Automated and Manual Testing of an Athlete Progress Tracking Application

Radić, Dajana

Sveučilište J.J. Strossmayera u Osijeku
Filozofski fakultet Osijek
Dvopredmetni diplomski studij informatologije i informacijskih tehnologija

Dajana Radić

Automated and Manual Testing of an Athlete Progress Tracking Application

Diplomski rad

Mentor: doc. dr. sc. Tomislav Jakopec

Osijek, 2024.

Sveučilište J.J. Strossmayera u Osijeku
Filozofski fakultet Osijek
Odsjek za informacijske znanosti
Dvopredmetni diplomski studij informatologije i informacijskih tehnologija

Dajana Radić

Automatizirano i ručno testiranje aplikacije za praćenje napretka sportaša

Diplomski rad

Društvene znanosti, Informacijske i komunikacijske znanosti, Informacijski sustavi i informatologija

Mentor: doc. dr. sc. Tomislav Jakopec

Osijek, 2024

Prilog: Izjava o akademskoj čestitosti i o suglasnosti za javno objavljivanje

Obveza je studenta da donju Izjavu vlastoručno potpiše i umetne kao treću stranicu završnog odnosno diplomskog rada.

IZJAVA

Izjavljujem s punom materijalnom i moralnom odgovornošću da sam ovaj rad samostalno napravio te da u njemu nema kopiranih ili prepisanih dijelova teksta tuđih radova, a da nisu označeni kao citati s napisanim izvorom odakle su preneseni.
Svojim vlastoručnim potpisom potvrđujem da sam suglasan da Filozofski fakultet Osijek trajno pohrani i javno objavi ovaj moj rad u internetskoj bazi završnih i diplomskih radova knjižnice Filozofskog fakulteta Osijek, knjižnice Sveučilišta Josipa Jurja Strossmayera u Osijeku i Nacionalne i sveučilišne knjižnice u Zagrebu.

U Osijeku, datum 10.04.2024.

Dajana Dadić, 0122228889
ime i prezime studenta, JMBAG

Abstract

This master thesis aims to show the comparison of automated and manual testing processes delivered within a practical and theoretical scope. The theoretical overview consists of highlighting some advantages and disadvantages of manual and automated testing and defining the capacities to which both types of testing extend in terms of time efficiency and overall effectiveness. The practical part consists of carrying out automated and manual tests on a mobile application tracking athletes' progress. Manual and automated tests are carried out on a physical Android device and the execution of tests in both respects is described. The project environment consists of IntelliJ Integrated Development Environment (IDE) and Selenium, Appium, and TestNG as the main tools. The implementation of automated scripts is carried out using the Java programming language. Final reports of the test execution are created to attest that the software is consistent within itself and high in overall quality. The defined test cases cover the functionality and design of the application. Finally, a full and detailed bug report is created which consists of opportunities where improvements could be made.

Keywords: automated testing, manual testing, quality assurance, testing tools, Selenium testing tool, Appium

Table of Contents

1. Introduction

Throughout the history of software development, ever since the appearance of the first computers, the demand for sophisticated and complex software has been on the rise. The consumer of today wants the software to be reliable and to have all the components they desire along with being fast in the execution of tasks, which is sometimes hard to accomplish. It is no longer sufficient for an application to just be fast, good quality and easy to navigate. These qualities have become a must-have for any application that seeks to have medium or large usage and achieve success. With that being said, the average successful software today is very complex. Customers have a higher demand and expectations compared to the early stages of technology. To ensure that software meets the expectations of clients, testing is of vital importance. Customers have a low tolerance for apologies when experiencing slow software, often ceasing the use and seeking better options. In the wide selection of software applications at the average user's disposal, software that performs badly or is badly designed makes the customer angry. They are quick to discontinue the usage of software that does not meet their standards. This is the difference in expectations between users of the past and today's users who are ruthless toward badly executed software. That is a vital change in the user perspective and as such, it is taken into account by testers, who are there to ensure that the software once presented to the customer will perform by expectations, standards, and required quality levels. In such cases, Quality Assurance proved to be an essential part of the process of software creation. It is vital to the process of creating software, and it finds its place from the beginning to the end. Testing can be considered the last polish and shine to the software, but contrary to popular belief, polishing and shining is not the main role of software testers. Testers are included in the whole process of creation of software at the beginning of a project, but before it as well. The testers have a say in the requirements, review them, and work alongside the development team to prepare the project itself and deliver it to the production.

2. Theoretical foundations of software testing

There are three main types of testing. Some scientific literature talks about the Testing Pyramid as it is illustrated in Image 1.

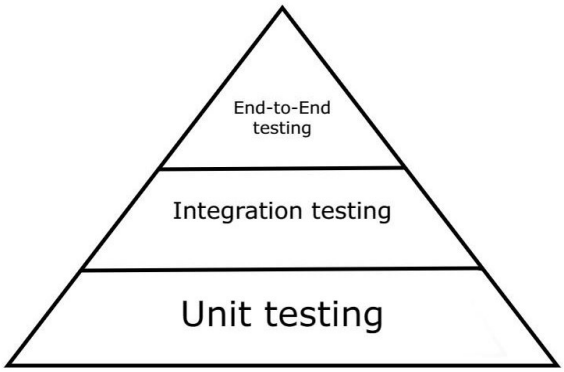


Image 1. The Testing Pyramid

The Testing Pyramid is a scheme that portrays the three different levels of testing, and they are as follows:

- Unit testing
- Integration testing
- End-to-End (E2E) testing

Unit tests are carried out by the developer. They are a sort of introduction to the testing process in the software development life cycle and serve as a type of foundation for building software quality. Integration tests, which are next on the pyramid, are generally more complex since they are the activities involving testing outer layers of the software, in other words, testing external elements and how they integrate with the software.[1] Integration testing tests if all portions of the software have been integrated into one feature the way they were expected to be.[2] The highest level in the pyramid is E2E testing, which is the most sophisticated and thorough testing process, ensuring the integral entirety of software. This is being done to catch bugs before they reach the late stages of development, let alone the very last stage in which a consumer will identify those errors.[3] This means testing software *from top to bottom*, covering all the aspects of software as much as possible.[4] The purpose of E2E testing is to test the software to find inconsistencies, bugs, breakage, faulty code, and errors, to validate data

[1] Khan, Salman. What is End-to-End Testing? E2E Testing Tutorial with Examples and Best Practices. URL: https://www.lambdatest.com/learning-hub/end-to-end-testing (2023-04-24)
[2] Johal, Dilpreet. Software Testing Fundamentals: Guide to Concepts and Processes. URL: https://testsigma.com/blog/fundamentals-of-software-testing/ (2024-07-04)
[3] Khan, Salman. What is End-to-End Testing? E2E Testing Tutorial with Examples and Best Practices. URL: https://www.lambdatest.com/learning-hub/end-to-end-testing (2023-04-24)
[4] Ibid.

processing and transfer between databases and the software, and to ensure that the User Experience (UX) and User Interface (UI) are up to the standard and as described in the documentation.[5] This kind of activity would be very delicate and time-consuming. Therefore, we have two types of E2E testing, one being manual testing where a tester goes through the software manually to ensure full quality and coverage, and the other type is automated testing, where a tester writes up a script to go through the entirety of the software through a myriad of tools.[6] As an addition, we also have two methods of testing regarding code access.[7] One of them is black-box testing and the other is white-box testing.[8] White-box testing is considered to be the type of testing where the code to develop the software itself is available to the tester for an overview.[9] As a result, testers are able to closely examine the project structure, data usage, how the data is handled and work with the testing process from there, allowing for more precise testing outcomes. White-box testing is sometimes called transparent testing.[10] It requires a tester to understand data structures and the programming language to perform white-box testing.[11] Black-box testing tests the application without any access to the code or structure within the application.[12] This type of testing greatly considers the finished product and that which the customer sees. The black-box testing approach relies heavily on the tester's ability to look at the software from a user's standpoint.[13] While black-box testing and white-box testing are on different ends of the testing spectrum, grey-box testing strives to blend both approaches and minimize the appearance of their negative sides, while retaining the advantages.[14] Grey-box testing implies that the tester has limited knowledge of the internal processes of the software, as well as the data which is contained within the software and some of the code on which the software runs.[15] It is worth mentioning that grey-box testing tests the

[5] Ibid.
[6] Ibid.
[7]Differences between Black Box Testing vs White Box Testing. URL: https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/(2024-04-03)
[8]Ibid.
[9]Ibid.
[10] Fulber-Garcia, Vinicius. Software Engineering: White-Box vs. Black-Box Testing. URL: https://www.baeldung.com/cs/testing-white-box-vs-black-box (2024-04-08)
[11] Ibid.
[12]Differences between Black Box Testing vs White Box Testing. URL: https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/(2024-04-03)
[13] Fulber-Garcia, Vinicius. Software Engineering: White-Box vs. Black-Box Testing. URL: https://www.baeldung.com/cs/testing-white-box-vs-black-box (2024-04-08)
[14] Murray, Adam. Gray Box Testing Guide. URL: https://www.mend.io/blog/gray-box-testing/ (2024-04-08)
[15] Ibid.

