.4 HIL Lab

Authors of the state-of-the-art study [2] on the topic of hardware in the loop simulation propose a general structure of the HIL lab which is depicted in figure 5. The lab building blocks are command and control unit which is used for loading test script as well as other commands during simulation. Then, the processor, based on given commands, produces corresponding commands to electrical and mechanical blocks, while the system response on them is provided back to the processor.

5.3.5 XIL integration

There is ongoing research on how to integrate all these previously mentioned testing environments into one common testing framework. The approach presented by [16] is based on the functional mockup interface (FMI) standard, which enables the exchange of simulation models created by different tools. Aside to the transition between different levels of XIL testing, it provides test case reuse by applying the black-box testing method. Next, a cross-platform test system proposed by [17] uses black-box testing and allows reuse of the test cases across different testing platforms (MIL, SIL and HIL), as well. The similar research work is done by [18]. Furthermore, the proposed framework uses an evolutionary black-box testing which has an ability to generate test cases automatically.
5.4 Test case generation

One of the hot topics in the field of embedded software testing automation is the automatic test case generation. Even though the most common form of contribution offered by the academic community is a tool for test case generation, it is a significant input for designing any type of testing framework. This topic is covered in many aspects and different level of abstraction. In the context of previously mentioned XIL levels, test case generation is studied on MIL as well as on HIL level.

For example, authors of the paper [19] propose a search-based approach to automate the generation of MiL level test cases for continuous controllers. Next, authors of the paper A Search-Based Approach to Functional Hardware-in-the-Loop Testing [20] present an approach in which evolutionary functional testing is performed using an actual electronic control unit for test case evaluation.

Test case generation is not limited only to the white-box testing, nevertheless, there are more papers on this topic [21], there is research done on how to perform a black-box test generation. As presented in [22] search-based testing techniques could be used for unit level testing of real-world applications. Next, the paper [23] presents a tool for automatic test data generation in C programming language based on specified criteria. Similarly, [24] demonstrates evolutionary test environment. Authors of the paper [25] are focused on black-box testing and present an Automated Test Case Generator (ATCG) that uses Genetic algorithms (GAs) to automate the generation of test cases from output domain and the criticality regions of an embedded system.

As it can be noted from previously referenced papers, evolutionary algorithms (also known as genetic algorithms) are a very popular method in the field of automated test generation. Aside to notion gained by performing a scientific database search (IEEE Explore, ACM, Springer), the results of Systematic Mapping Study in Automatic Test Case Generation [21] confirms the popularity of genetic algorithms where it is a focus of 27 papers out of 85 included in the study, leading with a large margin in comparison with other approaches.

Evolutionary algorithm is based on Darwin’s theory of evolution where the fittest genes get passed to the next generation [9]. In more technical words, the algorithm starts with an initial population, which is often randomly selected from the input space. The population is passed to fitness function responsible for assigning the fitness factor to every population individual. The next step is to select individuals who satisfy the selection criteria, breed them (apply mutation
and crossover) and repeat the whole process until the determination criteria. The way most of the reviewed papers use evolutionary algorithm is to generate optimal test cases, put in other words, the algorithm is trying to find optimal input space with the respect to the fitting function. Fitness function could be used to select a test based on some coverage criterion, in case of structural testing. Furthermore, it can be used for functional testing when fitness function is designed in such a way that data sets tend towards a system behavior failure [25]. The structure the evolutionary testing is well depicted by [24], where the authors of the paper indicate that fitness evaluation phase is the one where the other testing activities occur.

Although popular topic with many proposed solutions, the field of automated test case generation requires more research in the future. Nonetheless, it brings time savings, its effectiveness is still not confirmed. The research done on the topic of comparison manually written tests with automatically generated ones [26] shows that manual tests are able to detect more faults, that they achieve similar code coverage, while automatic test generation introduces extensive time savings.

5.5 Limitations

It is important to list the limitations of the research presented in this section, which may affect its validity. The first limitation is the scope of searching, which included three major scientific databases. However, the results of the research may be different and more complete if the number of searched databases is higher. Another limitation that has to be considered is search terms that are used. There is a possibility that there is a better set of search terms or better combination of them (using different operators). For example, it is possible that there is an alternative name for one of the search terms which the authors of this research are not aware of.
6 Prototype development

This section aims to describe the prototype that was developed during this thesis work to put the reader in context and to provide better understanding for the following sections.

Developing a software prototype is an integral part of this thesis for a numerous reasons. One purpose of the prototype development is to understand and compile guidelines for performing software migration (defined in section 7). Consequently, this project provided an opportunity to advance the understanding and determine challenges that may be encountered in software migration from one language to another. Furthermore, this development process seeks to result in a codebase that can be used as an object for the experiment run with the purpose of comparing Python and Golang codebases with respect to the code refactoring and navigation. In consideration of all the previously mentioned goals, the output of this development process is of significance for the reason that it is the input for the further research that aims to answer the research questions posed in this thesis.

6.1 Introduction

The software prototype that was developed as a part of this thesis is a testing framework for an embedded operating system, WeOS. Before proceeding to discussing test framework details, it is important to present the software that is the subject of the testing. WeOS is a proprietary operating system that is developed by the Swedish company Westermo with the goal of ensuring reliable and robust communication between industrial network devices running it. WeOS has been in the market for a long time and it is recognized as one of the leaders in the field. Part of its success has been due do its quality that can be attributed to the exhaustive and continuous testing performed by the testing team.

At the moment of writing this thesis, all the automated testing is done by in-house build testing framework - Fawlty. Fawlty is a large application with 100 000 lines of code that is being continuously developed and updated from 2011 to satisfy WeOS needs and ensure its quality. However, as the company representatives pointed out, after seven years of development the framework has become resistant to changes and hard to maintain by the test framework development team. Also, the company experts stated that they believe that the problems with the current implementation are partly caused by the fact that the framework is implemented in a dynamically-typed language, Python. Furthermore, the support for the Python 2, which is being used for Fawlty development, ends in 2020. Therefore, the company is interested in feasibility of migration of the testing framework to another programming language and whether the static typing system will increase maintainability of the software.

6.2 Defining testing context

Before the legacy system and the prototype are discussed further, it is important to put them in the context of definitions presented in section 5. First, the testing framework provides support for system level testing. The subject of testing are network devices and the test ensures the correct behaviour of the device as a whole, including its hardware components and software that runs on it but also the integration of multiple networking devices. Next, it is a black-box testing focused on external behaviour of the device without considering its code structure or architecture. Finally, when put in context of XIL (x-in-the-loop) testing, this framework supports HIL (hardware-in-the-loop) as well as SIL (software-in-the-loop) testing.

HIL testing is performed in nightly builds, where each night a set of tests is run on the target
Figure 6: Testing process on local test setup

hardware devices. During this test a set of devices, specially made for the purpose of environment simulation, are used to support the testing process. For example, a device, which is connected to a cable, that simulates breaking of the cable and disables the link between two DUTs (devices under test). However, in order to provide better support for defect detection and make time that passes between commit and detection as short as possible, the testing framework supports SIL testing. In other words, it enables testing of the WeOS that is being run on virtual machines on developer’s computer. This feature helps several stakeholders. First, it supports the developers of the WeOS and the testers in the system testing. Additionally, it helps the developers of the testing framework during the development and maintenance of the testing framework.

Besides Fawlty, there is an additional system that is integrated with it and supports test scheduling as well as automatic WeOS building and test execution. The continuous testing process is presented in the following section.

6.3 The framework architecture and testing process

The testing framework in relation to the SIL testing is represented in figure 6. As discussed in previous section, the whole process of testing is conducted on developer’s machine. Towers, the virtual topology manager, starts virtual machines that run WeOS and connects them based on the topology specified for the particular test. Then, Fawlty testing framework communicates with the virtual machines in order to perform the tests described in the provided testing scripts.

The testing framework in relation to the HIL testing is represented in figure 7. The WeOS’s source code as well as test scripts are pushed to GIT repositories. Next, the developers schedule tests for the execution for each night. During the night, the test scripts are run by the test manager, that delegates scheduled tests to Fawlty testing framework and configures the hardware devices for the test execution. Finally, the tests are executed by the testing framework, which communicates with the hardware devices during the test execution and reports the outcome of the test to the test manager.
In order to provide background needed for the following text and to make it easier to understand, a generalized overview (abstraction) of Fawlty implementation details is described. Fawlty can be divided into three different logical sections. The first is basically a wrapper around the WeOS command line interface (CLI), represented as WeOS abstraction in figure 9. It is very important since the WeOS CLI is a way of configuring the operating system for a specific purpose. The following list of commands presents an example of configuring the WeOS via CLI.

```plaintext
> config
> ports eth1
> enable
> exit
```

And once it is wrapped it can be used in a test script as following:

```plaintext
1 dut1.Ports["eth1"].Enable() // dut1 = device under test No.1
```

However, having a wrapper around the CLI is not enough, Fawlty contains additional features. The next functionality Fawlty provides is the ability to take the output of the WeOS (console output), parse it and return it back to the test script for the developer to decide if it is the expected output or not. Besides the functionality of writing and reading from WeOS, Fawlty also provides support for executing test scripts and formating the test execution results.

Finally, Fawlty testing framework provides helper classes for writing test scripts. Each test script uses WeOS abstraction functionalities in order to describe communication with the device. However, many tests share common sets of functionalities needed for their execution. For example, each device requires from user to login upon its startup. These common functionalities are grouped
into helper classes in order to provide better code reuse. The relation between test helpers, test scripts and WeOS is presented in the figure 8.

6.3.1 Scope of the prototype

As suggested in the introduction of this section, the current implementation of the testing framework that is being in use is a large application. Having this in mind and this thesis time limitations, it is obvious why the prototype developed for the thesis focuses only on covering a small part of the existing testing framework. However, even though limited in size, the developed prototype is sufficient as input for answering two research questions that it was developed for. It provides a set of functionality that is enough for writing, executing and evaluating a particular subset of tests. The scope of the prototype in comparison with the legacy system is depicted in figure 9.

6.4 Challenges encountered during the prototype development

There are two quite different challenges that are encountered during the prototype development. The first one is code understanding. There are several factors that can make code understanding and navigation quite troublesome. The first one is the fact that the language used for the current framework implementation is dynamically-typed. Therefore, the IDE did not provide the support needed for effective code navigation. Furthermore, generic names, which many software parts use for functions and properties, made searching the code base more difficult. The second factor is the lack of domain knowledge when it comes to computer networks and WeOS operating system. That increased the time needed for understanding the functionality of system. The second challenge that has been encountered is the lack of support for exceptions and default method parameters etc. in Go programming language. Even though, the creators of the Golang have good and justified
reasons for such a design of the programming language, it requires additional code to be written. For example, default values of function parameters are heavily used in the legacy implementation of Fawlty and it requires additional effort and creativity for corresponding implementation in Go programming language. However, it is important to note that the main reason for that is making codebase less fragile and ambiguous, which significantly impacts the code maintainability.

6.5 Prototype implementation

The prototype developed is not a very large project, however, it is sufficient enough to satisfy all of its purposes stated in the introduction of this section. In consideration of the thesis time limitation, the scope of the prototype was increasing gradually. Having in mind that the main goal is to run tests, the functionality of the testing framework was developed in order to support writing tests. Put in other words, flow of development was the following: identifying features that are necessary for running any test (core features, for example the ability to enter configuration mode), after it was implemented, one test is chosen for development and functionality needed for writing it in Go is implemented, and then the next test is chosen. Nevertheless, the most time is spent on the core functionalities of the testing framework due to their importance. In the remaining text, the goal is to present important features and their relation to the concrete implementation depicted on the UML diagram 10. The diagram presents the code after implementing support for the test focused on the FRNT protocol (FRNT is a layer-2 network ring proprietary protocol), that is presented by FrntStartup class on the diagram.

The main purpose of Fawlty is to provide a programmatic way of reading and writing from a WeOS device. This communication process is done via additional external Linux tool conserver allowing multiple users to watch and write to a serial console. Fawlty adds a new layer on top of conserver that is tailored for WeOS. The code of this layer is in the class ConsoleDriver. However,
Figure 10: Fawlty class diagram
the ConsoleDriver is a part of a bigger class \textit{WeOSdevice} that is at the core of testing framework since it aggregates all the essential features. Next, the \textit{TestHelper} aggregates all the functionalities that are common for a set of tests, which is explained in section 6.3. Finally, the \textit{Reset} class assures that one test does not have an impact on the other tests by resetting all devices that were part of in the test. The rest of the classes in the diagram are wrappers around the WeOS functionalities that are of interest for the test.

What is not visible on the class diagram is support for logging. It is important to log complete communication with the devices (output provided by its console) in order to provide support for later debugging process in a case of a test failure. In consideration of a fact that this software is a prototype, a logger is implemented as a simple file logger meaning that it provides a file per device with its output during the test.

