Ignaz Ötzlinger, BSc

Comparison of Automated Catrobat UI Testing with BrowserStack Integration

Master's Thesis
to achieve the university degree of
Master of Science
Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor
Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute of Software Technology
Head: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Bad Ischl, September 2023

Ignaz Ötzlinger, BSc

Vergleich von automatisiertem Catrobat UI testen mit BrowserStack Integration

Masterarbeit
zur Erlangung des akademischen Grades eines
Diplom-Ingenieur
Masterstudium: Software Engineering and Management

eingereicht an der

Technische Universität Graz

Betreuer
Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Institute of Software Technology
Vorstand: Univ.-Prof. Dipl.-Ing. Dr.techn. Franz Wotawa

Bad Ischl, September 2023

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Dissertation identisch.

Datum	Unterschrift

Abstract

Today, there are a wide variety of devices with various features, particularly in the field of mobile applications, on which any particular application must run without any problems. For mobile app testing, you can easily use virtual devices, which are often integrated into most development environments. However, it is reasonable to ask whether these test results are comparable to those of real devices. Services such as BrowserStack allow you to run tests on a variety of real devices from different vendors and OS versions. The mobile applications Pocket Code and Pocket Paint are used as test objects in this work, and the two test frameworks Espresso and XCUITest are used to test them. The first goal of this work is to integrate BrowserStack into an existing Jenkins CI implementation. The second main goal of this research paper is to compare the BrowserStack test results with those of virtual devices and finally to show how they differ from each other.

Kurzfassung

Gerade im Bereich der mobilen Anwendungen gibt es heute eine Vielzahl von Geräten mit unterschiedlichen Funktionen, auf denen eine bestimmte Anwendung problemlos laufen muss. Für das Testen mobiler Anwendungen kann problemlos auf virtuelle Geräte zurückgegriffen werden, die meistens bereits in Entwicklungsumgebungen integriert sind. Es stellt sich jedoch die Frage, ob diese Testergebnisse mit denen von realen Geräten vergleichbar sind. Mit Diensten wie BrowserStack können Tests auf einer Vielzahl von realen Geräten verschiedener Hersteller und Betriebssystemversionen durchgeführt werden. Die mobilen Anwendungen Pocket Code und Pocket Paint werden in dieser Arbeit als Testobjekte verwendet, und die beiden Test-Frameworks Espresso und XCUITest werden zum Testen dieser Anwendungen eingesetzt. Das erste Ziel dieser Arbeit ist die Integration von BrowserStack in eine bestehende Jenkins CI-Implementierung. Das zweite Hauptziel dieser Forschungsarbeit ist es, die BrowserStack-Testergebnisse mit denen von virtuellen Geräten zu vergleichen und schließlich zu zeigen, wie sie sich voneinander unterscheiden.

Contents

List of Figures

List of Tables

1. Introduction

Software testing is equally crucial as creating the software that is being tested. There are now numerous frameworks available for testing mobile apps, each of which has benefits and drawbacks. Since Catrobat projects already have CI implied in Jenkins, the goal of this work is to integrate BrowserStack here. BrowserStack is a cloud-based testing platform for web and mobile applications. It is used in this work to run Espresso and XCUITest automation tests on real devices for the Pocket Code and Pocket Paint applications. While an official Jenkins CI solution from BrowserStack exists for Appium, a BrowserStack solution does not yet exist for Espresso and XCUITest at the time of writing. Therefore, a special focus is on the integration of BrowserStack for the two testing frameworks Espresso and XCUITest (BrowserStack, 2023b). Appium, like Espresso and XCUITest, is a test framework that can be used to test both Android and iOS projects. This point is clearly an advantage over the other two mentioned. Espresso can only be used for Android applications and XCUITest only for iOS applications. Another clear advantage of Appium is the fact that there is only one code base and that it does not need to be managed separately for iOS or Android projects. However, one drawback of Appium is that it requires more work to implement and is harder to understand and learn compared to the other two. For example, Appium uses the client-server approach. The configuration of the Appium server is more difficult and requires strong programming skills. The advantages and disadvantages of the different testing frameworks will also be explained in more detail later. The fact that the Catrobat projects are developed and tested to a large extent by students can play an important role. This fact should be viewed as a drawback for Appium in this situation. Moreover, Espresso and XCUITest have already been used for testing on virtual devices so far. Therefore, it makes sense to stay with Espresso and XCUITest instead of migrating to Appium. Moreover, Appium runs slower than the other two. For these reasons, it also makes sense to use Espresso and XCUITest on real devices. This is the only way to achieve a valid comparison between real and virtual devices to test the Pocket Code and Pocket Paint applications (Unadkat, 2023a).

1.1. Problem Statement

A thorough and reliable testing process is necessary during the progressive development of software applications to guarantee quality and functionality. Critical phases in mobile application development include choosing an appropriate test framework and deciding whether to run tests on virtual or real devices. To use real test devices within the existing Jenkins CI integration solution for Pocket Code and Pocket Paint applications, a custom solution must be found. The reason is that there is no official BrowserStack Jenkins CI solution for the Espresso and XCUITest frameworks.
The test results and the effectiveness of test execution are affected by the choice to use virtual or real devices. Additionally, given that there are various test devices, this situation should also be considered. Depending on the manufacturer, or size of the display, etc., differences should be found here as well. The effects of this decision on the calibre of the tested applications and the effectiveness of the testing procedure for the Pocket Code and Pocket Paint applications have not yet been thoroughly examined in research.

1.2. Research Question

The following significant concerns surface when taking into account the integration of BrowserStack for test execution on real devices with the Espresso and XCUITest test frameworks in the current Jenkins CI environment for Pocket Code and Pocket Paint applications.

RQ1. How do virtual and real devices compare in terms of testing time, number of tests executed, and percent of tests passed for the Pocket Code and Pocket Paint applications?
RQ2. How strong are the differences between individual device versions and between device vendors for real devices?

2. State of the Art

In this chapter, important terms needed in connection with this work are explained in more detail. The literature that served as the basis for this work is also provided, along with a comparison of it with this research paper.

2.1. Related Work

Fantini (Fantini, 2022) describes continuous integration, which is also involved in CI and testing frameworks. This work began with an analysis of state-of-the-art automated testing frameworks that can be utilised in a CI pipeline, specifically for mobile applications. From these, the tools for creating and maintaining CI pipelines, particularly Jenkins, have been further analysed. In contrast to Fantini, this thesis placed more emphasis on the comparison of Espresso and XCUITest, while Appium was entirely ignored. In addition, instead of concentrating on the frameworks themselves, this thesis focuses more on the variations in output between various testing devices. Finally, the sole purpose of this research is to determine the distinctions only for the Pocket Code and Pocket Paint application and not for any other applications.
Lefterov and Enkov (Lefterov & Enkov, 2020) explore a detailed approach to the automation of cross-browser, device, and compatibility testing on the BrowserStack platform. To provide direct configuration output to this environment and make future web and mobile development easier, this work uses a customised extended automation framework. For the integration of BrowserStack, Lefterov and Enkov have focused more on expanding the fundamental software testing framework. In contrast to their efforts, a BrowserStack integration approach was put into place in Jenkins for the Espresso and XCUITest frameworks for the Pocket Code and Pocket Paint applications.
Guerra-Manzanares, Hayretdinm and Nõmm (Alejandro et al., 2019) made a comparison of real devices versus emulators with more attention to malware identification. In contrast, the main topic of this paper focuses on variations in test outcomes across various devices.

2.2. Catrobat Projects

Catrobat, a nonprofit project, seeks to find ways to encourage adults and children to learn programming, exercise their creativity, and prepare for the future of the digital world (Catrobat, 2023a). The terms Catrobat Catty, Catroid and Paintroid[1] are the respective projects that contain the code base for the iOS and Android versions of Pocket Code and Pocket Paint. Their code is available in public repositories on GitHub[2] and was also forked during the course of this work. Since there is an Android version as well as an iOS version for Pocket Code and both have a different code base, the BrowserStack integration has to take this into account. There is also a separate Jenkins CI implementation for both versions of Pocket Code. Moreover, different test frameworks are employed; for particular, the iOS version of Pocket Code uses XCUITest, while the Android version uses Espresso.

2.3. Pocket Code and Pocket Paint

The applications that have been tested on BrowserStack are Pocket Code and Pocket Paint. The Android application called Pocket Paint, which is connected to Pocket Code, is a graphical paint editor that, among other things, allows you to make certain parts of images transparent (GitHub, 2023). The Pocket Code application is available in both an iOS and an Android version. The purpose of this application is to introduce programming in a fun way to kids and teens between the ages of 10 and 17. As a result, users' ability to use logic and problem solving should increase. The application was created at the Institute of Software Technology at the Graz University of Technology under the supervision of Prof. Wolfgang Slany. It is inspired by Scratch (Scratch, 2023) and is designed to be used very easily. Pocket Code could be used without much prior knowledge and without additional costs, since users can use their own smartphone to develop their own applications and games with simple graphical blocks (Slany, 2014).
An overview of the iOS version of Pocket Code is shown in Figures 2.1 and 2.2. Figure 2.1 shows the home screen of the application in the first picture. For example, you can create a new project or continue working on an existing one. The second image in figure 2.1 shows the games created by the community. These games can be downloaded and used as templates. Finally, the last picture in figure 2.1 shows the individual components of a game project that can be edited. Figure 2.2 shows the graphical blocks that are used for "programming". The last two pictures in figure 2.2 show the

[1] https://github.com/Catrobat, visited on: July 27, 2023.
[2] https://github.com/, visited on: July 27, 2023.

game created within the Pocket Code application and how the game items can be graphically customised.

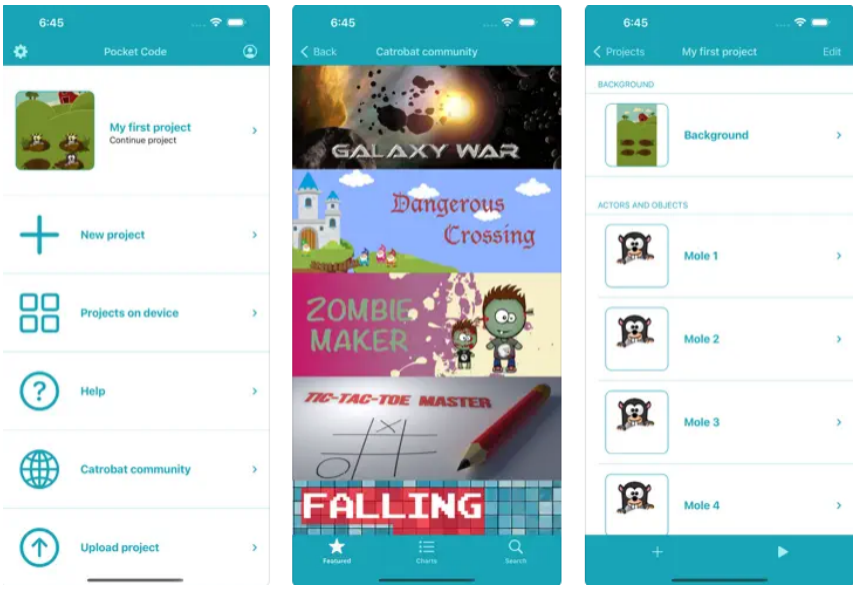

Figure 2.1.: Overview for iOS Version of Pocket Code I (Store, 2023)

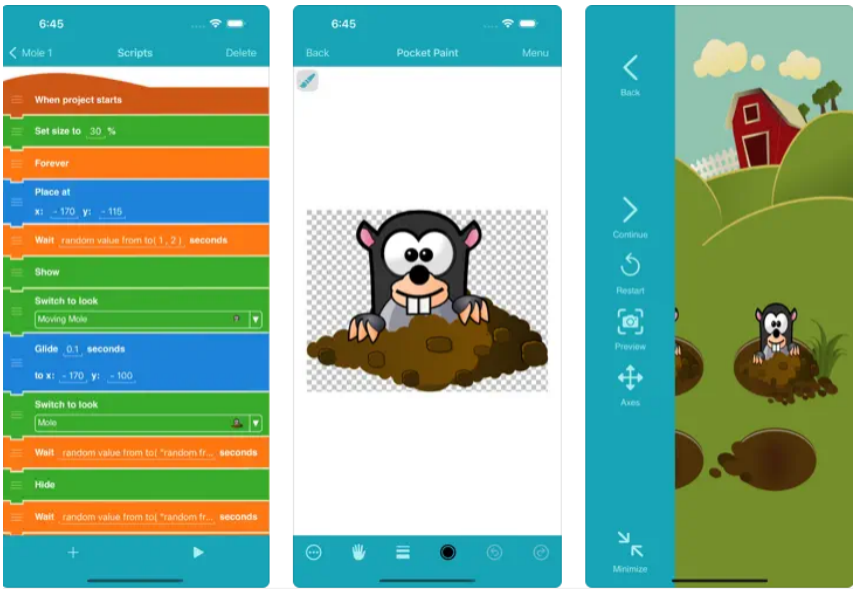

Figure 2.2.: Overview for iOS Version of Pocket Code II (Store, 2023)

2.4. Docker

Applications can be run in isolation using a platform called Docker. The term "container" refers to this isolated environment. With Docker, the so-called images, or Docker images, can be downloaded and executed in containers. These images consist of several layers and are used to execute code in a Docker container. You can also consider the containers as running instances of an image, and these containers are also temporarily stored, unlike the images. The big advantage of Docker is that Docker images run on any OS that Docker itself runs on (Jenkins, 2023b).
To test the integration of BrowserStack, a custom Jenkins server was configured and run locally through Docker. Docker eliminates tedious and repetitive configuration processes and enables rapid, simple, and portable application deployment throughout the development life cycle (Docker, 2023).

2.5. Fastlane

With fastlane, beta deployments and iOS and Android application releases can be easily automated. It takes care of all time-consuming tasks, including creating screenshots, managing code signing, and publishing your application. Although fastlane can be used for both Android and iOS, its use here is limited to the iOS version of Pocket Code. The main motivation is to maintain continuity with the iOS version of Pocket Code's prior implementation, where nearly almost all CI steps of Jenkins have already been outsourced to fastlane. In the Jenkins file for iOS Pocket Code, the corresponding lanes of fastlane are called, and the same applies to the BrowserStack integration. A lane outlines all the operations that must be performed when the lane is executed. In addition to passing parameters, the following syntax can be used: (Fastlane, 2023).

```
$ fastlane [lane] key:value key2:value2
```

2.6. DRY

DRY, or do not repeat yourself, promotes staying away from anything that is repeated twice or more because it is more challenging to keep up with. This is particularly true since whenever a change or correction is made, numerous locations must be updated, increasing the likelihood of errors and inconsistencies (Wilson et al., n.d.).

2.7. APK

The file format for Android applications is called an APK file. The acronym stands for "Android Package Kit file format". An APK file has all the information an application needs. This includes all the resources, assets, and the source code of the software (Gillis, 2023).

2.8. Software Testing

Before a programme or system is released to end users, software testing is used to detect problems. Software testing's primary goal is to make sure the programme works properly, complies with all criteria, and performs as intended. The overall quality of the software product should increase as a result and development expenses should be reduced. Software testing comes in a variety of forms, including manual, automated, and other testing techniques, including black-box, white-box, and grey-box testing. Testers can use a variety of tests during the testing process, including functional, performance, security, and usability tests (Doshi, 2023; Muccini et al., 2012).