UI as well as the functionality of the software. This provides excellent testing coverage. When the person testing the software has a perspective of a user and knowledge of a developer, security, time execution, coverage, and overall quality are raised to a new level.[16]

3. Manual software testing

Manual testing is carried out manually, by the hand of the tester. Regardless of the job title, position, or even field of work, one will always find themselves testing regularly.[17] This behaviour is natural to humans, and it is vital to the development process. Anything that has not been tested is of questionable quality. Information technology was able to advance through persistent testing. This approach was beneficial even before the advent of modern software development, even with the very first computer. The team developing the computer had an interesting problem that included mice chewing on the cables.[18] When they realized that the mice were compromising their hardware, they did not speculate on the dietary habits of rodents but rather took the mice and put them in a cage, then starved them and offered them different types of wires, after which they selected only those wires which the mouse did not eat. This is a practical example of hardware testing and its ability to preserve the project. In terms of software, a performance test can tell us how well a system performs under a very high load once applied to the database for example, or the software itself.[19] Usability tests can tell us whether users are able to navigate the software successfully and without assistance.[20] Surprises can be found when deviating from test cases because the human mind cannot anticipate all the problems, especially when a bug is crucial and difficult to find, or a bug that is in plain sight but easily overlooked. This is why a small degree of exploratory testing should be included in every testing action. Exploratory testing is another important testing activity a manual tester should perform, other than following test cases. It covers the areas of software that the test cases have not covered. Exploratory testing allows the manual tester to be creative in his testing routine. To spot as many bugs as possible, a tester needs curiosity, an eye for details, and a good attention span. An exercise that can test these qualities is called the Moonwalking Bear exercise.[21] It is a test that targets the tester's ability to overlook details while looking at the big

[16] Ibid.
[17] Elisabeth Hendrickson. Explore It: Reduce Risk and Increase Confidence with Exploratory Testing. Dallas; Raleigh: The Pragmatic Bookshelf, 2012 page 3
[18] Ibid.
[19] Ibid.
[20] Ibid.
[21] Ibid., page 25

picture. The test begins with the observer being asked to track how many passes the team in white makes.[22] Naturally, the tester is drawn to focus on counting how many times the ball has been passed around, and it is easy to overlook a man in a bear suit dancing between the people passing the ball. This happens because we, as humans, cannot process all the impulses around us because that would overwhelm us. This story is a good reminder that sometimes it is good to think outside of the box when testing because some bugs can be hiding in plain sight. An example of *out-of-the-box* thinking is entering the wrong date format to the data input and seeing if it produces an error. Another good example is to try to upload a large file size to see if it is possible to make the application crash.[23] The frequency of inputs is also a part of durability testing. An action done by the user once will yield no bugs, but when it is done repeatedly it might result in an unexpected error.[24] Using a large number of whitespaces, odd characters, and symbols that are not the targeted data is a way to test if the UI is up to standards. Experienced testers know how to break software simply by examining the software's ecosystem, like network connections, databases and libraries.[25] Additional bugs can be found by running the application for a long time, forcing it to break. The author mentions how in one instance he was testing his system for very long periods of time and got very tired when he heard a noise coming from the computer, suggesting that the software was overloading the hardware.[26] This poses a serious problem as it can mean that the application will ultimately destroy the hardware. Had he not been tired, working for so long on the same laptop with the project, he would not have noticed it.[27] These are just some of the testaments of why it is important to introduce testing into the early stages of software development. Apart from testing bugs and errors, a tester validates the project documentation throughout its creation and maintenance. Testers create test cases to match the flow of the documentation and their execution is crucial to the process. A tester will manually document observations they have found, the success and failure rate of each test case, bugs that were found while testing, and additionally communicate all of this to colleagues, with added respect to those colleagues whom this particular test execution concerns, like developers, team leaders, and other testers.[28]

[22] Ibid., page. 26
[23] Ibid.
[24] Ibid., page. 43
[25] Ibid., page. 87
[26] Ibid., page. 29
[27] Ibid., page. 30
[28] Ibid.

4. Automated software testing

As the software project grows, manual testing becomes unfeasible and difficult to plan and execute. The process involves testing all levels or components of a program, including the UI and the database. The main goal of automated testing is to achieve a testing process that is fast, simple, and reliable.[29] This is why it is important to select tools that are of high quality as well as easy to use and understand, and versatile enough to be able to test different operating systems, features, and devices.[30] A tester should always get very familiar with the tools that are used in the project. The tools should support the programming or script language and make it simple enough for a person to master with little to none prior programming skills.[31] We mostly talk about automated testing that requires knowledge of programming, which is partially true.[32] It helps if a person already has extensive programming knowledge or experience, but depending on the project, this can be easily overcome if the person has the right attitude and mindset. Some approaches that should be mentioned are Test Driven Development (TDD) and Data Driven Development (DDD). Test Driven Development is a way of writing automated test cases before the code itself, after which the code validates those test cases.[33] In this development, tests are given an advantage before the code to be the drive for code execution.[34] In this project, an approach of Data Driven Testing was used. It is an a appropriate approach because it cuts down the time needed to format the data which will be used in testing.[35] With this approach, a large number of data sets are stored in a secluded file or a class and then called in a test execution. It saves money, time, and effort by organizing and speeding up the process of data handling.[36] Closely related to this is the Don't Repeat Yourself (DRY) principle.[37] The main point of the DRY principle is to remove all the things we don't need in the system, thus simplifying it.[38] The point of doing this is because we aim to get code that is easy to decipher,

[29]Johal, Dilpreet. Software Testing Fundamentals: Guide to Concepts and Processes. URL: https://testsigma.com/blog/fundamentals-of-software-testing/ (2024-07-04)
[30]Ibid.
[31]Ibid.
[32] Lambdatest. What is Automation Testing? URL: https://www.lambdatest.com/automation-testing (2024-04-07)
[33] Elisabeth Hendrickson. Explore It: Reduce Risk and Increase Confidence with Exploratory Testing. Dallas; Raleigh: The Pragmatic Bookshelf, 2012 page 135
[34]Test Driven Development (TDD). URL: https://www.geeksforgeeks.org/test-driven-development-tdd/(2024-04-03)
[35]Banu, Shakura. What is Data Driven Testing: All You Need to Know. URL: https://www.lambdatest.com/learning-hub/data-driven-testing (2024-04-03)
[36]Ibid.
[37] Anton Angelov. Design Patterns for High - Quality Automated Tests: Clean Code for Bulletproof Tests. Automate the Planet. page 15
[38] Ibid.

operate, handle and read.[39] The tools should be durable and handle large data transmission without breaking, slowing down the test execution, or making it difficult for the tester to do its job.[40] Framework should also have libraries, test data, modules, and third-party tools integrated. The idea is that once created, an automated test can last for a long time and be reused. The preparation and planning process is vital. If it is done correctly, the entire project will be well executed. The first step is to define the objective of testing, which is essentially planning the process of testing, and also the end goal.[41] The initial part of planning may be time-consuming and it could feel like it is not very productive, but this act of preparation allows the tester not to waste time later in the project. To start, a tester should prioritize tests and make sure that the most important tests are carried out immediately and the least important ones are saved for last. This is being done to ensure that test steps don't accidentally get left out.[42] The next step is to write the code. We build the automated test cases with the help of documentation and attempt to completely cover it with tests. At this point we consider a wide range of scenarios, for example, testing on different devices like iPhone and Android, or Samsung smartphones and Samsung tablets.[43] The next important step is ensuring the ease of testing, which happens almost consequently to the prior steps as the planning makes sure that the tests will be carried out smoothly. The last step, and a very important one, is communication within the team. Nothing can be done in a project if there is no successful communication or if a team has teammates who are unwilling to talk and communicate through problems.[44] These kinds of people pose a problem for the project because the development plans must be communicated openly. With all of that being said, testing itself will not eliminate all the problems the software has. It is impossible to have all the problems accounted for in a software, and sometimes it is possible that these errors get noticed but they cannot be fixed by the development team. There are also some misconceptions about automated testing versus manual testing. Common one is that automated testing replaces some aspects of manual testing. This is not true because some tests can be carried out with automated tests, while some can be carried out only in a manual manner. Automation holds some advantages over manual testing, but each has its respective benefits. Automation is a good tool for a project but it cannot replace the tester's ability to

[39] Ibid., page 16
[40] Johal, Dilpreet. Software Testing Fundamentals: Guide to Concepts and Processes. URL: https://testsigma.com/blog/fundamentals-of-software-testing/ (2024-07-04)
[41] Lambdatest. What is Automation Testing? URL: https://www.lambdatest.com/automation-testing (2024-04-07)
[42] Ibid.
[43] Ibid.
[44] Ibid.

notice errors or have critical and innovative thinking which is a very important part of a project or software.