Finally, the process of implementation was continuously followed by testing of the framework itself. All the written tests were unit tests, and therefore, in order to provide support for the testing the communication with the devices, the WeOS responses were mocked.

Parts of the implementation details are omitted from the text as well as from the UML diagram due to its confidentiality.
7 Software modernization

Software modernization is a codebase transformation of a system while preserving the semantics of the existing software system. Software modernization is a subcategory of a software maintenance, which is an important aspect of software lifecycle [27]. Old system that needs transformation is often referred to as legacy system [28], and the process of modernization is often referred as migration [29].

Reasons for modernizing software are numerous. Modernisation can be a result of business needs a software has to fulfill, for example, when a specific technology can not deliver the performance a business process needs. Also, another problem could be that technology is outdated and there is no support for it by the vendors delegating the patching and maintenance work to the company who uses it. One more problem that is different from the already mentioned issues is that it is hard for a company to attract and find good talents to work with the technology that is considered outdated. Therefore, aside from the situations when software migration or modernization is an obvious choice or necessary in some manner, there are situations where migration is an option that has to be considered. In such case, benefits and drawbacks need to be thoroughly reviewed in order to make the decision. The possible advantages and disadvantages of software modernization are discussed in section 7.5.

The need for software modernization comes naturally in a life cycle of a software, especially if it a successful product, due to its needs to satisfy the market. In such a case, a software is being developed and used successfully for some time, however, technology has changed in the meantime. In other words, regardless how successful software is, it needs to be changed and adapted to modern trends in software development industry in order to satisfy users needs.

Nevertheless, a migration is a complex engineering but also, business and organization task to perform. More precisely, the complexity partly comes from difficulties to understand the current system that is subject of migration, recovery of the knowledge that is implemented in the system, methodology of migration process etc [30]. Details regarding migration complexity are explored in the sections 7.5 and 7.6.

7.1 Reusing components from a legacy software

Sometimes parts of the system are working flawlessly or are too expensive to be implemented in a new technology. One way of approaching this problem is by isolating the existing software functionality and encapsulating it into a new software component. Some of the benefits of doing this are cost and risk reduction. However, this is not a simple task to perform, it requires reengineering of existing functionalities, integration with the new system and at the end, validation and verification that the component function properly in the new environment. Also, another challenge that can arise from this approach is that encapsulated component cannot exchange information with the rest of the system due to the technology constraints. This can be overcome by introducing a standard protocol between software parts that is technology independent. Yet, it adds additional overhead and can affect the system performance.

Additionally, existing code may be replaced with commercial-off-the-shelf (COTS) components. This option affects system’s finances by reducing maintainability costs due to codebase reduction. However, on the other side, it may introduce new expenses caused by writing and maintaining a glue code, as well as licensing [31].
7.2 Migrating to a new programming language

Due to the fact that software systems are designed to operate in a specific environment, there are several types of migration which cover migration of different aspects of a system [29]. Therefore, the migration is a broad term and can target hardware, programming language, architecture (for example, migration to Service Oriented Architecture - SOA), migration with the goal of transferring to the cloud, and more. This thesis report covers the migration activity of transferring to a new programming language, and the rest of this research question’s answer is only interested in this specific type of migration activity. Furthermore, the term migration in the following text refers to migration to a new programming language.

The process of modernizing a software to a new language might look like a simple and easy task to do. One might argue that all the complex domain logic is already implemented and that the migration is about transforming it the new language of choice. Nevertheless, it is true that having already implemented logic, that developers who are responsible for migration can look and refer to, is of a huge help migrating software. However, in the migration there are other issues that come up making it challenging, time and resource consuming task. Those concerns are discussed in the great detail in section 7.6

One of the approaches to the software migration to a new language is a source code translation. This method converts codebase to a new programming language in a way that program logic and data structures changes are minor [32]. Code translation requires automation of the process in order to be cost-effective [31]. Although it ensures consistency of translated code, automated translation cannot apply coding practices that are specific to the target language. It will rather produce code written in “style” of the programming language that legacy code is written in. This does not need to be huge difference necessarily. In a case of translation between versions of the same programming language, the negative impact of automatic translation will less likely be vast. However, if the two programming languages belong to two different programming paradigms, the translation will be less effective. Furthermore, resulting code will be hard to read and maintain.

7.3 White-box and black-box migration

White-box and black-box concepts are already mentioned and explained in the section 5.1.2 where they were put in the context of software testing. Even though it is a completely different field in which these two terms are used, they can also be applied in the context of software migration. Therefore, the fundamental meaning of the terms can be transferred to the context of software modernization, meaning that white-box migration requires knowledge of the system internals while black-box migration examines only inputs and outputs of a legacy system.

In order to understand system internals, white-box modernization involves reverse engineering that helps to interpret the current (legacy) system and its design and architecture. According to [33], in this process “components of the system and their relationships are identified, and a representation of the system at a higher level of abstraction is produced”. This task is quite troublesome and time costly and its reasoning is elaborated in the section 7.6. After the comprehension of the system implementation is gained, the next step is to revise the insights and perform system restructuring in order to improve maintainability of the software, its performance, or any other of its characteristics [33]. This process is depicted on figure 11.

Similarly to testing, black-box modernization completely ignores internals of the system and focuses only on inputs and outputs of an examined system [33]. The ignorance of the system architecture provides both benefits and drawbacks. On one side, it gives the developers advantage of a fresh start when they are able to develop without being affected by the current internal design and therefore are less biased. In other words, fresh start gives an opportunity for even better enhancements and more suitable system design and architecture. On the other side, gaining
the understanding of the system interfaces is a challenging task. Furthermore, developers need to understand and reimplement whole domain logic from the start, which is time and resource consuming task. Another common approach to black-box migration is wrapping. In a nutshell, wrapper translates the existing interface to fit the new requirements and still uses the legacy system in the background.

7.4 Reverse engineering

According to [13], “Reverse engineering is the process of analyzing software to identify the software’s components and their interrelationships and to create representations of the software in another form or at higher levels of abstraction.”

It is important to note that reverse engineering does not affect current implementation, but rather, it aims to provide understanding of the software. Better understanding of the system is achieved by utilizing different views of the software provided as results of this process [34].

Reverse engineering is one of the key aspects of the software understanding. It can be performed for several goals like the recovery of lost information, better support for decision making, discovering bugs or vulnerabilities. Furthermore, it is the first step of white-box modernization process preceding the forward engineering. Therefore, the result of the reverse engineering process has a high impact on the success of the whole modernization process since reverse engineering output is input to forward engineering.

There are several techniques used to perform reverse engineering. Different authors categorize those techniques differently. However, the two main approaches, that are frequently elaborated, are static and dynamic analysis.

Static analysis is a process of studying the source code with the goal of understanding how the system functions. The source code is approached from different angels from inspecting business rules to how the data flows in it [35]. The information that static analysis outputs are software artifacts (for example, classes, interfaces, methods etc.) and their relations. [36]

Another technique that is worth mentioning when it comes to program comprehension is dynamic analysis. In contrast to static analysis, which is previously elaborated, dynamic analysis is performed on a running software. The idea behind it is to observe the running software and collect useful information regarding the software’s behavior. Additionally, while developing software, dynamic analysis is used for debugging, testing and profiling. However, before doing any of this,
it is necessary to understand what the analyzed piece of software is doing or is intended to do. Therefore, dynamic analysis is a useful methodology when it comes to program comprehension and perceiving accurate picture of the software system [37].

The authors of *A Systematic Survey of Program Comprehension through Dynamic Analysis* [37] have come to the conclusion that dynamic analysis is rarely applied to legacy systems, based on the number of papers that focused on the topic. This is caused by three factors. The first one is limited access to the legacy software by researchers. The second one is the lack of available instrumentation tools. Finally, the third one is the fact that is hard to deploy and run the software that is being analyzed.

However, the authors of the same paper have pointed out that half of the surveyed papers use a combination of static analysis along with the dynamic analysis. Accordingly, the dynamic analysis could contribute to completing the understanding of the system and, therefore, should be used if previously mentioned limitations can be overcome.

### 7.5 When to perform software migration?

As already presented in the previous sections, the process of modernizing software can be a very complex and time-consuming. Due to its complexity, it does not affect only the software product that is the subject of the process but also other aspects that play important role in the software development lifecycle. Before starting the process, software stakeholders should ask themselves the question whether is this migration worth doing.

There are numerous important factors to consider. The first one is financial perspective, taking into account short time and long time cost-benefit analysis, which is directly caused by migration’s complexity. And the other perspective comes from the fact that a new product is being developed even though it is a product of migration.

Software stakeholders have to be aware that modernization means changing or reimplementing the source code that works and that has been tested on many different levels. Yet, a change or completely new implementation requires testing of the system to make sure it functions properly and to regain confidence in it. Furthermore, in some cases, the system is a subject of major changes in the ways how to interact with it. Those changes can be in the range of low-level changes such as product application programming interface (API) to high-level changes as user interface. Having said that, the modernized system might require training for people who are interacting with it. Besides training for the end users once the software is developed, there is one more training that has to be considered. A company that has a software product in a specific technology or a stack of technologies is highly likely to have developers who are experts in those technologies. Even though the technology the legacy system uses and the new technology of choice might be very similar, the introduction of a new technology requires training of engineers that are the part of the process. And one last possible disadvantage we have discovered while reviewing literature is resistance to change. Put in other words, some of the people involved in the modernization might feel comfortable with the way the software currently works and therefore resist the change to a new unknown technology or design.

Therefore, it is crucial to consider all the aspects and alternatives before proceeding with the act of modernization. Furthermore, it is important to consider benefits that are going to be gained if the migration is performed. In such a way, a clear insight can be achieved showing the advantages and disadvantages of the discussed migration. Therefore, in the following text, possible benefits of a migration are presented.

One of the key benefits of performing a modernization is the financial benefit, which is, in essence, very interconnected with all other benefits that are mentioned in the following text. One
of the drivers of modernization that has a huge impact on the organization’s decision whether modernize or continue using the current system is the maintenance cost. According to the survey performed by [38], 80.4% of participants indicated that maintenance cost is one the most impactful forces for software modernization.

Next benefit the modernization brings is developer’s satisfaction. In contrast to the previously discussed resistance to the new technologies, some software developers prefer to work with modern technologies in general, whether it is a framework, programming language, database management system (DBMS), or something else. One of the reasons is that it is never appealing to invest time in learning some obsolescent technology. Therefore, developers would less likely strive for mastering the technology they are using. Moreover, by using modern technologies, developers will more likely come across better support by the development community. This will impact their development time and support their programming decisions which will directly impact developers’ satisfaction. Furthermore, a company that is utilizing modern technologies is in a better position when trying to attract new workforce talents but also keeping the current employees.

Another benefit to consider is the system performance. Performance improvement is not a necessary consequence of software migration. However, it may be one of the outcomes gained from the migration since modern technologies are more likely to use new and more efficient approaches. Furthermore, performance improvement can be one of the reasons for performing migration. In such a way, particular technologies are chosen with the goal of system performance enhancement. The advantage gained can impact on the financial benefit. By improving performance, less time needed for the same operations will reduce resource consumption. Moreover, performance enhancement will affect users’ impression of the system, which can further be reflected in the revenue.

Finally, an additional benefit is improved maintainability. With time, maintenance of a legacy code becomes more and more challenging. Therefore, the outcome of any successful migration needs to be an improved maintainability. Moreover, it is one of the most usual reasons for performing a migration.

Having in mind all the obstacles that have to be overcome when performing a software migration and extent of benefits that are expected to be gained, the process has to be justified and worth doing.

### 7.6 Software modernization challenges

The challenge of software modernization lies in the understanding of the legacy system that is subject of modernization. The reasons why understanding is challenging, even when there is a working system in place, are numerous. First reason, which is common for systems that are in use for a long time, is that the documentation is out of date or it might not even exist for some parts of the system that were developed in an ad-hoc way [30, 34, 32]. Besides the fact that out of date documentation does not provide information needed for system understanding, it can also negatively affect the process since the reader might think it represents the current state of the system.

The second challenge that occurs is knowledge extraction for the legacy system. One of the ways of performing the knowledge extraction is by collecting it from the developers who have knowledge about the system, however, this is not always possible because they might have left the company or retired. Nevertheless, even if the developers of the system where available, they usually only have detailed knowledge of a part of the system they were responsible and just a general idea of how the rest of the system functions. The second way of doing the knowledge extraction is by exploring and analyzing the source code. Yet, even the fact that the legacy source code contains business rules, they are usually hard to find and extract due to the complicated structure. One factor that could affect the understanding of the source code is that the code base can contain
source code that is not in use but still there as a product of changes in requirements during years.

An additional challenge for migrating a software is how to perform data migration from the old system to the new. Due to data complexity and its importance to the organization, the process has to be planned carefully at the early stages of the migration planning since it is crucial to ensure data consistency between the old and the new system.

