UNIVERZA NA PRIMORSKEM
FAKULTETA ZA MATEMATIKO, NARAVOSLOVJE IN
INFORMACIJSKE TEHNOLOGIJE

Zaključna naloga
(Final project paper)

Zagotavljanje kakovosti programske opreme na primeru uporabe zunanjih izvajalcev pri razvoju

(Software quality assurance on a case study of outsourced development)

Ime in priimek: Rudi Kovač
Študijski program: Računalništvo in informatika
Mentor: doc. dr. Matjaž Kljun
Somentor: doc. dr. Klen Čopič Pucihar
Delovni somentor: So-Young Kang

Koper, maj 2017

Ključna dokumentacijska informacija

Ime in PRIIMEK: Rudi KOVAČ

Naslov zaključne naloge: Zagotavljanje kakovosti programske opreme na primeru uporabe zunanjih izvajalcev pri razvoju

Kraj: Koper

Leto: 2017

Število listov: 38 Število slik: 11 Število tabel: 1

Število referenc: 23

Mentor: doc. dr. Matjaž Kljun

Somentor: doc. dr. Klen Čopič Pucihar

Delovni somentor: So-Young Kang

Ključne besede: testiranje programske opreme, zagotavljanje kakovosti, razvoj z zunanjim izvajalcem, testne procedure

Izvleček:
Diplomsko delo opisuje postopek zagotavljanja kakovosti in testiranja programske opreme, težave pri testiranju pri vpletenosti zunanjega izvajalca, ter predstavlja prilagojeno vpeljano rešitev v podjetju, ki se je znašlo v opisani situaciji. V diplomskem delu najprej na kratko predstavimo nekaj najbolj razširjenih standardov za zagotavljanje kakovosti in v nadaljevanju opišemo uporabljane izraze. V diplomskem delu nato opišemo postopek testiranja programske opreme in ga nato prilagojenega implementiramo na dejanskem primeru. V postopku podrobno opišemo proces testiranja in podamo analizo različnih vidikov kot so uporabljena orodja, izbira jezika in postopkov, ki se uporabljajo za opis in izvedbo preizkusov in pravil, ki so bila določena za to, da bi ohranili pravilen proces testiranja programske opreme. Delo zaključimo s pregledom vsebine in viri, ki omogočajo bralcu nadaljne izobraževanje na področju.

Key words documentation

Name and SURNAME: Rudi KOVAČ

Title of final project paper: Software quality assurance on a case study of outsourced development

Place: Koper

Year: 2017

Number of pages: 38 Number of figures: 11 Number of tables: 1

Number of references: 23

Mentor: Assist. Prof. Matjaž Kljun, PhD

Co-Mentor: Assist. Prof. Klen Čopič Pucihar, PhD

Working Co-Mentor: So-Young Kang

Keywords: Software testing, Quality assurance, Outsourced development, testing procedures

Abstract:
This paper describes software quality assurance and testing, why it is a problem when the development is outsourced to an external company, and examines a solution by analyzing the process of a real life company in this specific situation. The thesis starts by giving a short presentation of some of the most used standards in Quality Management in general, and proceeds by briefly introducing some of the most used terms in the paper. The third chapter tackles software testing specifically, giving an overview of some of the most used techniques, and describing how each of them relates to the specific use case examined by this paper. The last chapter describes the testing process in details, analyzing a variety of aspects such as the tools, language and procedures used to describe and execute tests, or the rules set in place in order to keep the software testing operation clean and concise. The work concludes by providing a short overview of its content, as well as further references for the interested readers.

Acknowledgement

Thank God it is finally over.

Contents

List of Tables

List of Figures

List of Abbreviations

i.e. that is
e.g. for example

1 Introduction

Several models and standards exist for quality management such as ISO 9000, Capability Maturity Model Integration (CMMI) and Six Sigma. The CMMI is described as a capability improvement model, which provides guidelines and recommendations for helping an organization diagnose problems and improve performance at any level of the organization in any industry. Similarly, the Six Sigma methodology consists of a set of techniques and tools to provide foundations for eliminating defects and process improvement "from manufacturing to transactional and from product to service" [8].

Probably the mostly used standard for quality management is ISO 9000 [9]. A Quality Management System (QMS), as described in ISO 9000:2015, is a predefined and structured set of policies, processes and procedures required for planning and delivering the services or products to meet demands and requirements of consumers and other stakeholders while meeting statutory and regulatory requirements related to a product or program [13]. It requires the company to set in place detailed work instructions, quality manuals, written quality policies and other necessary documents that define expectations as well as actions to fulfill desired quality goals and ensure a sustainable growth on the market.

QMS has four main components: (i) quality planning, (ii) quality assurance, (iii) quality control and (iv) quality improvement [rose2006]. While planning, control and improvement are as important as assurance, this thesis will focus on the later. Quality assurance (QA) is an intermediate step between planning and control that ensures the prevention of mistakes or defects in manufactured products or executed services since control takes place retrospectively.

One part of the quality assurance component is Software Quality Assurance (SQA). It is described as the process that helps prevent defects, mistakes and other issues in the final product, and helps to provide confidence that the product delivered to customers meets specific quality requirements [ISO 9000:2005, Clause 3.2.11]. SQA has a very important role in mobile software development, as by ensuring the quality of the service or product it tries to provide a positive user experience while keeping negative feedback and bad ratings at bay.

This thesis will focus on SQA of a mobile e-learning application designed by the author of a global mobile e-learning service provider - from hereafter we will refer to

it as Provider. Its main goal is to offer (and deliver) a service that meets high quality standards and expectations set by our target customer (mobile e-learning content providers such as universities and other higher education institutions).

Content providers (lecturers at higher educational institutions) using our service should have a reliable, stable and easy-to-use platform that gives them the ability to prepare and spread high quality content among students within the courses. Students on the other hand should have available fast and cross-platform applications (that can be used on mobile devices just as easily as on laptops) to follow and complete the courses provided, interact with each other by sharing their answers or posting questions, and collect their certificates of completion. Both parties must trust the mobile e-learning service provider (our company) with their data, and rely on it being always available and up to date, both while online and offline.

Since our application is available on both major mobile platforms in use today (iOS and Android), as well as a responsive web application, we (developers) need to make sure to deliver cross-platform applications that look, feel and perform the same on a large number of different devices with different specifications (the most important of them being the screen resolution). This is the reason why we decided to develop a custom SQA process that allows us to easily verify and confirm that every detail and feature looks and works the same way on a variety of different devices. The development process is outsourced and we have no control over it, so our quality assessment process must start well before the company we outsource to starts developing. We have to make sure we provide teams with design and functional specifications that are as detailed as possible and we also have to make sure all of the teams are up to date with all of the changes or issues that might occur.

This thesis will analyze the above described SQA process in detail. Despite all of the standards and best practices (e.g. ISO 9000, CMMI and Six Sigma), there are scenarios in which none of them simply "fit" a particular SQA process. There are factors that are not taken into consideration when creating standards, like the collaboration with an external company, or the outsourcing of the development process. For this reasons our approach to SQA makes an interesting case study and even though the QA of a deliverable is usually in the domain of the company developing the software, we have to make sure the software delivered is appropriate.

This thesis will focus on software testing, and will start by providing definitions and short presentations of different testing types, approaches and processes in Chapter 2. It will then propose descriptions of some of the already available quality assurance methods that are widely used, and describe whether they are, are not, or to what extent, suitable for our QA needs in Chapter 3. It will continue with a short presentation of the methods we used before developing the current process, and will end by providing

a detailed description of the process we have today, describing the tools we use, test structures we planned and testing methods we follow in Chapter 4.

2 Theoretical Background

The chapter starts by providing the most important definitions and concepts related to software testing, and describes our own interpretation and approach we have towards them when applicable.

Software Quality Management (SQM) is defined as "the collection of all processes that ensure that software products, services, and life cycle process implementations meet organizational software quality objectives and achieve stakeholder satisfaction" [11, 17]. It defines processes, process owners, requirements for the processes, measurements of the processes and their outputs, and feedback channels throughout the whole software life cycle.

There are three main quality layers used by Ian Sommerville [15] to describe SQM:

1. The **Software Quality Assurance (SQA)** layer is defined as a quality guide that includes standards, directions and various strategies to create and assess the quality of a software product during its development life cycle. Sommerville also mentions a knowledge base and tools that help apply the guides mentioned above.

2. The **Software Quality Plan (SQP)** layer is a quality plan that describes goals and possible risks (goal setting and risk management), commits projects to follow the guides and procedures provided by the SQA layer, as well as presents new tools and procedures that might arrive from different sources, that are missing from the layer described above or that are too specific to the project to be described elsewhere. The SQP and SQA are usually consistent with each other and when deviating should have a plausible justification.

3. The **Software Quality Control (SQC)** layer is the layer that describes activities that make sure both SQP and SQA guidelines are being followed by developers. Among other things it provides tools such as standard templates used to produce good documentation and the know-how required to perform good quality reviews.

For the purpose of this thesis we are mainly interested in the first layer - **Software Quality Assurance (SQA)**. SQA is defined also as a collection of activities that *"de-*

fine and assess the adequacy of software processes to provide evidence that establishes confidence that the software processes are appropriate for and produce software products of suitable quality for their intended purposes" [7]. According to ISO/IEC 15504 v.2.5 (also called SPICE - a set of technical standards related to the Software development Process (SDP)), **SQA** is defined as a *"supporting process that has to provide the independent assurance in which all the work products, activities and processes comply with the predefined plans and ISO 15504"* [10]. In other words, these activities are needed in order to ensure that the final product meets the required quality expectations, and that the software works as expected, by continuously looking for, and correcting, weaknesses found in the development process. The collection of **SQA activities** is also referred to as the **Software Development Processes (SDP)**. These processes include, but are not limited to, software design, coding, source code control, code reviews, software testing and release management. **Software Testing** is the focus of this thesis as a part of SQA. The relation of Software testing in respect to Quality Management System can be seen in Figure 1.