2.8.1. Principles of Software Testing

The testers use a collection of recommendations called testing principles to help them create and carry out efficient software tests. This comprises recognising the existence of defects, time-consuming exhaustive testing, early testing reduces costs and time, clustering of defects, paradox of pesticides, context-dependent testing and the absence-of-errors fallacy (Doshi, 2023; Muccini et al., 2012; Vashishtha et al., 2014).
Software testing does not aim to demonstrate that the programme is bug-free. Instead, it involves identifying flaws that must be corrected. Code errors, missing requirements, erroneous functionalities, and other issues are examples of defects. With the current plan for BrowserStack, this would not be possible for the Pocket Code and Pocket Paint applications anyway, at least within one test run. This is because only five threads can be tested at the same time and five more can be queued. But execution on one device requires at least one thread. However, testing all possible combinations and scenarios in software applications is impractical and requires a lot of time. In the context of CI, such a treatment would be extremely impractical, regardless of the question if it would even be possible. Therefore, it is crucial to focus on the most risky and crucial regions. An important reason for testing early in the software development life cycle is to be able to detect errors at an early stage. This can reduce costs, since in many cases it is usually easier to correct errors at this stage than at a later point in time.

Groups of errors often occur. This indicates that a very small number of modules or components are in charge of the vast majority of faults. Therefore, it is crucial to focus on these high-risk regions. The phenomenon known as the pesticide paradox occurs when the same tests are run repeatedly until they no longer detect any new bugs. To discover new bugs, testers must frequently modify and update their test cases. The testing procedure should also be customised to the unique circumstances and specifications of the software application being tested. After all, the discovery of a few minor problems does not guarantee that a software programme is bug-free, and one should be aware of this (Doshi, 2023; Muccini et al., 2012; Vashishtha et al., 2014).

2.8.2. Functional Testing

As the name implies, functional testing is concerned with examining the functionality of the software system. This kind of testing is done to make sure that the system functions as planned and adheres to the functional specifications established by the stakeholders. Functional testing is concerned with the actions and operations of the software system. Unit tests, integration tests, and acceptance tests are a few examples of functional tests (Doshi, 2023; Vashishtha et al., 2014).

2.8.3. Types of Functional Testing

Unit Testing

Individual units or components are tested in isolation from the rest of the system as part of unit tests. This will guarantee that things work as planned. A test object is the smallest testable part of a software application that executes a certain function or action. A ”unit” can be a method, a function, a class, or even a module, and it can be used alone or with other units to execute. Unit tests allow for the early detection of errors, the facilitation of debugging, and an overall improvement in the quality of the code. It also promotes code reuse and allows continuous integration (Doshi, 2023; Hooda & Chhillar, 2015).
An example of a unit test is the so-called ”gorilla test.” To find any potential issues or flaws, a single module or component of the software system is thoroughly and carefully examined. This is to ensure that a portion of the software can endure and perform well even under challenging circumstances and heavy loads. One of the benefits of gorilla testing is the ability to spot potential bottlenecks or flaws in a certain module. On the other hand, it helps to locate issues or errors that other testing techniques could miss. Finally, it has the ability to manage high loads (Doshi, 2023).

Integration testing

Integration testing involves evaluating several software application modules or components collectively to make sure they function as intended and are properly integrated. The basic goal is to find potential issues that may arise when several components work together. It makes ensuring that the various units/pieces of code work together cohesively as a whole. With integration testing, early problem detection is possible and software quality is improved. Moreover, the danger of production errors is reduced and the cooperation of team members is increased. Finally, it increases trust in the software as a whole and can also result in a more precise estimation of the project timeline (Doshi, 2023; Hooda & Chhillar, 2015).
Component, system, and end-to-end integration testing are additional categories under which integration testing can be further divided. Interactions between various components or modules are the focus of component integration testing. On the other hand, system integration testing is concerned with examining how various layers or subsystems of software application interact with one another. The interaction between the complete software programme and the external systems on which it depends is the focus of end-to-end integration testing (Doshi, 2023).

System Testing

To guarantee that the programme satisfies the functional and non-functional criteria, system tests examine the entire software application as a whole. Often, this kind of testing comes after integration tests. To ensure that the application can handle a variety of situations efficiently, testers analyse the behaviour of the software application in various scenarios and under various settings, including both normal and atypical use. It can be utilised to find and address flaws or issues that could have gone unnoticed during earlier testing stages. It results in an evaluation of the overall quality of the software application, including its dependability, maintainability, and scalability. User satisfaction increases, while the chance of issues decreases (Doshi, 2023; Hooda & Chhillar, 2015).
End-to-end tests, monkey tests, and smoke tests are different types of system tests. The complete software programme, including all systems, components, and integrations involved in the workflow of the application, is tested during end-to-end testing. E2E testing focuses on business processes and user scenarios to make sure that they are reliable and satisfy user needs. Real-world scenarios, quality improvement, and comprehensive testing are all advantages of E2E testing. Monkey testing uses random inputs to determine whether the application crashes. The unique aspect of this situation is that there are no unique test cases. Its advantages include the fact that it requires less in-depth knowledge, can guarantee reliability, and can be

relatively inexpensive. Additionally, it enables the detection of problems that are impossible to find using traditional techniques. Before moving on to more extensive testing, smoke testing is performed to make sure that an application or system's fundamental and necessary functionalities operate as intended. This offers immediate feedback and early error discovery (Doshi, 2023).

Acceptance Testing

Acceptance tests ensure that a piece of software satisfies the required standards and is usable. User Acceptance Testing and Business Acceptance Testing can be distinguished from each other. User acceptability testing determines whether the programme is user-friendly and meets the end-user criteria. On the contrary, business acceptability testing verifies that the stakeholders' business/functional criteria are met and are consistent with the organisation's objectives. This increases stakeholder engagement, reduces risk, and reduces expenses (Doshi, 2023; Hooda & Chhillar, 2015).
There are different types of acceptance tests, including alpha testing, beta testing, user acceptance testing, and sanity testing. Alpha Testing is the first phase of testing that is carried out before the software is released to the public or external users. Often, the developer team or a small group of users conducts these internally. Early error detection is possible, the user experience is improved, and internal users provide feedback. However, beta testing is performed by a group of outside users who are not employees of the development team. This will improve user experience and get input from actual users. Additionally, it also has marketing and promotion purposes. User acceptability testing focuses on confirming that the software system satisfies user needs and requirements and is legitimate from the user's point of view. The goal is to improve customer happiness while reducing costs. Finally, sanity testing is the process of quickly verifying whether a specific feature or compact section of the software performs as expected after minor adjustments and prior to more extensive testing. This kind of testing is quick, efficient, and reasonably priced. It also saves time and effort (Doshi, 2023).

2.8.4. Non Functional Testing

In contrast to functional testing, nonfunctional testing focuses on assessing the nonfunctional elements of the software system. For example, performance, usability, reliability, scalability, and security testing are included. Finding out how well the software system executes its functions is more important than learning about what it does. Examples of tests include load tests, stress tests, usability tests, and security tests (Doshi, 2023; Vashishtha et al., 2014).

2.8.5. Types of Non Functional Testing

Security Testing

Security tests help identify weaknesses and vulnerabilities in a system's security and help to guarantee the protection of sensitive data. As a result, compliance is strengthened, sensitive data can be protected, and system security increases (Doshi, 2023; Hooda & Chhillar, 2015).
Security tests can be divided into penetration tests, vulnerability scans, and authentication tests. By simulating an attack, penetration testing attempts to take advantage of any potential weaknesses in a software system. However, fuzz testing involves sending a lot of unexpected or incorrect input data to the software system to find any potential flaws in the input validation and processing. Last but not least, access control testing verifies that only authorised users have access to sensitive data by testing access control measures of the software system (Doshi, 2023).

Performance Testing

Performance tests are used to assess the performance and reaction time of a software application under various workloads to locate system bottlenecks. Therefore, there should be an improvement in customer pleasure and experience, as well as a better scalability (Doshi, 2023; Hooda & Chhillar, 2015).
Performance testing includes tests of load, stress, volume, scalability, and endurance. A software application's performance and response time are assessed during load tests under a certain workload. This is done to ascertain the system's maximum capacity and make sure that it can manage the anticipated user demand. It is necessary to increase the scalability and dependability of the system. In contrast, stress testing measures the performance and response time of a software application under heavy workloads. This is done to identify the weak point of the system and make sure that it is capable of handling unexpected demands. This also seeks to increase the scalability and dependability of the system. Lastly, it should result in better scenario planning for real-world situations. Performance and response time of a software application are measured during volume testing. It is determined whether the system can handle huge amounts of data. Volume testing should improve the scalability and dependability of the system. The software's capacity to manage a growing workload and scale up or down in response to shifting user needs is tested through scaling tests. In order to determine how well the software system functions and whether it can withstand growing volumes of traffic, data, or transactions, various load conditions are tested on it. This should improve system performance optimisation and generally improve scalability. Finding out how effectively a

software system can manage a workload over an extended period of time without sacrificing performance or stability is the main goal of endurance testing. To find long-term problems, a typical or average workload or traffic scenario is recreated over a period of several weeks to months. Endurance testing should lead to less downtime and a better user experience (Doshi, 2023).

Usability Testing

A software application or system user interface and overall user experience are assessed by usability testing. The main objective is to test usability, learning ability, effectiveness, and overall user satisfaction (Doshi, 2023; Hooda & Chhillar, 2015).
Exploratory, user interface, and accessibility testing fall into this category. Exploratory running without script. This indicates that there is no adherence to a predefined test strategy or test scenario. Instead, a tester looks at the software and discovers bugs using his or her own knowledge, intuition, and creativity. This can speed up the testing and provide testers with more leeway in how they approach the task. Last but not least, it is utilised to evaluate real-world scenarios. User interface tests examine an application's graphical user interface. This is done to make sure that the application's graphical user interface works properly and satisfies the needs and expectations of the end users. The goal is to improve productivity and usability, as well as reduce costs and detect visual errors. Last but not least, accessibility testing is concerned with determining how accessible a software programme or system is to users with disabilities. In addition to being potential positive PR, it also attempts to improve user experience (Doshi, 2023).

Compatibility Testing

The compatibility of a software application or system with various hardware, software, operating systems, browsers, and other devices or components is evaluated using compatibility tests (Doshi, 2023; Hooda & Chhillar, 2015).
Compatibility testing can be divided into cross-browser and cross-platform testing. The goal of cross-browser testing is to ensure that a web application or website functions properly on a wide range of browsers, operating systems, and devices. This will improve brand recognition, customer satisfaction, and website traffic and conversion. At the same time, problems should be discovered early. Cross-platform testing is used to ensure that a software system or application functions properly across many platforms, operating systems, and devices. This involves evaluating the performance, user experience, and operation of the application on many platforms, including iOS, Android, and others. By doing this, the quality and marketability

of the software will improve (Doshi, 2023).

2.8.6. White box, Black Box and Grey Box Testing

Black box testing is a technique for performing tests without having any prior knowledge of the application's internal workings. It only looks at the fundamental elements of the system and makes little to no reference to its internal conceptual model. White-box testing, in contrast, involves a thorough analysis of the code's internal logic and structure. A tester must be thoroughly aware of the source code in order to do white-box testing. On the other hand, grey box testing combines the strategies of black-box and white-box testing. The tester tests the system from the developer's point of view because they only have a partial understanding of the internal details of the system. Testing some, but not all, of the core components of the system is the main objective (Doshi, 2023; Khan & Khan, 2012).

2.8.7. Manual Testing

In contrast to App Automate, manual testing involves looking for software bugs by hand while adhering to a defined test plan that includes a number of test cases. To determine whether the application behaves as a user might anticipate or whether deviations from this are possible, testers take on the role of a user. If deviations are discovered, the testers notify the developers, who must then fix the problem. Manual testing is a good way to test functionality, user interface, user experience, website and app behaviour, features, and user acceptance. Focusing on individual tests rather than repeating tests can be easily achieved through manual testing. However, since testing is manual, the knowledge of the tester is crucial. As a result, the tester should adopt a multifaceted strategy and have a thorough understanding of the components of a technical and business application. With BrowserStack Live, BrowserStack provides manual testing on actual hardware (Enoiu et al., 2017; Pai, 2023b).

2.8.8. Automated Testing

The automated testing market has recently grown, surpassing $20 billion in 2022, and is expected to increase at an annual growth rate of over 15% from 2023 to 2032 (Preeti Wadhwani, 2023). These statistics demonstrate the recent shift away from manual testing and toward automation. Consequently, agile approaches and rapid development have evolved from ”nice-to-have” to ”must-have” features. Manual testing is insufficient for organisations that have adopted agile practises. As part of agile methodologies, new features should be created and tested in a few weeks due to quicker development

cycles. If the testing time gets shorter and shorter, this may result in problems and a bad user experience. However, manual tests are effective when rapid and early findings and analysis are needed, but they are ineffective when code execution and multiple iterations are needed. They are also unable to keep up when the scope is broad, since they take time and may cause unneeded delays in a field of technology that is fast developing. Automation testing, on the other hand, offers the benefit of being able to conduct iterative parallel tests on a variety of devices, browser versions, and operating systems in a single pass, while automatically generating error logs and reports. This capacity can reduce bugs, which will improve the user experience. Automated testing therefore makes it easy to carry out tests on a larger scale. The faster turnaround time also reduces costs and increases test accuracy (Enoiu et al., 2017; Pai, 2023b).
As has been shown, both manual and automated tests have their advantages and disadvantages. Therefore, a combination of both can make the most sense in many cases.

2.9. Test Case and Test suite

A test case is a series of actions necessary to verify a certain feature or functionality of the software. Prerequisites, post-conditions, techniques, and necessary data are explained for this type of function validation. A test case outlines the actions the tester must take, in what order, and the outcomes that they should anticipate. For instance, if you need to confirm the login functioning, one of the test cases could be: ”Check results when a valid Login Id and Password are entered.”
On the other hand, a test suite is a collection of several test cases that are used to validate a test scenario. Test cases are added to test suites. The ongoing, active, and completed states of the test execution process can also be expressed in a collection of stages. It also goes by the name ”validation suite” and includes detailed information and goals for various test cases and system configurations needed for testing (Harrold et al., 1993; Kitakabee, 2023).
BrowserStack requires a test suite in addition to the actual application to be tested. This is needed so that the test cases for the application under test can be executed on real devices. For Espresso, BrowserStack requires the test suite in APK format. For XCUITest, a normal zip file is specified (BrowserStack, 2023b).