It is important to note other activities, which are performed after the system functioning is comprehended, that affect the difficulty of migration. As presented by the authors of a paper [30] complexity of a migration process has two more aspects, aside from the understanding of a legacy system. The first one is a problem of decomposition of the legacy system under examination, which relies on the results gained from the previous system analysis. The second one is covering organizational aspects where the goal is to organize the migration process in a cost-efficient way.

### 7.7 Proposed method

This section describes a method for migration that is proposed based on the literature review and experience gained during the migration of the testing framework from Python to Go programming language. The proposed method is covering only the technical part of the modernization process that is discussed in great details in this section while ignoring financial and organizational challenges. Put in other words, the proposed steps assume that the organization interested in the modernizing software has already done the evaluation of the modernization process and made a decision to proceed with it. Also, it is assumed that the organization has specified what are the reasons for the modernization and what should be results of the process, for example, improved maintainability or system performance.

Before getting into details regarding the proposed steps, it is important to connect it with the previously elaborated modernization techniques. The proposed method represents a hybrid approach, which utilizes white-box modernization process, as well as, the black-box approach. Furthermore, for the reverse engineering step of the white-box process, both static and dynamic analyses are used.

In a nutshell, the process consists of five main steps as presented in the figure 12. The goal of the first step is getting an understanding of the current system. The next step is focused on the development of an abstract representation of the system. The third step introduces improvements to the previously extracted abstract representation. Finally, the software implementation and its testing. It is important to note that the whole process can be iterative. This causes several benefits. The first one is the fact that the iterative nature of the process allows parallelization and better organization of development teams. This means that the system can be split into separate subsystems which can be modernized at the same time and by different developers (development teams). The second benefit is the ability of modernization parts of the system and later integrating it. The process is depicted in the figure 12. The bidirectional arrow between the implementation and the testing steps emphasizes the importance of testing during the development (in an iterative manner). Put in other words, it is important to test as soon as it is possible during the development and then go back and continue the development process. In consideration of previously mentioned option, the term system can be replaced with the term subsystem (for example, component or module).

One way of grouping the process steps is the following. The first two steps of the process are, in essence, reverse engineering process. The third step represents the phase where modernization process utilizes fresh start and introduces improvements to the system design and architecture. Moreover, this step focuses on investigating possible benefits of the target programming language with respect to the system architecture. The last two steps are basically forward engineering part of the process where the new system is developed based on the collected artifacts and improvements.
In the following sections, each of the steps will be presented in more detail with corresponding substeps.

### 7.7.1 Understanding of the current system

The first step, initial system information collection, is consisted of gathering as much as possible information regarding the system and the modernization. The quality of this task is crucial since it affects all the following steps. Therefore, if not executed with a great caution it can lead to long-term consequences such as software that does not fulfill business needs. First, system requirements are needed in order to gain insight of the software’s purpose and significance. Next, the causes that lead to the migration and the expected outcomes of the migration and improvements to the current system are needed to be able to focus on them during the rest of the steps.

If the documentation is complete and the precise system comprehension is achieved, the next step is interviewing the software stakeholders with the goal of gathering suggestions for software improvement. In essence, the question is what would they change about the current system if they had a magic stick. The interview questions are open-ended but guided by the goals of modernization that have been identified before the start of the technical part of the modernization. For example, if the goal is the performance improvement, the focus will be on parts of the system that are performance bottlenecks, and if the goal is enhanced maintainability, for example, the focus will be on software architecture parts with poor changeability or parts with low readability.

On the other hand, if documentation of the current system is not available or is out of date, the engineers responsible for the migration have to take a different approach to the information collection. The first step is to check if there is somebody who is familiar with the system internals who can provide valuable information. The value of information provided by a person or group of people with a deep understanding of the system is reflected in reduced time and complexity. First of all, it saves time since they can answer potential questions the people responsible for the modernization might have. Next, parts of the system or even the whole system might be complex due to its domain complexity or as a result of a bad design, nevertheless, in such situation, the input from system expert is highly helpful. Besides giving new information that was not collected during the previous steps system expert can also help validate the information that has been gathered.

However, if the information gathered from the system experts is not enough for the people involved in the modernization process or in a case where the previous step is skipped due to unavailability of system experts, the modernization process enters a new step of reverse engineering. That process is very challenging and troublesome. However, if it cannot be avoided, statical analysis of the legacy code base will provide needed information regarding system artifacts and their relationship. Furthermore, as noted in the section 7.4, dynamic analysis is rarely applied on a legacy system. Nevertheless, its results can help completing the insights of the system internals and, therefore, help in software comprehension. Therefore, if it is feasible to perform dynamic analysis of the legacy system, it should be conducted. As emphasized in the previous text, reverse engineering is time and resource consuming task. Therefore, developers participating in the
modernization process, who are already familiar with the legacy system, should be involved in the reverse engineering tasks, if there are any. Their experience will contribute in a great deal to the time needed for the task for the same reasons as discussed in the previous text regarding the importance of interviewing system experts. Furthermore, the quality of the produced insights will most likely be better in comparison with the one conducted by new developers who are not familiar with the system or automated codebase translation. As depicted in figure 13, the static and dynamic analysis should be performed in parallel. Having in mind the complementing nature of the two processes, they should be conducted in the same time (or one after another for the part of the system), in order to gain more through understanding and provide different views of the system.

7.7.2 Creating an abstract representation of the system

Once the previous step, understanding of the current system, is completed successfully with the great confidence in the gathered information, it is time to move to the next step in the modernization process. In a nutshell, the next step is converting knowledge gained in the previous step into documents needed for the following steps.

The outputs of the previous step are delivered in different formats. For example, non-standardised diagrams, graphs, text such as interview transcripts or similar. All the information gathered could be ambiguous if it is not merged and organized in a proper way. Now it is up to this step to convert
the extracted knowledge from the previous step into standardized form documents, for example, UML (unified modeling language). It is of great importance for this step to result in documents in a standardized way because it helps reducing ambiguity for the future system developers and makes the communication easier.

![Diagram of sub-steps of creating an abstract representation of the system]

Figure 14: Sub-steps of creating an abstract representation of the system

The first substep is a decision making on what representations of the system (documents) are needed for the further system redesign and implementation. The suitable abstract representation differs from system to system due to different system complexity and application requirements. Therefore, the decision on which concrete documents should be provided is left open. In such a way, the proposed framework offers higher flexibility. After the type of documents has been established, the next substep is creating the defined abstract representation of the system. Some examples of the documents that can be an output of this step are architecture diagram, module dependency diagram etc. After that comes a redocumentation of the legacy system where newly created artifacts are merged with the existing documentation. The process is visually represented in figure 14.

### 7.7.3 Improving the abstract representation of the system

This step as depicted in figure 15, improving the abstract representation of the system, is consisted of two major parts. The first one focuses on improvements in general. The second one is focused on improvements that are possible due to the target programming language and platform.

The process starts with analyzing the abstract representation of the system with the goal of identifying possible design flaws and improvements possibilities. Put in other words, documents which are the output of the previous step are being examined.

Next, the legacy system tends to have parts that are obsolete or parts which functionality is replaced by other subsystems. This results in parts of the system that are not being in use but still available in the system. In order to avoid unnecessary work of modernizing a part of the software that is not part of the business process anymore, it is crucial to identify such parts and remove them from the final system representation.

Once the obsoleted parts of the system are identified and removed, it is time to make improvements to the software design and architecture. All the improvements done here should be aligned with the software requirements and modernization goals specified at the beginning of the process. Also, the people involved in the design process can take advantage of a fresh start to make the new system more flexible, scalable etc.

After the general improvements have been established, the system is analyzed again in order to detect possible improvements which are related to the target platform and programming language.
In essence, the following process is very similar to the previous one but now the abstraction is on a lower level. For example, Go programming language has great support for creating coroutines (goroutines), which should be taken advantage of in order to introduce concurrency. This will further impact the performance and introduce the possibility for parallelism. Thus, in this sub-step, language specific design patterns (for example, Go’s concurrency patterns) and recommended practices and their applicability to the system should be investigated.

The substep focuses on system’s external dependencies. Packages and libraries that are used in software implementation need to be examined in order to discover corresponding ones for the target platform and programming language or better alternatives. If system strongly relies on external libraries, its replacement may impact the system design. Therefore, the next substep focuses on system adjustments and possible improvements.

### 7.7.4 Implementation and testing

After the final architecture and design of the target system are established, developers can proceed to software implementation. The implementation is a forward engineering task, a traditional process of transforming high-level abstraction to low-level implementation [39]. Additionally, after and during the implementation, testing step is conducted.

Even though those steps are part of the modernization process, they can be viewed as independent since they are a part of the traditional software engineering challenges, and are not specific to the process of modernization. Therefore, these two steps are left out of this thesis report.

### 7.7.5 Final remarks

Even though the proposed method has clear steps (and substeps) and their order, it is possible to jump from one step (substep) to another even when there is no clear connection between them. Besides the ability to execute steps in adjustable order, it is also possible to execute step or group of steps in an iterative manner. This results in a flexible framework that is possible of adjusting to the modernization needs. The flexibility is very important since proceeding with the next steps
before making sure the system is well understood or designed is not a good practice and can result in problems down the road. Moreover, another reason why the flexibility is very important is due to the legacy systems diversity and the framework applicability in different domains.

7.8 Statically-typed to dynamically-typed language

So far this section presented challenges and proposed methods on how to perform software modernization in general, regardless of language type system. In this subsection the focus is outlining how the type system affects the process. Furthermore, since the proposed steps in this are only addressing the knowledge extraction and creating abstract representation out of the gained knowledge, the implementation part is out of the research scope and therefore left out. For the same reason, the focus of this section is what effects do language type have on the knowledge extraction and system understanding.

In the case where the legacy system is implemented in a statically-typed language, one of the advantages it has over the system in dynamically typed language is the solid type system. This is an advantage for the following reasons. First of all, if the code analysis with the goal of understating it is performed manually by software developers, statically typed language together with a modern IDE help developers navigate throughout the system design easily. Put in other words, it provides developers with the option to have confidence in object fields but also to identify all the references between objects. Next advantage of the type system is that tools for creating diagrams can utilize type system data and create a corresponding diagram, for example, a class diagram and therefore help to understand the current system.

However, even though the codebase is written in a dynamically-typed language and types are absent, modern IDEs are able to support code navigation to some extent and, therefore, enable at least partial information extraction.

7.9 Related work on software modernization

For gathering relevant literature three major scientific libraries have been searched, IEEE Xplore, ACM and Springer. The term used for querying is (white-box or whitebox) and (migration or modernization). In the following paragraphs it is presented how the papers are related to the work in this thesis.

The process presented by the authors of the paper [3] is partly overlapping with the methods proposed by this thesis. The main similarity between the processes is that it is divided into multiple steps with the clear goal that contributes to the understanding of the legacy system. However, there is a difference in the steps details, the process discussed in [3] focuses on creating a technology agnostic model out of the legacy system while the method proposed in this thesis work takes a different approach by creating models that are technology specific and therefore taking the full advantage of the target platform. Moreover, another difference worth mentioning is the difference in processes flexibility. Both processes offer a certain amount of flexibility, where the [3] method is less flexible since only steps three to six can be iterative while steps one and two are fixed (steps are presented in the figure 16). On the other hand, while the process proposed in this thesis have clearly defined steps and connections between them, it is possible to jump from one step to another with the goal of understanding and creating a better model representation.

The second paper we analyzed, [40] is focused on the process of modernizing by use of the model-driven method. Even though this thesis is not interested in the model-driven engineering, there are still similarities and value to be extracted from the mentioned research. The part where this research and method presented in this thesis overlaps is the importance of creating an abstract
model out of the legacy system. However, once an abstract model is created, the work proposed in the [40] takes a completely different path that is out of the scope of this research question and this thesis in general.

7.10 Applying the proposed method

This subsection is going to explain the method applied during the prototype development i.e testing framework migration. Put in other words, the purpose of this subsection is to connect the work done during the prototype development and the method proposed in subsection 7.7. During the process of developing, the mentioned prototype was not following the proposed steps, but rather a tool to discover all the necessary steps for modernizing software with the help of related literature. Besides developing a method for this specific case, additional effort was put to cover cases that were not needed or considered important for this particular prototype. To further clarify the point an example is provided in the following text. The first step’s understanding the current system substep is asking if the software stakeholders have a clear insight into system internals, in case of the subject system the answer was no, however, the proposed method still offers a solution in a case of a positive answer. Next, another important aspect of the proposed method that was put to the test was its flexibility, addressed in section 7.7.5. Having in mind the thesis limited time and development experience of its authors, deciding how much of the legacy framework should be modernized was a big challenge, therefore the proposed method is developed to fit short iterations that in turn makes it flexible. The following text provides a detailed overview of the method’s five steps and its sub-steps taken during the prototype development.