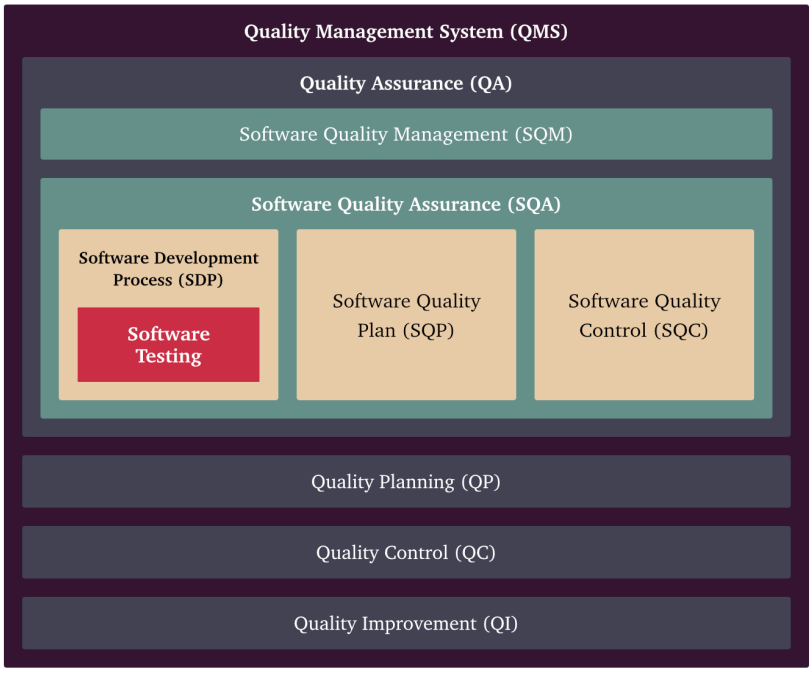


Figure 1: Quality Management System structure and relation to Software testing

Prior to describing software testing and related terms the Provider's software development process is explained. The Provider is outsourcing the development of the mobile applications (the web platform is developed in the house) to an external company - which we will refer to as Outsourcee (the party hired by the Provider that provides outsourcing services). The result is the quality of their products being assessed twice. The first time the quality is assessed by the developers and quality assessors employed

by the Outsourcee. When providing their test plans and the results of the tests performed the Provider starts its assessing process. It follows its own SQP and performs tests described in the test suite that will be described further on. It is worth noting that the Provider handles most of the SDP, with the exception of coding (managed entirely by the Outsourcee), and code reviews (executed by both parties).

2.1 Test Artifacts

Before talking about Software Testing it is imperative to define a couple of terms that will be frequently used throughout this thesis. These terms are categorized as **Test Artifacts** by WideSkill [18], which defines them as products developed by the test team during the software testing life cycle to guarantee that the testing process proceeds smoothly, and that the communication between the test team and the customer proceeds without misunderstandings. It is important to notice that in the case of the Provider some of these artifacts are not only provided by the test team, but also by the team that designs new features. Although WideSkill examines six of the several different types of artifacts, this paper will focus on just the three that are most relevant for our case-study.

The first artifact we will focus on are **Test Plans** (or Suits). According to Wikipedia, a test plan is defined as *"a document detailing the objectives, target market, internal beta team, and processes for a specific beta test"* [22]. The test plans that will be executed are known in advance to developers and the management, allowing them to be more cautious during the development phase or during a code refactor. Various elements are considered while defining a test plan, and are usually used to determine things like: (i) the purpose of the tests, (ii) the scope of testing, (iii) a testing approach, (iv) exit criteria, and (v) possible risks and appropriate mitigation plans. Sometimes the higher-management of test teams might produce higher-level documents called **Test Strategies**, which are the basis for future test planning (e.g. defining the objectives of testing, testing guidelines, requirements, etc.)

Two other important artifacts are **Test Scenarios** and **Test Cases**. Although the two are commonly used as synonyms, they cover two different aspects of testing. While the first is usually prepared after reviewing the functional requirements of a software, and is used to verify particular areas of an application, the latter is essentially a list of steps (also called actions) to take in order to verify that when given a specific input the application will return the expected result. Test scenarios can be described by a single test case or can be a combination of many.

Figure 2 depicts an example of a test scenario, in which testers have to verify the validity of a login module that might be present on different screens. A single test case

would not be enough to test such a module, as there are many distinct variables to take into consideration, such as the current location (URL), locale, types of input fields, etc. As further discussed in Chapter 5, at the Provider, test scenarios are described in the description field of tasks separating different application features.

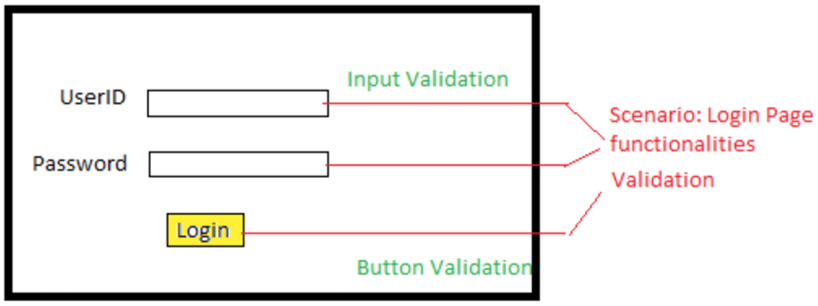


Figure 2: An example of a test scenario in which a validity of a login module is tested [22].

Figure 3 on the other hand depicts an example of a single test case used to verify the validity of a single input field, by checking that given a specific input parameter the response will be the one expected by the tester. Test cases are described by an introductory user story followed by a step by step guide describing how to execute them, each in its own task on Asana. This process will be further discussed in Chapter 5.

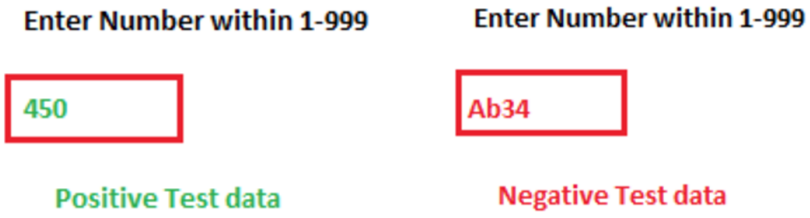


Figure 3: An example of a test case and its outcome [22].

2.2 Software Testing

SWEBOK v.3 [6, chapt. 4 page 1] defines software testing as a process that *"consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain."* Simply put, despite all of the test approaches available (mentioned later in Chapter 3.1), software testing is the only process that, given a limited number of carefully selected test cases (discussed later in Chapter 3.2), assesses how a software will behave, and

if it will behave correctly within a target environment, based on a set of predefined expectations and specifications. It is important to note that software testing has two main goals: finding and removing bugs and evaluating a specified quality of the software (performance testing, reliability, usability, etc.)

As already mentioned, one of the primary purposes of software testing is the detection of possible software failures, so that defects may be discovered and corrected before delivering the final product to the customer or end users. As the literature [12] puts it, *"testing cannot establish that a product functions properly under all conditions but can only establish that it does not function properly under specific conditions."* These conditions are called test cases, and a good tester will try to predict and reproduce as many of these conditions as possible.

During the development phase of the mobile e-learning application described in this thesis, a test suite was compiled (and is maintained on a regular basis) that lists and describes all of the issues encountered during this process. The test cases provide testers and developers with a short user story (or persona, defined as *"a simple story of the user's goals, behaviors and pain points"* by [16]) followed by a step-by-step process that helps them reproduce and debug the issues that arose. Additionally, the test suite has another important role in the development phase, as it allows development teams working on other platforms to have the same understanding of the problem and a clear description of the desired functionality, in case it was not clearly understood when analyzing the functional specification.

Swebok divides software testing into four separate levels (examined in details later in Chapter 3.3): *unit testing, integration testing, component interface testing, and system testing*. These **test level**s include different kinds of tests grouped together based on the different situations they tackle during the software development process [6, chapter 2]. Customers on the other hand have a different perspective on software testing and tend to divide it into two main categories: low-level and high-level testing. The first one meant to test specific components of the product while the second one meant to test the overall functionality of the deliverable.

In the business case described in this thesis, the Provider has a specific need to seamlessly blend different test levels while thinking of and writing test cases. In many instances, the test cases need to check the functionality of the application, the functionality of the API, and more importantly how the integration of the two affects the whole system.

Software testing does not consist of specific step-by-step procedures, but has general guidelines and rules that can usually be followed in order to perform it well. As the system gets more and more complex, these guidelines and procedures may not suffice, and a customized way of testing has to be set in place. This is the case of

the Provider in this thesis. By developing a system that is quite complex and by outsourcing the development process to another company, it was not enough to simply follow the Quality Assessment standards, principles or best practices described by Swebok and other books that tackle this problem.

This brings us to the next chapter, in which we analyze software testing in general, inspect some of the most common testing levels and techniques, describe in details some of the testing methods employed by the Provider, and determine why they are useful in their case.