5. The purpose and goals of testing

The goal of testing in this Master's Thesis is to portray the testing activities within the software development life cycle which aims to find bugs or faults in a system and then point them out so that they can be removed, as well as to point out weak points of the system with the intent of prevention.[45] The purpose of testing is to increase the quality of the software, using different methods when aiming for the earlier established goal, achieving software excellence.[46] To emphasize the distinction, it is possible to say that when we talk about the purpose, we take it as a philosophical concept, where a metaphysical value is established in the form of the idea of what testing means to us and what it means for the software development process.[47] Meanwhile, the engineering portion is where we are getting concerned with the actual feasible goals and actions which will be taken to embody the said purpose. This means performing the testing process, choosing tools, developing frameworks, writing test cases, and studying the use cases or documentation.[48] The goal of testing here is to bring the software-making process as close as possible to the written documentation. In this regard, a tester has a goal to take the requirements written for this specific software and then do the necessary action that will ensure that the product itself matches the formed and approved documentation as much as that is humanly possible.[49] We can divide the testing goals into a few sections of actions. We have short-term goals, long-term goals, and goals after the implementation. The short-term goals are dealing with bugs in a manner of controlling them and preventing them so that if a bug does occur the solution can be easily found. Ideally, the solution is fast-paced, easily done, and can be used as a lesson for future endeavors.[50] Here lies the importance of documentation as well. Everything that is documented can be used in the future as a standpoint for what went wrong at what point and how we managed to solve it.[51] This is a great learning curve for the team. Long-term goals can be somewhat subjective as well. In the long term, we talk about which

[45] Goals of Software Testing. URL: https://www.geeksforgeeks.org/goals-of-software-testing/ (2024-04-03)
[46] Testing for Transportation Management Systems. URL: https://tmcpfs.ops.fhwa.dot.gov/cfprojects/uploaded_files/Final%20Q&A.pdf
[47] Goals of Software Testing. URL: https://www.geeksforgeeks.org/goals-of-software-testing/ (2024-04-03)
[48] Ibid.
[49] Ibid.
[50] Ibid.
[51] Ibid.

steps can be taken to ensure that this quality is seen through. We talk about managing risk and the importance of how a customer sees the product because, in the end, it is the customer who will determine how well the software fits the purposes for which it has been intended.[52] Another key point is the company's trustworthiness, that the software we produce is up to the level of quality we were expected to deliver.[53] Important to note is that the portion of the processes that come after testing the product gives us insights about what a company can learn from mistakes that were made, and how we can apply this knowledge in future projects or how this can be done concerning ongoing software maintenance.[54]

6. Selecting testing tools

Testing tools are selected based on the needs of the project and the goals of testing. Tools used in the context of the Master's Thesis are Appium with Selenium and TestNG. The test script is written with Java programming language. Appium is an automated testing tool used with a variety of programming languages to fulfill the testing needs of a project, some of which are Java, Python, and the testing framework Cucumber.[55] It allows for different operating systems and different platforms to run tests on. This brings about its versatility. Appium tool can also be used with the Appium Inspector tool, which is a tool used to locate identifiers of elements on a screen.[56] The Appium framework was brought by Dan Cuellar and Jason Huggins and was intended for web, native and hybrid application testing.[57] The versatility of the tool cuts down on the possible issues a tester can run into while testing an app with a wide number of users who naturally all have different operational systems, platforms, and phones. When choosing tools, it is good to know what kind of devices the users have. Sometimes applications are intended for just one type of device, which makes the choice of the tool rather easy, but most times the pool of users will have different operation systems, system versions, etc. However, using Appium eliminates the need to worry about problems featuring different testing platforms as it supports a wide range of them. IntelliJ IDE contains several tools to help with the testing process and incorporates Appium with Java and Selenium very well.[58] When it comes to Java

[52] Ibid.
[53] Ibid.
[54] Ibid.
[55] Kumar Deo, Tanay. Appium Tutorial: A Detailed Guide To Appium Testing. URL: https://www.lambdatest.com/appium(2024-04-03)
[56] Appium Inspector. URL: https://appium.github.io/appium-inspector/2023.12/overview/ (2024-04-10)
[57] Kumar Deo, Tanay. Appium Tutorial: A Detailed Guide To Appium Testing. URL: https://www.lambdatest.com/appium(2024-04-03)
[58] Ibid.

programming language, it is a good choice for automated testing because it is platform-independent. As such it is described as an object-oriented language.[59] What this means, essentially, is that Java programming language works as a set of grouped objects where each portion of the group does its separate job and as one creates a program. This allows for a very organized background of a program.[60] It's important to have some knowledge about the following:

- JRE (Java Runtime Environment) - an environment where the program is run.[61]
- JDK (Java Development Kit) - tools for development.[62]
- JVM (Java Virtual Machine) - executes the bytecode, this is why we can say that Java is platform-independent.[63]

To write any program in the Java programming language, JDKs are necessary. They are incorporated into the project by being downloaded from the official webpage and without the proper JDKs the project itself cannot be run.[64] When writing the test scripts in Java, the Selenium testing tool is a good choice because of their compatibility. It is an open-source tool and somewhat the industry standard. It's versatile and supports programming languages like Java, Python, and C#.[65] Selenium imitates user behaviour on a screen and one of the most notable advantages of Selenium is the cross-browser interoperability.[66] Some negative sides of Selenium are complicated installation, the demand for knowledge of a programming language, and no built-in mechanisms to handle errors and Java exceptions.[67] TestNG is used within the project as a testing tool.[68] TestNG allows the use of annotations like *@Test* annotation, the annotation used to mark a test case, *@BeforeMethod* and *@AfterMethod* which run before and after each method, *@BeforeSuit*e and *@AfterSuite* which run before and after each suite, and *@BeforeClass* and *@AfterClass* which describe behaviour before and after each class. These are used as an environment set-up to describe desired behaviour surrounding the test

[59] Introduction to Java. URL: https://www.geeksforgeeks.org/introduction-to-java/ (2024-04-03)
[60]What Are Java Classes and Objects and How Do You Implement Them? URL: https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects (2024-04-03)
[61] Introduction to Java. URL: https://www.geeksforgeeks.org/introduction-to-java/ (2024-04-03)
[62] Ibid.
[63] Ibid.
[64] Ibid.
[65] Rungta, Krishna. What is Selenium? Introduction to Selenium Automation Testing. URL: https://www.guru99.com/introduction-to-selenium.html (2024-04-03)
[66] Khan, Salman. What is End-to-End Testing? E2E Testing Tutorial with Examples and Best Practices. URL: https://www.lambdatest.com/learning-hub/end-to-end-testing (2023-04-24)
[67] Rungta, Krishna. What is Selenium? Introduction to Selenium Automation Testing. URL: https://www.guru99.com/introduction-to-selenium.html (2024-04-03)
[68] TestNG. URL:https://testng.org/(2024-04-03)

execution.[69] TestNG also allows the definition of test execution structure. DataProvider is within the scope of TestNG and was used as a helper tool to define test data.[70] Prior to using DataProvider, the written tests were messy, with a lot of data being difficult to read and not very presentable as such. Right next to the advantage of easier data handling, DataProvider also allows for readable tests.[71] After defining the parameters of data that are planned to be handled through testing, it is easy to connect the DataProvider *name* in the method to the data file, which acts as the only attribute connecting the data handling class and the actual test class, and incorporate it into the Test as it is shown in Image 2.

```
@Test(dataProvider = "Menu", dataProviderClass = PlayerProvider.class,
        groups = {PlayerProvider.GROUP_SMOKE})
public void smokeTestNavigation(String survey, String player, String team) {
```

Image 2. Test annotation usage in a class

[69] Ibid.
[70] Ibid.
[71] Rajora, Harish. TestNG DataProviders. URL: https://www.toolsqa.com/testng/testng-dataproviders/(2024-04-03)

7. Forming use cases

The flow of the application is divided into three menus as shown in Image 3, and the project is built on that premise as well.