2.10. Continuous Integration

Software projects are becoming more and more complicated these days. Additionally, there are now more devices than ever before, and the number of different platforms on which mobile applications can run has also increased significantly in the last few years. Consider the market for smartphones and the numerous manufacturers that have all released their own devices. For instance, the company Staircase studied their own application "OpenSignalMaps" for six months and discovered that 599 different manufacturers and nearly 4000 distinct Android devices could be recognised (JOHNSTON, 2023). Every software team must at some point ask themselves how they can guarantee, with a certain probability, that their own app will function properly on all devices. Manual and automated tests can be used to accomplish this. For smaller projects, manual testing may still make sense, but for larger projects, you quickly reach your limits. Given that even a 10% change in the code requires 100% of the features to be re-tested, manual testing appears to be worthless in these circumstances (Hayes, 2004). The term CI is closely related to the alternative option of automated testing. The term "continuous integration" (CI), which essentially specifies concepts that can be applied to daily workflow, was first thought to be part of extreme programming methodologies (Meyer, 2014). The following guidelines are intended to demonstrate the benefits that CI can offer over manual testing. Since people also make mistakes, it should be obvious that manual testing alone is insufficient to properly test applications over an extended period of time, especially if they are upgraded and developed on occasion (Collins, 2023; Ram, 2021). It seems tempting to spend less time testing and more time on new features, especially when teams are under pressure for time. However, this can lead to serious problems because bugs might not show up at all or may show up much later. Fixing them later can take a lot of time and cost additional money (Smart, 2011). The complexity of apps will continue to grow, and they are also much more complex now than they were a few years ago. Because of this, an application cannot be evaluated independently anymore. Instead, it must be used in combination with other components, such as other code parts of the same application or interfaces. As a result, manual testing alone can easily miss errors that occur only in combination. Among the many advantages of CI, it could be challenging to choose which tests can be omitted while still testing the full set of features to save time. The interesting thing here is that tests that take more than ten minutes can decrease productivity on the developer's side and thus also lead to negative effects (Meyer, 2014).
Meyer (Meyer, 2014) states that in addition to all the advantages, the following must be taken into account with regard to CI. New modifications should be merged back into the main branch as soon as possible. The reasoning

behind this is that minor adjustments that have been shipped quickly to production are also simpler to repair if they lead to errors. This is due to the fact that troubleshooting is facilitated and often only minor changes need to be made to correct the errors. This applies to daily and frequent commitments. In addition to this, there are also extreme approaches to prevent errors. These approaches claim that changes should be rejected if they have not been committed before the end of a working day. Even if it is an extreme approach, these methods are designed to make it simpler to overcome the habit of making only major complex adjustments rather than small and simple ones. All of these things can be summarised with the words "Commit Daily, Commit Often" (Meyer, 2014).
Checking if changes can cause errors or code breakdowns is a crucial component of CI. It is best that this check can be performed as soon as possible. This is also understood as "Build Early, Build Often", and for this, a variety of tools are available, like Jenkins. These tools essentially have the job of identifying code changes and testing them so that developers can determine if their changes work or not. The basic idea is that the earlier such errors are detected, the easier it is to fix them. Furthermore, it is important that all developers can easily handle such tools. The advantage of Jenkins in this case is that it has great community support and is also very flexible in how new jobs can be created (Meyer, 2014; Tutorialandexample, 2023).
Moreover, the builds should always be green to avoid hindering the work of other developers. Therefore, if a project does not yield a green result, fixing it should be the main priority (Meyer, 2014).
The importance of fostering a culture in which a development team sees itself as accountable and accepts responsibility for providing value promptly to the client is recognised as a matter of culture (Meyer, 2014).
It was previously demonstrated how important it is to provide developers with the ideal environment, especially what is understood by "Build Early, Build Often". We will thus look at how easy it is to set up a Jenkins server and how we will use it later with our own modifications and testing purposes as well.

2.11. Jenkins

Jenkins, formerly known as Hudson, is an open-source CI tool written in Java that is widely used by many teams. Its ability to be used with a variety of technologies, including .NET, Ruby, Groovy, Grails, PHP, and others such as Java, has contributed to its market domination of about 47.09% in 2023 (Sense, 2023). Additionally, Jenkins is easy to configure and use, based on a simple and intuitive user interface. Moreover, Jenkins is also simple to set up for specific jobs and easy to extend with the help of a wide variety

of different plugins. Finally, the community is quite active, and updates with new features, bug fixes, and plugins are released almost weekly. Older Jenkins versions are also still supported for a long time, allowing teams that do not want to update every week to still use Jenkins (Smart, 2011).
Jenkins is based on the Jenkins Master Slave Architecture as shown in Figure 2.3. Because the slave's primary responsibility is to carry out the master's instructions, the word "slave" was coined. However, the Jenkins community made the decision to remove objectionable phrases from the project in 2016 (Jackson, 2023). As a result, the term "slave" was changed to "agent" in Jenkins 2.0. In this context, one should also talk about primary and replicas instead of master and slave architecture (McKenzie, 2023). The Jenkins master, or Server, is responsible for holding all important configurations. This includes sending builds to agents for actual execution and monitoring the agents. Furthermore, the recording and presentation of results fall also under this. Finally, a master instance of Jenkins can also directly execute build jobs. In this case, the master node itself assumes the role of an agent. A Java executable file, called an agent, runs on a different computer. It can operate on many operating systems and basically take requests from the Jenkins master. Agents serve as work nodes and are assigned a task. You can use a combination of Windows, Linux, and even container servers as build agents, and you can attach as many Jenkins agents as you like to a master. Alternatively, Jenkins can choose the next available agent on its own, or projects can be specified to always run on a particular agent machine or a kind of agent machine (Jethva, 2023; Smart, 2011).

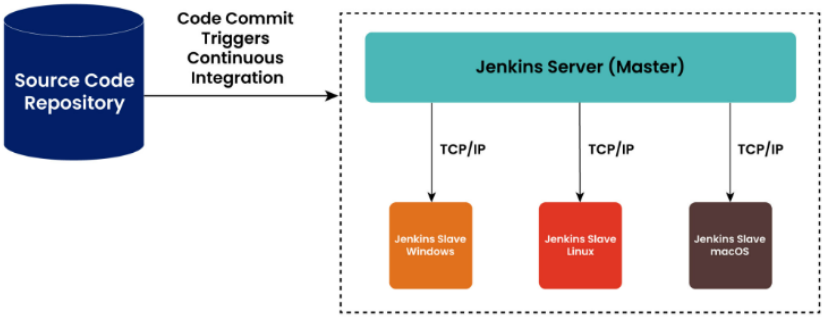

Figure 2.3.: Architecture of Jenkins (Solutions, 2023)

2.12. BrowserStack

BrowserStack is a cloud-based testing platform for web and mobile applications (BrowserStack, 2023b). It was made available to everyone in 2011 after leaving the beta phase and currently has over 4 billion active users. Browser-

Stack supports more than 9,000 different devices, 21 different operating systems, and eight popular browsers (BrowserStack, 2023d).
Figure 2.4 shows the BrowserStack architecture and a detailed overview of the cooperation of all tools and libraries in the architecture of the software testing framework. "Hubs" and "Nodes" are the foundation of the BrowserStack architecture. The hub is the central location where all requests can be found, together with details of the platform, browser, and device that will be used to run the test. It will distribute them to the registered nodes according to the request. The corresponding tests are run on the nodes. Each node is a genuine mobile device registered with the hub, a physical or virtual machine, or both. As soon as a node registers with the hub, the hub has access to all the node's information (Lefterov & Enkov, 2020).
The advantage is that you can use them for testing without having to purchase or maintain the equipment yourself. Since this work is intended to test mobile applications, the mobile part of BrowserStack is of greater interest. This part can be divided again into BrowserStack App Automate and BrowserStack Live.

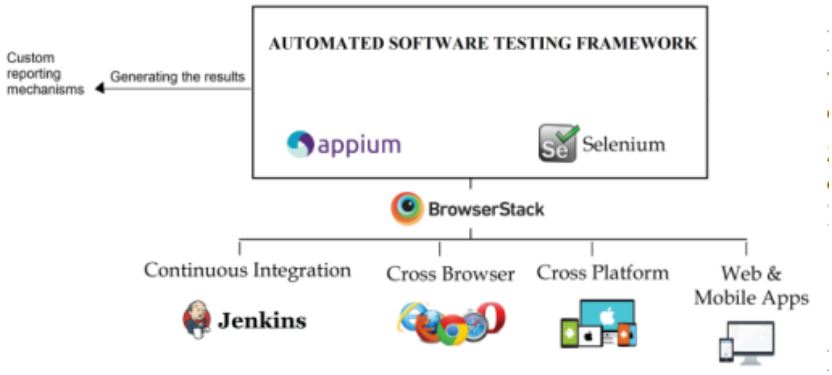

Figure 2.4.: Architecture of BrowserStack (Lefterov & Enkov, 2020)

2.12.1. BrowserStack App Automate

Through BrowserStack App Automate, automated testing can be performed on real devices. For this purpose, the application to be tested must be uploaded to BrowserStack together with the test suite. BrowserStack supports a number of testing frameworks for automated testing. These also include Appium, Espresso, and XCUITest. Furthermore, BrowserStack already provides a Jenkins CI integration option for Appium. For Espresso and XCUITest, this was missing at the time of this work (BrowserStack, 2023c).

2.12.2. BrowserStack Live

Along with the option to automatically test the submitted application based on the uploaded test suite, BrowserStack also provides interactive testing. With BrowserStack Live, interactive testing can be done instantly on a wide range of real iOS and Android devices in the cloud. This kind of manual testing might be quite beneficial if you want to validate automated failed test cases on real devices (BrowserStack, 2023a).

2.13. UI Testing Frameworks

The functionality and design of a user interface play a major role in its success or failure. It is essential to be able to ensure the functionality of this user interface to a particular extent in order to convince and satisfy clients. As a result, testing the user interface has become a crucial step in the creation of applications. UI tests are created to ensure that each function performs as intended and that a set of standards are met. Moreover, UI testing helps to establish whether the website or app's audio and visual components are appealing and can maintain users attention (Bose, 2023).
Currently, there are many UI testing frameworks available, each with unique benefits and drawbacks. This thesis will concentrate on the frameworks XCUITest and Espresso because of their special significance for our work. In addition, the Appium framework will be introduced to make the differences between XCUITest and Espresso clearer. The main differences between the individual testing frameworks are also shown in the table 2.1.

2.13.1. User Interface Testing

User interface testing is intended to ensure that user interface features operate without errors and according to the plan. Instead of evaluating the fundamental logic of the product, user interface testing focuses on testing the components with which the user must interact. This comprises the graphical and structural components of the software. The full range of graphic-based symbols and visual indications are covered by UI testing. For example, menus, text fields, buttons, check boxes, toolbars, or colours. Generally speaking, user interface tests are meant to evaluate usability, functionality, compliance, responsiveness, performance, and visual design (Bose, 2023; El-Morabea & El-Garem, 2021; Min & Cai, 2018).
The range of test cases that can be included in UI testing is fairly broad. Field widths, navigational elements, progress bars, type-ahead, table scrolling, and key-user operations are just a few examples of this. The test determines whether only legitimate data may be entered for specific data fields, such as date, currency, etc., in the case of data type problems. When it comes to field

widths, the test determines whether certain text fields prevent users from entering data that exceed a predetermined character limit. For navigation components, the test verifies that all buttons are functional and sends users to the appropriate page. In the case of progress bars, the test determines whether a progress bar is shown to the user to indicate that the page is loading when viewing pages or screens that take some time to fully load. The idea behind scrolling tables is that a user should be able to scroll through all the data while keeping the headings visible and in place if the website has data tables and the table extends to a second page. Critical user activities are routine tasks that should be executed without difficulty by the user. Nevertheless, these actions may be crucial for the application's operator or the company behind it. For example, this includes the steps necessary for payment transactions to ensure that the business does not lose revenue (Bose, 2023; El-Morabea & El-Garem, 2021; Min & Cai, 2018).

2.13.2. Appium

Appium is an open source test automation framework for mobile applications and is used to check whether an application's functionality operates as expected. It enables the creation of code that executes user scenarios in the application's actual user interface, as nearly as possible imitating real-world behaviour. By using a standardised set of tools, Appium seeks to facilitate this type of automation on a variety of platforms. Most platforms have tools that enable some level of user interface automation, but these tools are often platform-specific and require expert knowledge and experience with specific programming languages. Numerous other programming languages, including Java, Python, Ruby, JavaScript, and many others, can be used to create Appium scripts (Appium, 2023; Manoj, 2015; Unadkat, 2023a).
Appium utilises the mobile JSON wire protocol. This protocol, which is used for client-server communication, is an extension of the Selenium JSON wire protocol. This makes it possible for test scripts to be developed in various programming languages rather than just one. The fact that Appium works with both Android and iOS platforms is an additional benefit. This makes it simple to reuse test scripts across several platforms, which should help to make code maintenance simpler (Nishant, 2017; Unadkat, 2023a).
However, it is made much more difficult by the fact that Appium utilises the client-server approach. To configure the Appium server, you will need solid programming skills. XCUITest and Espresso do not require this. Moreover, Appium executes tests more slowly than Espresso and XCUITest. Each command specified in the test script must be executed after the server has started, which takes some time. Moreover, Appium is unable to automatically detect web elements. So, in the test script, the location of the element must be explicitly defined (Unadkat, 2023a).

The Android versions of Pocket Code and Pocket Paint are already being tested with the Espresso framework on virtual devices. Appium has not been used so far and therefore will have to be introduced first. The aim of this work is to compare virtual devices with real devices. However, this would only make sense if both use the same testing framework. It would therefore make sense to also use Espresso for testing on BrowserStack with real devices. Naturally, this also holds true for the iOS version of Pocket Code, which has already been tested with XCUITest on virtual devices.
In contrast, there is currently only an official BrowserStack CI integration solution in Jenkins for Appium. For Espresso and XCUITest, this option is currently not available. Therefore, a separate CI solution must be found for both frameworks. It makes reasonable that creating your own solution requires more work than using one that is already created.

2.13.3. Espresso

Espresso is a native open source Google framework that was created to test the Android UI automatically. It is used to test native Android mobile applications and is a component of the Android SDK. Espresso offers a set of APIs for developers to create trustworthy Android UI tests and replicate user interactions in the application. In addition, Espresso offers automatic synchronisation of test operations and interface elements. Its automatic synchronisation determines when the main thread is not active and launches the test commands at the right moment. The scripts are written in Java, and there are only three different types of actions, viewMatchers, viewActions, and viewAssertions. They are designed for a very compact and lightweight API, which makes it therefore easy to maintain and customise. To determine whether an element satisfies a specific criteria, ViewMatchers are used. For instance, it decides whether a certain text is present or not, or whether the element is displayed. Use ViewActions to perform actions within the application, such as clicking and typing. Finally, to determine the state of the selected element, ViewAssertions and ViewMatchers are used. The check to verify whether the object is visible or not serves as an example. The Espresso testing framework is also included in Android Studio, a native Android development environment. This implies less setup work than Appium. Furthermore, the Espresso Test Recorder is another feature that makes Espresso less challenging than Appium. This makes it possible to test Android applications without having much programming experience. The necessary code is automatically generated, so you only have to record the user interactions with the application. Finally, Espresso is also faster and more reliable than Appium and does not require server communication (Lämsä, 2017; Meiliana et al., 2018; Unadkat, 2023a).
Espresso is a popular tool for testing native Android applications. Only

Java and Kotlin are supported, making it possible to write tests in the same language as the application being tested. But it is also a restriction, since the language stack is constrained at the same time. In some cases, the applications should also run on Android and iOS devices. This also affects Pocket Code, as there is both an Android and an iOS version. For each of the two versions, there is a separate project with its own distinct test scripts. Compared to Appium, this situation seems more complicated because Espresso by itself is insufficient in this situation. For the iOS version, XCUITest is also needed (Meiliana et al., 2018; Unadkat, 2023a).