3 Software Testing: A Detailed Overview

Software testing is an important part of the software development life cycle for various reasons. For one, it guarantees that the product delivered to customers will perform as expected, and that it will achieve the expected quality standards. Continuous testing during the development phase also ensures that the development speed is quicker, as the issues that might arise during the beta testing phase (or even worse when the application is already in production) can get fixed right away, without the need for a new development iteration. Lastly, it helps protecting the reputation of the company that issues the product. A curious example is the story of a climate orbiter NASA sent to Mars in 1999, built and developed by a subcontractor. Due to a silly bug left in the code by the subcontractor, which had the orbiter use English units instead of the metric system, the orbiter's thrusters worked incorrectly and it crashed upon arriving on Mars. The money loss was assessed to $327 million, but it is also worth noting that it took about a year for the orbiter to reach the planet (and we all know that time equals money) [19]

Importance of software testing of the Provider's applications by the Outsourcee's developers

Performing software tests before deploying a new build is especially important in this case study due to the fact that the applications are already in production and installed on more than 5.000 devices. Thoroughly testing new builds before their release to the general public helps preventing two very important scenarios: (i) new features might not work as expected, and more importantly (ii) they might break existing functionalities that customers are already relying on. Additionally, since the application is multi-platform, tests are also needed to guarantee that the new version on a platform does not break the applications available on other platforms, even though they are usually released at the same time, making them aligned feature-wise.

Importance of software testing of the Provider's applications by the Provider's

team

Making sure developers deliver high quality builds is still something the Provider has to verify by itself. The fact remains that the product is theirs and their reputation depends on it. Personal experience has shown that developers usually do not do a satisfying job when writing and performing tests, and quality assessors do not usually test many of the common use cases. Additionally, they usually tend to test best case scenarios to ease their work. Provider's employees, on the other hand, thoroughly test the applications and try to simulate as many scenarios as possible. For example. testing the applications online and offline, simulating situations in which users lose their network connection (or have a really bad one), exploring cases in which users cancel specific operations while they are still being executed, or manically tapping on buttons just to see if such a behavior would break anything.

For the sake of briefness, this chapter will only discuss topics that are relevant to the Provider, and the experience they acquired in the software testing field while working with the Outsourcee.

3.1 Test Approaches

During the development life cycle of an application there are two main testing techniques that can be chosen from: the first one is called proactive while the second one is called reactive. The main difference between the two is the development phase in which the tests are performed: while the proactive technique initiates the testing process as soon as possible, preventing bugs from leaking into actual application builds, the reactive technique initiates it only after the development (coding) part has been completed, hence tackling issues in a reactive manner. Due to the way the applications are developed and delivered to the Provider, its testers are forced to combine the two techniques: tests are performed after a development build has been delivered, but before the production build hits the market. An approach like this can (and unfortunately does) cause a lot of delays during the development life cycle, which in consequence slows down the time needed to deliver new features.

Depending on the context of the project, one can choose to adopt different approaches to software testing, including (but not limited to) approaches, that according to [14] are:

1. Dynamic and heuristic (or experience based) - tests rely on educated guesses based on past data.

2. Consultative - tests depend on the advice of technical experts outside the test team.

3. Model-based - tests are designed based on mathematical or statistical models.

4. Methodical - tests depend on a predefined testing method.

5. Standard-compliant - tests are based on externally developed industry standards.

There are, however, several factors to be considered when deciding which approach to follow as not all of them might suit one's need due to a number of reasons. Some examples being:

- Testers might not have the appropriate experience or expertise to work with the tools proposed by the test approach.
- The development process and test approach might not be compliant with the internal regulations of a company.
- The proposed approach might simply not fit the nature of the product.

The Provider decided to opt for a combination of the dynamic and methodical approaches, seeing how their existing versions of the applications are being used numerous customers, and based on the issues have already been discovered (and fixed) during their lifetime.

3.2 Test Selection

As mentioned in the introduction of **Chapter 3**, test selection takes a very important role in software testing. By carefully crafting a list of the most important and relevant test cases, many of the issues that might arise while using an application can be prevented beforehand. Furthermore, a list of meticulously selected and well described test cases will usually save testers a lot of time, which consecutively saves the client a lot of money. An important thing to remember is that the definition of a good test case does not actually exist. A particular test that might simulate one particular situation might not cover all the elements in another one. As discussed in a paper related to software testing [3], a frequent interpretation of what a good test case is, is the ability of the test case to detect as many failures as possible, opposed to a test so simple and generic that will always pass. This is why, even though we have defined different test approaches, a good practice is to always combine different testing techniques, instead of focusing on a single one.

3.2.1 Selection Based on Code

"*[...] These criteria are also called path-based, because they map each test input to a unique path p on the flowgraph corresponding to the reference model. The ideal and yet unreachable target of code-based testing would be the exhaustive coverage of all possible paths along the program control-flow. The basic test hypothesis here is that by executing a path once, potential faults related to it will be revealed.*" [4]

The selection of tests based on code will be the primary focus of this thesis. Being the designer of their applications gives the Provider the ability to think and predict different paths their users will take to access and use new features. Although an aim to strive for, full path coverage is technically not applicable, but with Provider's specific approach and a careful selection of test cases examined in detail in **Chapter 4**, they try to cover as many different and relevant paths as possible. Additionally, describing them in a step-by-step manner helps testers and developers find and reproduce issues that might otherwise affect end users.

The test suite is always available to all developers as an Asana[1] project, and is constantly updated, either by the designer that writes the specifications of new features, or by another member of the Provider's team that handles testing. New test cases are regularly added to the suite, including in instances like:

- while designing new features,
- while using the application,
- when developers require additional specifications,
- after reproducing issues reported by users, or
- after receiving notifications about crash reports.

When using the application to try and predict possible issues, the Provider tends to use the destructive testing technique, which is a technique that tests the stability of the application and tries to find its points of failure by testing it in an uncontrolled manner (such as sporadic tapping on buttons, turning on and off airplane mode, etc.)

3.2.2 Other Selection Criteria

Error guessing

This is another testing technique extensively used by the Provider. The applications are tested by skilled employees with a lot of experience in the development sector, as

[1] A web and mobile application designed to help teams track their work on projects: `https://asana.com`

well as past experiences testing applications with similar capabilities. Their experience, intuition and skills are crucial in identifying defect that more formal techniques might have missed. This technique is usually applied at the end of the testing period, when all of the more formal ones are completed.

Operational Testing

Operational acceptance testing (OAT) is a testing procedure that focuses mainly on testing the application in an environment that simulates the production environment. This technique verifies that the product is operationally ready, and is often included as part of the Software Testing Life Cycle (STLC)[2]. This technique will not be extensively discussed in this thesis as the Provider focuses primarily on system tests (described in **Chapters 3.3 and 4**) and performs them in a staging environment that mimics the production one.

Many more testing techniques are available and widely used globally, like the Selection of tests based on specifications, and Mutation testing, but this thesis will not focus on those as they are not used by the Provider during the testing phase. Nevertheless, due to the complex nature of the applications examined in this document, it is not possible for the Provider to simply focus on a single technique. The testing process is a combination of several methods (described above) that range from code reviewing and analyzing to operational testing on copies of the production environment.

3.3 Testing Levels

"*Software testing is usually performed at different levels throughout the development and maintenance processes. Levels can be distinguished based on the object of testing, which is called the target, or on the purpose, which is called the objective (of the test level).*" [6] In other words, Swebok defines two main testing levels categories: (i) tests that have specific targets such as modules, groups of modules, or entire systems, and (ii) tests that have specific objectives, that can aimed at functional properties (e.g. verifying that the functional specifications were correctly implemented), or at non-functional properties (e.g. performance and usability tests), discussed in Chapter 3.3.5.

Levels are generally divided into four categories (see Table 1 for a brief description), based on when they are added in the SDP, or based on their specificity [21]: (i) unit testing, (ii) integration testing, (iii) component interface testing, and (iv) system

[2]Software Testing Life Cycle (STLC) is the testing process, which is executed in systematic and planned manner. In STLC process, different activities are carried out to improve the quality of the product.

Level name	**Short description**
Unit Testing	Used to verify that specific sections of code work, and that they respond as expected. They are usually very specific, and an example of a very minimal unit test could be simply testing the constructor or destructor of a class.
Integration Testing	Used to verify that different components are correctly integrated into the system, and that they correctly interact with it. Test cases can include testing procedure calls or the activation of different processes.
Component Interface Testing	Used to verify that the data is correctly passed between different units (or subsystems) and processed by the next component. A simple test could be sending a bogus data package from a component and verifying that the next one processes it correctly.
System Testing	Used to verify that a completely integrated system meets the requirements specified by the customer. An example could be testing that the registration process for a new user works as expected: starting from the user creation form, to the dispatching of a welcome email, to the user login form and then user logout.

Table 1: Brief description of the four main testing levels

testing, with (i), (ii) and (iv) defined as main levels during the development process by Swebok. Additionally, as previously mentioned in Chapter 2, there are also two levels of tests from a customer's perspective, one being low-level (components) testing, and the other one high-level (whole system) testing.

The rest of the chapter is dedicated to a detailed description of the various testing levels, and how they fit into the Provider's software testing procedures.

3.3.1 Unit Testing

Unit tests are not meant to be a replacement for classic Quality Assurance (QA) processes, but are instead meant to eliminate errors during the coding phase, before a build is pushed to the next stage (QA), and should be kept up to date (new features require new unit tests). Their goal is to prevent functions (in the code) from returning

the wrong responses, increasing the quality and efficiency of the product before it reaches QA. These tests can be written either before actually producing any code relevant to the software (and then write the software based on unit tests), or after the software has been written.