The first menu is the *Survey* menu, the second is the *Players* menu, and the third is the *Team Stats* menu. Manual and automated tests were conducted regarding some basic navigation between screens.

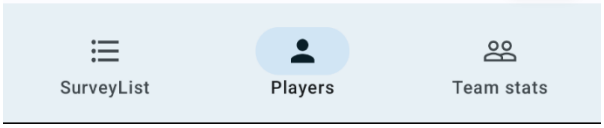


Image 3. Menu

The first screen a user sees when opening the application is a screen asking the user what type of training he had. On this screen, a user can choose the type of training, with the options being *Offensive*, *Defensive,* and *Physical* training as shown in Image 3. Depending on the choice of the user, the blocks of surveys on the Survey screen will appear purple for Offensive, red for Defensive, or green for Physical. The remaining features like the two sliders and an input field are optional. The sliders are formatted with a decimal value. This is not the best value type to offer to the user, because the average user is accustomed to values ranging from 1 to 10 when describing his experience. This design flaw has been reported as a bug.

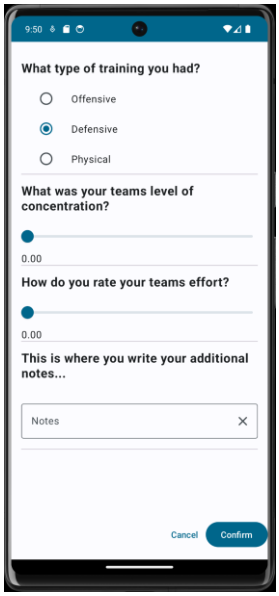


Image 5. Survey menu

After clicking the *Confirm* button, the user will have a visual of the surveys they took on the previous screen, which can be seen in Image 5. However, if the user clicks the *Cancel* button they will be taken back to the Survey menu and no results will be saved. The design flaw that is noticed here is that the only way the user can interact with the blocks of surveys that were saved is by deleting them. The user also has no option of editing the survey block, like rearranging the surveys or similar activities. This has been characterized as a design issue and has also been reported in the bug report.

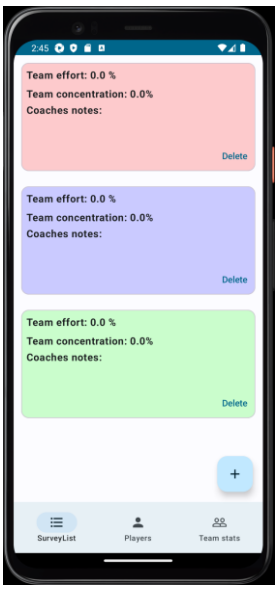


Image 4. Entrance screen

After going through the *Survey* menu it is possible to go to the *Players* menu with a block of *Players* visualizing important data of the player, or navigate to *Team Stats* screen. In the *Players* screen, there is a main button and a section under the button with dots, as portrayed in Image 6, hinting at the possibility of interaction with the element. However, it is not possible to interact with these dots nor are they clickable, leading the user to a false functionality, which is reported as a bug. Furthermore, the testing portion of this screen dealt with the click functionality of the main button and the swipe functionality, as well as if all the elements were present on the screen as described in the documentation. If all the conditions were met then the tests passed. The *Player* button is clickable and leads the user to a screen of the chosen player's details.

Image 6. Players menu

After clicking the button, the user navigates to the *Players* screen as visible in Image 7. On this screen, the user has the option to interact with the button *Improvements* and scroll down to view elements regarding player statistics, but the user cannot interact with any of the elements except the *Improvements* button or change their values. This goes to say that the majority of the screen elements are static.

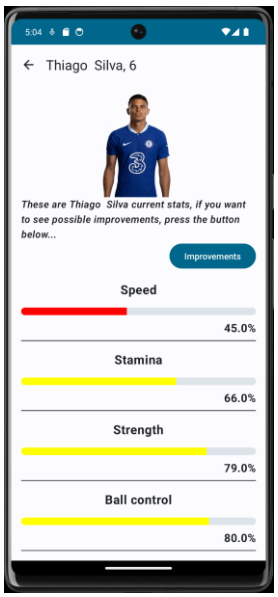
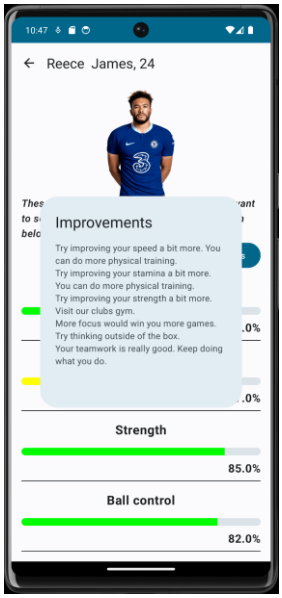


Image 7. Players screen

When the user clicks on the *Improvements* button they are met with a popup screen containing the title and text element. In this particular popup screen, there is also no *Confirm*, *OK,* or *Exit* button to offer the user a button to navigate back. Because of that the user can only exit the pop-up by clicking outside of the element. This is not a bug in itself but it is an example of bad design. Another bug is that there is no *plus* button on the screen, which makes the application seem inconsistent as it's supposed to be either only visible on the Survey screen or every screen. This detail makes the application lack consistency. This issue is visible in Image 8 below:



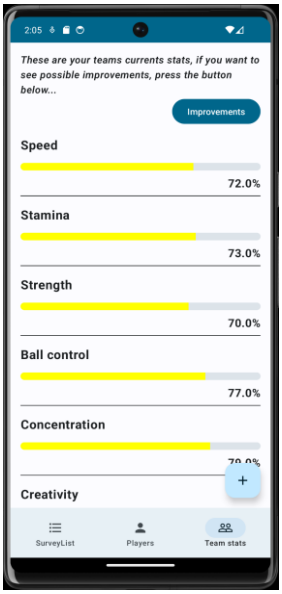


Image 8. Improvements popup **Image 9.** Teams statistics screen

The last menu is the *Team Stats* menu, where a user sees overall team statistics as shown in Image 9. From here the user would think that they can add a player by pressing the *plus* button, but it takes them back to the survey input. This is also a bug as it is not a good practice. The *Improvements* button has the same functionality as described in the *Players* screen. On this screen, just as in the *Players* screen, the user cannot interact with any of the presented elements except the *Improvements* button.

8. Creation of test cases for manual and automated testing

A test suite and a bug report were made to present documentation that is built within the usual testing process. Table 1 depicts test cases that have passed, and Table 2 depicts a bug report for those test cases that have failed. The respective test cases tables are on the link: https://puh.srce.hr/s/KFbRAQMRttyLJo3

9. Project implementation

For the needs of the project an application that tracks the progress of players was used. A project was created in IntelliJ IDE and a simple structure was created. For the project, a real device was used along with the possibility of an emulator provided by IntelliJ, however, based on the idea that the end user will be using a real device instead of an emulator it is the better option to, whenever possible, use the real device. The physical device used was POCO M5s, and the Android version 13. An IntelliJ IDE emulator Pixel 6 Pro was used with Android version 12 as well for some of the needs regarding the execution of automated testing, and along with the emulator the Appium Inspector tool was used to identify elements in the application.

The repository for the source code is on the link: https://github.com/Dajana-jpg/Masters_Thesis

9.1 Prerequisites

The project has the structure as seen in Image 10 below:

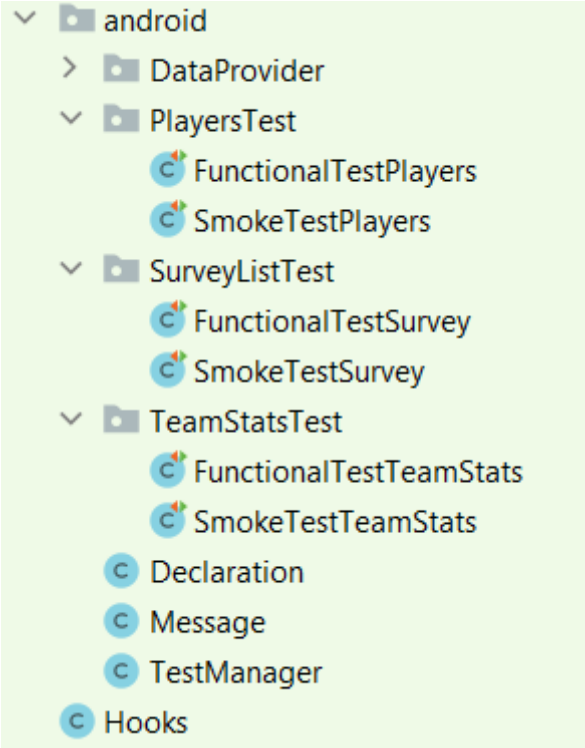


Image 10. IntelliJ project structure

File *testng.xml* was created to help facilitate the order of classes and how they should be running in test execution. This file can be edited and configured based on the needs of the project. The test cases themselves along with all the tools for tests, helper functions, and added classes are stored in the *android* directory. The project was organized into a structure of three sections that correspond to the three screens in the project application, which helped to put the project into segments and make the testing process easier. *PlayersTest* class, *SurveyTest* class, and *TeamStatsTest* classes were made with two corresponding test activities: smoke testing and functional testing. This approach increased the readability of code and made the organization of the test cases easier.