2.13.4. XCUITest

A testing framework called XCUITest was initially presented by Apple in 2015 as a significant improvement to improve testing capabilities. XCUITest itself is built on XCTest, which is a testing framework that is integrated into Apple's Xcode IDE. XCUITest can be used to test the user interface of native iOS apps with Swift or Objective C. Compared to Appium, the tests run faster and are far more consistent and dependable. This is due to the fact that the XCUITest testing framework was specifically developed to facilitate UI testing for iOS apps. The inclusion of XCode in the IDE offers an additional benefit. It is possible to keep the tests and the source code together, which makes it easier to quickly determine whether changes to the source code were successful or not (Unadkat, 2023b).
The focus on iOS is not only a strength, but also has the potential to be a weakness. If the aim is to have an application run on both Android and iOS, XCUITest alone is no longer sufficient. As a result, two separate code bases must be maintained, which increases the workload. Another drawback of XCUITest is that it only provides limited language support. In contrast to Appium, XCUITest is limited to Swift and Objective C (Unadkat, 2023b).

2.13.5. Key differences of Appium, Espresso and XCUITest

Table 2.1 shows and summarises the main differences between Appium, Espresso and XCUITest (Nuriel, 2023; Unadkat, 2023a, 2023b).

Table 2.1.: Key differences of Appium, Espresso and XCUITest

Parameters	**Appium**	**Espresso**	**XCUITest**
Purpose	Automated testing of both mobile and desktop applications	Automated testing of mobile Android applications	Automated testing of mobile iOS applications
Backers	Open source	Google	Apple
Setup	Hard	Easy	Medium
Language Supported	All leading languages like Java, Python, Ruby, etc.	Only Java and Kotlin	Only Swift and Objective C
Flakiness of test	Very	Low	Low
Test type Instrumented	Black Box \ No	White Box \ Yes	White Box \ Yes
Speed	Slow (approx. 8t)	Fast (approx. t)	Fast (approx. 2t)
Jenkins CI solution for BrowserStack	Official BrowserStack solution	No Official BrowserStack solution	No Official BrowserStack solution

2.14. Virtual and Real Devices

One of the main goals of this work is to investigate the differences between virtual devices and real devices to test the mobile applications Pocket Code and Pocket Paint. For testing with real devices, BrowserStack (BrowserStack, 2023b) was used, which provides a large number of real devices for testing mobile applications for both Android and iOS. Virtual devices can be divided into emulators and simulators (Manohar, 2023; Pai, 2023a) and have already been used for Pocket Code and Pocket Paint applications.

2.14.1. Virtual Devices

A virtual testing device is a type of computer programme that offers features similar to those of a real device. This enables testers to assess how their applications would work on a real smartphone by mimicking its features. Emulators and simulators are two types of virtual devices. (Gao et al., 2014; Manohar, 2023; Pai, 2023a).

Emulators

An emulator is computer software that simulates the hardware and operating system of the target device. The emulation techniques used by different types of emulators may vary. However, reproducing the experience of using the original equipment or programme is always the main objective. This is achieved through binary translation to transform the target device's ISA (Instruction Set Architecture) into the ISA of the host computer. Each processor family has its own set of machine language instructions known as ISAs, which are used to create individual device configurations that reflect the capabilities and behaviour of the device. You can create a virtual test environment by simulating the functionality of the target mobile device on your PC and converting the ISA. Unfortunately, it comes at the cost of delay. Automation, unit testing, and debugging all make use of emulators (Gao et al., 2014; Manohar, 2023; Pai, 2023a).

Simulators

A simulator is a piece of software that allows your computer to run specific applications created for an alternative operating system. These are primarily used for iPhone and iPad devices, as opposed to easily emulated Android smartphones. The iOS simulators sit on top of the computer's operating system to simulate iOS and execute the necessary applications inside of it. Nevertheless, as the iOS emulator requires Apple's native Cocoa API, one can only use macOS to run it. The GUI, run-time, and numerous other activities depend on this Cocoa API (Manohar, 2023; Pai, 2023a).
Hardware configurations cannot be duplicated by virtual devices. In addition to this drawback, performance validation on virtual devices is unreliable because these observations may change when the operating system is improved. You cannot be sure that your test results will be accurate on a real device, even if everything goes smoothly. Virtual devices are also often accessible only for a specific platform, and sometimes they will not work with your application. Emulators, for example, are limited to testing Android devices and do not support other operating systems. The benefits of virtual devices, which are cheaper and work with both local and cloud-based solutions, outweigh these drawbacks. In many instances, virtual devices are not only cheaper, but also more accessible than real devices. Moreover, debugging is simpler on virtual devices, since many development environments already provide debug facilities for virtual devices, and faults may be quickly captured via screenshots. Finally, virtual device testing provides the ability to test your application across many platforms and use cases that demand a variety of hardware and operating systems (Manohar, 2023; Pai, 2023a).

2.14.2. Real Devices

Real testing devices are mobile phone models that are used to evaluate the functionality and usage patterns of a website or app. These handsets are actually used by the end customer. With testing on real devices, additional different factors, such as temperature, can be taken into account. Also, a smartphone can have many different interruptions. These include phone calls, battery problems, or screen lock. Of course, these factors should also play a role in testing, and testing on real devices allows you to do this. Testing teams typically purchase mobile devices running iOS, Android, phones, tablets, and iPad's to test their software applications. To test the Pocket Code and Pocket Paint applications on real devices, BrowserStack is used (Gao et al., 2014; Manohar, 2023; Pai, 2023a).
Testing on real devices is more expensive than testing on virtual devices, given that virtual devices are generally available for free. For optimal testing on real devices, different devices are needed, which all cost money. In addition, many new devices have entered the market in one year. This device fragmentation must, of course, also be taken into account and again costs money. Services such as BrowserStack offer real test devices, but a licence is usually required for this. In most cases, these licences come at a price. In any case, the use of real devices for testing is more expensive and it is much more challenging to discover issues on real devices in the early stages of software development. These drawbacks can be offset by the fact that testing on real devices typically yields more insightful results. Additional aspects may also be included, such as network-related actions. For example, a call or text received in the middle of a transaction falls under network-related actions. On real devices, performance issues can also be found more quickly. Since these cannot be simulated on virtual devices or can be simulated much less accurately. For example, the battery cannot be simulated in virtual devices. Finally, testing on real devices will enable you to understand the user experience taking into account memory, size, CPU, and other factors (Gao et al., 2014; Manohar, 2023; Pai, 2023a).

2.14.3. Virtual vs. Real Device Testing

Since both virtual and real testing tools offer a variety of benefits and drawbacks, you should consider when to utilise each. It is a good idea to use virtual devices in the early stages of code development because they have a better debugging function than real devices. Because frequent iterations are necessary in the initial phase, virtual devices could speed up and improve the cycle. Furthermore, the user's choice to test on emulators or simulators is also influenced by how quickly and frequently tests are conducted. However, it is more difficult to find network-related problems with virtual devices.

This is because the testing is done on real devices in a live environment. Additionally, testing on actual devices will typically be more expensive than testing on virtual ones. On the other hand, when more precise test results are required, it makes sense to utilise real devices. On real devices, performance issues are also easier to spot (Gao et al., 2014; Manohar, 2023; Pai, 2023a).
A combination of virtual and real testing would be ideal for Pocket Code and Pocket Paint applications as well. As before, testing could typically be carried out on virtual devices. In the case of releases or at regular larger intervals, additional testing could be performed on real devices. It would then also make sense to run tests simultaneously on multiple real devices to increase the significance of the test results.

2.14.4. Differences of Emulators and Simulators and Virtual and Real Devices

Tables 2.2 and 2.3 show the most important differences between emulators and simulators and between virtual and real devices (Manohar, 2023; Pai, 2023a).

Table 2.2.: Differences of Emulators and Simulators

Criteria	**Emulators**	**Simulators**
Target Area	Duplicates all of a device's operating systems, hardware, and software. Typically, it is a full re-implementation of the original software	Mimics the behaviour and settings of a real device by creating an environment. Typically, a modified version of an existing programme.
Provided by	Provided by device manufacturers	Provided by device manufacturers and other companies
Internal Structure	Written in Machine-level assembly language	Written in High-level language
Suitable for Debugging	Reliable and more suitable for debugging	Less reliable and not so suitable for debugging
Performance	Slower due to latency based on binary translation	As there is no binary translation, it is quicker
Example	Android SDK	iOS Simulator

Table 2.3.: Differences of Virtual and Real Devices

Criteria	**Virtual**	**Real Device**
Cost	Low expense, often open source and free	Costs prevent large-scale purchases
Reliability	Cannot accurately reproduce real user conditions, such as software and hardware setups. Appropriate for executing specific kinds of functional test cases.	Provide realistic findings and enable testing in a user-like environment. Real-time performance testing more appropriate (e.g. Network, battery, location, notifications)
Processing Speed	Slower due to latency based on binary translation	Faster, no binary translation
Suitable for Debugging	Step-by-step debugging simple and allow you to detect flaws.	Could be challenging, especially when trying to find faults.
Cross-Platform Testing	Easy to do cross-platform testing.	Is not supported

3. Methodology

3.1. Jenkins docker-compose.yaml File

To use the same version as in the actual implementation of Jenkins for Catrobat at the time of this work, we run Jenkins in Docker and use version 2.337 of Jenkins as an image. To start the Jenkins container, the following command is used.

```
$ docker compose up -d
```

The -d stands for detach mode and let's the jenkins container run in the background. This keeps the command line clear of output from the started container so that it can still be used freely. The contents of the docker-compose file are shown in the listing3.1, (Goebelbecker, 2023).

Listing 3.1: Jenkins Docker Compose YAML File

```
1  # docker-compose.yaml
2  version: '3.8'
3  services:
4    jenkins:
5      image: jenkins/jenkins:2.337
6      privileged: true
7      user: root
8      ports:
9        - 8080:8080
10       - 50000:50000
11     container_name: jenkins
12     volumes:
13       - C:/Users/ijoet/iCloudDrive/MT/jenkins_compose
     /jenkins_configuration:/var/jenkins_home
14       - /var/run/docker.sock:/var/run/docker.
     sockdocker logs jenkins | less
```

- The version of the docker-compose file used is specified in the file's first line. The Services section contains a list of the containers that should run. Here we are using the name Jenkins.
- The docker image to use for the Jenkins container is specified in line 5 of the listing 3.1. It pulls the image "jenkins/jenkins" with the tag "2.337".

- The container is instructed to operate in privileged mode on line 6 in listing 3.1. It enables the device to have access to the host. For example, to be able to easily retrieve the test files that are kept on the host computer. The application that will be tested and its associated test suite file are both considered test files in this context. It is more practical to store the test files on the host rather than rebuilding or downloading them for every test run.
- The user inside the container is changed to ”root”.
- The next environment variables are set for the Jenkins container. The ”JAVA_OPTS” variable is specified to enable local checkout for Git SCM.
- The following parameter connects the ports in the container to their equivalent ports on the host machine. Jenkins’ online user interface runs on port 8080, whereas Jenkins’ agent communication runs on port 50000.
- The docker container name is then set to ”jenkins.”
- The volumes to mount inside the container are defined next. The first volume transfers the ”/var/jenkins/home” directory inside the container to the local directory ”D:/MT/jenkins compose/jenkins configuration” that holds Jenkins’ data. Jenkins can further communicate with the Docker daemon by using the second volume, which transfers the Docker socket on the host machine to the Docker socket inside the container. Finally, the command ”docker logs jenkins” allows you to inspect the Jenkins container’s logs. It retrieves the logs produced by the Jenkins container. In combination with the ”less” command, it enables a developer to examine the output on a pager, which enables scrolling and searching through the logs.

3.2. Jenkins Setup

Jenkins can already start after the docker-compose file definition. Calling localhost on port 8080 when the container of the above docker-compose file is running should trigger the unlock page shown in Figure 3.1 to appear. The page asks for a required password, which should be available in the docker logs. Select Install Suggested Plugins on the next page. You can specify a new log-in and password as soon as Jenkins is finished. In addition, the hostname of Jenkins can be optionally changed on the next page. After all, Jenkins is supposed to be ready for usage at this stage and the Jenkins homepage should appear.

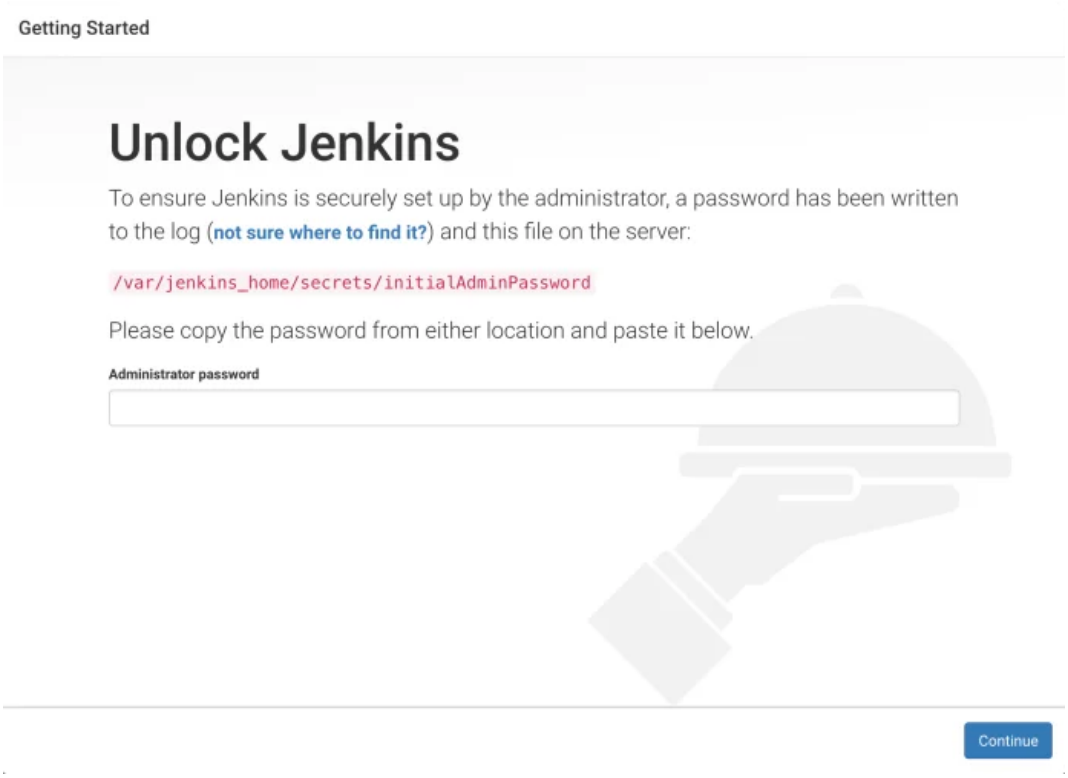

Figure 3.1.: Jenkins Unlock Page

3.3. Modifications on the Jenkins Files

The following examples are taken from the Catrobat Paintroid implementation. However, with just a few minor adjustments, this also holds for Catrobat and Catty as well. The lines below refer to appendix A. A new parameter was included on line 3 of the file seen in the appendix A to enable the option of doing or not doing local testing. In this context, local means that BrowserStack is not used. Since the default value is true and to avoid running tests on the Jenkins agents themselves, this option must be deselected before running the pipeline. Moreover, on line 100 of the file seen in the appendix A, the BrowserStack testing can also be turned on or off. Tests will run only on BrowserStack if that option has previously been selected, since the value of the parameter BROWSERSTACK_TESTING has been set to false. On lines 5 to 17 of the file seen in the appendix A, a separator has been added. A separator similar to this one already exists in the Catrobat Catroid Jenkins file. Compared to other parameters, this divider makes it simpler to select the relevant parameters that apply only to BrowserStack. Lines 18 to 57 of the file seen in the appendix A contain a list of all Android test devices for BrowserStack. For Catrobat Catty, BrowserStack also supports a wide variety of different iOS devices. In this case, the Catty Jenkins file

is notably different from the other two. The format of each listed device is <device name>-<OS version> (BrowserStack, 2023b). A special case is the first value in the list, which is called "AndroidDevices" for Catrobat Catroid and Paintroid and "IosDevices" for Catrobat Catty. Unlike the other options, where only one test device is specified, in this case multiple test devices are selected at once. The reason behind it is that testing on multiple devices rather than just one should lead to more insightful test results. Finally, the option of choosing between 1 and 5 concurrent threads was provided at line 101 of the file seen in the appendix A. This option was given to enable parallel testing on BrowserStack and therefore reduce the execution time. The first number provided, 1, is the default value in this case. In addition to the selection of threads, BrowserStack calls them shards. Furthermore, a clickable link has been added to open the BrowserStack dashboard. The idea behind this is that it should be simple to query BrowserStack's current utilisation status and to take this into account when choosing the number of shards. Because at the time this work was being done, the BrowserStack plan allowed a maximum of five shards to run simultaneously and an additional five to be queued.