The Provider does not have control over such tests. Based on past experiences this usually results in buggy software builds delivered to the QA team, that have a lot of issues that could be easily prevented by unit tests (e.g. the handling of unexpected values in input fields). This behavior consequently results in a delayed delivering of application updates (with requested features) to their customers.

3.3.2 Integration Testing

As briefly described above, integration testing is the phase in which multiple components (that have previously been unit tested) are combined into groups, and tested as such. If unit tests prevent specific components from returning invalid data, integration tests are used to make sure that the aggregation of these components work correctly and that their output is a correctly integrated system, ready to be passed to the next phase: system testing.

Same as with unit testing, since integration testing is a process that occurs before an application is built and passed to the QA team, and since it is built on unit tests, in the case study examined in this thesis these tests cannot be run by the Provider but should be performed by the developers.

3.3.3 Component Interface Testing

Component interface testing is a process that succeeds integration testing. The main focus of these tests is to verify that data (or message packets) passed from one component (or a group of components) to the other is valid, instead of being interested in how the components work. These tests can check the validity of data types passed between units, the reaction of units when they are passed unusual data, or data with extremely large values, making sure that the preceding components do not affect the performance of their successors.

In the case of the Provider's mobile applications these tests are not performed in the code but rather by the QA team — only after receiving a new application build. The testing process is done by hand which makes it much slower, but the QA team has a wider knowledge of the data (and data types) that is allowed and will be used by the applications. The latter essentially act mainly as the "view layer" of the whole platform, and rely on the API provided by the Provider to filter and clean bad inbound data. A substantial amount of component interface tests are performed as part of the

API testing process, which is not the topic of this thesis, as it is developed internally by the Provider.

3.3.4 System Testing

System (also known as end-to-end) testing is the process that verifies that a completely integrated system meets the requirements set by the functional specification. Tests are written so that they can confirm that multiple components correctly interact between them, that their outputs are the expected ones and that the functionality of the system as a whole correctly processes the requests made by users of the application. An additional example to the one briefly described in Table n could be the process of signing into the application using a Single Sign-On[3] (SSO) access (e.g. Facebook login), verifying that the user data is correctly fetched from the local database, posting something as the user, removing the post, and then logging off.

Based on the description above, we can conclude that system testing is a full suite of tests, that are usually ran only after all of the other levels have been processed, as they take a lot of time and manpower to complete. At the Provider's', system tests are executed manually based on the test suite that will be further discussed in Chapter 4, usually (i) before issuing new features that have a major impact on the system, (ii) before issuing a major update, or (iii) after a large refactor. The Provider recently had to rewrite a large portion of the iOS code, updating it from Swift 2.2 to Swift 3, due to a new major iOS version release by Apple. The new iOS version impacted not only the code written by the developers working on the iOS application, but also the code produced by other developers that was included in the application in the form of external components. This was an example of the perfect moment to perform a complete system checkup, to make sure everything worked as expected, before declaring that the new version of the application was ready to be rolled out to users.

3.4 Testing by Objective

"*Testing is conducted in view of specific objectives, which are stated more or less explicitly and with varying degrees of precision. Stating the objectives of testing in precise, quantitative terms supports measurement and control of the test process.*" [6] Although Swebok categorizes Testing by Objective as a subcategory of Test Levels, we are going to discuss them under a separate chapter, as they have an important role in the testing

[3]Single sign-on (SSO) is a session and user authentication service that permits a user to use one set of login credentials (e.g., name and password) to access multiple applications.

process at the Provider's. While Swebok analyses more than ten different types of tests, this sub-chapter will only discuss those that are used the most by the QA team verifying the suitability of new builds, and give a short comment as to why those tests are relevant in their specific case.

The quickest tests performed are called **smoke or sanity tests**, which are quick and shallow tests used to determine whether it makes sense to actually proceed with further, more in depth testing. These tests usually consist of basic operations that cover the most important functionality of a software, and are the first tests to be performed when a new build is delivered to the QA team. The failure of these tests will most likely result in a rejection of the build, which will be returned to the developers. An example of a smoke test could be to simply run the application and try logging in. At the Provider's, when a build with new features is delivered to the QA team, a couple of pre-defined sanity tests are executed. If those pass, they move on to a more detailed testing technique. To prevent additional delays during the test phase testers may choose to continue testing (even though one (or more) of the sanity checks failed) if they decide that the affected features are not crucial, and do not prevent them to perform additional tests.

A different type of testing is called **Graphical User Interface (GUI) testing**, which is described as the process of verifying and ensuring that the GUI of a software meets the design specifications that was provided along the specifications of new feature requests, and usually includes more than a single test case. These tests are especially important in the case-study examined by this thesis, and a lot of effort is put into testing this aspect of the two outsourced mobile applications. Considering the mobile applications are the Provider's main product, as well as the main tool used on a daily basis by their users to follow and complete the mobile courses made available to them, it is very important that they are programmed exactly as they were designed. The design team invests a lot of time into making sure that the user experience will be as best as possible, and a lot of effort is put into carefully designing new features and providing developers with exact blueprints of how the applications should look and feel. When delivered a new test build, the test team has to make sure that the final product looks exactly as designed by their colleagues.

Another type of testing is called Regression testing, which is the process that focuses on detecting issues with existing features after a large code refactor. As Ammann and Offutt describe in their book, "*Regression testing is the process of re-testing software that has been modified.*" [2] Issues could include old bugs and lost or broken features that used to work before code modifications, and might happen due to a variety of reasons, one of which could be the collision of parts of code handling the same features. This type of testing can be performed by e.g. retesting old test cases, or verifying

that issues that might have been fixed in the past are not present in the new build. "*Regression testing constitutes the vast majority of testing effort in commercial software development and is an essential part of any viable software development process.*" [2] The Provider gives regression testing an important role in the testing process and a lot of effort is always put into testing new releases, especially those related to the back-end, which affects all of their mobile applications. Thoroughly testing the backward compatibility of their API and other server-side code guarantees that customers and end users using their mobile applications will never even notice any of the code was changed, as the applications will continue working without interruptions and other issues. Regression tests are also performed on their mobile applications before rolling out a new update, to make sure they still function at least as well as the previous version.

Alpha testing is usually performed at the developer's site, either by potential users or a dedicated test team, both of which work in unison with the developers. "*During alpha testing, a serious effort is made to identify and correct problems resulting from new code by a coordinated effort between the programmers and testers.*" [5] When all alpha tests pass, the software gets promoted into the **Beta testing** phase, which is a form of user acceptance testing usually performed by a limited audience of external users not related to the team of developers (also known as beta testers). Recently, many large software development companies (such as Google, Microsoft and Apple) started opening their beta software programs to the general public, increasing the input they get from their actual users, allowing them to really optimize their products and deliver high quality versions. Software can include (but is not limited to) web and mobile applications, video games, desktop software, etc. A good example of a software left in a perpetual beta ("*keeping of software at the beta development stage for an extended or indefinite period of time*" [20]) is Gmail by Google, which finally exited the beta phase after more than five years of being available to the general public. As expected, alpha tests on the Provider's mobile applications are performed by a test team employed by the company developing the applications. When these pass a new software build is pushed to the beta testers at the Provider's, which follow specific test suites to verify that the build is appropriately developed.

After verifying that an app works as expected while using it for basic operations, the test team proceeds with a more destructive testing technique, which is a method that "tries really hard" to crash or cause a software to fail. Examples of this technique could include (i) voluntarily providing software with bad, invalid or extremely large amounts of input data, or (ii) quickly tapping on buttons that perform specific operations. At the Provider's this type of testing is usually performed by the test team as the software is primarily mobile, but a lot of free and open-source software dedicated to destructive

testing can be found online. This testing technique has proven really useful when trying to find and reproduce issues caused by edge case scenarios.

The last testing technique we are going to discuss is the usability testing technique. This method is used to verify that a user interface is straightforward and easily understandable by end users. It usually consists of designers or testers heading out of the office to meet users in person, in places where they feel comfortable using the application and expressing their "feelings" using body language while doing so. According to an example from Wikipedia, "*ninety-five percent of the stumbling blocks are found by watching the body language of the users. Watch for squinting eyes, hunched shoulders, shaking heads, and deep, heart-felt sighs. When a user hits a snag, he will assume it is "on account of he is not too bright": he will not report it; he will hide it*". They continue by explaining that in circumstances like this the person watching the struggling user should "*not make assumptions about why a user became confused*", but instead ask him for an explanation, as "*learning what the user thought the program was doing at the time he got lost*" could be an invaluable information that would never be available to them otherwise [23].

3.5 The reason behind a "custom" approach

As already mentioned in the previous chapters, testing is not one size-fits-all solution, and what works in a specific scenario might not function well in another. The situation described by this thesis is so specific, that using a single approach would have been fruitless, and completely trusting the test team employed by the external company would have been a disaster. This is why the Provider had to think of a not-so-standard approach that would allow the external and internal teams to be aligned. Since techniques like unit and alpha testing start well before developers deliver new builds to the beta testers, a system had to be put in place that would try to mitigate the issues that could arise, and provide developers with user stories and test cases relevant to the new features they would have to develop. Additionally, seeing how adding, modifying or removing a feature does not only affect the feature itself, a system like this also helps by providing an overview of all the features related to the one in discussion. By working in a tool called Asana (further described in **Chapter 4**), used as the main project management tool, the Provider is able to keep a link between relevant test cases, previously solved issues, or anything else that might be useful during the test phase. Additionally, they always provide developers with sample API calls they will need when integrating and testing new features, and although the developers always perform basic tests to see if new features work as expected, the Provider's test team always has to perform additional checks to verify the API behaves as expected on the

backend, and that the responses returned are the correct ones.