Step 1: Understanding the current system.

The first sub-step of this important step is to gather initial system information. In the case of the prototype, this sub-step took place over a couple of days and consisted mostly out of presentations and conversation with one of the main developers of the legacy system. Once the general understanding of the whole system was gathered it was time to move further to understanding the system details. The details gathering was planned to be done using the available documentation. However, the documentation was not sufficient enough to understand all the necessary details and therefore it was needed to continue with the collecting information. At this stage, once again we consulted with one of the main developers of the legacy system and extracted all the required information as well as suggestions for improvements (problems the current system has and should be removed in the new implementation). However, even though the interview resulted in much useful information additional static code analysis and dynamic analysis (watching the system running) was of great help for the complete system understanding.
Step 2: Creating an abstract representation of the system

As the main tool for representing the system in an abstract way, a UML class diagram was used. More details regarding the UML class diagram can be found in the section 6.5. The step of creating an abstract representation also suggest re-documenting parts of the system that are out of date or missing documentation completely. During the process, we have discovered that the legacy system suffers from both problems, missing and out of date documentation, however, during the time limitations it was not possible and of a big importance for us to execute the step.

Step 3: improving the abstract representation

The third step takes an output of the previous one, abstract representation of the system, and analyses it in order to introduce improvements.

The first group of sub-steps are focused on the system at a higher level of abstraction. The sub-step detecting and removing obsolete parts of the system was not present in our work due to the fact that the focus of modernization was at the core functionalities of the system. Consequently, these parts that were modernized have not had any obsolete parts. The next sub-step is improving the design and architecture. In a case of this thesis, this sub-step was limited due to the fact that the prototype will be used in the experiment. Consequently, in consideration of the future comparison with the legacy system, the prototype’s architecture should not differ from the original one.

However, there are changes that have to be introduced in order to adjust the code to the target language - Go. One quite straightforward step performed was to change inheritance with composition. The first reason and the primary cause for doing so is the fact that Go programming language does not have a full support for inheritance. It does have a limited support for it with provided embedded types, in which case, for example, it lacks support for method overriding. Therefore, the use of classes and its inheritance, in the way it is in the legacy Python implementation, is not possible in a case of Go. Moreover, favoring composition over inheritance is a recommended principle in software engineering. The next sub-step was to investigate corresponding packages and libraries. The suitable example is package used for creating and running an external process due to the fact that it is responsible for connecting to the virtual machines. While the subprocess module is used in the Python implementation, the exec package is chosen Go implementation, whose only drawback is the fact that assumes a Unix system. However, that is not an issue for testing framework due to the fact that the WeOS is Linux based. Consequently, the next step was to get familiar with the package and implement the functionality responsible for communicating with the device (an external process). Finally, an example where the improvements were introduced in order to take advantage of the target language is Go’s excellent support for concurrency. In consideration of the fact that most of the tests use more than one device, the commands that need to be executed on all devices should be executed in parallel. This can be easily implemented in Go, however, the recommended concurrency patterns need to be investigated first. For example, the following is the recommended pattern for setting timeouts explained in detail in The Go Blog [41]:

```go
timeout := make(chan bool, 1)
go func() {
  time.Sleep(1 * time.Second)
  timeout <- true
}()

select {
  case <-ch: // a read from ch has occurred
  case <-timeout:
    // the read from ch has timed out
```
This pattern is used for reading from the console where the timeout needs to be set to prevent an infinite waiting for the output in case of an error.

```go
scanned := make(chan bool, 1)
timeout := make(chan bool, 1)
go func() {
    scanLoop:
        for scanner.Scan() {
            output += scanner.Text()
            for _, pattern := range patterns {
                r, err := regexp.Compile(pattern)
                if err != nil {
                    panic(fmt.Sprintf("There is a problem with your regexp.\n"))
                }
                if r.MatchString(output) {
                    matchedPattern = pattern
                    scanned <- true
                    break scanLoop
                }
            }
        }
    scanned <- true
}()

go func() {
    time.Sleep(20 * time.Second)
    timeout <- false
}()

select {
    case <-scanned:
    case <-timeout:
        log.Println("Timeout: ", output)
}
return
```

Steps 4 and 5: Implementation and testing

A detailed overview of the prototype implementation process and its testing is presented in the section 6.5 and therefore will be left out of this section.
8 An experiment: the effects of the type system on software maintainability

The research with the focus on effects of the typing system on the software maintainability have shown that the static typing eases software maintainability when it comes to bug fixing [4, 5, 6]. However, on the other side, dynamic programming languages are gaining more and more popularity. Therefore, it is a question whether static typing impacts software maintainability and whether that effect is large enough to make a significant difference in real-life software maintenance. Additionally, Westermo is posing the same question in the interest of discovering whether a migration to a statically typed language would help in the maintenance of their testing framework. Consequently, an experiment is conducted in cooperation with Westermo R&D, with the goal of answering the third research question: What is the difference in the time needed for refactoring and navigating between the new system written in a statically-typed language in comparison with the legacy implementation in dynamically-typed language?

The similar experiments have been already carried out on the topic of what effects does the type system have on the software maintainability. However, even though this experiment asks the same question there are factors that differ from the previous research done and could potentially have effects on the experiment results. The first difference comes from the fact that all the experiments on this topic that authors of this thesis are aware, used Java and Groovy programming languages for treatments where this experiment utilises new treatments that have not been explored. Next, all the subjects of this experiment are people with the industry experience with a number of years of developing software, unlike in the mentioned studies where the most of the participants if not all, were students with limited set of skills and industry experience. Furthermore, the third and the final difference is that the software that is object of this experiment, Python and Go codebases, are either software that is being used in the production or there is a plan of using in the future unlike in the other experiments where a new software was developed for the purpose of the experiment.

The following text is organized as following: Background section 8.1 which will present the ground knowledge needed for understanding the rest of the text and clarify the connection between time needed for refactoring and navigating, which is the focus of the research question of interest, and the software maintainability. The next section presents the overview of the related work 8.2 followed by experiment design section 8.3. Next, analysis section 8.4 presents the results of the experiment and explains the way the missing data is handled. The next section 8.5 is covering data interpretation by performing statistical analysis, which is discussed in the following section 8.6. Finally, the conclusions are drawn and future work is discussed in section 8.7.

8.1 Background

This subsection is covering the basic terms whose understating is essential for the understanding of the following text. First, the typing system and its relation to software maintainability are explained. Next, the software maintainability is addressed in more detail with a focus on how to measure it since it is of high importance for this experiment’s design. Finally, code refactoring and navigation are discussed and connected to the software maintainability measuring.

8.1.1 Statically-typed languages and dynamically-typed languages

Every variable in a statically-typed language has an assigned type which allows compile-time analysis which provides an early code inspection while dynamically-typed languages determine a variable’s type at run-time. Since both approaches are well-known to the academia, as well as the industry, their advantages and drawbacks are established and discussed in many papers. The
advantages of statically-typed languages are identified by Erik Meijer and Peter Drayton [42]:

- early disclosure of programming mistakes
- more opportunities for compiler optimizations
- increased run-time efficiency (since variables’ types do not need to be assigned dynamically)
- better developer experience (due to the help of programming IDE (Integrated development environment))

The last item that is mentioned, the help of the programming IDE which produces better programming experience, has several more handy features which reduce likelihood of misunderstanding of code and its original intention:

- code completion
- hints while typing
- documentation inside IDE
- enhanced navigation

Furthermore, as it is summarized by [4], strong typing can enforce discipline leading to better structured and cleaner code. It is naturally achieved by type declarations for variables, methods parameters and return types, which by itself contributes to the code documentation. Herewith, code naturally speaks for itself without an extra need for writing comments that would describe intended behavior of code (e.g. methods).

Authors of the paper [43] point out advantages of dynamically-typed languages, stating that type system can be unnecessary too restrictive and tiresome when it comes to making simple changes. Therefore, “softness of dynamically languages makes them ideally suited for prototyping systems with changing or unknown requirement” [42]. Furthermore, dynamically-typed languages provide convenience and good programming experience, however different from the ones presented earlier. In the lack of compilation step, development is more rapid and agile by the elimination of waiting for the compilation process to complete, which is not negligible in medium or large projects. Therefore, the developers’ thoughts are not interrupted during development or debugging process, which can increase their productivity.

8.1.2 Software maintainability

Maintenance represents a huge, important and costly part of a software development lifecycle and it is an issue that has been interesting to software stakeholders for a long time [44]. According to [45], most of software that is considered useful is developed in a short time span and the rest of effort and time is put in the maintenance work, adapting to internal and environment changes. Consequently, it is a critical part of ensuring software quality and its operation. Therefore, it is very important to plan the maintainability in the early stages of development life cycle [46]. Maintainability is a very broad term and it can be perceived differently by different people. Furthermore, another term worth understanding is maintenance. For that reason, it is important to clarify the definitions of the terms.

IEEE standard 14764 [47] defines maintainability as
“the capability of the software product to be modified. Modifications may include corrections, improvements, or adaptation of the software to changes in environment as well as changes in requirements and functional specifications.”

The same standard classifies software maintenance as one of the many technical processes with the objective to modify existing software while preserving its integrity.

Maintenance is done with the goal of [13]:

- correcting faults
- improving the design
- interface with other software
- adapt programs so that different hardware, software, system features, and telecommunications facilities can be used;
- migrate legacy software; and
- retire software.

**Categories**

In order to better understand the nature of the maintenance process, its purpose and structure of maintenance costs, the categories of software maintenance and the factors that influence the maintainability will be presented.


- **Corrective maintenance** - is performed after a problem is discovered in the delivered software product.
- **Adaptive maintenance** - work done on the software after its delivery with the goal of adjusting to operate in changed or changing environment.
- **Perfective maintenance** - modification that aims at improving the software for its users, software documentation, performance etc.
- **Preventive maintenance** - process of detecting faults in delivered software before they become operational faults.

**Measuring maintainability**

In order to properly design the experiment it is important to find a suitable way to measure maintainability. There are many aspects of the software that impact maintainability, some of them are more easy to quantify and measure than other. The list below is compiled by [13].

1. **Analyzeability** - indicates the effort or resources invested into discovering deficiencies or causes of failure but to identify parts that are subject of modification.
2. **Changeability** - measures the effort needed for implementing a specified modification.
3. **Stability** - targets the unexpected behavior of software.
4. **Testability** - measures the effort put in testing the modified software.

Other measures that are highly interconnected with each other are:

1. **size of the software** - lines of code and number of files
2. **complexity of the software** - module dependency, software cohesion and coupling, etc.
3. **understandability** - code comments, naming conventions

The connection between maintenance categories and its aspects to the work presented in this report is clarified in further text.

The measurement of the software maintainability is composed of various factors. However, it is very difficult to measure and quantify most of them. For example, analyzability and understandability highly depend on the particular maintainer and its capabilities. In other words, many aspects of maintainability are internal process of humans and, consequently, are hard to measure. Nevertheless, there are many measurements proposed in order to calculate software maintainability. For example, maintainability index (MI) is a well-known measure. However, many authors [48] agree that these numbers are a poor indicator of degree of maintainability in general, especially when used to compare on different technologies. Additionally, a recent literature review on software maintainability measurements [48] discusses different algorithms for calculating or estimating maintainability.

However, the static analysis of code is not suitable for the comparison between the two applications if there is a significant difference in their size. For example, if we compare the prototype with the legacy system, it is clear that the maintainability of the prototype will be higher by reason of its much smaller codebase, fewer modules etc. Consequently, in order to compare the prototype developed during this thesis with the legacy software, an experiment is conducted in order to decrease the impact of the software size, technical debt etc. Furthermore, considering the goal of the comparison, which is related to the typing system, the experiment is the best fitting option due to the possibility of adjusting the experiment and its measurements to show the dependence on the typing system.

For the reason of the previously mentioned difficulty of measuring of all aspects of maintainability, in the experiment conducted during this thesis, the focus of software maintainability is narrowed down to the code refactoring and code navigation. From the maintainability definition provided in the text above, there is no clear connection between the concept of maintainability and refactoring or code navigation. However, if approached from the other side and the definitions of refactoring and code navigation are analysed the relationship becomes more clear. The next section is focused on clarifying the connection between the experiment and the software maintainability.

### Refactoring

The following statement from the SWEBOK [13], gives refactoring definition and provides a clear connection between the refactoring and the software maintainability:

> “Refactoring is a reengineering technique that aims at reorganizing a program without changing its behavior. It seeks to improve a program structure and its maintainability. Refactoring techniques can be used during minor changes.”

The minor changes are suitable for the experiment due to its time limitation. Therefore, the refactoring tasks can, in a time efficient way, display the ease of this technique which clearly impacts
the software maintainability. Furthermore, it is important to note that the refactoring tasks reflect
the code changeability, one of the previously listed aspects of software maintainability. Finally,
related to the listed categories of maintenance, the refactoring belongs to perfective maintenance
by its nature, however, it can be a pre-phase to the other maintenance categories.