The tests are executed from the *pom.xml* file which is configured with different sorts of dependencies helping to carry the project and bring ease of use of the testing framework. TestNG served to facilitate test execution as portrayed in Image 11, while an Appium dependency, as seen in Image 12, made the methods and driver initialization possible. Selenium dependency for a tool is responsible for interactions with elements which is seen in Image 13.

```xml
<dependency>
    <groupId>org.testng</groupId>
    <artifactId>testng</artifactId>
    <version>7.7.1</version>
</dependency>
```

Image 11. TestNG dependency

```xml
<dependency>
    <groupId>io.appium</groupId>
    <artifactId>java-client</artifactId>
    <version>8.5.1</version>
</dependency>
```

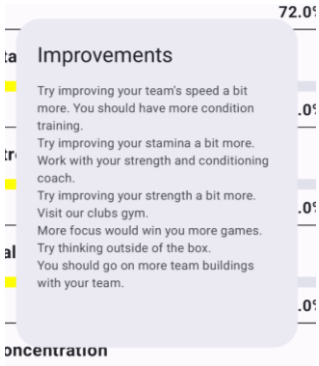
Image 12. Appium dependency

```xml
<dependency>
    <groupId>org.seleniumhq.selenium</groupId>
    <artifactId>selenium-remote-driver</artifactId>
    <version>4.13.0</version>
</dependency>
```

Image 13. Selenium dependency

9.2. Classes

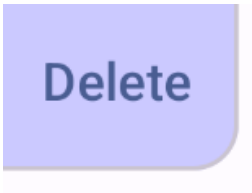
Additional two separate helper classes were made. One helper class was the *Messages* class, which acted as an umbrella class for all messages that appeared in the documentation and the application's actual state as visible in Image 14.




```java
15 usages
public class Message
{
    1 usage
    public static final String
            TITLE_TEXT_DOES_NOT_MATCH
            = "Title text does not match";
```

Image 14. Message helper class for title text

Another helper class was the Declaration class, where all the repetitive and unnecessarily long methods were taken out of the test classes and separated, in hopes of improving readability.




```
2 usages
public static void clickDeleteButton() {
    driver.findElements(By.className("android.widget.Button"))
            .get(0).click();
}
```

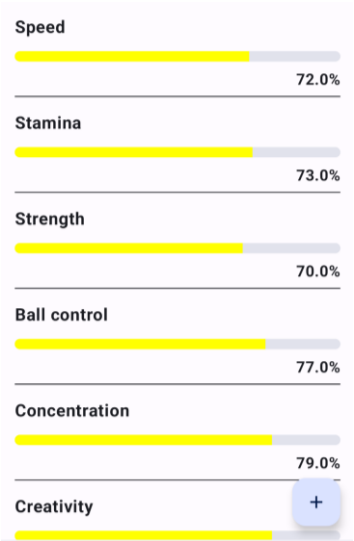
Image 15. Example of method declaring in Declaration class for the Delete button

By declaring methods and calling them back in the tests, as is shown in Image 15, it was possible to easily cut down on the need to go through a great deal of code just to call one method, or even go through many steps. Some redundant steps were easily conveyed into one method in the Declaration class and then used in the test. This method is shown in Image 16.

```
Declaration.clickDeleteButton();
```

Image 16. Example of method usage

Data itself was pulled from the DataProvider package, as is shown in Image 17, and then tested the actual state of data in the application.




```
@DataProvider(name = "Team")
protected static Object[][] team() {
    return new Object[][]{
            {"Team stats", "Speed", "72.0%"},
            {"Team stats", "Stamina", "73.0%"},
            {"Team stats", "Strength", "70.0%"},
            {"Team stats", "Ball control", "77.0%"},
            {"Team stats", "Concentration", "79.0%"},
            {"Team stats", "Creativity", "79.0%"},
            {"Team stats", "Teamwork", "85.0%"},
            {"Team stats", "Shot Power", "74.0%"}
    };
}
```

Image 17. Example of data handling

The three visible DataProvider classes are also corresponding to the screens that were segmented within the tests. Within those DataProvider classes data was divided, based on the

needed sets of data. In such a way the whole project structure is very well-rounded. We have the *PlayerProvider* class in the structure which corresponds to the *PlayerTest* class tests, *SurveyProvider* for *SurveyTest* class, and *TeamStatsProvider* for *TeamStatsTest* class tests.

```
13 usages   7 inheritors
public class TestManager {

    27 usages
    protected AppiumDriver driver;
    17 usages
    protected WebDriverWait wait;
```

Image 18. Defining the drivers and the wait method

TestManager class as shown in Image 18 is a class used to set up the prior needed methods, for example, the methods of *@BeforeMethod* and *@BeforeSuit*, which helped the tests run seamlessly. Setting the environment properly is important so that problems in execution do not resurface while running tests, as this can be a cause of flaky tests.

```
no usages
@BeforeMethod(alwaysRun = true)
public void beforeMethod(Method method) {
    String methodName = method.getName();
    String udid = "9e490bfdd263fc9f";
    System.out.println(String.format("Running test '%s' on '%s'", methodName, udid));
    driver = createAppiumDriver(udid);
    wait = new WebDriverWait(driver, Duration.ofSeconds(50));
    Declaration.setDriver(driver);
}
```

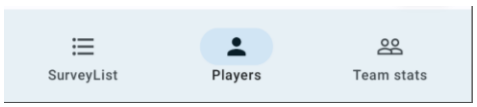
Image 19. Driver configuration in TestManager class

In the scope of the project, a method was declared containing the driver, perhaps the most important configuration, as seen in Image 19. It was decided on an Android device which was used for testing purposes. Consequently, a *udid* number was extracted from the physical device and defined in the *@BeforeMethod*, which helps the driver locate the correct device to run tests on. The last class to mention in the scope of configuration is the *Hooks* class, which is a standard class in the framework, and was edited and tailored to the project's specific needs which is visible in Image 20.

```java
4 usages
public class Hooks {
    10 usages
    private static AppiumDriverLocalService localAppiumServer;
    1 usage
    private static String APPIUM_SERVER_URL = "http://127.0.0.1:4723/";
    1 usage
    public static void startAppiumServer() {
        System.out.println(String.format("Start local Appium server"));
        AppiumServiceBuilder serviceBuilder = new AppiumServiceBuilder();
        // Use any port, in case the default 4723 is already taken (maybe by another Appium server)
        serviceBuilder.usingPort(4723);
        serviceBuilder.withIPAddress("127.0.0.1");
        serviceBuilder.withArgument(GeneralServerFlag.SESSION_OVERRIDE);
        serviceBuilder.withAppiumJS(new File( pathname: "./node_modules/appium/build/lib/main.js"));
        serviceBuilder.withLogFile(new File( pathname: "./target/appium_logs.txt"));
        serviceBuilder.withArgument(GeneralServerFlag.ALLOW_INSECURE, value: "adb_shell");
        localAppiumServer = AppiumDriverLocalService.buildService(serviceBuilder);
        localAppiumServer.start();
        APPIUM_SERVER_URL = localAppiumServer.getUrl().toString();
        System.out.println(String.format("Appium server started on url: '%s'",
                localAppiumServer.getUrl().toString()));
    }
    1 usage
    public static void stopAppiumServer() {
        if (null != localAppiumServer) {
            System.out.println(String.format("Stopping the local Appium server running on: '%s'",
                    localAppiumServer.getUrl().toString()));
            localAppiumServer.stop();
            System.out.println("Is Appium server running? " + localAppiumServer.isRunning());
        }
    }
    1 usage
    public static URL getAppiumServerUrl() {
        System.out.println("Appium server url: " + localAppiumServer.getUrl());
        return localAppiumServer.getUrl();
    }

}
```