To conclude, it is also important to note that the developers do not usually have a direct access to the whole infrastructure, which is set up and maintained by the Provider. There are specific situations or edge cases that can be only correctly tested by the Provider's test team, which has a complete overview and understanding of the whole system, from the data to the view layers. Hence, it is very important that final tests are always performed by this team, and that the confirmation that a new build is ready to be deployed to production is issued exclusively by them.

4 The Provider's Approach

This chapter will provide an insight into the testing methods, tools and procedures employed by the Provider during the testing phase. It will start by giving a short presentation of Asana, a project management tool (that has been in this case also used for the implementation of test plans), and proceed by analyzing their strategy regarding the structure of test suites and test cases, describe how the Provider tackles the actual testing, and conclude by briefly describing their approach toward bug reporting.

4.1 Asana as the Tool of the Trade

Asana is a web and mobile application designed by Facebook co-founder Dustin Moskovitz and ex-engineer Justin Rosenstein whose work has been improving the productivity of employees at Facebook. Because of the numerous features Asana offers it was first used as a project management tool by Provider's team. Its features assist the teams with their daily work, such as support for quick project duplication, a simple but powerful user management, a straightforward reorganisation of tasks, facilitating and tracking communications between team members, the support for file attachments, and many more. These are also the reasons why the team decided to use Asana as their main tool for documenting and tracking software testing.

In a nutshell, Asana holds a collection of Projects that reside under different Workspaces, and a list of tasks grouped by Projects. It features a narrow sidebar that lists the Projects that belong to the currently active Workspace, while the main window features a two panel design that perfectly adjusts on screens with different resolutions. The left panel displays a list of tasks grouped by Sections (or tasks) that can be highlighted, while the right panel displays the information of the highlighted task. Each task has a wide selection of attributes, options, and can have its own subtasks. A graphic representation of the hierarchy can be seen in Figure 4, and a screenshot of the actual design is available Figure 5. Each of the features will be further examined in this chapter along with a short explanation of why it is useful to the Provider.

Figure 4: Graphical representation of the hierarchy in Asana.

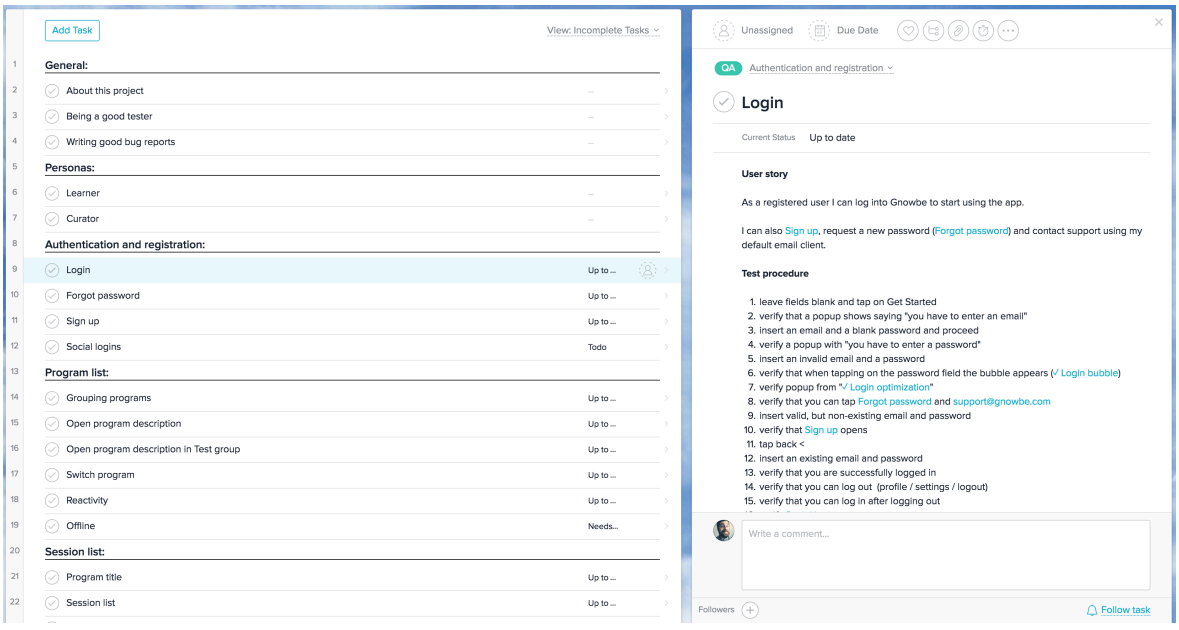


Figure 5: Left panel with a list of tasks and right panel with details of the task highlighted on the left.

4.1.1 Workspaces

A Workspace allows for people to collaborate together on projects and tasks. They can include as many people as wished and do not require a common company email domain. This makes them an ideal solution when collaborating with users from different organizations that work on the same projects. Administrators can easily manage accounts and their access rights, making sure that each user sees nothing more than what it needs in order to finish the job.

The Provider decided to work with a single Workspace accessible by all of its employees and the employees working on their mobile applications employed by the Outsourcee. While their employees have access to all of the Projects available in the Workspace, the teams provided by the Outsourcee only have accesses to specific Projects relevant to their positions. Such a procedure guarantees a quick but detailed overview on all of the users working on specific projects and tasks, as well as provides the administrators with a tool that helps them easily manage users.

4.1.2 Projects

In Asana projects are made of lists of tasks. Project are typically used for a larger goal while tasks are used for actions needed to be taken in order to achieve that goal. Projects are one of the main features in Asana. A Project is a list of Tasks that can be grouped by Sections. Each Project can allow access to different users from the same Workspace. Projects can easily be duplicated in the same Workspace or copied to another one. This process also handles the duplication and/or copying of the Tasks created inside of them. Users using the duplication feature can easily choose which information to include in the process: the procedure can be as simple as simply copying the name and description, or it can be as complex as copying every bit of information, including Task due dates, attachments, project members, etc.

Duplication makes Projects a very powerful tool, and this feature is used a lot by the Provider during the testing phase, as it allows them to always work on a clean copy of the test suite while keeping a detailed record of the past test results. Figure 6 depicts a list of projects within a workspace, with a menu showcasing options for one of the projects.

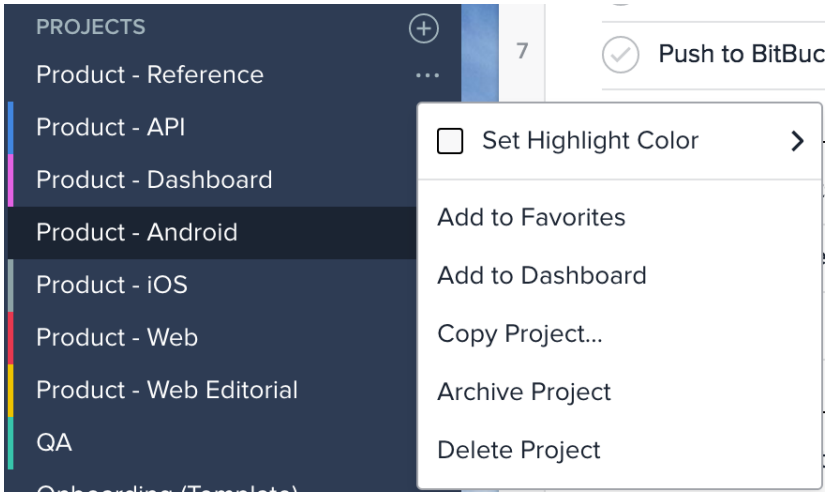


Figure 6: List of Projects inside a Workspace.

4.1.3 Sections

Sections are usually used to group Tasks that fit into the same categories inside Projects. Section separators are nothing more than Tasks that have their titles ending with a colon (:). To make it obvious they are separators they feature a slightly different styling when compared to normal Tasks; they appear with bold and underlined text. Additionally, they work the same way as Tasks do, and can include a description, attachments, assignees, a due date, etc.

The Provider makes extensive use of Sections in order to clearly group test cases targeting specific groups of features as can be seen in Figure 7. Their descriptions usually include a generic user story and relevant links to other available documentation.

4.1.4 Tasks

Tasks are the basic unit of action in Asana. One can create new tasks, duplicate existing ones, merge tasks together, print or delete them.

The Provider uses Tasks to describe specific test cases and has defined a set of guidelines detailing how to write them, which will be analyzed later in this thesis. Each Task has a set of attributes that can be set, such as the title, a description, a category, tags, etc. Figure “sections and tasks” shows a group of Tasks belonging to the same Section, with titles set so that testers know right away what is the feature they are going to test. An important attribute of Tasks is the current assignee, as it allows the Provider to keep track of the user that is currently working on the Task. Another powerful feature is the notification system implemented by Asana: modifying a Task broadcasts a notification to all of the users that were assigned to the Task at least once or added as followers. All of the above (i) allows the communication between

users working on the same Task to be prompt and clear, and (ii) helps the Provider to optimize the whole process when needed by keeping track of all the changes.

Figure 7: Sample Section with Tasks in Asana.