3.4. Build Test Suite Stage

In addition to the Pocket Code and Pocket Paint application itself, a test suite app will also be built for BrowserStack testing. For the iOS version, it is a zip file, for the Android applications, it is an apk file, just like the actual application itself. All the tests that will run on BrowserStack are included in this test suite. The test app is then uploaded to the test device and installed in the same way as the actual application that should be tested (Google, 2023).
The command ./gradlew with all the options specified in the listing 3.2 is used to create the test application. The script known as the Gradle wrapper is a powerful tool and can also be used to build projects (Gradle, 2023). The Jenkins environment variables BUILD_NUMBER and BRANCH_NAME are used for a custom property that is set by the -Pindependent flag. To create the test application, the flag 'assembleAndroidTest' is provided, and the Debug Build Type is used as the default option if the test build type is not explicitly specified (Google, 2023). If all went well, a separate apk file for the test suite would be generated, which is then also available via Jenkins if desired. Jenkins' plotting or graphing plugin function is used (Jenkins, 2023d). It outlines a number of parameters, including the CSV file name, the location of the CSV file, and options for graph styling. Finally, the numBuilds argument indicates that data from the previous 180 builds will be shown. The overall goal is to visualise the build data.

Listing 3.2: Build Test APK Stage

```
1 stage('Build Test-APK') {
2     steps {
3         sh "./gradlew -Pindependent='#$env.
    BUILD_NUMBER $env.BRANCH_NAME' assembleAndroidTest"
4         archiveArtifacts 'Paintroid/build/outputs/apk
    /androidTest/debug/*.apk'
5         renameApks("${env.BRANCH_NAME}-${env.
    BUILD_NUMBER}")
6         plot csvFileName: 'dexcount.csv', csvSeries:
    [[displayTableFlag: false, exclusionValues: '',
    file: 'Paintroid/build/outputs/dexcount/*.csv',
    inclusionFlag: 'OFF, url: '']], group: 'APK Stats,
    numBuilds: 180', style: 'line, title: 'dexcount' }
7 }
```

3.5. BrowserStack Testing Stage

A new separate stage was introduced for BrowserStack testing in listing 3.3. The stage for BrowserStack testing only runs if the parameter "BROWSERSTACK_TESTING" has been previously selected. If this is the case, the shared libraries function named 'BrowserStack' will be executed. This shared libraries function is used for the entire BrowserStack execution process. The Pocket Code or Pocket Paint App itself and the Test App's locations are defined by the first two parameters. There is an additional third argument for the android versions of the Catrobat project. It specifies whether it is the Catrobat Catroid or Paintroid project. In the given example in listing 3.3, the Catrobat Paintroid project was chosen. As there is only one iOS project for Catrobat, this parameter is not required for the iOS Pocket Code version of the Catrobat project called Catty. It is also mentioned that this script is called inside a 'withCredentials' block. This is done to safely send the login information to BrowserStack while allowing the shared libraries "browserStack" function to use these data as well. The values of the credentials are accessible in Jenkins with the ID 'browserstack'. These values must be added to the Jenkins configuration with the following steps (Jenkins, 2023a):

- Go to the Dashboard after opening the Jenkins user interface.
- Under the left menu, select Credentials.
- Select the type of credentials (username with password).
- Finally, to store the credentials, enter them and then click OK. This will give the credentials a unique ID.

Following that, the obtained values are given correspondingly to the environment variables BROWSERSTACK_USERNAME and BROWSERSTACK_ACCESS_KEY. The shared libraries function "browserStack" can then use these variables as well. These credentials are only revealed in encrypted form and not as plain text for non-admin pipeline users. Compared to simply putting the credentials in the environment variables, this solution provides the benefit of keeping the credentials more private.

Listing 3.3: BrowserStack Testing Stage

```
1  stage('Browserstack testing') {
2      when {
3          expression { params.BROWSERSTACK_TESTING ==
       true}
4      }
5      steps {
6          withCredentials([usernamePassword(
       credentialsId: 'browserstack', passwordVariable: '
       BROWSERSTACK_ACCESS_KEY', usernameVariable: '
       BROWSERSTACK_USERNAME')]) {
7              script {
8                  browserStack('app/build/outputs/apk/
       debug/', 'Paintroid/build/outputs/apk/androidTest/
       debug/', 'Paintroid')
9              }
10         }
11     }
12 }
```

Finally, it becomes also possible to enable or disable device testing directly on Jenkins agents as shown in listing 3.4. Hence, in the example below, the stage "Devices Tests" will only run if the parameter "DEVICE_TESTING" has not already been deselected before starting the pipeline.

Listing 3.4: Enable or Disable of Device Testing

```
1  stage('Device Tests') {
2      when {
3          expression { params.DEVICE_TESTING == true }
4      }
5      ...
6  }
```

3.6. Selection of Devices

Choosing the right test devices is an important part of getting meaningful test results. BrowserStack itself contains a large number of iOS devices and Android devices for testing purposes, which can be seen in the appendices B and A. On the one hand, using all devices at once would produce highly insightful test results. On the other hand, it would already fail for BrowserStack due to the fact that, at the time of this work, the selection of test devices that can be used simultaneously is limited.
Finally, test times would also be a major problem. Especially in the case of pipeline integration, we do not want to wait forever for the test results. So how do you find a perfect selection of test devices that can still represent an ideal compromise in terms of test time, meaningfulness of the test results, and finally with the BrowserStack limitation to five test devices that can be run simultaneously? The official BrowserStack documentation itself contains information on this. It lists the test devices most commonly used separately by region and unique visitors per month (BrowserStack, 2023e). The respective tables 5.4 and 5.5 contain exactly these test results for Pocket Code and Pocket Paint for Android, after the 'Additional devices tested to the device matrix' row. It is already apparent that not all test devices provide the same results. The other test devices in the respective tables were self-selected. They differ as much as possible in terms of vendor, screen size, and OS. As a result, you might conclude that BrowserStack not only allows testing on real devices, but also offers a larger selection of devices that can be used for this purpose.

3.6.1. Multi Device Testing

Vilkomir (Vilkomir et al., 2015) tested 15 different Android applications on 30 mobile devices and came to the conclusion that 24 devices had specific errors. He examined the effectiveness of different types of coverage and distinguished between OS, resolution, manufacturer, screen sizes, RAM, each choice (that is, the coverage of all device characteristics at the same time), and random, as can be seen in Table 3.1 (Vilkomir et al., 2015). For all types, except for each choice, the effectiveness was tested with four and with five devices. The interesting thing is that coverage of different types of OS delivered the best results. This comes as a bit of a surprise considering that some device-specific errors occur due to a different resolution or screen size, as the information is presented differently. The number of test devices can already have a great impact on the usefulness of test results. Additionally, this research provides clues regarding the number of devices tested. Vilkomir came to the conclusion that a random selection of five test devices already means 85% effectiveness and 100% could be achieved with 13 devices. The

OS coverage shows the best results with almost 90% with only five test devices.
When it comes to Pocket Code and Pocket Paint, this implies that the best outcomes should be obtained using a variety of test devices with various OS versions. For the iOS version of Pocket Code, fewer test devices were selected than for Android devices, as the devices differ less and the OS versions that should be supported are lower. Tables 5.1 and 5.3 show the selection of test devices for the iOS and Android versions of Pocket Code and Pocket Paint.

Table 3.1.: Effectiveness of the different types of coverage

Coverage	**Effectiveness with 4 Devices in %**	**Effectiveness with 5 Devices in %**
OS	85.6	89.2
Resolutions	77.5	80.6
Manufactures	78.3	84.2
Screen Sizes	81.7	85.6
RAM	81.1	85.8
Each-choice	was not tested	87.5
Random	81.4	85.0

3.7. Summary

This section focused mainly on describing the modifications that had to be made to the relevant Jenkins files to include BrowserStack. It also demonstrated what steps are needed to run Jenkins locally. This was done to be able to test changes to the Jenkins files. On the one hand, unrestricted testing and trying is possible due to the admin permissions. However, on the actual Jenkins server itself, you do not obstruct any resources that others might use.
In addition, it was demonstrated how and why particular test devices were chosen. To comply with the BrowserStack limitation, these several test devices run concurrently on BrowserStack with one thread each. The simultaneous execution of the various devices was also utilised as a default setting for the pipeline integration. Therefore, the defined device matrix is used unless a specific test device was chosen before executing the pipeline. Tables 5.2, 5.4 and 5.5 demonstrate that these results are also more substantial than when performed on a single device. This fact alone makes it tolerable that the execution time is much longer than it would be if only one device were executed, with several shards running simultaneously. Finally, the option of turning on or off the device matrix at any time is also available.

4. Implementation

This chapter explains the integration of BrowserStack into the existing Jenkins CI implementation. For a better understanding, code examples are also used and explained in more detail.

4.1. Differences of Catrobat Catty

The most significant differences for the Catrobat Catty project from the Android versions of Catrobat are displayed in section 4.2 below. A list of test devices that are compatible with iOS can be found in the appendix B.
The main distinction between the Android and iOS versions of the Catrobat project is that the pipeline development of the iOS version called Catty is more evident in the Fastlane file rather than in the Jenkins file itself. In the Jenkins file of Catty, only the Fastfile lanes are called, and the integration of BrowserStack has been modified to account for it.

4.2. Catrobat Catty Fastfile

The simplest method for automating beta deployments and releases for your iOS and Android applications is fastlane. With fastlane developing screenshots, managing code signing and distributing your programme is very easy to handle (Fastlane, 2023). Instead of executing these tasks in the Jenkins file as in the Android versions of Catrobat, only the corresponding lane is called in the Jenkins file of Catty. The command for this looks like this:

```
sh 'cd src && bundle exec fastlane ios ui_tests_app_automate'
```

The second part of the command above "bundle exec fastlane ios ui_tests_app_automate" runs the lane called "ios ui_tests_app_automate" of the Catrobat Catty Fastlane file.
The code snipped 4.1 contains the description of the lane called "ui_tests_app_automate" and this lane can be called from the corresponding Jenkins file. By deleting the derived data directory, the clear derived data action ensures that the build environment is clean. The gym action part, which is an Xcode's xcodebuild command-line tool, is used to build the iOS

version of Pocket Code application. It details the export method, configuration, output directory, output name, and xcodebuild formatter. The full UI-test launch settings require a debug ipa file, but the previously used build was a release. Therefore, the iOS version of the Pocket Code application also needs to be generated particularly for the BrowserStack integration in this instance. The xcodebuild test command is used to execute tests as part of the scan action. It details the options scheme, xcodebuild formatter, destination, disable slide to type, and build for testing parameters. Finally, the "CattyUITests-Runner.app" directory, which includes symbolic links, is compressed, and "pushd" is used to move to a specified directory. To go back to the previous directory that was added to the directory stack using the "pushd" command, the "popd" command is used. The path indicated by the variable "$runner_path" is used to save the created zip file in the previous working directory.

Listing 4.1: Catrobat Catty Fastfile

```
desc "Build UI tests for App Automate"
lane :ui_tests_app_automate do
clear_derived_data

$output_dir = "fastlane/app_automate/"
$runner_path = "../" + $output_dir + $build_name + "-
   UITests.zip"

gym(
  scheme: $catty_schemes["debug"],
  xcodebuild_formatter: "xcbeautify",
  configuration: "Debug",
  export_method: "development",
  output_directory: $output_dir,
  output_name: $build_name
)

scan(
  scheme: $catty_schemes["debug"],
  xcodebuild_formatter: "xcbeautify",
  destination: "generic/platform=iOS",
  disable_slide_to_type: false,
  build_for_testing: true
)

sh("pushd " + lane_context[SharedValues::
   SCAN_DERIVED_DATA_PATH] + "/Build/Products/Debug-
```

```
iphoneos && zip --symlinks -r $OLDPWD/" +
$runner_path + " CattyUITests-Runner.app && popd")
```

4.3. Jenkins Parameter Selection for BrowserStack

Figures 4.1 and 4.2 both show a part of the BrowserStack integration in Jenkins for Catty. By default, BrowserStack testing is disabled and can be enabled using the BROWSERSTACK_TESTING parameter. The BROWSERSTACK_SHARDS parameter specifies the number of threads to use and enables parallel testing on the same device or on different devices. Finally, the BROWSERSTACK_IOS_DEVICES parameter specifies the mobile device for the BrowserStack testing. The value IosDevices indicates that the device matrix is to be used and thus the tests run on several devices simultaneously. The settings also analogously apply to the Android Catrobat projects.