Image 20. Hooks class

The Test classes were defined by the Java programming language principles and led by the *@Test* annotation, as seen in Image 21.




```java
@Test(dataProvider = "Menu",
        dataProviderClass = PlayerProvider.class,
        groups = {PlayerProvider.GROUP_SMOKE})
public void smokeTestNavigation
        (String survey, String player, String team) {
    Declaration.getPlayer(player).click();
    Declaration.getTeam(team).click();
    Declaration.getPlayer(player).click();
    Declaration.getSurvey(survey).click();
}
```

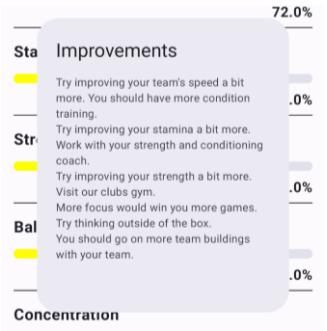
Image 21. Test class example of the menu navigation

This is an example of the basic navigation principles within the application. As can be read from the code, the written code navigates the driver through a plus button to open the screen on which the test will be further executed onto the three menu buttons and performs clicks on each of them. After the test is successful, the command line shows a message that the test was successful. The tester can also monitor the test execution in real-time on a physical device or emulator of choice.

```
@Test(dataProvider = "Team",
        dataProviderClass = TeamStatsProvider.class,
        groups = {TeamStatsProvider.GROUP_SMOKE})
public void teamData(String team, String stats, String percentage) {
    SoftAssert softAssert = new SoftAssert();
    Declaration.getTeam(team).click();

    try {
        softAssert.assertEquals(Declaration.findTeamStats(stats), stats);

    } catch (NoSuchElementException e) {
        softAssert.assertEquals(Declaration.findStatsTeam(stats), stats);
    }

    try {
        softAssert.assertEquals(Declaration.findTeamPercentage(percentage), percentage);

    } catch (NoSuchElementException e) {
        softAssert.assertEquals(Declaration.findPercentageTeam(percentage), percentage);
        softAssert.assertAll();
    }
}
```

Image 22. Example of using the *try-catch* block to solve the issue

The portrayed example of code in Image 22 is executing a test on the data in the application. The reason to use *try-catch* statement is because of the nature of Android Compose. The problem was that even with a *scroll* method and with an *id* of the element provided the driver could not find the text within the mobile application screen, and it could not interact with the element the way it interacted with other elements. The problem we ran into persisted throughout different approaches and fixes, hence the only solution was to insert the action into a *try* statement and catch the problem within the *catch* block. This way ensured that the problem would be solved and would not appear in execution.




```
@Test(dataProvider = "TeamMenu",
        dataProviderClass = TeamStatsProvider.class,
        groups = {TeamStatsProvider.GROUP_REGRESSION})
public void improvementsButtonFunctionality(String team) {
    Declaration.getTeam(team).click();

    By improvementsButton = AppiumBy.id( selector: "team_stats_improvements_button");
    wait.until(ExpectedConditions.visibilityOfElementLocated(improvementsButton));
    driver.findElement(improvementsButton).click();
    driver.navigate().back();
}
```

Image 23. Test checking the improvement popup screen functionality

The code example as seen in Image 23 aims to examine the functionality of the *Improvements* button within the app. We use the *@Test* annotation with the *DataProvider* name attribute which we call from the *PlayerProvider* class, where we call the data needed in the test form, and then call the group *GROUP_REGRESSION* from the *testng.xml* file to connect the test with the test definition in the *testing.xml* file. The *SoftAssert* assertion was called to the test structure as the *SoftAssert* assertion has the advantage of running multiple test steps without breaking the test. The next step in the test is to call the *explicit wait* functionality, which commands a certain amount of seconds for the test to be on hold before the continuation. After this step, the test continues to click on the *Player* menu button, then find and click the main block in the screen that portrays the *Players* screen. From that point, the driver performs a click on the *Improvements* button and examines all the elements, as seen in Image 23. After the driver confirms all the elements are as described we get a confirmation that the test is passed.

9.3. Test execution

Maven test execution can be done with the command *test -f pom.xml.* An example how a report can be generated is by going through the TestNG report, which is generated automatically every time a tester runs the script. It can be found in the file by the name *surefire-reports* in a HyperText Markup Language (HTML) file. Furthermore, opening it in a Browser of choice like Edge or Google Chrome gives the tester an overview of test execution, success, time and other information. In a project it is good to run partial tests, each one separately, because tests can get very large in number. Then a targeted test report will be executed which can be seen in Image 24.

Test	# Passed	# Skipped	# Retried	# Failed	Time (ms)	Included Groups	Excluded Groups
Tests							
SmokeTests	36	0	0	0	480,197	regression, smoke	

Class	Method	Start	Time (ms)
Tests			
SmokeTests — passed			
com.radic.masterthesis.sample.android.PlayersTest.FunctionalTest_Players	improvements	1712414266698	6434
com.radic.masterthesis.sample.android.PlayersTest.SmokeTest_Players	smokeTestBlock	1712413818285	3835
	smokeTestData	1712413830529	4491
	smokeTestData		
	smokeTestData		
	smokeTestData		
	smokeTestData		
	smokeTestData		
	smokeTestData		
	smokeTestData		
	smokeTestNavigation	1712413945084	5375
	smokeTestUIView	1712413958785	4440
com.radic.masterthesis.sample.android.SurveyListTest.FunctionalTest_Survey	Defensive	1712414202650	2684
	checkSurveyDelete	1712414212647	5230
	saveFunctionCheck	1712414225618	4745
	screenFunctionCancel	1712414239086	4700
	xButtonNoteFunctionality	1712414252098	6215
com.radic.masterthesis.sample.android.SurveyListTest.SmokeTest_Survey	emptyScreen	1712413971340	940
	plusButtonToDelete	1712413981167	5420
	smokeTestNavigation	1712413994955	4507
	trainingButtons	1712414008999	4464
	trainingText	1712414053451	1959
	trainingText		
	trainingText		
	trainingText		
com.radic.masterthesis.sample.android.TeamStatsTest.FunctionalTest_TeamStats	improvementsButtonFunctionality	1712414281893	5450
com.radic.masterthesis.sample.android.TeamStatsTest.SmokeTest_TeamStats	improvements	1712414062634	4972
	smokeTestNavigation	1712414075934	5916
	teamData	1712414145230	5936
	teamData		
	teamData		
	teamData		
	teamData		

Image 24. TestNG report

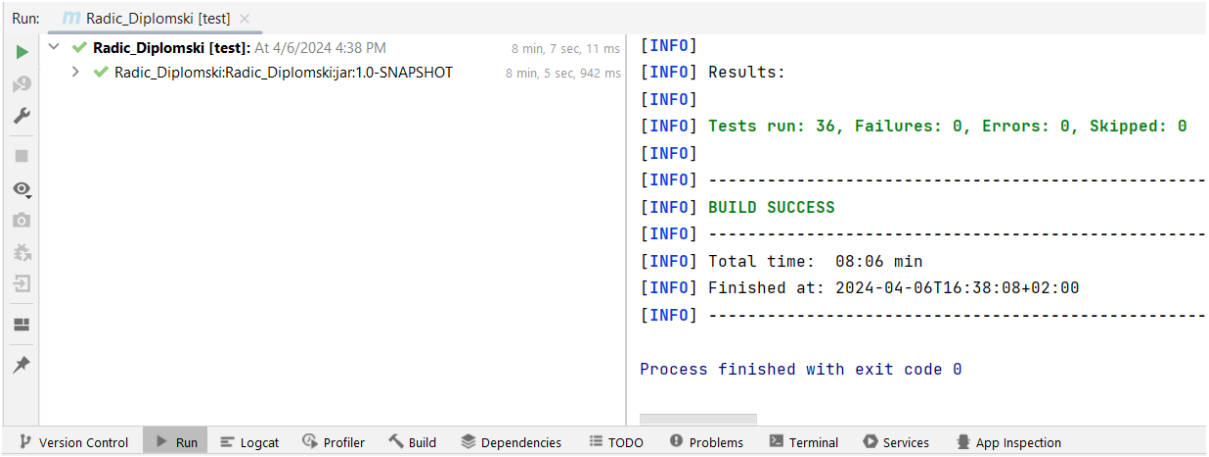


Image 25. TestNG execution success in Terminal

After an execution with *maven clean test,* the testing project has run with 36 tests successfully, as seen in Image 25. After a successful run of all collective tests from the directory *android*, the project is marked as successful with 36 tests executed successfully in total, as visible in Image 26.