4.2 Structuring Test Suites

As briefly described in the introduction of this chapter, each test suite resides in its own project, each project is divided into sections that represent a high level structure of applications, and each section includes a group of tasks that either represent and describe a single feature, a group of related features, or a whole process.

Single test cases try to cover as many use cases as possible by providing developers with one or more short user stories (a story for each persona (see Chapter 2.2 Software Testing) when applicable), and a step-by-step testing procedure. Furthermore, test cases also include a set of implementation instructions, design guidelines and references to old issues that might have occurred while developing similar features (or the same one when doing a code refactor), that are already described in bug reports and have most likely already been fixed once. While most of the test cases are written while designing new features, many of them are added or modified by the test teams while testing new application builds.

Issues encountered during the testing phase are reported inside their appropriate Asana project based on the platform the application runs on. After being verified and confirmed by a supervisor, tasks are re-assigned to the leader of team responsible for that application, which then assigns them to the developer in charge of the feature causing issues.

As already mentioned, tasks can include references to other relevant tasks, especially when they are based on the same components or if they are somehow related to each other. Having these links available has proved to be a useful tool for the developers during the development phase, as by perceiving features as a part of something bigger they are more prone to finding and solving issues before delivering a new testing build.

4.3 Designing Test Scenarios and Test Cases

After trying out several widely used methods for writing and structuring test cases, the test team employed by the Provider decided to try and develop a procedure that would be ultimately tailored to their specific use case. They ended up with a solution based on a set of specific guidelines briefly described in this subchapter that has both helped and allowed them to perform a good job when testing new builds and reporting issues to the developers.

Sections in Asana have a very important role in this procedure as they are used to group all tasks related to a specific functionality. As we briefly mentioned in Chapter 2, sections could be also called **test scenarios**, while tasks could be called **test cases**. As can be seen in Figure 7, it is important that their title is always short and as concise as possible, and that it briefly describes a high-level functionality of the application. The title itself should give enough information to returning developers and testers and allow them to understand what the tests will be about by reading just that. Nevertheless, opening the detailed view of a section should provide a brief overview of the functionality by providing one (or more) short user stories explaining its use.

The same is true for **task titles**, which should briefly describe features that the test cases will verify. However, when describing tasks titles should provide labels of specific features instead of their high-level description, and these features should be further described in detail in the description field provided by Asana, as titles alone could not give developers and testers the overview needed to perform precise tests.

This brings us to the **description** field, which is used to describe the actual test case in detail. The first thing that should be provided by the person writing the test case are relevant **user stories** [1] that should give a short example of how a feature is going to be used by the end user. These stories represent an extremely important part of the test case, as testers and developers can really understand what the feature is about, and can try to change their mindset into the mindset of the user. This way, the testing becomes more realistic, simulating a real life experience instead of focusing on testing exclusively that feature alone. An example could be a user story of the login functionality of an application: testing a story describing how a user opens the application, navigates to the login screen, inputs their credentials and taps the login button can cause issues that could not have been found by simply testing if the action triggered by tapping on the login button works correctly. Linking **relevant tasks** and providing **additional documentation** has also proven to be a really good strategy, as this gives developers and testers an overview of what has already been done, an idea on where they can find relevant code snippets, the information about who was working on the task, and an insight on issues that might have occurred during the development.

Figure 8 displays a screenshot of a task that includes the details mentioned above..

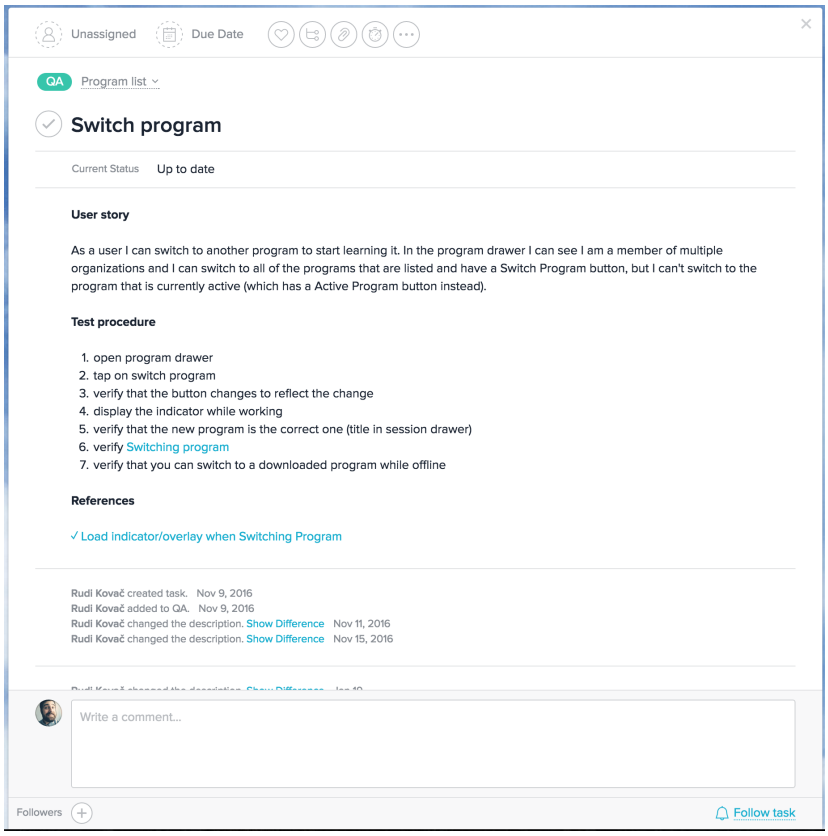


Figure 8: Sample task with description view opened.

The last two important features we are going to briefly describe in this subchapter are **attachments** and **comments**. **Attachments** are usually used to provide screenshots, sketches or documentation of new features that were not included in the original designs or functional specification. Additionally, when writing bug reports (explained in detail further on) attachments are used to provide screenshots that demonstrate the issue and how it affects the application.

Comments are used as the main communication tool between designers, testers and developers. Experience has shown that an open communication platform tends to work against productivity, as the information is usually shared only between the limited number of participants that joined a conversation. Using comments to communicate makes all of the information available to everyone involved. Nevertheless, other communication platforms are also used in specific cases, especially when developers and testers working remotely are trying to fix issues that would take much longer to resolve by going through a channel like Asana.

4.4 Performing Tests

Thoroughly testing new features, or verifying that existing features that had their code refactored still work correctly, has proven to be a task that is much more difficult to perform well than one could imagine. Especially as the complexity of an application grows with time. This is why even though the designer (or developer) will always prepare one or more test cases (as per the guideline set up by the Provider) that will be executed to make sure the feature works as expected, they will usually serve only as a guide to the quality assessor (QA), who will have to perform (and document) more thorough and detailed tests. Use cases described by the designer are used to better explain the requested functionality, and each of them will usually only describe the best-case scenario[1].

After performing the tests described in the test suite, a good QA will (almost) always be able provide a fresh selection of test scenarios and test cases (which will be initially most likely filed as bugs), by using series of different techniques like destructive and usability testing. Experience has shown that by simply using an application with the mindset of a power user[2] instead using it as an average user (or even worse as a developer) will almost always cause features to work in unexpected ways, or even worse cause them to stop working all together. Additionally, QAs must also think outside the box and try to imagine various edge cases, such as their applications running on slow or flaky mobile connections (on airplanes, trains, etc.), users having unexpectedly large amounts of content available, users that love the application and use it daily for vast amounts of time, or users that dislike it and only use it occasionally. All of the features that are available to users at any given time should work as expected, and more importantly should never cause users to lose any data.

The list below includes a couple of guidelines taken directly from the Provider's project in Asana and gives some directions that the QA should follow when performing tests. As previously mentioned, tasks might include links to related tasks, issues, or links to the original designs when they are available elsewhere, and the QA should take all of these details into consideration in order to do a thorough job.

- When there are **linked tasks** in the test procedure, go **check them out**.
- **Always** check the **Bugs** section (at least **read the titles**), as those are problems we **already had** with that feature and **had fixed them**.

[1]Being, relating to, or based on a projection of future events that assumes only the best possible circumstances

[2]A power user or experienced user is a computer user who uses advanced features of computer hardware, operating systems, programs, or web sites which are not used by the average user.

- Don't be afraid to **tap everything** there is to tap (**multiple times**), users will not always follow the same pattern to do something (ex. share - unshare, like - unlike).
- Don't be afraid to **cancel a process** every now and then, the app might crash while waiting for an API response, an upload to finish, or because of other weird stuff.
- Don't be afraid to **remove content** once it's there: remove photos, remove answers, remove journey posts, remove your name.
- Always check InVision[3] for design specifications when testing new stuff, and don't approximate: follow the design, it's there for a reason.

This subchapter will conclude with a quote taken from the same project on Asana, and the thesis will then proceed with the last topic examined: **Reporting Bugs**.

"*"Don't be afraid to improvise, break, or bend the rules, and use the app in ways the developer and designer did not think of. The main goal is to find all of the bugs and errors that might happen when using the app heavily, by 'pro' users. These users will use the app very quickly, will tap around a lot, and will perform actions that users that are just starting will not. These users might remove or edit things and will most definitely use the application while offline or with a bad network connection.*""

4.5 Reporting Bugs

The last topic tackled by this thesis is bug reporting. This subchapter will examine the procedure implemented by the Provider, and describe their interpretation on how bug reports should be submitted in order to allow developers to easily reproduce them, subsequently giving them the ability to start working on them without major delays.