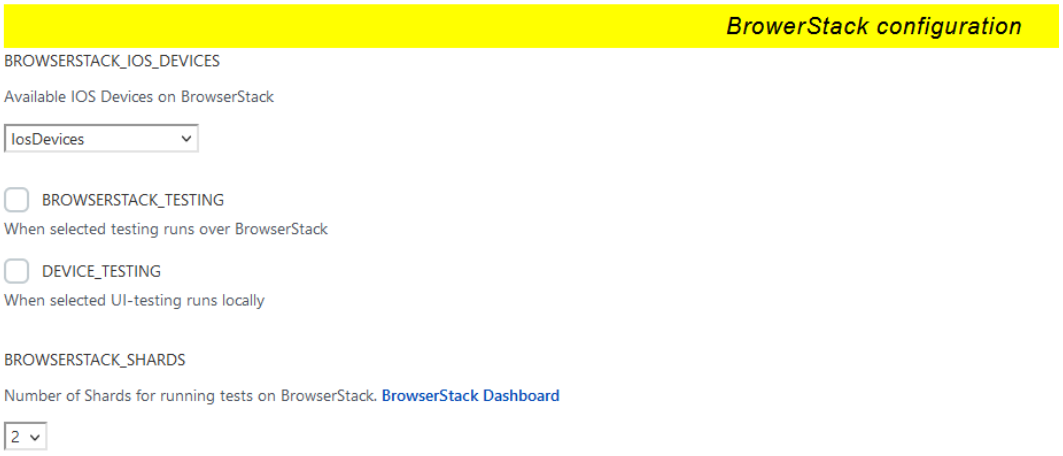

Figure 4.1.: Jenkins parameter configuration side for Catty

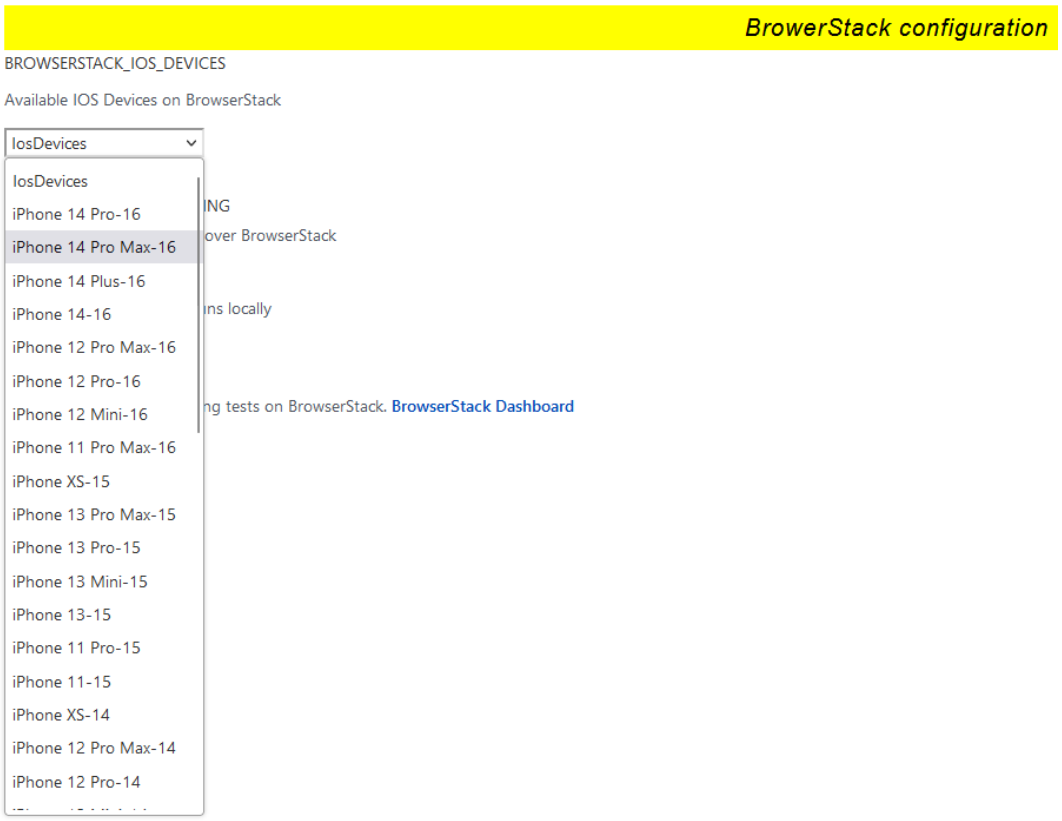

Figure 4.2.: List of available devices for Catty

4.4. Shared library for BrowserStack

Pipelines can be used for numerous projects within an organisation at the same time. This is applicable to other projects as well, not just Catrobat. For all projects, common patterns are likely to manifest themselves. Therefore, it is often beneficial to share pieces of pipelines between different projects to eliminate redundancy and try to keep the code DRY. For example, using ”shared libraries” that can be loaded into existing pipelines and defined in external source code repositories is one way to achieve this (Jenkins, 2023c). The starting point for the integration of BrowserStack are the methods called ”call” in 4.2 and 4.3. The two call methods, one for the iOS version of Pocket Code and one for the Android version and Pocket Paint, differ mainly in the parameters given. Depending on how many parameters are specified in the call in the Jenkins file, the corresponding call method is called. In principle, these two methods follow the same logic, but some things have to be handled specifically for a project, which is why it makes sense to split them up. For the iOS version of Pocket Code, the location of the test suite zip file and the iOS version of the Pocket Code application itself must be

specified. Only one path is required because they are both in the same folder, which is accomplished with the input "path to app". Unlike the iOS version of Pocket Code, the app and the test app are not in the same directory. Therefore, a separate path was specified for both, and also a third parameter was included that specifies the project itself for the Android "call" method in listing 4.3.
Both "call" methods use the method "browserStackPolling", and both use a method to upload the Pocket Code or Pocket Paint app and the test suites to BrowserStack in their own way. Also, after successfully uploading the app and the test suite, a message is returned and stored in a variable, as this is needed for polling the BrowserStack status.

Listing 4.2: Shared Library Entrance Method for Catty

```
1  /**
2   * Sending Catty app to BrowserStack for automated
      mobile app testing and fetch build result.
3   *
4   * @param path_to_app path of the apk/ipa app.
5   */
6
7  void call(final String path_to_app) {
8      final String credentials = '${
     BROWSERSTACK_USERNAME}:${BROWSERSTACK_ACCESS_KEY}'
9      final def message = browserStackCatty(credentials
     , path_to_app)
10
11     browserStackPolling(credentials, message)
12 }
```

Listing 4.3: Shared Library Entrance Method for Catroid and Paintroid

```
1  /**
2  * Sending the Catroid or Paintroid app to
     BrowserStack for automated mobile app testing and
     fetch build result.
3   *
4   * @param path_to_app path of the apk/ipa app.
5   * @param path_to_test_suits path of the apk ui tests
      suites file.
6   * @param indicate catroid or paintroid project
7   */
8  void call(final String path_to_app, final String
     path_to_test_suits, final String project) {
9      final String credentials = '${
```

```
    BROWSERSTACK_USERNAME}:${BROWSERSTACK_ACCESS_KEY}'
10      final def message = browserStackCatOrPaintroid(
    credentials, path_to_app, path_to_test_suits,
    project)
11
12      browserStackPolling(credentials, message)
13  }
```

4.5. Shards Selection Implementation for Catty

Unlike Catroid and Paintroid, BrowserStack does not support auto-sharding, i.e., the ability to run tests on one or more devices simultaneously. In the case of XCUI testing, the tests to run on a thread must be specified manually. However, the choice of tests to run on a thread was specified in the method "getShardsCatty" so that this could run automatically without encountering any significant problems. This approach in 4.4 aims to automatically distribute the tests among other threads, much as it is already automatically possible for espresso testing on BrowserStack. The method returns an empty string if "IosDevices" was chosen in the device selection step prior to running the pipeline, which implies that parallel testing has been turned off. This happens because numerous devices are already running on BrowserStack at the same time in this situation. However, each device requires at least one thread, and at the time of this study BrowserStack was able to only pick up five threads at once. Five more threads can be queued in addition to the five that can run concurrently. So, in this situation, it makes sense to limit the number of threads that can run on one device. In all other scenarios, it is still possible that the tests are split amongst different threads. To store the different test classes and the return value of the "getShardsCatty" method, respectively, two variables with the names "test_classes _shards" and "shards" are used. The "find" command is used to perform a recursive search for Swift files in the given directory, where the test files should be located. The output is then recorded and divided into a list of file paths. After that, only files that finish with "Tests.swift" or "Test.swift" are filtered out, and the ".swift" extension is removed. Moreover, only the file name is taken over and filtered out of the path. The "FormulaEditorSectionViewController" test class is an exception, since the name does not end with "Tests" or "Test" in the name, and must therefore be explicitly declared as such. By dividing the total number of test classes by the given parameter "BROWSERSTACK_SHARDS", the number of test classes per shard (n) is determined. An additional increase n by one if there is a remainder. A for loop iterates across the test classes and divides them into shards. For each shard, a sublist of test classes is extracted using

the ”subList” function, starting with current index i plus n, and each class name is surrounded by quotes using the collect method. Finally, the return value of the ”getShardsCatty” method is defined and stored in the variable shards. Here, the scenario where ”BROWSERSTACK SHARDS” equals 2 is explained. Naturally, the same reasoning also holds true for instances with values 3 to 5. The matching portion of the test cases, which was already calculated, is assigned to each unique shard in the example by creating a mapping here.

Listing 4.4: Shards Selection Implementation for Catty

```
1  final String getShardsCatty() {
2      if (params.BROWSERSTACK_IOS_DEVICES.equals('
       IosDevices')) {
3          return ""
4      } else {
5          def test_classes_shards = []
6          String shards = ""
7          def dir = "${workspace}/src/CattyUITests"
8          def files = sh(script: "find ${dir} -name
       \"*.swift\"", returnStdout: true).split()
9          def test_classes = files.findAll { it.
       endsWith("Tests.swift") || it.endsWith("Test.swift"
       ) }.collect { it.substring(it.lastIndexOf('/') + 1)
        }.collect {
10             it.replaceAll(/\.swift$/, "")
11         } + 'FormulaEditorSectionViewController'
12
13
14         int n = test_classes.size() / params.
       BROWSERSTACK_SHARDS.toInteger()
15         if (test_classes.size() % params.
       BROWSERSTACK_SHARDS.toInteger() > 0) {
16             n += 1
17         }
18
19         for (int i = 0; i < test_classes.size(); i +=
        n) {
20             test_classes_shards << test_classes.
       subList(i, Math.min(i + n, test_classes.size())).
       collect { "\"$it\"" }
21         }
22
```

```
23          if (params.BROWSERSTACK_SHARDS.toInteger() ==
        2) {
24              shards = """ \
25              "shards": {"numberOfShards": 2, "mapping"
        : [{"name": "Shard 1", "strategy": "only-testing",
        \
26              "values": ${test_classes_shards[0]}}, \
27              {"name": "Shard 2", "strategy": "only-
        testing", \
28              "values": ${test_classes_shards[1]}}]},
        \
29          """
30          ...
31          return shards
```

4.6. Check BrowserStack Status

Before testing on BrowserStack, the method "checkIfBrowserStackIsReady" in listing 4.5 checks to see if the BrowserStack service is accessible because the capacities on BrowserStack are limited and also because a connection to BrowserStack is required. If there are issues, the execution is stopped early on. The situation where "AnyDevice" was chosen before running the pipeline must be handled specifically in this instance as well. This is due to the need for extra resources on BrowserStack at the same time due to simultaneous testing on several devices. To determine if this is the case or not, a parameter is passed to the method "checkIfBrowserStackIsReady". Note that, in general, responses to status requests from BrowserStack are frequently provided in JSON format, which is also true in this instance. The command "jq" is frequently used to get the desired values in this message. This command-line tool is used to apply different filters to a stream of JSON data, and then the desired data can be pulled from the JSON stream in this manner. Additional pipes can be used to join various filters (ubuntuusers, 2023). The method "checkIfBrowserStackIsReady" uses two integer variables with the names "status" and "queued_sessions_max_allowed". The first variable specifies whether it is now possible to run BrowserStack or if all available threads have been utilised. The second variable is defined as 5, since a maximum of five threads can run in parallel at the same time on BrowserStack. An additional two string variables, "response" and "code" are declared. When queried, BrowserStack responds with a status in JSON format, which is stored in the first variable. The second variable specifies the connection status to BrowserStack itself. A curl command sends an HTTP GET request to the endpoint of the BrowserStack API,

using the supplied credentials for authentication. The ”response” variable stores the curl command's output, and the ”code” variable stores the HTTP response code. If it is present in the BrowserStack message, the value for the ”queued_sessions_max_allowed” variable is extracted from the JSON response using the jq command. An integer is created from the retrieved value. After that, a while loop begins, which runs until the API response code is not 200. This indicates a successful HTTP request, and BrowserStack is not ready at the moment. The queued_sessions variable is used inside the loop to hold the current number of queued sessions, which is taken from the response. In addition, this value is transformed into an integer. The value of ”queued_sessions” and the ”BROWSERSTACK_SHARDS” parameter value are added to the ”status” variable, which indicates the current status of BrowserStack. BrowserStack is ready in two cases. First, if ”isAnyDevice” is false and the value of ”status” is less than or equal to the value of ”queued_sessions_max_allowed”. In this case, the while loop end and the return statement of the method ”checkIfBrowserStackIsReady”, signal that BrowserStack is prepared. This is also true if the value of ”isAnyDevice” is true and if ”queued sessions” equals zero. Finally, the curl operation to retrieve the updated status from the BrowserStack API is repeated until none of the return conditions are satisfied.

Listing 4.5: Check BrowserStack Status

```
1  void checkIfBrowserStackIsReady(final String
       credentials, final boolean isAnyDevice) {
2      int status = 0
3      int queued_sessions_max_allowed = 5
4
5      String response, code
6      (response, code) = sh(script: """ \
7                            curl -w' \\n%{response_code}'
       \
8                            -u "$credentials" \
9                            https://api.browserstack.com/
       app-automate/plan.json \
10                           """, returnStdout: true).trim
       ().tokenize("\n")
11     queued_sessions_max_allowed = sh(script: "echo \'
       $response\' | jq '.queued_sessions_max_allowed'",
       returnStdout: true).trim().toInteger()
12     while (code.toInteger() == 200) {
13         final int queued_sessions = sh(script: "echo
       \'$response\' | jq '.queued_sessions'",
       returnStdout: true).trim().toInteger()
```

```
14          status = queued_sessions + params.
    BROWSERSTACK_SHARDS.toInteger()
15          if (status <= queued_sessions_max_allowed &&
    !isAnyDevice) {
16              return
17          } else if (queued_sessions == 0 &&
    isAnyDevice) {
18              return
19          }
20          (response, code) = sh(script: """ \
21                          curl -w' \\n%{response_code}'
     \
22                          -u "$credentials" \
23                          https://api.browserstack.com/
    app-automate/plan.json \
24                          """, returnStdout: true).trim
    ().tokenize("\n")
25      }
26  }
```

4.7. Device Selection

The method "getDevice" in listing 4.6 returns a list of devices that will be executed simultaneously if "IosDevices" or "AndroidDevices" have been chosen before executing the pipeline. Devices have been selected to work on a variety of different devices in terms of OS version, screen size, and, in the case of Android, different vendors. In all other cases, the method returns the device chosen before the pipeline is running.

Listing 4.6: Device Selection

```
1  final String getDevice(final String devices) {
2      if (devices.equals('IosDevices')) {
3          return '["iPhone 14 Pro-16", "iPhone 13-15",
    "iPhone XR-15", "iPhone 8-15", "iPhone SE 2022-15",
     "iPhone 12 Pro Max-14", "iPhone 11-14", "iPhone 12
     Mini-14"]'
4      } else if (devices.equals('AndroidDevices')) {
5          return ''' \
6              ["Google Pixel 7 Pro-13.0", "Samsung
    Galaxy S22 Ultra-12.0", "Google Pixel 5-11.0", \
7              "Google Pixel 4-10.0", "Samsung Galaxy
    S10-9.0", "Samsung Galaxy S9-8.0", \
```

```
8               "Google Pixel 2-8.0", "Samsung Galaxy S8
    -7.0", "Google Pixel-7.1", "Samsung Galaxy S7-6.0"]
     \
9           '''
10      } else {
11          return "[\"${devices}\"]"
12      }
13  }
```

The method "browserStackCatOrPaintroid" in appendix C uploads the apks and runs them on BrowserStack for the Android version of Pocket Code and Pocket Paint. Although the iOS version of Pocket Code uses an alternative method. The core reasoning is the same as the one presented here and takes the unique characteristics of the testing with XCUITest into consideration. The instances of providing the corresponding BrowserStack API's for Espresso or XCUITest show the biggest difference. The central purpose of the method is to send the Android version of Pocket Code or Pocket Paint application and the test suites to Browserstack. It utilises a curl command to submit the two apk files to the appropriate BrowserStack API and store the response in different variables. Once again, the "app_url" and "app_id" are filtered out of the response message using the command "jq". The "app_url" in particular is crucial because it is used to identify the uploaded app on BrowserStack and to launch a test run on this particular app. The "jq" command is used in the same manner as described above for test suites. Each shard will run on each of the devices, which is returned by the method "getDevice" in listing 4.6, if "all" is chosen. The value "all" is also the default setting for BrowserStack. The other option "any" means that each shard will operate on a single device that is selected at random. The single-runner invocation is also set to true to speed up the test running. By default, the test runner runs every Espresso test case separately. Due to the fact that each test runs in its own instrumentation instance, there are less shared states between various test cases. The benefit of using this default procedure is that the tests run more consistently overall and therefore should result in fewer errors. However, since each test case starts a new process, the overall build execution time will increase as a result (BrowserStack, 2023b). Lastly, the curl command instructs BrowserStack to run the previously uploaded test suite on the previously uploaded app. Furthermore, all the information required in the following stages for the BrowserStack integration is contained in the return value of the method.