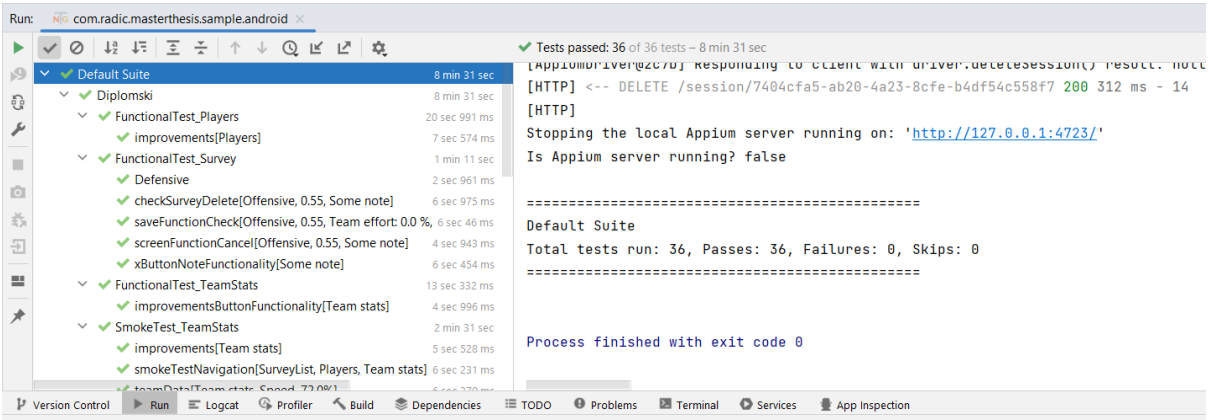


Image 26. Running all tests in the project from the *android* package

By doing this last step, a tester can be certain that all tests run successfully and that the testing execution is done and closed. A report from TestNG can be uploaded to the project's tool of documentation, for example, Jira, where it can be further handled and maintained.

10. Comparison of manual and automated testing

There are two primary methods of testing: manual and automated.[72] Each approach has its strengths and can be beneficial depending on the specific circumstances.[73] The primary distinction lies in the way the testing is performed, and in the area of application.[74] Manual testing allows for creativity and individuality as one of the main virtues and is known for giving a manner of human context to the software. Automated testing, on the other hand, is done with the use of previously written scripts and is executed automatically, but still handled and managed by a tester.[75] The process of testing involves both the preparation of a script and writing the code, which allows for some creativity on the part of the tester. The tester plans the process, manages the execution, reads documentation, compares it to the actual state of the application and documents the results.[76] There is a debate in the field of software testing about the effectiveness of manual testing versus automated testing. Some literature proposes that manual testing is not as much of an accurate testing process as automation testing, because it allows for the probability of human errors while the automated testing process is highly code-based and reliable as it runs by a predefined script. Another difference between the two is that manual testing is a highly time-consuming process. However, from writing test cases to analyzing features and executing tests, all of these efforts require time and attention.[77] If a tester wants to do a good job they must dedicate a large amount of time to exploratory testing and defining the tests as accurately as possible. Automated testing on the other side, does not open up space for randomized testing. What this means is that the automated tests run in a pre-defined structure and are seldom changed once accurately made.[78] While it may overlook certain bugs that a manual approach would catch, it eliminates the likelihood of human error. That is a good thing as well as a bad thing because automated testing does not allow *free-hand* testing to notice more bugs, but it is also not prone to error. Another point speaking in favor of automated testing is that it is fast. It can run as many tests as possible in a short time, and this results in significant time and cost savings for the organization.[79] A tester running automated tests can also overview the duration of test execution and monitor the behaviour of software,

[72] Hamilton, Thomas. Difference Between Manual and Automation Testing. URL: https://www.guru99.com/difference-automated-vs-manual-testing.html (2024-04-03)
[73] Ibid.
[74] Ibid.
[75] Ibid.
[76] Ibid.
[77] Ibid.
[78] Ibid.
[79] Ibid.

thus increasing accuracy. It is even safe to assume that the high level of reliability that automated testing boasts with is exactly because of the alertness of the human behind it.[80] Automated testing deals well with repetitive tasks that need to be done weekly, every two weeks, or monthly, with the aim of maintenance.[81] It is particularly effective for recurrent tasks that require periodic maintenance such as smoke testing, regression testing, load, and performance testing.[82] It is not intended to be applied in each testing task, and it does not perform its best work where manual testing is usually carried out.[83] Automated testing and manual testing have a key difference, which is cost-effectiveness.[84] Time-effectiveness is also important to consider.[85] For instance, manual testing of each screen in this project would take approximately 15 minutes for the *Survey*, 45 minutes for the *Players,* and 30 minutes for the *Team Stats* portion. In contrast, using automated testing, these same tests could be completed in just 8 minutes and 31 seconds. This shows how automation saves time and completes the same amount of work in a significantly shorter amount of time. However, it's important to note that while the automated tests were very quickly executed, it took a long time to build the framework, troubleshoot, write the code, make sure it is executable, edit tests, formulate them, and then bring the code to a satisfactory security level and high level of readability and function. In fact, writing the code for automated testing took more time than all manual testing combined. This is because manual testing is inherent to us humans. We always question our surroundings and check for inconsistencies. High-quality programming is something we must learn and get very good at before we can implement the knowledge into writing tests. Automated testing took time to learn to use the code, write it, and execute it without fail. Once everything was done and written every next test was quickly written and executed, which means regression testing can be done every two weeks or monthly in a much smaller time frame than manual regression testing. Therefore, it is safe to conclude that both types of testing are complementary with eachother, and should be done in a joined fashion. One would spend a large amount of time writing a code to check if a button is clickable, while a human would click on it to ensure it is in fact clickable within seconds. However, when it comes to regression testing, for example, a human can spend hours testing and going through repetitive tasks. This

[80] Ibid.
[81] Bhanushali, Amit. Impact of Automation on Quality Assurance Testing: A Comparative Analysis of Manual vs. Automated QA Processes. // International Journal of Advances in Engineering Research 4, 26(2023). page 40
[82] Ibid.
[83] Ibid.
[84] Ibid.
[85] Ibid.

is why automation was brought in, to save the tester's time and effort. It is mostly a matter of choosing the correct situation in which to use each one of the types of testing. While automation can save time and effort for the tester, it cannot replace human intelligence and intuition. Automated testing can only test what it has been programmed to test. In contrast, human testers can identify and report issues that automated tests might not catch. Moreover, automated testing requires a significant investment of time and resources to create and maintain. While the initial investment can pay off in the long run, it might not be feasible for smaller organizations with limited resources.

Differences between automated and manual testing		
	Automated	Manual
Tools [86]	Automated testing uses IntelliJ, Selenium, Appium, TestNG, and other tools.	Manual testing uses documentation tools, like Jira.
Reliability[87]	Automated tests are somewhat more reliable.	Manual tests allow for human error.
Execution time[88]	Automated testing is not as time-consuming.	Manual tests are time-consuming.
Exploratory testing[89]	Exploratory testing cannot be done with automated testing.	Exploratory testing is a part of manual testing.
Accuracy[90]	Automated tests are somewhat more accurate.	Manual tests allow for human error.

[86] Kumari, Bhawna; Chauhan, Naresh. A Comparison Between Manual Testing and Automated Testing. // Journal of Emerging Technologies and Innovative Research (JETIR) 5, 12(2018). Page 330
[87] Bhanushali, Amit. Impact of Automation on Quality Assurance Testing: A Comparative Analysis of Manual vs. Automated QA Processes. // International Journal of Advances in Engineering Research 4, 26(2023). page 49
[88] Ibid.
[89] Kumari, Bhawna; Chauhan, Naresh. A Comparison Between Manual Testing and Automated Testing. // Journal of Emerging Technologies and Innovative Research (JETIR) 5, 12(2018). page 330
[90] Ibid.

Cost-effectiveness[91]	Automated testing is cost-effective.	Manual tests are not as cost-effective.
Programming knowledge[92]	Automated testing requires basic programming knowledge.	Manual testing does not imply the use of programming.
Engagement[93]	Automation does not require plenty of tester engagement.	Manual execution must have human engagement.
Documentation, test cases[94]	Automated tests can be documented by uploading the test execution result.	Documentation is meticulously made by the tester and thus more reliable.
Parallel execution[95]	Automated testing performs parallel tests very well.	In manual testing, parallel execution is improbable.