Code navigation

The definition of the term code navigation is hard to find in the scientific literature. Neverthe-
less, it is quite common action taken by developers during software development or maintenance
work. It includes navigating to a file, definition or implementation as well as reference searches
of a symbol or its occurrences in the software and many more. The code navigation is one of
the essential actions in the developers’ toolbox and it is used very frequently. Furthermore, the
authors of study [45] report that developers spend 35% of development time on navigation within
and between source files, on average.

These actions are performed with one of the several goals. The first goal is related to the pro-
gram comprehension. For the sake of understanding the source code, the effective code navigation
is essential. For example, navigation through files with the goal of investigating implementation
of the method and understanding its effects. Consequently, it is connected to a modification task,
where developers need to identify, locate and understand the software modules that are subject of
change [45]. The previously mentioned activities relate to the understandability and changeabil-
ity, listed in previous text as aspects that impact maintainability. The second common task that
utilises the code navigation is is debugging, which relates to the software analysiability. Similarly
to the previously mentioned modification tasks, in order to debug the code, it is vital to first
identify the source of the bug. Furthermore, in the study which focuses on developers behavior
related to fixing performance bugs [49], which is even more complex than fixing functional bugs,
it is concluded that “fixing a performance bug is a code comprehension and navigation problem”.

The code navigation tasks are part of the experiment for several reasons. They are suitable
for the experiment due to its simplicity and the fact that those tasks are not time consuming, yet,
on the other hand, they are essential to the developers’ activities with the direct relation to the
software maintainability.

Finally, it is important to note that, both, refactoring and code navigation, at least the tasks
selected for this experiment are quite simple and common tasks which do not depend on the
developers’ cognitive abilities and experience in a great manner. In such a way, the impact of
the developers’ cognitive abilities are mitigated and through these simple but essential tasks, the
difference between the two codebases can be measured with the goal of discovering which is easier
to maintain.

8.2 Related work on the impact of type system on software maintain-
ability

Statically typed languages like Java, C++ used to be the leading force in the field of software
engineering. However, with the rise in the popularity of web applications, dynamically typed
languages such as Python, Ruby, PHP and JavaScript started to play an important role in software
engineering.

The type system of programming language, especially impact that static languages have on
software maintainability has attracted many researchers. Many papers have shown that impact
to be positive. Results of an empirical study [4] show that static type is beneficial with respect
to the maintainability, except for fixing semantic errors. Paper [5] argues that the studies that
have concluded advantage of the static type systems over dynamic type systems did not take into
account modern IDE features. However, the experiment they have conducted results in same conclusions as the previous findings. Authors of the paper [6] question whether the results that show the benefit of statically-typed languages come from the bias from the experiments. Therefore, the conducted experiment is designed in such a way to enforce benefit from dynamically typed languages. Nonetheless, the results showed the opposite and increased the doubt regarding the profit obtained by dynamically-typed languages. Also, another case study [50] showed that using Flow or Typescript (type support libraries for JavaScript - a dynamically typed language) could have prevented 15% of bugs. Aside from the mentioned experiments and empirical studies, a large-scale observational study of programming languages and code quality in GitHub [51] has also confirmed that statically typed projects have fewer bugs than dynamically typed.

Even though all the previously mentioned papers resulted in the same conclusion, they used same programming languages, Java and Groovy, for the study. Additionally, the mentioned research used tasks focused only on bug solving. Therefore, the contribution gained by this research could be beneficial by introducing diversity and providing results of comparing Go and Python programming languages as well as measurements related to code navigation and refactoring. Finally, the previously mentioned papers used software specifically made for the experiment and, on the other side, the experiment conducted as a part of this thesis uses real-world software, used in production. Nevertheless, the size of the software is reduced for the purpose of the experiment.

8.3 Experiment design

8.3.1 Goal

The goal of the experiment, according to the template provided by the authors of the book Experimentation in software engineering [52], is the following:

- Analyse static and dynamic typing
- for the purpose of evaluation
- with respect to code navigation and refactoring
- from the point of view of the developer
- in the context of Westermo developers maintaining Python and Go codebase

8.3.2 Participants

Participants in the experiment are software developers who work at Westermo R&D. The total number of participants in this experiment is six and all of them are experienced developers with years of development in the industry using different tools and techniques. Their experience that is particularly relevant to this experiment is knowledge and skill with the programming in Python and Go programming languages, which differ. Furthermore, subjects familiarity with the current Fawlt testing framework, which is one of the two objects used in the experiment, differ as well. According to the experience with Python, Go and familiarity with the framework, the subjects can be grouped as follows:

- Two developers have limited experience in using Fawlt and also have limited to none experience in both Python and Go programming languages.
- Two developers have experience with Fawlt as a testing framework and also basic experience in Python and Go programming languages.
- Two developers have experience with Fawlt as a testing framework and also intermediate experience in Python and Go programming languages.
Furthermore, the participants of the experiment are part of different development teams in the company. Finally, participants were not paid for the participation. However, the experiment is going to be conducted during their working hours. Nevertheless, their participation is voluntary and was based on their willingness to help and curiosity. Also, all the personal information such as years of experience and task performance data for example, how long did it take to finish a specific task are submitted via an anonymous form and there is no way to connect it to a specific participant.

8.3.3 Experiment materials

Objects

The objects used in the experiment are codebases of testing framework written in Python and Go programming languages. The code written in Python is used in production, however, on the other side, code written in Go is a prototype. In consequence, Python codebase is significantly bigger than the one written in Go. However, this difference is mitigated by removing parts of the software that are not needed for the solving the tasks and executing tests.

Guidelines

During the experiment, developers are going have at their disposal cheat sheets for both programming languages. These cheat sheets consist of description, short code excerpts, and notes regarding basic language usage. For example, how to create a class, how to add a method to a class, how to declare or define variable and similar. These cheat sheets can be found in the appendix, section A. The difference in subjects’ experience with respect to the programming languages used in this experiment and their familiarity with the testing framework are the main reasons for creating these documents. With the use of cheat sheets and in consideration of their experience, the lack of experience in working with the Python or Go is minimized. In other words, with the help of these documents, developers’ previous experience should have less effect on the experiment results.

Measurement instruments

Since the task performance is measured by the time it takes to complete it, every participant is required to have a stopwatch (on their phone or laptop) and use it to precisely capture the time it took him or her to finish a task. Once all the tasks are completed, the participants are expected to submit all the results to an anonymous web form that is being utilised in the further analysis. Additionally, they are required to provide a copy of their code with the objective of checking the validity of the task solutions they submitted.

8.3.4 Tasks

This experiment consists out of eight tasks that are divided into two groups, each for a corresponding treatment. As already mentioned in the previous sections, the treatments in this experiment are two programming languages, Go and Python and accordingly tasks are further divided into four tasks for Python and the corresponding four tasks for Go programming language. One of the challenges of designing tasks for two programming languages is to make them the same level of complexity so the task difficulty is isolated and does not affect the experiment results. This problem can be overcome by giving the same tasks for both languages. However, this approach raises a new issue. The problem is that experience gained while solving the same problem can affect
the outcome when the same task is done in another language. In order to avoid the mentioned problem, six out of eight tasks are completely different, utilising different parts of the systems with the goal of removing the issue of the learning effects. The two remaining tasks are not being affected by knowledge gained by performing the task multiple times and, therefore, are the same for both languages.

**Task 1. Change a class name**

The first task is the same for Go and Python part of the experiment.

**Description:** Fawlty test framework has a class that is a wrapper for WeOS and gives access to WeOS core functionalities and properties. Your task is to find the mentioned class which is located on the path specified below and rename it to "WeOS_5". Remember to change all its occurrences in the codebase.

- Go: go-fawlty/weOsDevice/weOsDevice.go
- Python: fawlty/src/weos/device.py

The first task that is given to the participants is to rename a class. At the first glance, this task might seem like a trivial and therefore its results meaningless. Yet, after consulting with an expert, we have discovered that it is a bigger problem than it seems and therefore decided to include it as a valuable part of this experiment. What is unique for this task, when compared with the other tasks in this experiment, is that it is the same for both treatments. As discussed in this section’s introduction, doing the same task twice in different languages can have an impact on the final results. However, the learning effect is not present in the execution of this task because its nature and it cannot affect ones performance when doing the corresponding task (for the second time). Nevertheless, all the remaining tasks are different and adjusted for the corresponding treatments.

**Task 2. Change a method declaration**

The task number two is designed with the goal of discovering how much time it takes to change a method declaration. Modifying a method declaration is not only changing the method itself but rather changing all its occurrences in the corresponding system. The description for the Task 2 is straightforward and does not require additional elaboration.

**Go:** The WeOS_5 class has a method Login that is responsible for logging in the user based on the current context. However, the current implementation of the login method does not accept any parameters and uses hardcoded values inside the method as the login credentials. Your task is to change the method declaration so it accepts username and password. Remember to change all its occurrences in the system.

**Python:** All tests inherit from a base class named Test which can be located in fawlty/src/common/test.py module. The Test class has a login_all() method. The method don’t have any arguments to give the test(s) any option to use any customized username and password. Update the function signature with a username and password arguments. Do not use any default values and update all references to that method. You do not have to do handle the arguments inside that function, just update the function signature.

**Task 3. Refactoring**

Refactoring is an integral part of any software development or maintenance work. It means changing the source code organization and structure without affecting its functionality. Having all of this in mind, measuring a time it takes to do a refactoring is a valuable information. However,
there is no single way to refactor a code, different people can come up with different solutions for
the same problem. Therefore, in order to isolate the developer’s previous experience and “style”
of refactoring, the Task 3 defines how to refactor. It is done by giving the participants a list of
methods that have to be extracted into a new class.

**Go:** The *WeOS_5* class has over 200 lines of code. Even though the properties and methods of
this class are very cohesive and 200 lines of code is a fair number for the class with this level of
complexity, the functionalities that are related to context can be grouped together and extracted
in a new, separate class. Your task is to extract following methods and run the validation test.

The Go implementation:

1. `func (weOsDevice *WeOsDevice) GetContext() (context string) {

2.    weOsDevice.console.Execute(""

3.    _, matchedPattern := weOsDevice.console.Expect(promptsRegexArray...) 

4.    contextArray := []string{SHELLCONTEXT, LOGINCONTEXT, CONFIGCONTEXT, SUBCONFIGCONTEXT, MAINCONTEXT}

5.    for i, pattern := range promptsRegexArray {

6.        if matchedPattern == pattern {

7.            return contextArray[i]

8.        }

9.    }

10.}

11. return

12.}

**Hint:** In order to simplify the task, *WeOS_5* can have the new class as a property and route
methods invocation through methods that are already existing in the *WeOS_5*. See the code example
bellow.

Before:

1. func (weOsDevice *WeOsDevice) GetContext() (context string) {

2.    weOsDevice.console.Execute(""

3.    _, matchedPattern := weOsDevice.console.Expect(promptsRegexArray...) 

4.    contextArray := []string{SHELLCONTEXT, LOGINCONTEXT, CONFIGCONTEXT, SUBCONFIGCONTEXT, MAINCONTEXT}

5.    for i, pattern := range promptsRegexArray {

6.        if matchedPattern == pattern {

7.            return contextArray[i]

8.        }

9.    }

10.}

11. return

12.}

After:

1. func (weOsDevice *WeOsDevice) GetContext() (context string) {

2.    // Extracted context that is now a separate class

3.    return context.Get();

4.}

**Python:** The *WeOS_5* class is a huge file with over 2700 lines of code. Your task is to extract
following methods into a new class and run the validation test.

1. `_cb_login_boot(self, user, passwd):

2. `_cb_login_ser(self, user, passwd):

3. `_cb_login_fr(self, user, passwd):

4. `_cb_login_pr(self, user, passwd):

5. `_cb_login_sh(self, _, __):

6. `_cb_login_prompt(self, _, __):

**Hints:**
In order to simplify the task, WeOS_5 can have the new class as a property and route methods invocation through methods that are already existing in the WeOS_5. See the code example below:

**Before:**

```python
1 def cb_login_boot(self, user, passwd):
2     """Callback for login when the device is booting."""
3     self.device._console.writeln("reset")
4     self.device.wait_login()
5     return self._cb_login_ser(user, passwd)
```

**After:**

```python
1 def _cb_login_boot(self, user, passwd):
2     return self.cblogin.cb_login_ser(user, passwd)
```

Since the login needs to have a reference to the device object it can be accomplished by passing the object via constructor in the following way:

```python
1 def __init__(self, device):
2     self.device = device
3     return
```

and used like this:

```python
1 def cb_login_boot(self, user, passwd):
2     self.device._console.writeln("reset")
```

In the WeOS_5 class do the following: in the self.__init__ method create an instance of the CbLogin class like so and pass a reference to the device itself:

```python
1 self.cblogin = CbLogin(self)
```

**Task 4. Follow a test execution and extract commands given to the WeOs**

The goal of the fourth, and the final task is to reflect ease of code navigation. All test scripts communicate with the WeOS by executing commands and observing the output. Therefore, this particular navigation task consists out of discovering a set of commands that are executed in the background by a test script. Besides its importance for validating that the system is behaving in a way it is supposed to, discovering commands is a good indicator of the navigation ease since commands can be found in different parts of the system.