When opening a new task in the section dedicated to bug reports, reporters should make sure to submit issues that are well structured, have a description that explains the problem as clearly as possible, and possibly even provide a list of steps needed to reproduce it. Delivering a report like this will give developers the essential information needed to find the bug, and chances are they will realize right away what went wrong.

The Provider identified the **title** of the bug report as the most important piece of information. Their directions are to write short and very concise titles, that should include the name of the broken feature and use the word "should" to describe the

[3]InVision is the name of the service the Provider uses to deliver designs to the Outsourcee.

expected behavior - in the least possible amount of words. An example taken from Asana is the following: "Switching to an open program should put users to 1st session". Ideally, the developer should be able to understand what the issue is and locate it by simply glancing at the title. If the issue needs further explanation, as the amount of space available in the title is limited, the **description** field should be used to describe a short user story, the expected behavior, and obviously why the feature failed the test. This is also the place to include a step-by-step procedure that the developer and other testers can use to reproduce the issue. If the issue is only related to specific steps and not the whole feature, those steps should be clearly marked.

Attachments should be used to provide developers with relevant screenshots, or even better videos, that clearly display what the issue is, where it is located on the screen, why it works incorrectly or how the bug affects the content of the application.

Comments should be used as the main communication medium between the bug reporter and the developer handling the issue, should always try to be on point and provide as much information as possible when needed by the other party.

And finally, a very important part of bug reports are the **assignees** currently working on them. This feature allows the Provider to stay on top of the current situation at any time, by simply looking at the profile photos of the users currently assigned to the tasks. Known faces usually mean tasks are ready to be re-tested, while the unknown ones mean the tasks are being worked on by the developers.

5 Discussion and Conclusion

Even though several well-defined models and techniques tackling the problem of quality assurance and more specifically software testing already exist, one should always try to develop an approach that would best fit their specific case. The approach described in this thesis is far from perfect and might not be useful in a different set of circumstances, but it has proven to be useful to the Provider, that has been successfully using it for the past year. Additionally, due to the ever-changing nature of a software product, the guidelines have to be constantly kept up to date, making it harder for them to become obsolete. The important thing to remember is that software testing is not a simple task, should be approached with an open mind and ready to break features: finding and fixing issues during the development phase is much better than shipping a product that will cause pains to users.

By approaching software testing the way it was discussed in Chapter 4, the Provider is able to always release a stable and reliable product. Additionally, constantly testing the applications before releasing new versions to the general public allows them to deliver updates that are at least as good as the application already installed on the device. It can be argued that the major downside of this approach is definitely the time needed to deliver updates to customers, as testing has to be performed at least twice when the Outsourcee delivers a version that is flawless as can be seen in Figure 10. Before employing the method described in this thesis, the testing has been done by the Provider only (see Figure 9). However, compared to current testing procedure, the old one took even more time as the product could be returned to the Outsourcee several times even for the smallest issues that might have been discovered by performing relatively simple tests. Moreover, the fact that the Provider and the Outsourcee might be located in different time zones, this process can be further prolonged by the fact that both parties work out sync. With the new approach the developers first push a build to their local test team which takes care of the initial testing and, when done, releases the build to the Provider's team of testers, that tests the build one last time.

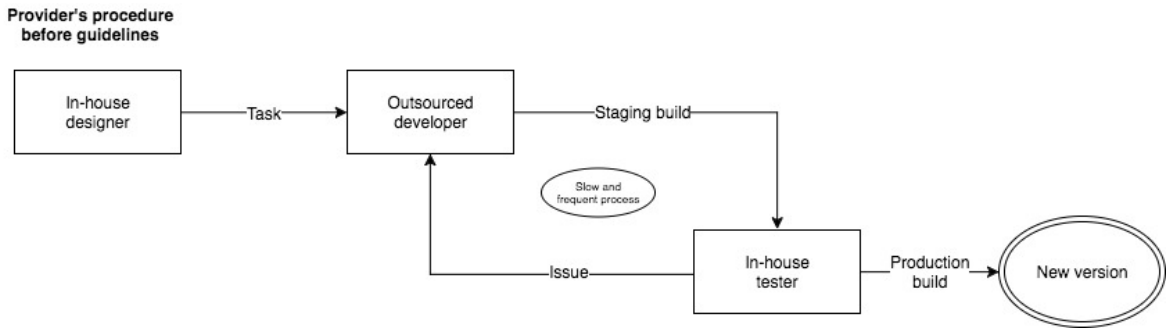


Figure 9: Diagram of the procedure **before** the approach examined in the thesis.

This system - examined in details in Chapter 4 - has now been in use for over a year and it has proven to aid developers and testers employed by the Outsourcee to produce and deliver better builds with fewer issues. Since most tests are already performed by the Outsourcee, the test team employed by the Provider has a relatively simple task, which consists mainly of testing with a destructive approach, to make sure the applications are really stable. Since testing has now been performed by both parties there are less returns to the Outsourcee. This approach also improved the time needed to deliver new features to end-users, which dropped for more than 25% per feature. One might even argue that the Provider now needs to pay the Outsourcee to do the tests on their side. Even if this is the case, the time saved by testing the application builds by both parties is (i) saving the Provider's testing team time, which can be devoted to other things rather than to writing bug reports for the developers, and (ii) saving money since features can be rolled out to customers quicker than before.

The approach described in this thesis is far from being perfect. It currently suits the Provider's workflow. Nevertheless, it is still regularly being updated and the rules written in Asana are not set in stone but are instead used as guides that help better perform a specific task. Test scenarios and test cases have proved to be useful documentation resources, as they are always available to the whole team when in doubt. The guidelines defined to better fill bug reports are also a perfect literature for the business team in contact with the actual end-users, as they are actually the ones that always encounter real life issues (compared to issues encountered by testing teams) and feel pain when something does not work as expected.

The diagram in Figure 10 could even be further complicated by the Outsourcee outsourcing the testing part. As explained earlier, the Outsourcee employs both developers and testers locally, but in the globalised world this could easily be the case. However, there is certainly a limit in the number of subcontractors if the software to be delivered is to be any good since the of communication between the Provider and the last subcontractor in chain can be distorted.

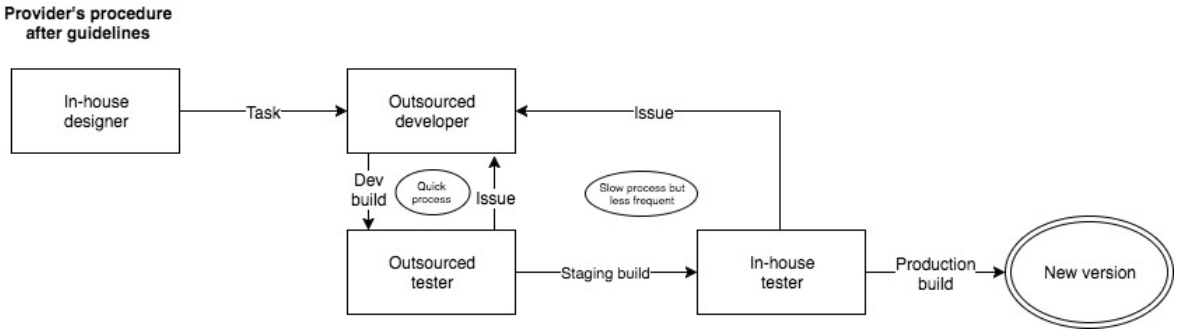
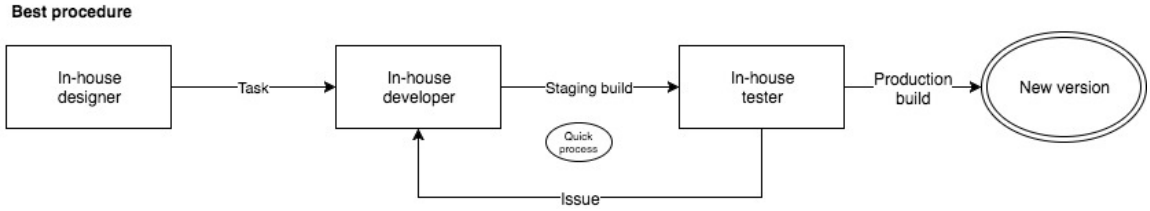


Figure 10: Diagram of the procedure **after** the approach examined in the thesis.

The best solution, albeit more expensive for the Provider at the current stage of company's development, would definitely be having an in-house team of developers directly supervised by the Provider, which could constantly work alongside the team focused on testing, drastically lowering the time needed for feedback to travel between the two parties, subsequently decreasing the time needed to ship new versions as can be seen in Figure 11. The reason behind the choice made by the Provider to outsource the development of their products to an external company was the fact that they started as a startup company, and the costs were much lower than the alternative (having a local team of developers).



pov

Figure 11: Diagram of the best approach with an in-sourced development team.

6 Povzetek naloge v slovenskem jeziku

V zaključni nalogi so predstavljeni najbolj razširjeni standardi za testiranje programske opreme, čemer sledi pregled uporabnosti teh standardov v primeru, ko je razvijalec programske kode zunanje podjetje ali zunanji izvajalec (v nadaljevanju Izvajalec), ki ga najame lastnik programskega izdelka (v nadaljevanju Podjetje). Zaradi omenjenega pregleda, kljub vsem že preverjenim standardom v uporabi v drugih podjetjih in projektih, so v Podjetju v zadnjem letu razvili lasten sistem za vodenje testnih primerov in sistem za sledenje poteka testiranja. Sistem sicer črpa iz omenjenih standardov za testiranje programske opreme. a se od teh tudi razlikujeta. Naloga opisuje kako je prirejen sistem Podjetju omogočil, da je porabljen čas, ki je potreben za razvoj, testiranje in pripravo novih verzij aplikacij veliko krajši od časa, ki so ga za enako delo porabili pred tem. Opisan pristop je daleč od popolnega in bi lahko v nekaterih primerih bil celo neuporaben, vendar se je v tem specifičnem primeru izkazal za zelo koristnega in je zato v zadnjem letu postal eno najpomembnejših orodij Podjetja.