Automated testing is frequently used in mundane testing tasks. It shows great advantages in terms of reducing repetitive tasks and speeding up the testing process. On the other hand, exploratory testing is not possible in the scope of automated testing which cuts down on the possibility of finding a large number of bugs.[96] The approach that does not consider executing exploratory testing allows space for errors and bugs, which happen frequently outside of the ideal, happy path of testing. Sometimes the tester makes mistakes, and sometimes the tools and hardware are overheating, lacking in quality and this leads to failed tests. It is up to the tester to find the reason and cause of tests failing and provide a solution. This is where experience comes in handy because out of all the possible reasons a test could fail the tester must troubleshoot and find the real cause, and then remedy it. At this time, the testing process must be on hold while the tester finds a solution, which could take a long time and push back the planned release date until the software environment is ready to work with. Additionally, some

[91] Ibid.
[92] Bhanushali, Amit. Impact of Automation on Quality Assurance Testing: A Comparative Analysis of Manual vs. Automated QA Processes. // International Journal of Advances in Engineering Research 4, 26(2023). page 49
[93] Kumari, Bhawna; Chauhan, Naresh. A Comparison Between Manual Testing and Automated Testing. // Journal of Emerging Technologies and Innovative Research (JETIR) 5, 12(2018). page 330
[94] Ibid.
[95] Ibid.
[96] Hamilton, Thomas. Difference Between Manual and Automation Testing. URL: https://www.guru99.com/difference-automated-vs-manual-testing.html(2024-04-03)

mistakes may occur during the testing process from the tester's side, hindering the quality of test execution. Sometimes a tester might skip over tests believing that they have been tested before, or it could happen that the tester has limited attention from fatigue caused by executing repetitive tasks. This can not happen with automated testing as the script goes by the plan through the determined portions of software without leaving out pieces.[97] This is why automated testing is also considered a safe confirmation strategy to manual testing and goes hand in hand with it. With the help of automated testing by their side, manual testers can focus on innovative parts of the testing process, for example, documentation which includes test case creation, as well as risk assessment and user-oriented testing which are of immeasurable worth to the software development process. The exact premise of agile testing is where automated and manual testing synergy exists.[98] The notion is that we have to find what works, and then apply it there where it performs best. Therefore, it is safe to conclude that automated testing is not a replacement for human testers but a tool to assist them. Automated testing can help testers focus on more critical tasks, leaving routine tasks to automation which can run these tasks in the background while the tester uses this time to perform other types of actions toward testing. By using automated testing and manual testing together, testers can leverage the strengths of both methods to ensure high-quality software.

[97] Bhanushali, Amit. Impact of Automation on Quality Assurance Testing: A Comparative Analysis of Manual vs. Automated QA Processes. // International Journal of Advances in Engineering Research 4, 26(2023). page 50

[98] Kumari, Bhawna; Chauhan, Naresh. A Comparison Between Manual Testing and Automated Testing. // Journal of Emerging Technologies and Innovative Research (JETIR) 5, 12(2018). page 330

11. Conclusion

The differences between automated and manual testing can be found. The project upon which this Master's Thesis was built can attest to that fact. Sometimes manual testing is a better choice than automated testing to tackle a particular testing journey, sometimes it is vice versa. It would be a mistake to think of automated testing as better than manual testing, or to degrade manual testing just because it relies heavily on the abilities and the naked eye of the tester.

If companies go as far as to stop using manual testing and leave it all to automation, then their software might minimize bugs that are more complex and unusual, but will have very noticeable bugs present. Software that is tested manually rarely has very obvious bugs, but may contain more complex ones. A project can implement human creativity and the ability to detect bugs and the preciseness of the automated scripts and benefit from both. It will be impossible to eradicate the precise and detailed human hand from testing simply because the companies that do that face severe problems that are obvious to the users and are influencing the user's experience. Just as manual testing is dependent on the creativity and the preciseness of the tester, the scripts are written by a tester and depend on it. After the code is written one might think that this is where the engagement of the tester diminishes, but the main challenge of automated testing for a tester lies exactly in the additional editing, updating, and maintaining of the automated tests. Sometimes the software itself fails in different ways, which we have experienced many times in the duration of this project. The human desire to fix things, find a solution, research, and do the best job, is the exact decisive factor whether a project will succeed or fail.

Out of some possible challenges regarding the project one can be noted is the problem of the driver not locating elements on a screen. This problem was bypassed successfully with the use of a try-catch statement, but this can be solved in future iterations. Another challenge was swiping action on one of the screens not being possible to execute because of the nature of Android Compose. When facing this problem it was easy to lose morale as the issue was not in our control, but rather in the environment we used to perform testing. This is another problem where manual testing comes hand in hand with automated testing, as a tester here tests the clickability and advises a different approach to the developer. Facing the problem in automation, we turned to manual testing and solved the problem by determining that the tests pass manually. This is a great example of how a tester can be an advisor to the development process as well, because sometimes within the development it is easy for developers to have tunnel vision and only work as much is expected of them instead of investing time and pushing

the development process to a higher level of quality, but the task of the tester is to try to push for excellence within the software as much as possible, hence securing the quality of the software.

12. Literature

1. Anton Angelov. Design Patterns for High - Quality Automated Tests: Clean Code for Bulletproof Tests. Automate the Planet.

2. Appium Inspector. URL: https://appium.github.io/appium-inspector/2023.12/overview/ (2024-04-10)

3. Kumar Deo, Tanay. Appium Tutorial: A Detailed Guide To Appium Testing. URL: https://www.lambdatest.com/appium(2024-04-03)

4. Banu, Shakura. What is Data Driven Testing: All You Need to Know. URL: https://www.lambdatest.com/learning-hub/data-driven-testing (2024-04-03)

5. Bhanushali, Amit. Impact of Automation on Quality Assurance Testing: A Comparative Analysis of Manual vs. Automated QA Processes. // International Journal of Advances in Engineering Research 4, 26(2023). URL: https://www.researchgate.net/publication/375342615_Impact_of_Automation_on_Quality_Assurance_Testing_A_Comparative_Analysis_of_Manual_vs_Automated_QA_Processes (2024-04-05)

6. Differences between Black Box Testing vs White Box Testing. URL: https://www.geeksforgeeks.org/differences-between-black-box-testing-vs-white-box-testing/(2024-04-03)

7. Elisabeth Hendrickson. Explore It: Reduce Risk and Increase Confidence with Exploratory Testing. Dallas; Raleigh: The Pragmatic Bookshelf, 2012

8. Fulber-Garcia, Vinicius. Software Engineering: White-Box vs. Black-Box Testing. URL: https://www.baeldung.com/cs/testing-white-box-vs-black-box (2024-04-08)

9. Goals of Software Testing. URL: https://www.geeksforgeeks.org/goals-of-software-testing/ (2024-04-03)

10. Hamilton, Thomas. Difference Between Manual and Automation Testing. URL: https://www.guru99.com/difference-automated-vs-manual-testing.html(2024-04-03)

11. Introduction to Java. URL: https://www.geeksforgeeks.org/introduction-to-java/ (2024-04-03)

12. Johal, Dilpreet. Software Testing Fundamentals: Guide to Concepts and Processes. URL: https://testsigma.com/blog/fundamentals-of-software-testing/ (2024-07-04)

13. Khan, Salman. What is End-to-End Testing? E2E Testing Tutorial with Examples and Best Practices. URL: https://www.lambdatest.com/learning-hub/end-to-end-testing (2023-04-24)

14. Kumari, Bhawna; Chauhan, Naresh. A Comparison Between Manual Testing and Automated Testing. // Journal of Emerging Technologies and Innovative Research (JETIR) 5, 12(2018). URL: https://www.researchgate.net/publication/354555692_A_COMPARISON_BETWEEN_MANUAL_TESTING_AND_AUTOMATED_TESTING (2024-04-05)

15. Lambdatest. What is Automation Testing? URL: https://www.lambdatest.com/automation-testing (2024-04-07)

16. Murray, Adam. Gray Box Testing Guide. URL: https://www.mend.io/blog/gray-box-testing/ (2024-04-08)

17. Rajora, Harish. TestNG DataProviders. URL: https://www.toolsqa.com/testng/testng-dataproviders/(2024-04-03)

18. Rungta, Krishna. What is Selenium? Introduction to Selenium Automation Testing. URL: https://www.guru99.com/introduction-to-selenium.html (2024-04-03)

19. Test Driven Development (TDD). URL: https://www.geeksforgeeks.org/test-driven-development-tdd/(2024-04-03)

20. Testing for Transportation Management Systems. URL: https://tmcpfs.ops.fhwa.dot.gov/cfprojects/uploaded_files/Final%20Q&A.pdf(2024-04-03)

21. TestNG. URL: https://testng.org/(2024-04-03)

22. What Are Java Classes and Objects and How Do You Implement Them? URL: https://www.simplilearn.com/tutorials/java-tutorial/java-classes-and-objects (2024-04-03)