**Go:** Navigate to weOsTests/frnt/basic/frntBasicTest.go and extract all the WeOS CLI commands that were being used for the test execution.

**Python:** Navigate to test/gre/connectivity/main.py and extract all the WeOS CLI commands that were being used for the test execution.

**8.3.5 Hypotheses, parameters and variables**

Static typing should provide better support for code navigation, furthermore, the compile-time check that it provides should decrease time needed for code refactoring in some cases. Consequently, static typing should provide better maintainability than dynamic typing. The objective of this experiment is to test whether these assumptions are true. Consequently, hypothesis related to the code maintainability of programming languages Python and Go, representatives of different typing system groups, are stated below:
H01: There is no difference in the time needed for renaming a class in the code written in Go language in comparison with the one written in Python programming language.

H01: There is difference in the time needed for renaming a class in the code written in Go language in comparison with the one written in Python programming language.

H02: There is no difference in the time needed for changing a method declaration in the code written in Go programming language than in the one written in Python.

H02: There is difference in the time needed for changing a method declaration in the code written in Go programming language than in the one written in Python.

H03: There is no difference in the time needed for methods extraction to a new class in the code written in Go language in comparison with the one written in Python programming language.

H03: There is no difference in the time needed for methods extraction to a new class in the code written in Go language in comparison with the one written in Python programming language.

H04: There is no difference in the time needed for code navigation in the code written in Go language in comparison with the one written in Python programming language.

H04: There is difference in the time needed for code navigation in the code written in Go language in comparison with the one written in Python programming language.

The most straightforward description of the dependent variable(s) is given by authors of the book [52]: “The effect of the treatments is measured in the dependent variable(s)”. Therefore, in this experiment, the dependent variable is time taken to solve a task. Independent variables are manipulated in the experiment and they have an impact on the dependent variable(s) [53]. Therefore, in this experiment, the independent variables are objects of the experiment - code written in Python and Go. Finally, controlled variable, which represents independent variable whose effect should not affect the experiment outcome, are the types of tasks for the two languages. The controlled variables should be kept fixed at the same level during the experiment so it is clear that the independent variables are the ones that reflect on the dependent variable(s) [52]. Therefore, the types of the tasks in the experiment are the same for both programming languages.

8.3.6 Experiment design

Due to the limited number of participants, this experiment is of within-subject type. Put in other words, all the experiment subjects received both treatments. Consequently, this experiment design introduces learning effect issue. As already discussed in the task section, the learning effects are taken care of and reduced as much as possible. However, in order to further remove the learning effect bias, the subjects were divided into two groups. One of the groups first solved the tasks in Python language and after it was done they continued to the next set of tasks in Go language. The same was true for the other group that first completed tasks in Go language and later switched to the Python tasks. The experiment uses a balanced design, which means that there is the same number of persons in each group.

In order to divide participants into groups, stratified random sampling is used. All the participants are first divided into three groups based on their previous experience related to the programming languages used in the experiment. This division is described previously in the section 8.3.2, and the three groups will be referred to as experience groups in the following text. Next, from each of the three groups, participants are randomly assigned to one of the two groups for the experiment. Put in other words, each group of developers in the experiment consisted of three developers selected from different experience groups. The division of participants is depicted in figure 17 below.
8.3.7 Procedure

The experiment was carried out in the Westermo’s office, Västerås (Sweden). Furthermore, the equipment used in this experiment, computers, were the same computers used by the participants in their daily tasks. Therefore, a week before the workshop, the experiment subjects were sent an email with directions what and how to prepare their development machine for the experiment. The email contained a suggestion for integrated development environment (IDE), however, the participants were free to use the IDE of their choice as long as they install support for the required technologies. Moreover, the email contained two more additional files, cheat sheets. The purpose of those files was to provide the participants with an overview of language features that are necessary for successfully completing the tasks. Due to the tasks’ simplicity and participants previous experience, a detailed training in corresponding language was not necessary and the short text file with language features was enough. Moreover, suggestions and advice regarding learning material were sent as well.

On the day of the experiment, the participants were given a short presentation to get familiar with the rules and goals of the experiment. The participants were given unlimited time for every task and asked to measure the time it took them to complete a task and report it at the end of the experiment. Also, in a case where a task can not be completed in the given time, the subjects were asked to report the task as failed. Once the subjects were familiar with the experiment, they were given a printed version of the cheat sheet files they received in the email two days before, a paper containing eight tasks and suggested to start with the first task due to the fact that the tasks are sorted by their difficulty. Nevertheless, the participants had an option to do the task in any preferred order. Next, after all of the eight tasks were completed it was time to submit the results. At this point, developers were asked to submit times for every task but also to answer a couple of qualitative questions to give their feedback and opinions on the problem. The section 9 is dedicated to presenting and analyzing their responses for the qualitative part. Once the data is submitted the experiment was officially over and the participants were free to leave the room.

8.3.8 Validity evaluation

It is important to consider all possible threats to validity that may impact the results of the experiment. Therefore, this section is going to discuss how valid the results of the experiment are. The following text is organized according to main categories of threats to validity.
Conclusion Validity

According to [52] threats to the conclusion validity “are concerned with issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome of an experiment”.

The threat to validity that belongs to this group of conclusion validity and was considered is random heterogeneity of subjects validity threat. It states that in a case of a very heterogeneous group, individual differences may influence the results more than the differences in treatments. This can be considered in the context of the experiment conducted in this thesis in relation to the difference in experience of the subjects when it comes to target programming languages. However, all the participants in the experiment are highly experienced developers and the lack of knowledge of the programming language should not have a high impact on the results. Furthermore, the language specific features are not used and just “common” primitives are present in the codebases used for the experiment. Additionally, the participants will have cheat sheets at disposal during the experiment, which should further mitigate the effect of their previous experience. Moreover, it is important to note that even though there is a difference in experience between developers, there is no difference in familiarity with the two languages when it comes to one particular developer. Put in other words, all developers have the same level of experience for both target programming languages.

The second threat to conclusion validity considered in this experiment is the fact that it has small number of subjects. However, this is only partly a problem since all the subjects in the experiment are highly targeted audience.

Construct validity

As discussed in [52], construct validity, as all the other validity types, concerns generalising the results to the concept theory behind the experiment. However, construct validity can be affected by numerous social factors. Some of those factors are hypothesis guessing, where the participants try to figure out hypothesis and therefore adjust their results in a positive or negative way. Next social threat is evaluation apprehension, human tendency to perform better when they are being watched or evaluated. The first threat, hypothesis guessing might have an effect on one particular group of participants, the developers who have been developing the Python version of the testing framework. The bias might come from the fact that those developers have spent years developing the framework and do not want the change. Next, in this experiment, the second threat is addressed by giving the participants a way to anonymously submit the data with no way of associating the performance with the particular participant.

External Validity

As defined by the authors of [52], threats to external validity “are conditions that limit our ability to generalize the results of our experiment to industrial practice”.

In this experiment, chosen programming tasks could affect the external validity of the experiment. Even though the tasks are designed in such a way to expose the differences between static and dynamic typing, it is possible that the tasks are not right for the purpose of this type of evaluation. The process of designing the tasks was mainly based on consulting with the experts and discovering what kind of tasks usually introduce issues. Furthermore, the literature survey conducted in order to present the related work had an impact on the tasks’ design.

Another potential threat to external validity is the IDE participants used for coding during the experiment. The features provided by IDE have an impact on the time needed for code refactoring and navigation. For example, go to definition feature eases the code navigation. However, it is
decided to let developers choose their IDE for the experiment. In such a way, each developer will be offered to set of features that he or she is familiar with. Furthermore, this threat can be considered minor since many modern IDEs offer the similar support needed for executing the tasks of this experiment.

**Internal validity**

As defined by the authors of the [52], threats to internal validity “are influences that can affect the independent variable with respect to causality, without the researcher’s knowledge”.

A potential threat to internal validity in this experiment is **maturation** effect which implies that the subjects may react differently as the time passes. It may be the case that the participants get bored as the time passes that can result in a negative effect on their performance. Another situation that can have an impact is a possible demoralization in a case of poor performance on one of the tasks or unexpected bugs. Next, working with a source code for the first time might be intimidating for some people and therefore affect their performance. However, the tasks are quite simple with precise instructions and, therefore, the developers should not have problems with solving the tasks.

### 8.4 Analysis

This section presents a statistical analysis of the gathered data. As mentioned in previous sections the measurement in this experiment is time and the gathered data is a collection of time the 6 participants have taken to solve given 8 tasks (4 in Python and corresponding 4 in Go programming language). The data is presented in table 2 where the cells hold values which represent time in seconds.

<table>
<thead>
<tr>
<th>Participant No.</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>542</td>
<td>260</td>
<td>1899</td>
<td>2520</td>
<td>522</td>
<td>2569</td>
<td>3375</td>
<td>567</td>
</tr>
<tr>
<td>2</td>
<td>596</td>
<td>1072</td>
<td>3600</td>
<td>1897</td>
<td>339</td>
<td>2340</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>3</td>
<td>480</td>
<td>300</td>
<td>/</td>
<td>/</td>
<td>425</td>
<td>305</td>
<td>1565</td>
<td>1200</td>
</tr>
<tr>
<td>4</td>
<td>3000</td>
<td>900</td>
<td>3600</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
<td>/</td>
</tr>
<tr>
<td>5</td>
<td>1140</td>
<td>240</td>
<td>1980</td>
<td>480</td>
<td>600</td>
<td>/</td>
<td>2880</td>
<td>1200</td>
</tr>
<tr>
<td>6</td>
<td>320</td>
<td>409</td>
<td>1830</td>
<td>840</td>
<td>454</td>
<td>2075</td>
<td>664</td>
<td>815</td>
</tr>
</tbody>
</table>

Table 2: Data gathered during the experiment (time in seconds)

As it can be noted from the table 2, there is missing data. The way it is handled is discussed in great details in the next section.

#### 8.4.1 Missing data

The quality of data collected during the experiment has a direct influence on the results analysis. However, even when the data collected is of high quality, another issue that often arises is the
problem of missing data. Obviously, the best strategy to reduce the amount of the missing data is to plan the collection process carefully. Yet, it is very common for a research to have to deal with missing data [54]. According to the same author strategies for dealing with the problem are:

- **Listwise or case deletion** - the most frequent way of dealing with the missing data, it relays on completely removing the case and proceeding to analyze the remaining data. Case deletion is possible when there is a large amount of data preset, otherwise, other techniques have to be utilized.

- **Pairwise deletion** - way of handling the missing data by removing the information only in a case where the missing data-point has a direct impact on the assumption under the test. On the other hand, if the missing data does not take active part in the tested assumption, the information is not deleted.

- **Mean substitution** - The process where the missing variable gets replaced with the mean value for the target variable. The problem with mean substitution is that it can introduce bias if the sample is not randomly selected from a normal distribution.

- **Regression imputation** - is a method of replacing missing variable with a predicted value which is calculated based on the other values that are present.

- and others such as: last observation carried forward, maximum likelihood, expectation-maximization, multiple imputation.

The experiment outputted eight data sets for eight corresponding tasks. Furthermore, every data set has six values produced by six different experiment participants. In total, there are 48 data points as a result of the experiment. As already discussed in the introduction of this section, missing data is a common thing in almost all research and this experiment was no exception. In this particular experiment, one of the participants decided to give up after first tree tasks causing five missing data points. Next, two more persons were not able to finish two tasks and one person was not able to complete one task. This inability to finish all the tasks resulted in ten missing values. The missing data can be seen in the table 2 and it is marked with the “/” symbol.

Having in mind the already presented strategies for dealing with the missing data and the size of the available data, it is evident that deleting data is not an option. The alternative for deleting data is substitution of missing data by utilizing one of the outlined techniques. The technique applied to the missing data in this analysis is mean substitution. The reasons for this choice is its simplicity. The mentioned problems with mean substitution are not present since the data is normally distributed as a whole and random sampling is applied.

The table 3 shows the data after the mean substitution has been applied to the initially gathered data.

### 8.4.2 Results

Before the beginning of the experiment, the participants were asked to push their code to a GIT repository in order to make it available for later inspections. Once after the missing data is handled and the validity of the participants’ answers is confirmed via inspecting the mentioned source code repository, the next step is performing an analysis of the gathered data. In the following text, a visual representation of the data is going to be presented followed by its statistical analysis.