Naloga opisuje tudi kako se morajo zaradi narave programske opreme, ki se neprestano spreminja in razvija, tudi navodila in postopki, ki so opisani v omenjenem sistemu, redno osveževati in posodobljati. Kljub temu, da je Podjetje definiralo kar nekaj smernic in navodil za izdelavo in izvedbo testnih primerov, je eno najbolj pomembnih zapisanih načel to, da je testiranje programske opreme zahtevno opravilo, katerega se moramo vedno lotiti z odprtim umom. Poleg tega je zapisano tudi to, da nas ne sme biti strah, da bi karkoli uničili, saj je vsaka napaka, ki jo najdemo med testiranjem napaka, ki ne bo vplivala na uporabo aplikacije končnega uporabnika.

V zaključni nalogi je v prvem poglavju predstavljena struktura naloge in na kratko so opisani že obstoječi standardi. V drugem poglavju nadaljujemo s teoretično osnovo in izrazi, ki so potrebni za razumevanje le-te. V tretjem poglavju natančno predelamo temo testiranja programske opreme: opišemo pristope k testiranju, izbiro pravilnih tesnih primerov, glavne metode testiranja med samim razvojem, razložimo zakaj in kako so opisani modeli primerni v našem primeru, in poglavje zaključimo s kratkim mnenjem o tem, zakaj je bilo potrebno razviti rešitev po meri.

Ker je v v preteklosti Izvajalec večkrat dostavil Podjetju neprimerne izdelke, ki so

morali biti veliko krat, zaradi malenkosti, vrnjeni razvijalcem (kar je dodatno podaljšalo čas potreben za objavo posodobitve), se je Podjetje odločilo, da bo razvilo sistem, ki vsebuje natančna navodila, s katerimi si lahko tako razvijalci kot preizkuševalci aplikacij pomagajo pri svojem delu. V preteklosti so razvijalci, preden so izdelke dostavili Podjetju, le-te samo površno testirali. Podjetje je zato velikokrat zavrnilo izdelek in ga vrnilo Izvajalcu, saj so po lastnem testiranju našli precej napak. V predstavljenem prenovljenem sistemu testiranja Podjetje sedaj hrani in posodablja testne primere in postopke, ki jih morajo pri Izvajalcu izvesti preden dostavijo izdelek. Sistem, kateremu je Podjetje posvetilo veliko dela in truda, je zunanjemu izvajalcu omogočil, da Podjetju dostavlja veliko stabilnejše aplikacije z veliko manj napakami. Detaljno testiranje z omenjenim sistemom, ki se začne že pri Izvajalcu, je pripomoglo h krajšemu času do objave nove različice aplikacije. Ravno tako se je zmanjšal obseg dela, ki ga ima testna ekipa pri Podjetju, saj je večino napak odpravljenih že s strani Izvajalca. Z opisanim sistemom se je čas, ki je porabljen za razvoj posodobitev zmanjšal za več kot 25%.

Rešitev, ki je natančneje opisana v četrtem poglavju diplomske naloge, omogoča Podjetju redno objavo stabilnih in zanesljivih posodobitev ponujenih mobilnih aplikacij. Poleg tega, testiranje le-teh pred objavo novih verzij zagotavlja tudi to, da bo nova različica vsaj tako dobra, kot je trenutno nameščena na napravi uporabnika. Negativna plat opisanega pristopa (kot tudi celotne situacije, ki je nastala zaradi izbranega načina izvajanja del) je zagotovo ta, da je čas, ki ga porabijo za razvoj in dostavo novih različic precej daljši od časa, ki bi ga porabili v primeru, če bi razvojna ekipa bila del Podjetja (in ne zunanji izvajalec). Eden glavnih razlogov za počasnost je testiranje izdelkov, ki se mora v najboljšem primeru zgoditi vsaj dvakrat, če Izvajalec seveda dostavi popoln izdelek.

Omenjeni sistem se, kot že omenjeno, redno posodablja z novimi testnimi primeri in navodili, ki nastajajo med samim načrtovanjem in razvojem novih funkcionalnosti. Zapisana navodila niso pravila, katerih se mora testna ekipa strogo držati, temveč le smernice, ki izvajalcem omogočajo lažje in boljše izvajanje del. Opisani testni scenariji in primeri so se prav tako izkazali za odličen vir dokumentacije, ki je na tak način v vsakem trenutku na voljo celotni ekipi. Zapisane smernice za prijavo napak so odličen pripomoček za ekipo, ki se ukvarja s končnimi uporabniki, saj so končni uporabniki tisti, ki se srečujejo z napakami med samo uporabo aplikacij in dober opis le-teh pripomore k hitrejšemu reševanju s strani zunanjega izvajalca.

7 Bibliography

[1] Scott Ambler. User Stories: An Agile Introduction. In *Agile Modeling*. URL: http://www.agilemodeling.com/artifacts/userStory.htm. *(Cited on page 27.)*

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*, volume 54. 2008. URL: http://www.amazon.com/Introduction-Software-Testing-Paul-Ammann/dp/0521880386. *(Cited on pages 18 and 19.)*

[3] Antonia Bertolino. Software Testing Research and Practice. URL: http://www.cis.upenn.edu/$\sim$lee/05cis700/papers/Ber03.pdf. *(Cited on page 12.)*

[4] Börger Egon and Stärk Robert. Abstract State Machines. *Springer-Verlag Berlin Heidelberg*, page 437, 2003. doi:10.1007/978-3-642-18216-7. *(Cited on page 13.)*

[5] William D. Goran. Testing Guidelines for GRASS Ports and Drivers. 1989. URL: http://www.dtic.mil/dtic/tr/fulltext/u2/a221176.pdf. *(Cited on page 19.)*

[6] IEEE, Pierre Bourque, and Richard E. Fairley. *SWEBOK v.3*. 2014. doi:10.1234/12345678. *(Cited on pages 7, 8, 14, and 17.)*

[7] IEEE Computer Society. IEEE Standard for Software Quality Assurance Processes, 2014. doi:10.1109/IEEESTD.2014.6835311. *(Cited on page 5.)*

[8] ISixSigma. What Is Six Sigma? https://www.isixsigma.com/new-to-six-sigma/getting-started/what-six-sigma/. *(Cited on page 1.)*

[9] ISO. ISO 9000. https://www.iso.org/obp/ui/#iso:std:iso:9000:ed-4:v1:en. *(Cited on page 1.)*

[10] ISO. ISO/IEC 15504 - Information technology – Process assessment. https://www.iso.org/standard/37454.html?browse=tc. *(Cited on page 5.)*

[11] ISO. Quality management systems - Fundamentals and vocabulary. Technical report, 2015. *(Cited on page 4.)*

[12] Cem Kaner, Jack Falk, and Hung Quoc Nguyen. *Testing computer software*. Dreamtech Press, 1999. *(Cited on page 8.)*

[13] Bozena Poksinska, Jens Jörn Dahlgaard, and Marc Antoni. The state of ISO 9000 certification: a study of Swedish organizations. *The TQM Magazine*, 14(5):297–306, oct 2002. URL: http://www.emeraldinsight.com/doi/10.1108/09544780210439734, doi:10.1108/09544780210439734. *(Cited on page 1.)*

[14] Pramod. Software Testing Strategies. http://softwaretestinghome.blogspot.si/2008/08/testing-strategies-approaches.html. *(Cited on page 11.)*

[15] Ian Sommerville. *Software Engineering*. 2010. doi:10.1111/j.1365-2362.2005.01463.x. *(Cited on page 4.)*

[16] The Interaction Design Foundation. User Personas. https://www.interaction-design.org/literature/article/user-personas-for-mobile-design-and-development-a-winning-technique-for-great-ux. *(Cited on page 8.)*

[17] Mark Utting and Bruno Legeard. Practical model-based testing: a tools approach. *Book*, page 433, 2007. doi:10.1145/357474.355050. *(Cited on page 4.)*

[18] WideSkills. 03 - Test artefacts. In *Software Testing Tutorial*, chapter 3. WideSkills.com. URL: http://www.wideskills.com/software-testing-tutorial/test-artifacts. *(Cited on page 6.)*

[19] Wikipedia. Mars Climate Orbiter. https://en.wikipedia.org/wiki/Mars_Climate_Orbiter. *(Cited on page 10.)*

[20] Wikipedia. Perpetual Beta. https://en.wikipedia.org/wiki/Perpetual_beta. *(Cited on page 19.)*

[21] Wikipedia. Test Levels. https://en.wikipedia.org/wiki/Software_testing#Testing_levels. *(Cited on page 14.)*

[22] Wikipedia. Test Plan. https://en.wikipedia.org/wiki/Test_plan. *(Cited on pages VIII, 6, and 7.)*

[23] Wikipedia. Usability testing. https://en.wikipedia.org/wiki/Usability_testing#cite_note-apple1982-15. *(Cited on page 20.)*