4.8. Polling of BrowserStack Status for a specific Test Run

The implementation of polling for BrowserStack presented one of the biggest challenges. BrowserStack only provides a Rest API where the current state can be accessed for Espresso and XCUITest testing. The important thing to remember is that the client side must check this status and that the pipeline's final result also depends on it. The "browserStackPolling" method in appendix D receives two parameters. The information required to identify a specific test run on BrowserStack is contained in the second parameter, which is the return value of the method "browserStackCatOrPaintroid" in the appendix C. The basic idea behind the polling method is really simple. Check the status of a certain test run on BrowserStack every 60 seconds. This is done via a while loop, which continues running as long as the connection to BrowserStack is stable and until the status of BrowserStack is no longer "running" or "queued". If the status is "passed," "failed," "error," "time out," or "skipped," the while loop is ended and the "getBrowserstackEndMessage" method is called at the same time. Among other things, the "getBrowserstackEndMessage" method retrieves the test results if they are available. Furthermore, it brings the BrowserStack operation in general to a clean end. The pipeline will fail if the status is unknown. Therefore, it cannot be classified under one of the aforementioned categories, as this method is also called in this case.
If a problem should arise, it is noted in the catch block, and the uploaded application and the test suit are removed from the BrowserStack server. This deletion is crucial because it might be required in order to upload new apps and test suites in the future. This is due to the fact that the uploaded apps and test suites must be saved to the BrowserStack server, and at the time of this writing, BrowserStack only allows the upload of eight separate apps for Espresso and XCUITest App Automate within a 30-day period. Naturally, this prevents BrowserStack from being properly integrated into the pipeline, but BrowserStack can still be used to run tests on real devices periodically. Moreover, for each pipeline run, there is an option in the pipeline to turn on or off BrowserStack testing. In addition, it is also useful for release tests. The limitation to releases, or generally only at longer intervals, is due to the limitation of BrowserStack itself. The 30-day limitation makes it impossible to use it for pull requests, or for heavy testing in a short period of time. This limitation applies only to Espresso and XCUITest, not for Appium. The reason for this is that BrowserStack provides its own CI solution for Appium, but not for Espresso and XCUITest. However, this work has given the reasons why it makes more sense to use Espresso and XCUITest instead of Appium.

4.9. Fetching Test Reports

It makes sense to submit a test report to Jenkins in the event that all tests were successful or some went wrong. For Android versions, BrowserStack provides a Rest API that allows one to obtain these test reports in XML format. With the use of the method in appendix E, Jenkins can access the test reports that BrowserStack has collected for a particular test run. After the test results are obtained, the uploaded app and test suit should also be deleted. This will free up space on the BrowserStack server.

4.10. Finish BrowserStack Execution

In addition to ending BrowserStack execution, the method 'getBrowserstack-EndMessage' also causes the pipeline to fail if any errors are encountered, as shown in listing 4.7.
As explained above, the method "getXmlFile" in appendix E retrieves and makes the test reports on Jenkins available if BrowserStack's state following the execution of the test is "failed" or "passed." Similarly to how the Android version of Pocket Code and Pocket Paint use JUnit, the iOS version of Pocket Code receives the test report in the form of a test bundle that can be loaded into Xcode and has more specific information about the individual tests.

Listing 4.7: Finish BrowserStack Execution

```
1  void getBrowserstackEndMessage(final String
       credentials, final String status, final def message
       ) {
2      echo "Status of browserstack exectuion is $status
       "
3      println "https://app-automate.browserstack.com/
       dashboard/v2/builds/$message.build_id"
4      if (status.equals('passed') || status.equals('
       failed')) {
5          getReportFile(credentials, message)
6      }
7      if (!status.equals('passed')) {
8          catchError(buildResult: 'FAILURE',
       stageResult: 'FAILURE', message: 'Some tests failed
        on Browserstack') {
9              sh "exit 1"
10         }
11     }
12 }
```

4.11. Summary

In this chapter, the simplicity of setting up a local Jenkins server is demonstrated. Modifications to the pipeline could be tested in a simple way. On the one hand, it is feasible to lighten the load on the actual test system, while on the other hand, full access modifications are available. Having administrator access can be very beneficial. For example, if you want to test new plugins. It was also shown how to include BrowserStack in the Catrobat Jenkins implementation. For this, the following changes were made to all Catrobat Jenkins files:
First, the appropriate new parameters for BrowserStack were added to the Jenkins file. Second, in addition to building the Pocket Code and Pocket Paint applications, building the test suites was also added. Third, both the Pocket Code and Pocket Paint applications and the test suite were sent to BrowserStack via a new shared library with the name ”browserStack”. All Jenkins files can access this common library, which also controls all of the BrowserStack integration logic. The test results were then fetched from BrowserStack and made available via Jenkins. Any problems during this process would result in a failure of the BrowserStack step.

5. Discussion

This chapter explains the test results for Pocket Code and Pocket Paint on BrowserStack. Finally, consideration is given to the research questions.

5.1. BrowserStack Test Results for the iOS Version of Pocket Code

The test results and test devices used in BrowserStack are presented for the iOS Version of Pocket Code.

5.1.1. Device specifications of iOS devices

Table 5.1 shows the specifications of the test devices used for the iOS version of Pocket Code on BrowserStack (Specifications, 2023).

Table 5.1.: Device specifications of iOS devices for the iOS version of Pocket Code

Device	OS Version	Screen Size	Resolution	Density (ppi)	Year
iPhone 14 Pro	16	6.1″	1179x2556	460ppi	2022
iPhone SE 2022	15	4.7″	750x1334	326ppi	2022
iPhone 13	15	6.1″	1170x2532	460ppi	2021
iPhone XR	15	6.1″	828x1792	326ppi	2018
iPhone 8	15	4.7″	750x1334	326ppi	2017
iPhone 12 Pro Max	14	6.7″	1284x2778	458ppi	2020
iPhone 11	14	6.1″	828x1792	326ppi	2019
iPhone 12 Mini	14	5.4″	1080x2340	476ppi	2020

5.1.2. Test Results

The test values in table 5.2 are from the develop branch of Catrobat/Catty on 14.04.2023 and three runs were performed for each device.

Table 5.2.: Device running time and test coverage of the iOS version of Pocket Code

Device	Average Execution Time	Coverage (%)	Tests
iPhone 11-14	20m	98.20%	109/111
iPhone 12 Mini-14	18m	98.20%	109/111
iPhone 12 Pro Max-14	23m	98.20%	109/111
iPhone 8-15	22m	100%	111/111
iPhone 13-15	23m	95,49%	106/111
iPhone XR-15	20m	95,49%	106/111
iPhone SE 2022-15	23m	95,49%	106/111
iPhone 14 Pro-16	24m	95,49%	106/111

5.1.3. Scatter plot of iOS devices for the iOS version of Pocket Code

Figure 5.1 shows a graphical overview of the test results for the iOS version of Pocket Code.

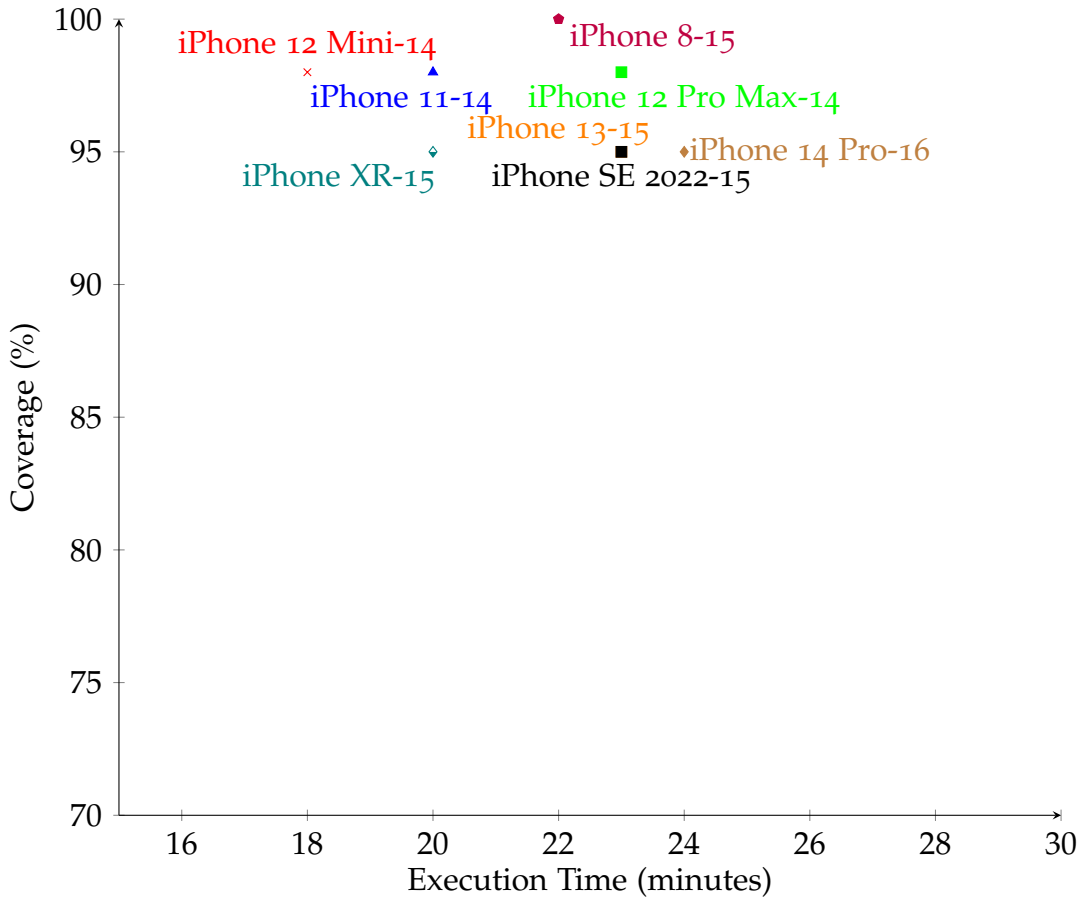

Figure 5.1.: Scatter plot of devices based on execution time and test coverage for the iOS version of Pocket Code

5.2. BrowserStack Test Results for the Android Version of Pocket Code

The test results and test devices used in BrowserStack are presented for the Android Version of Pocket Code.

5.2.1. Device specifications of Android devices

Table 5.3 shows the specifications of the test devices used for the Android version of Pocket Code and Pocket Paint on BrowserStack (Gizmochina, 2023; Specifications, 2023).

Table 5.3.: Device specifications of Android devices for the Android version of Pocket Code and Pocket Paint

Device	OS Version	Screen Size	Resolution	Density (ppi)	Year
Samsung Galaxy S7	6.0	5.1″	1440x2560	577ppi	2016
Google Pixel	7.1	5.0″	1080x1920	441ppi	2016
Samsung Galaxy S8	7.0	5.8″	1440x2960	570ppi	2017
Google Pixel 2	8.0	5.0″	1080x1920	441ppi	2017
Samsung Galaxy S9	8.0	5.8″	1440x2960	570ppi	2018
Google Pixel 3	9.0	5.5″	1080x2160	443ppi	2018
Samsung Galaxy S10	9.0	6.1″	1440x3040	550ppi	2019
Google Pixel 4	10.0	5.7″	1080x2280	444ppi	2019
Samsung Galaxy S20	10.0	6.2″	1440x3200	563ppi	2020
Google Pixel 5	11.0	6.0″	1080x2340	432ppi	2020
Samsung Galaxy S22 Ultra	12.0	6.8″	1440x3200	515ppi	2022
Google Pixel 6 Pro	12.0	6.7″	1440x3120	512ppi	2021
Samsung Galaxy S23 Ultra	13.0	6.8″	1440x3088	516ppi	2023
Google Pixel 7 Pro	13.0	6.7″	1440x3120	512ppi	2022
Additional devices tested to the device matrix					
Xiaomi Redmi Note 8	9.0	6.3″	1080x2340	409ppi	2019
Xiaomi Redmi Note 9	10.0	6.67″	1080x2340	395ppi	2020
Motorola Moto G7 Play	9.0	5.7″	720x 1512	294ppi	2019
Motorola Moto G71 5G	11.0	6.4″	1080x2400	411ppi	2021
Vivo Y50	10.0	6.53″	1080x2400	403ppi	2020
Vivo Y21	11.0	6.51″	720x1600	270ppi	2021
Huawei P30	9.0	6.15″	1080x2312	415ppi	2019
OnePlus 7T	10.0	6.55″	1080x2400	402ppi	2019
OnePlus 9	11.0	6.55″	1080x2400	402ppi	2021

5.2.2. Test Results

The test values in table 5.4 are from the Catrobat Catroid develop branch on 28.03.2023 and three runs were performed for each device.

Table 5.4.: Device running time and test coverage of the Android version of Pocket Code

Device	Average Execution Time	Coverage (%)	Tests
Samsung Galaxy S7-6.0	5m	95,51%	85/89
Google Pixel-7.1	14m	98,52%	1802/1829
Samsung Galaxy S8-7.0	21m	99,17%	3001/3026
Google Pixel 2-8.0	18m	99,22%	4301/4335
Samsung Galaxy S9-8.0	20m	96,96%	2140/2207
Google Pixel 3-9.0	18m	98,98%	4270/4314
Samsung Galaxy S10-9.0	21m	98,85%	4287/4337
Google Pixel 4-10.0	29m	98,61%	4253/4313
Samsung Galaxy S20-10.0	29m	98.72%	4254/4319
Google Pixel 5-11.0	30m	98,50%	2162/2196
Samsung Galaxy S22 Ultra-12.0	29m	98,53%	4209/4272
Google Pixel 6 Pro-12.0	28m	98,41%	4204/4272
Samsung Galaxy S23 Ultra-13.0	28m	98,60%	4222/4282
Google Pixel 7 Pro-13.0	29m	98,38%	4195/4264
Additional devices tested to the device matrix			
Xiaomi Redmi Note 8-9.0	22m	98,57%	3028/3072
Xiaomi Redmi Note 9-10.0	36m	98.11%	4210/4291
Motorola Moto G7 Play-9.0	7m	98.85%	946/957
Motorola Moto G71 5G-11.0	31m	93.06%	2141/2190
Vivo Y50-10.0	47m	94.90%	3228/3403
Vivo Y21-11.0	30m	98.19%	4165/4236
Huawei P30-9.0	20m	98.31%	4124/4192
OnePlus 7T-10.0	27m	97.45%	2133/2189
OnePlus 9-11.0	29m	98.06%	4219/4293

5.2.3. Scatter plot of Android devices for the Android version of Pocket Code

Figure 5.2 shows a graphical overview of the test results for the Android version of Pocket Code.