The visualisation of the experiment results is presented using box plots and a bar plot. The box plots are selected due to its suitability for visualizing groups of scores by presenting the dispersion and skewness of samples. Furthermore, the box plots clearly show outliers and median of the data.
set. The combined view presented using bar plot is chosen in order to visualise the mean value of data related to each task and each programming language. The mean value is complementing the median value shown on box plots, which is of significance to the overall presentation of results.

First, the results of the tasks are going to be presented separately. Accordingly, each figure below represents box plots of the data related to each task (figs. 18 to 21). The $x$ axis holds two box plots, one for each treatment - Go and Python. The $y$ axis presents time taken by developers to solve a task in seconds.

<table>
<thead>
<tr>
<th>Participant No.</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
<th>Task 1</th>
<th>Task 2</th>
<th>Task 3</th>
<th>Task 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>542</td>
<td>260</td>
<td>1899</td>
<td>2520</td>
<td>522</td>
<td>2569</td>
<td>3375</td>
<td>567</td>
</tr>
<tr>
<td>2</td>
<td>596</td>
<td>1072</td>
<td>3600</td>
<td>1897</td>
<td>339</td>
<td>2340</td>
<td>2121</td>
<td>946</td>
</tr>
<tr>
<td>3</td>
<td>480</td>
<td>300</td>
<td>2582</td>
<td>1434</td>
<td>425</td>
<td>305</td>
<td>1565</td>
<td>1200</td>
</tr>
<tr>
<td>4</td>
<td>3000</td>
<td>900</td>
<td>3600</td>
<td>1434</td>
<td>468</td>
<td>1822</td>
<td>2121</td>
<td>946</td>
</tr>
<tr>
<td>5</td>
<td>1140</td>
<td>240</td>
<td>1980</td>
<td>480</td>
<td>600</td>
<td>1822</td>
<td>2880</td>
<td>1200</td>
</tr>
<tr>
<td>6</td>
<td>320</td>
<td>409</td>
<td>1830</td>
<td>840</td>
<td>454</td>
<td>2075</td>
<td>664</td>
<td>815</td>
</tr>
</tbody>
</table>

Table 3: Experiment data with applied mean substitution (time in seconds)
Even though a hypothesis is posed for each task separately, the impact of typing system on code refactoring and navigation in general is of interest as well. Therefore, in the following, a combined view of results for all tasks is presented, first using box plot and then using bar plot (figs. 22 to 23). The $x$ axis holds groups of two box plots/bar plots for each task. The $y$ axis presents, same as in previous figures, time taken by developers to solve a task in seconds.

From the charts representing the experiment results, it can be seen that maintenance activities conducted on the Python codebase took less time on average when it comes to the tasks 1, 3 and 4: renaming a class, methods extraction to a class and code navigation. However, on the other side, results related to the Task 2, changing a method declaration, show that all developers took less time to work with the Go codebase. The following text addresses the charts in more detail.

The box plot 18 that represents results related to the first task for Python programming language is skewed which indicates that the developers reacted similar to the task and that the time they took for solving it does not differ a lot. However, on the other side, the box plot representing results related to the same (first) task and Go programming language indicate variability in developers’ performance. Moreover, this box plot contains one outlier which value is quite alarming. The value that outlier has is almost ten times higher than the lowest value in the chart and more than six times higher than the median of the sample. However, the outlier presents the value related to the developer who gave up after three tasks and overall performed differently than the other participants.

The second pair of box plots depicted on figure 19 related to the second task shows overall better performance of developers when working with Go, with the exception of one outlier which belongs to the box plot reflecting results in Python. Regardless the outlier, which shows a lot better performance than the others, ranges that the box plots take clearly show the difference. Furthermore, the median of the data related to Go is more than five times lower than the one related to Python.
The box plots on figure 20 related to the third task do not indicate a clear difference. Even though the median related to the data representing results in Go is higher than the one representing results in Python, the box plots are “tall” indicating high variability in results, which makes it hard to extract a pattern or draw conclusions. The difference between the lowest and the highest value in the data set is almost 50 minutes.

Finally, the fourth pair of box plots (on figure 21) indicates that the results related to the fourth task are in favor of Python. The median representing the data related to Go is much higher than the one representing the data of Python. Furthermore, the lower quartile of box plot related to Go is higher than the median of box plot related to Python which shows that the 50% of results related to Python are in the same range with 25% results related to Go.

One thing then that majority of box plots have in common is their “hight”, which indicates high variability in the results. This, as well as outliers, can be caused by unexpected bugs. For example, if a developer encounters a bug, that he or she is not familiar with, that can greatly impact their results. Moreover, the bug does not have to be related to the refactoring task at all and could be just a completely non-deliberate mistake (for example, typo). Additionally, these mistakes can have an impact on the developers who have less experience with the programming language. Furthermore, the same reason can be a cause of the fact that the median is close to lower quartile in all but sixth and seventh box plots on chart 22.

8.5 Interpretation

In order to test hypotheses, a statistical test needs to be chosen. Tests can be classified into two groups, parametric tests or non-parametric tests and the first step is to select the test type. A parametric test is more suitable in the case of this experiment since it is more powerful and its
hypothesis tests mean, which is better representative of the collected data than the median (tested by a non-parametric test). However, in order to conduct the parametric test, a few conditions need to be satisfied. The first one requires parameters to be measured at least on an interval scale. This condition is satisfied for all cases for the reason that the time is measured, which belongs to the ratio scale of measurement. The second condition requires a normal distribution of data. Even though the data is normally distributed if all the collected data is observed as a whole, a separate test for normal distribution needs to be conducted for each task separately since a separate hypothesis test is run for each task. In the table 4 below, the resulting p-values of the Shapiro-Wilk normality tests are presented. The Shapiro-Wilk test tests the null hypothesis that the samples came from a normal distribution. Therefore, if the p-value is less than 0.05 the null hypothesis is rejected meaning that the data is not normally distributed. Likewise, the p-value greater than 0.05 indicates a normal distribution of the data.

<table>
<thead>
<tr>
<th>Task</th>
<th>p-value</th>
<th>Normal distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>3.432e-05</td>
<td>False</td>
</tr>
<tr>
<td>Task 2</td>
<td>0.04915</td>
<td>False</td>
</tr>
<tr>
<td>Task 3</td>
<td>0.5061</td>
<td>True</td>
</tr>
<tr>
<td>Task 4</td>
<td>0.2415</td>
<td>True</td>
</tr>
</tbody>
</table>

Table 4: Results of the Shapiro-Wilk normality test for each task

As it can be seen from the table 4, a parametric test cannot be run on the data related to the first and the second task. However, on the other side, the data belonging to the third and the fourth task satisfies all the conditions needed for the parametric test. For the reasons stated above, higher power and better suitability, the parametric test is used where possible - the third task and the fourth task.

Another issue that was considered is small sample size and different recommendations can be found:

1. Non-parametric tests “are also likely to be accurate at very small sample sizes than parametric methods” [53].

2. “The power of parametric tests is generally higher than for non-parametric tests. Therefore, parametric tests require fewer data points, and therefore smaller experiments, than non-parametric test if the assumptions are true” [52].

Finally, guided by [52], paired t-test is chosen as a parametric test and Wilcoxon test is selected as non-parametric test based on the design of the experiment.

The next step is to conduct the hypotheses tests. Additionally, effect size is calculated in order to complement the results and get an insight into the difference between the results related to the two languages, which can be of a practical significance. Guided by [55], the effect size of data related to the first two tasks, where the data is not normally distributed, is calculated based on the z-value outputted by Wilcoxon test and the total number of samples(n) using the following equation: \( z/\sqrt{n} \). Likewise guided by [55], in order to calculate the effect size of the data related to the third and the fourth task, where the data is normally distributed, the Cohen’s \( d \) value is computed.

The table 5 represents test type used for hypothesis testing, resulted p-values, interpretation whether corresponding null hypothesis is accepted or not and effect size. The same procedure is repeated for all of the four tasks.
Table 5: Results of the hypotheses testing

<table>
<thead>
<tr>
<th>Task</th>
<th>Test</th>
<th>p-value</th>
<th>Null hypothesis accepted</th>
<th>Effect size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>Wilcoxon test</td>
<td>0.1563</td>
<td>True</td>
<td>0.4092828</td>
</tr>
<tr>
<td>Task 2</td>
<td>Wilcoxon test</td>
<td>0.03125</td>
<td>False</td>
<td>0.6217701</td>
</tr>
<tr>
<td>Task 3</td>
<td>paired t-test</td>
<td>0.4258</td>
<td>True</td>
<td>0.5136354</td>
</tr>
<tr>
<td>Task 4</td>
<td>paired t-test</td>
<td>0.2435</td>
<td>True</td>
<td>0.9001963</td>
</tr>
</tbody>
</table>

As it can be seen from the table, only the hypothesis testing of the data related to the second task leads to rejecting its null hypothesis. Therefore, the accepted hypotheses are following:

**H0₁**: There is no difference in the time needed for renaming a class in the codebase written in Go language in comparison with the one written in Python programming language.

**H1₂**: There is difference in the time needed for changing a method declaration in the code written in Go programming language than in the one written in Python.

**H0₃**: There is no difference in the time needed for methods extraction to a class in the codebase written in Go language in comparison with the one written in Python programming language.

**H0₄**: There is no difference in the time needed for code navigation in the codebase written in Go language in comparison with the one written in Python programming language.

Since the three null hypotheses were accepted the reported effect sizes related to tasks 1, 3 and 4 are not relevant. However, the effect size related to the second task is of significance and its value indicates that the difference between the time needed for solving the second task in Go and Python languages is meaningful.

Now that the difference has been identified with respect to the Task 2 and its effect is shown to be of significance, the question has to be posed whether the difference is in favor of Go or Python programming language. Put in other words, the research has shown, so far, only that there is a difference in time needed to complete the task in the two languages, however, it is not clear which one takes less time. Even though it is quite clear from the charts 19 and 23 that developers took less time to work with the code written in Go, a one-tailed hypothesis test is conducted in order to increase the trustworthiness. Therefore, alternative hypothesis of the previously run Wilcoxon test for the second task is adjusted. Instead of being "two-sided", meaning that the distribution of data could be shifted one side or another, it is changed to be "less" meaning that the alternative hypothesis is: **Less time is needed for changing a method declaration in the codebase written in Go programming language than in the one written in Python**. The p-value for this test (0.01563) which is small (< 0.05) indicates that there is a statistically significant evidence that developers took less time to complete the second task in the Go code than in the Python code. Finally, the hypothesis that needs to be added to the list of accepted hypotheses is:

**H1₂**: Less time is needed for changing a method declaration in the codebase written in Go programming language than in the one written in Python.

### 8.6 Discussion

Reading the raw experiment data for the first time was a big surprise. Based on the tasks’ requirements, related work on the topic and previous experience of authors of this thesis it was
expected from the experiment to give results in favor of Go programming language. After analysing the responses from a questionnaire that was filled after the experiment (section 9) and informal conversation, it became more clear why the experiment results are the way they are. Both pointed out to the fact that the participants did not take advantage of a modern IDEs support that is available for statically-typed language. One example where Go in combination with modern IDE should have had a significant advantage was the first task, renaming a class. It was assumed that the participants will use built-in refactoring features for renaming a class in the whole code-base in just a couple of seconds. However, the experiment showed that participants approached the problem with the tools they are used to, such as Linux command for searching and replacing (grep, sed etc) and IDE find and replace features. While designing this experiment the question of IDE and its influence was considered but due to the time limitation and participants solid experience in software development, it was decided to leave the choice to the participants themselves. It was assumed that, if participants use IDE of their choice, it will have a positive impact on their productivity and therefore lead to shorter time needed for completing tasks. However, after the experiment was done, it was clear that a short demo of using a modern IDE and its features could have an impact on the final result. The effect would be the most obvious in the task number one, but also, in other tasks.

All the previous research that the authors of this thesis are aware of, interested in answering the questions of the impact type system has on software maintenance have designed an experiment in such a way that its tasks are focused on debugging. Even though the bug fixing is at the core of software maintenance and plays an important role during the process, it is not the only aspect that makes software maintainable. Therefore, this experiment is designed to test and evaluate two other factors of maintainability, the impact of type system on refactoring and code navigation. Besides measuring different software maintenance processes the software used as objects in the experiment is a real-world software reduced in size, in contrast of the one used in other research which is made just for the purpose of the experiment. Moreover, the results differ from the results of the previous research as well. The related work gave results in favor of statically-typed language and even a study designed in a way to give an advantage to a dynamically-typed language failed to do so. However, this experiment showed different results pointing out there is no statistical difference when it comes to refactoring and code navigation except for one task (out of four), the task number 2.

8.7 Experiment conclusions and future work

The main goal of the experiment was to further contribute to the existing body of knowledge with the respect of effects a typing system has on software maintainability. The related work regarding software maintainability and typing system [4, 5, 6] was focusing on a single aspect of maintainability, bug fixing and therefore the experiments conducted in the studies were targeting this aspect of maintainability. In contrast to the previous research, this thesis was interested in other aspects of software maintainability, refactoring and code navigation. The relation between maintainability and tasks selected for this experiment is discussed in great details in the section 8.1.2. Furthermore, opposed to the related research previously mentioned which used software specifically made for the experiment testing, this experiment used real-world software as experiment object.

The experiment took place at Westermo, a Swedish company specialized in industrial data communication on Tuesday, 3rd of May 2018. Furthermore, all of the participants were Westermo's employees with years of experience and, therefore, highly targeted audience for the experiment. The experiment had two treatments, two programming languages, Go and Python as representatives for statically and dynamically typed languages. The participants were given two groups of four tasks for each programming language. Next, the experiment had six participants in total where one of the participants decided to give up after the first set of tasks. Additionally, two more participants were not able to finish two tasks and one participant reported failing to complete one task. The
uncompleted tasks introduced the problem of missing data. In total, the data set of forty-eight
data points had ten points that were empty. Simply ignoring the missing data was not an option
since the data set is very limited, therefore, the problem of missing data was overcome with the
mean substitution method, a simple technique common when dealing with missing data.

After the experiment data was collected and missing data was handled the next step was to
analyze the data and draw conclusions. As explained in the discussion section, the first look at
the raw data was surprising since the results were different from previous research on the similar
problem. However, the additional questionnaire that was not a part of the experiment provided
insight why the results are the way they are. The results have shown that there is no significant
statistical difference in the time it takes to do refactoring and navigation related tasks in statically
and dynamically-typed programming language. The results of three out of four tasks, presented
on box plots and bar plot and the corresponding mean and median values, have shown that the Go
programming language takes more time on average to solve the tasks: renaming a class, methods
extraction to a class and code navigation. However, statistical analysis of the data related to
the three tasks has shown that their null hypotheses cannot be rejected meaning that there is no
difference between Go language and Python with respect to the time taken for performing the
previously mentioned three tasks. The only difference was found in the second task where the
goal was to change method declaration. The further analysis has shown that Go, a statically-
typed programming language outperformed Python, a dynamically-typed language. Therefore,
the answer to the third research question is that the typing system does not affect the time needed
for code refactoring and navigation except in the case of method declaration changing.

The already mentioned questionnaire revealed that the participants have not taken the full
advantage that comes from a combination of modern IDE and statically-typed language. Put in
other words, features that were supposed to give advantage to the statically-typed language, were
not utilized and the statically-typed code was treated by most participants as same as a text file.
To further clarify the point, a concrete example is described in the following text. In the first
task the goal was to change class name, a trivial task to do in a statically typed language with
an IDE that has the support for this kind of actions, however, if the feature is not used and the
task is completed by manually changing name on all occurrences or using find and replace method
it takes significant more amount of time. Therefore, for the future experiments on this problem,
it would be of significance to give a short tutorial on how to fully utilize a modern IDE or have
participants with more experience in that matter. Besides the tutorial, another factor that could
be beneficial for future experiments is to find many more participants. Finally, as discussed in
previous section 8.6, an experiment which will combine the aspects of maintainability used in this
experiment and ones used by research mentioned in the related work section could provide a more
complete comparison.
9 Qualitative questionnaire

Besides doing the experiment the participants were asked to fill in a questionnaire with qualitative information regarding their experience with the tasks. The purpose of the questionnaire was to further collect information that could be beneficial for understanding the developer’s preferences. For example, it might be possible for the experiment to show that one treatment, in this case, programming language has better quantitative results but at the same time that the developer’s satisfaction is lower. It is important to note that the questionnaire is not conducted as any of types of formal empirical investigations, and it is rather an informal short survey with the objective of gathering additional information of interest. Questions asked in the questionnaire are the following:

1. What are your thoughts on developing in Python vs. Go?
2. Which language in your opinion gives a better support for the tested tasks?
3. Did you feel lost/confused in the codebase?
4. Which language would you prefer to use in a new project and why?
5. Do you have any additional comments or suggestions?

Even though, as seen from the experiment results analysis there is no significant statistical difference between performance in the two different languages (except in one case) the questionnaire showed that most of the developers prefer Go language. Five out of six developers with different levels of experience in Go programming language voted in favor of it. The reason for the preferences stated in the responses was the compiler checking and therefore increased confidence when performing refactoring. Here are some of the original responses to the question: “What are your thoughts on developing in Python vs. Go”

“Having a compiler helps a lot when refactoring as it makes it easier to know when the change has been applied on the entire codebase.”

“... However I really like all the feedback you get from the Go compiler. You get the feeling that when your tree builds, you’re actually finished with your refactoring.”

Once again the answers to the second question were in favor of Go language as well. Three participants were neutral when asked about the support for the tasks given by the language, however, the three stated that Go offers a better support.

“Go seems faster to implement in.”

The answers submitted for the third questions interested in knowing did the developers feel lost in the code base did not provide any useful information. The developers felt neutral regarding both codebases.

The fourth question once more confirmed the fact that the most participants prefer working in Go. Nevertheless, there was one exception, one of the participants stated:

“Personally I have to say Python - to be efficient.”
However, this can be explained by the fact that Python is used for the development for a long time. Therefore, it can be expected to provide better efficiency from the perspective of developers who are used to it.

Finally, additional comments given as an answer to the fifth question are just a general feedback and do not provide any additional information of interest for this thesis.

The overall conclusion that can be drawn from this questionnaire is that the compiler-time analysis is of value to developers and positively impacts their development satisfaction.
10 Conclusion

As the flow of this thesis report follows the software migration process of the test framework from dynamically to statically typed language, it resulted in three units. Therefore this section is a short overview of results that each section provided. It starts with the summary of the literature study results that were later used as an input for the prototype development. Then, findings and proposed method for software migration is summarized. Finally, the findings of the experiment and its contribution are presented.

During the literature study on testing frameworks for embedded systems, it is discovered that x-in-the-loop (XIL) testing brings advantages of simulation-based testing, which reduces costs and development time in exchange for accuracy that can only be achieved with real hardware and testing environment. Nevertheless, XIL testing provides a range of possibilities with different trade-offs with respect to the accuracy, costs and development time. The highest profit of XIL testing can be achieved when model-in-the-loop (MIL), software-in-the-loop (SIL) and hardware-in-the-loop (HIL) testing environments can be integrated into one testing framework with the ability of test case reuse.

One of the artifacts of this thesis is a fully functional testing framework, in this report referred to as a prototype. The prototype has been developed during the software migration process from dynamically typed to statically typed language. Referring back to the first research question finding the prototype can be classified as HIL and SIL testing framework. This is achieved by providing a possibility of testing the system as virtual devices on developer’s machines (SIL) as well as providing support for simulating environment while running tests on target hardware (network equipment in this case). The prototype was developed with two goals, the first goal was to gain practical experience from the process of migration which is later, with the help of related literature, used to propose a generic method for migrating software. The second goal was to create an object for the experiment conducted with the goal of investigating the effects of the type system on software maintainability.

The above mentioned generic method is for white-box migration and consists of five main steps and numerous sub-steps. First comes understanding of the current system followed by transformation of the legacy system to its abstract representation. Next is enhancement of abstract representation that serves as an input for the later process of forward engineering that includes implementation and testing. Even though the method was partly a product of the experience gained from a migration of a single software project, additional effort was put into making the method generic, flexible and applicable to other software projects.

The goal of the experiment was to evaluate the migration work done and to contribute to the research body by conducting an experiment that uses Go and Python programming languages, a real-world software, and measures the effects of typing system on code navigation and refactoring. Language design and language syntax can have an impact on software maintainability, therefore, the diversity with respect to the languages used in research done on this topic is of significance. Next, a use of real-world software, which is not the case in previous research that we are aware of, is of importance as well due to the fact that it brings experiment closer to the real-world problems. Finally, all the previous work have measured maintainability with respect to the bug fixing tasks, which is only one of the aspects affecting the maintainability quality of the software. By using tasks that are designed to measure the effects of the typing system on code navigation and refactoring, that in turn reflect on maintainability directly as well as code changeability and understandability (aspects that impact maintainability). Furthermore, with the careful design of the task set used in this experiment, an effect of participants’ cognitive abilities is minimized. In contrast to the previous related research where statically-typed languages have a clear advantage over dynamically-typed languages when it comes to maintainability the experiment results show that only for one task (changing a method declaration) developers’ need less time when working in Go than when working in Python code. Other tasks, three out of four, do not show statistical
difference between Go and Python. However, the mean and median of the resulting data indicate that in the case of the three tasks, developers took less time for solving tasks in Python.

The qualitative questionnaire conducted helped to explain why the experiment results differ from the previous research on the topic. It showed that participants did not take advantage of IDE features available when working with statically typed language. Therefore, there is a need for further research on this topic in order to fill the research gap identified. Accordingly, a research with participants familiar with modern IDEs and a set of tasks that focus on a high variety of aspects which affect software maintainability could provide an additional contribution to the field.
References


A  Cheat sheets

Participants of the experiment received the cheat sheets which consisted of short code excerpts and notes with respect to the basic usage of the languages used in the experiment, Python and Go programming languages. Furthermore, the participants had these cheat sheets at their disposal while they were solving the tasks during the experiment. The objective of providing the cheat sheets is to mitigate the difference between the developers’ experience regarding the two programming languages. This is further discussed in the section 8.3.3. The documents are in the following text.
Golang cheat sheet

In following text, you are going to find everything you need regarding Golang syntax in order to solve given tasks.

Create a new class

In the Go programming language the keyword class is not available and the struct is used instead. It is defined as it follows:

```go
type Console struct {
    cmd    *exec.Cmd
    reader bufio.Reader
    stdin  io.WriteCloser
    Logger iLogger.ILogger
}
```

Create a function

If we want to add a method to the class defined above, Go language does not allow to put methods inside of the struct type as in other OOP languages such as Java, c++ or C#. Instead, Go requires a method to be outside of the Console struct curly brackets and have the following structure.

```go
func (console *Console) Terminate() {
    if console.Logger != nil {
        console.Logger.Finish()
    }
    console.cmd.Process.Kill()
}
```

Where the (console *Console) is a special receiver argument, and it is the indication that this method belongs to the Console type. Furthermore, if we want to return a value from a method we specify the type after the method name.
func (console *Console) Terminate() (returnValue string) {
    returnValue = "success"
    return
}

Note:
- It is possible to return multiple values
  a, b := swap("hello", "world") fmt.Println(a, b)
- In Go, a name is exported if it begins with a capital letter.
  When importing a package, you can refer only to its exported names.
  Any "unexported" names are not accessible from outside the package.

Create a constructor

Simple constructor for the struct types example:

type Vertex struct {
    X int
    Y int
}
...
v := Vertex{1, 2}

Another way, if you need more control:

func New() *Console {
    console := &Console{}
    console.property = ...
    ...
    return console
}

Note: Golang has pointers that work the same as in C/C++ (memory).
However, there is one exception. To access the field X of a struct when
we have the struct pointer p we could write (*p).X. However, that
notation is cumbersome, so the language permits us instead to write just p.X, without the explicit dereference.

**Instantiate an object**

```go
console := &Console{} // console is a pointer to the
weOsDevice.console = *console.New("console", serPort)
```

**Create a new variable**

Declare variable: `var i int`

Variables with initializers: `var c, python, java = true, false, "no!"`

Short assignment statement:

```go
var i, j int = 1, 2
k := 3 // equivalent to var k int = 3
```

**Assign variable**

```go
isProcessed = true
```

**Print to the console**

```go
fmt.Println("string to print")
```

**Comments**

Comments are the same as in C, C++, C# or Java.

One line comment `//`

Multiline comment `/* ... */`

**Note:**

- A struct cannot contain two methods with the same name even when the methods accept different parameters.
- Pay attention to the `:=` operator
Python cheat sheet

In following text, you are going to find everything you need regarding Python syntax in order to solve given tasks.

Creating a class

Creating a class in Python is a very straightforward to implement. The keyword `class` followed by name and semicolon.

Notice: Unlike Go, python does not use curly braces for defining the scope but code indentation.

    class WeOSDevice:
        """A simple example class""
        i = 12345

Adding a method to a class

    class WeOSDevice:
        """A simple example class""
        i = 12345

        def login(self):
            return 'hello world'

Creating a constructor

Classes defined in python have a default constructor, however there is a possibility to add additional constructor via the following syntax.

    class Complex:
        def __init__(self, realpart, imagpart):
            self.r = realpart
            self.i = imagpart
Instantiating a new object

x = Complex(3.0, -4.5)

Create a new variable

isProcessed = true

Print to the console

print "string to print to the console"

Comment

Comment can be added by wrapping the text inside of triple quotation marks """" for the multiline comments or # for the single line comment.