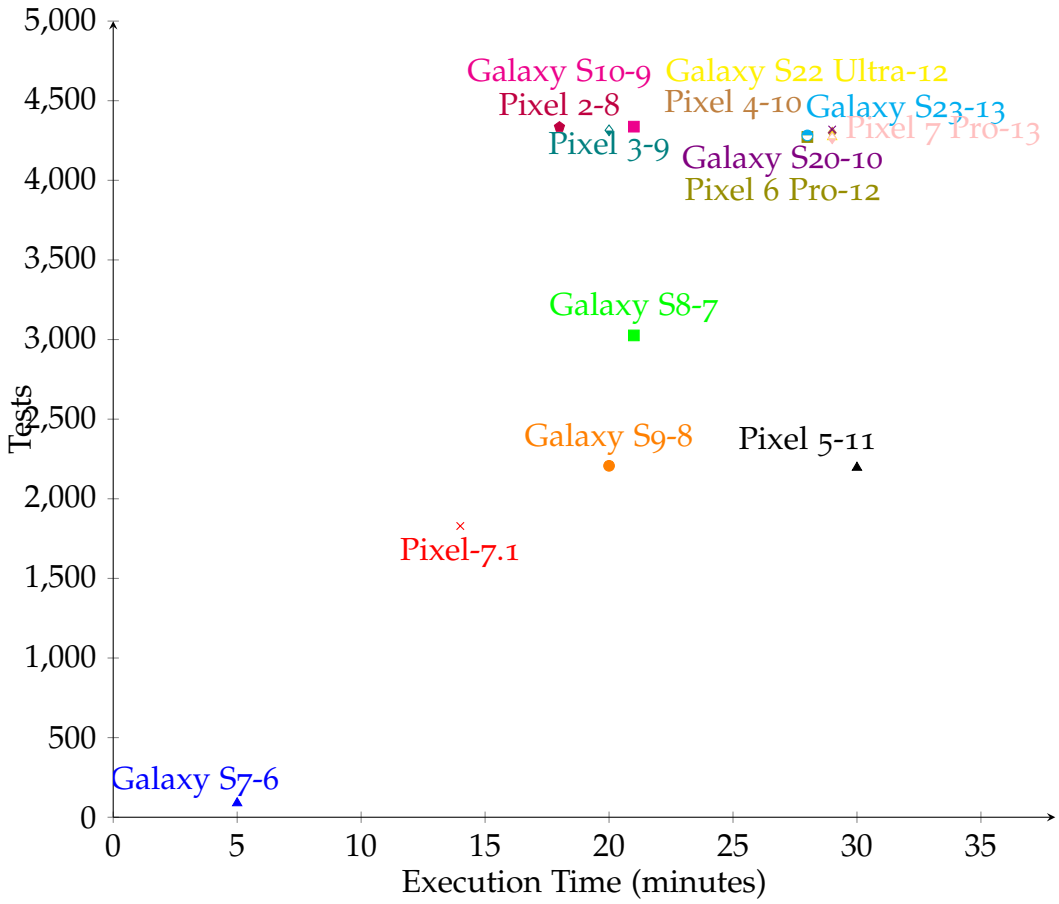

Figure 5.2.: Scatter plot of devices based on execution time and test coverage for the Android version of Pocket Code

5.2.4. Scatter plot of additional Android devices for the Android version of Pocket Code

Figure 5.3 shows a graphical overview of the test results of additional Android devices for the Android version of Pocket Code.

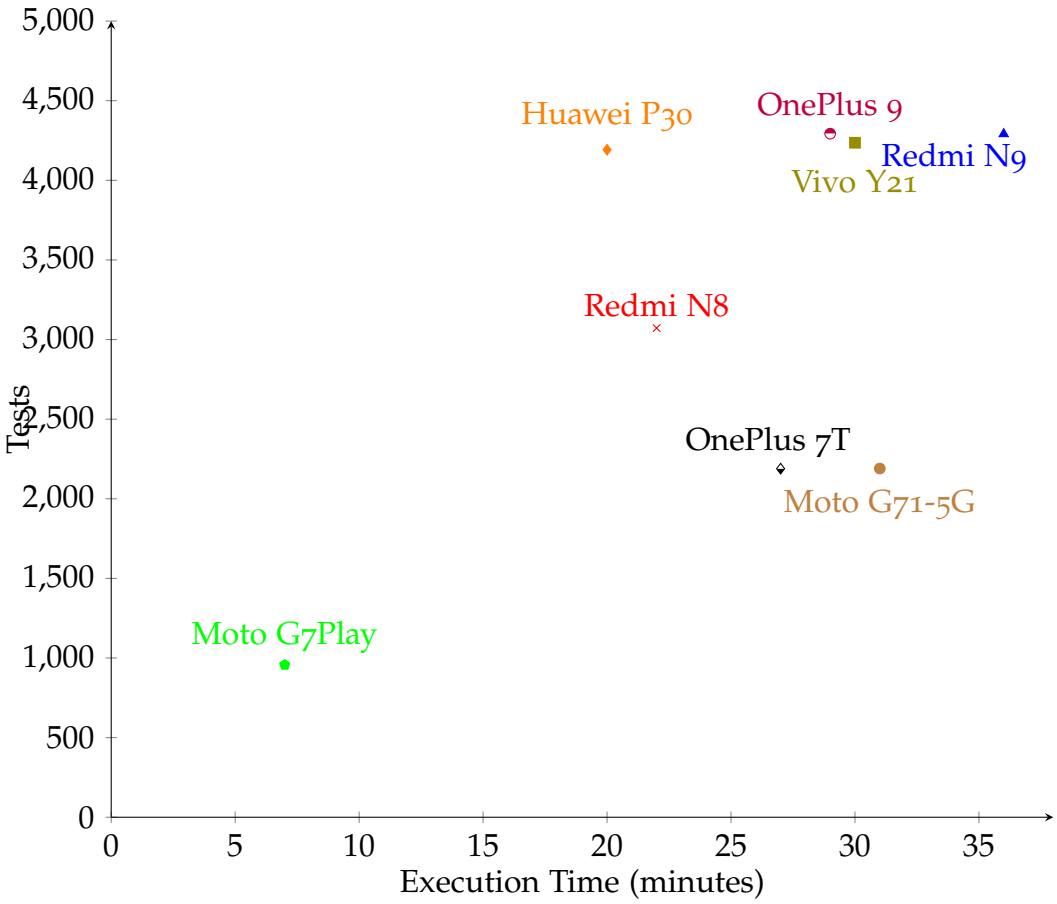

Figure 5.3.: Scatter plot of additional devices based on execution time and test coverage for the Android version of Pocket Code

5.3. BrowserStack Test Results for Pocket Paint

The test results and test devices used in BrowserStack are presented for Pocket Paint.

5.3.1. Test Results

The test values in table 5.5 are from the Catrobat/Paintroid development branch on 07.04.2023 and three runs were performed for each device.

Table 5.5.: Device running time and test coverage of Pocket Paint

Device	Average Execution Time	Coverage (%)	Tests
Samsung Galaxy S7-6.0	14m	91.47%	118/129
Google Pixel-7.1	Time out	74.90%	194/259
Samsung Galaxy S8-7.0	7m	88.71%	55/62
Google Pixel 2-8.0	12m	84.16%	472/561
Samsung Galaxy S9-8.0	15m	86.21%	483/561
Google Pixel 3-9.0	15m	86.27%	484/561
Samsung Galaxy S10-9.0	14m	85.39%	479/561
Google Pixel 4-10.0	11m	82.87%	465/561
Samsung Galaxy S20-10.0	12m	85.04%	477/561
Google Pixel 5-11.0	15m	70.04%	393/561
Samsung Galaxy S22 Ultra-12.0	Time out	68.50%	384/561
Google Pixel 6 Pro-12.0	Time out	29.23%	129/442
Samsung Galaxy S23 Ultra-13.0	19m	80.14%	226/282
Google Pixel 7 Pro-13.0	7m	45.04%	127/282
Additional devices tested to the device matrix			
Xiaomi Redmi Note 8-9.0	17m	87.34%	490/561
Xiaomi Redmi Note 9-10.0	19m	98.03%	549/561
Motorola Moto G7 Play-9.0	18m	88.63%	497/561
Motorola Moto G71 5G-11.0	16m	86.53%	485/561
Vivo Y50-10.0	19m	64.56%	362/561
Vivo Y21-11.0	23m	82.44%	462/561
Huawei P30-9.0	13m	86.68%	486/561
OnePlus 7T-10.0	12m	77.07%	432/561
OnePlus 9-11.0	15m	80.39%	451/561

5.3.2. Scatter plot of Android devices for Pocket Paint

Figure 5.4 shows a graphical overview of the test results for Pocket Paint.

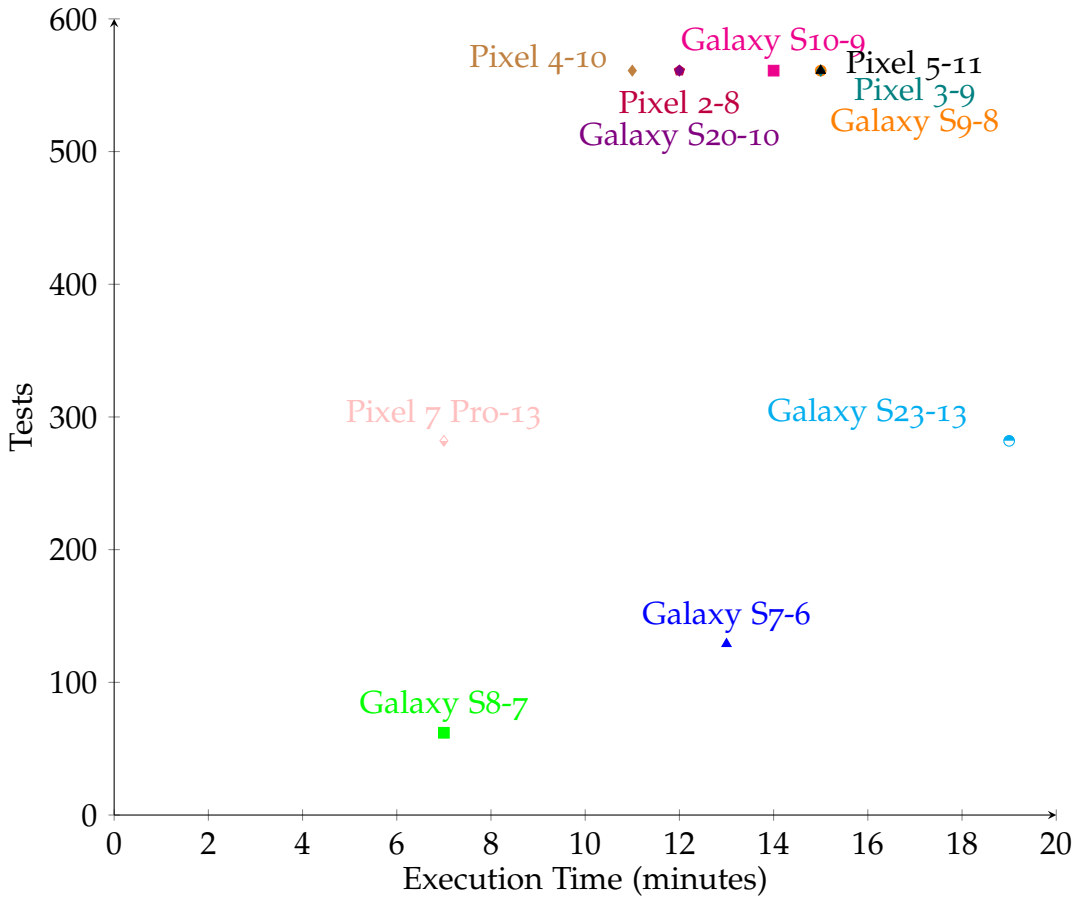

Figure 5.4.: Scatter plot of devices based on execution time and test coverage for Pocket Paint

5.3.3. Scatter plot of additional Android devices for Pocket Paint

Figure 5.5 shows a graphical overview of the test results of additional Android devices for Pocket Paint.

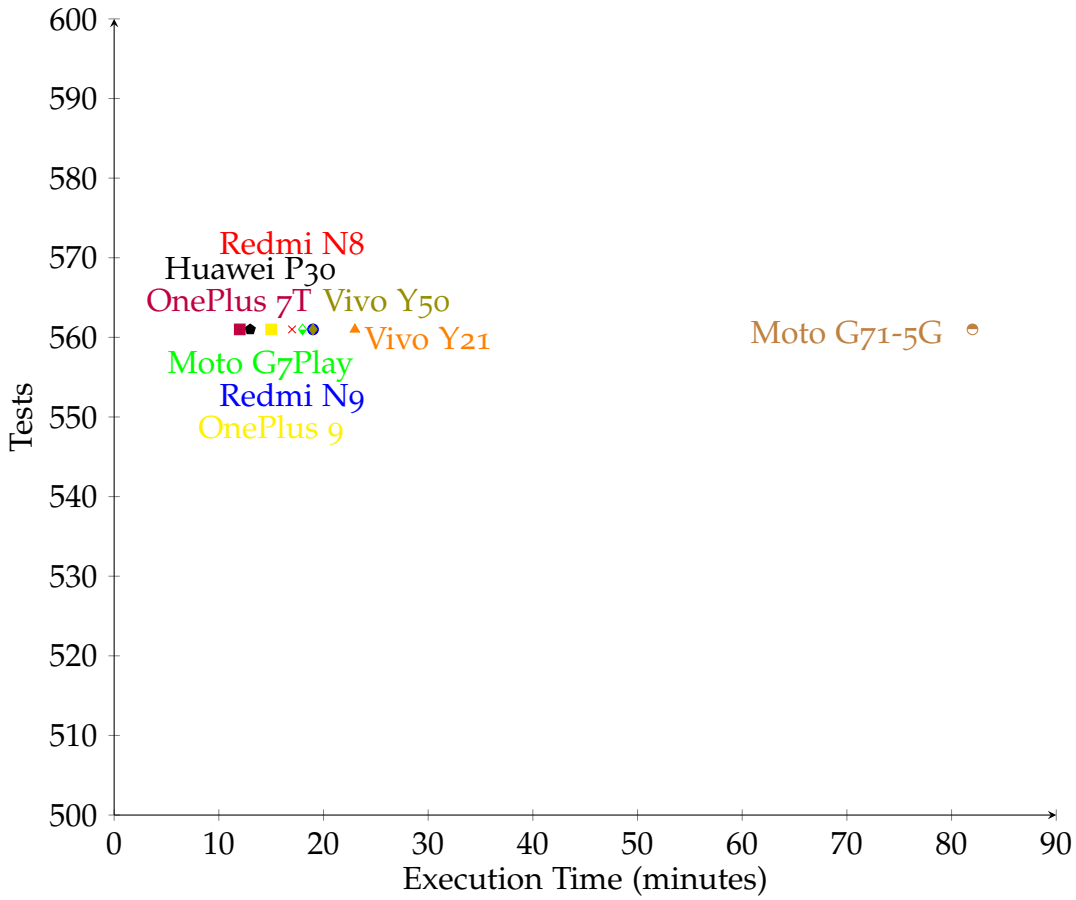

Figure 5.5.: Scatter plot of additional devices based on execution time and test coverage for Pocket Paint

5.3.4. Test Coverage on devices with time out for Pocket Paint

Figure 5.6 shows a graphical overview of the test results where the maximum execution time is exceeded for Pocket Paint.

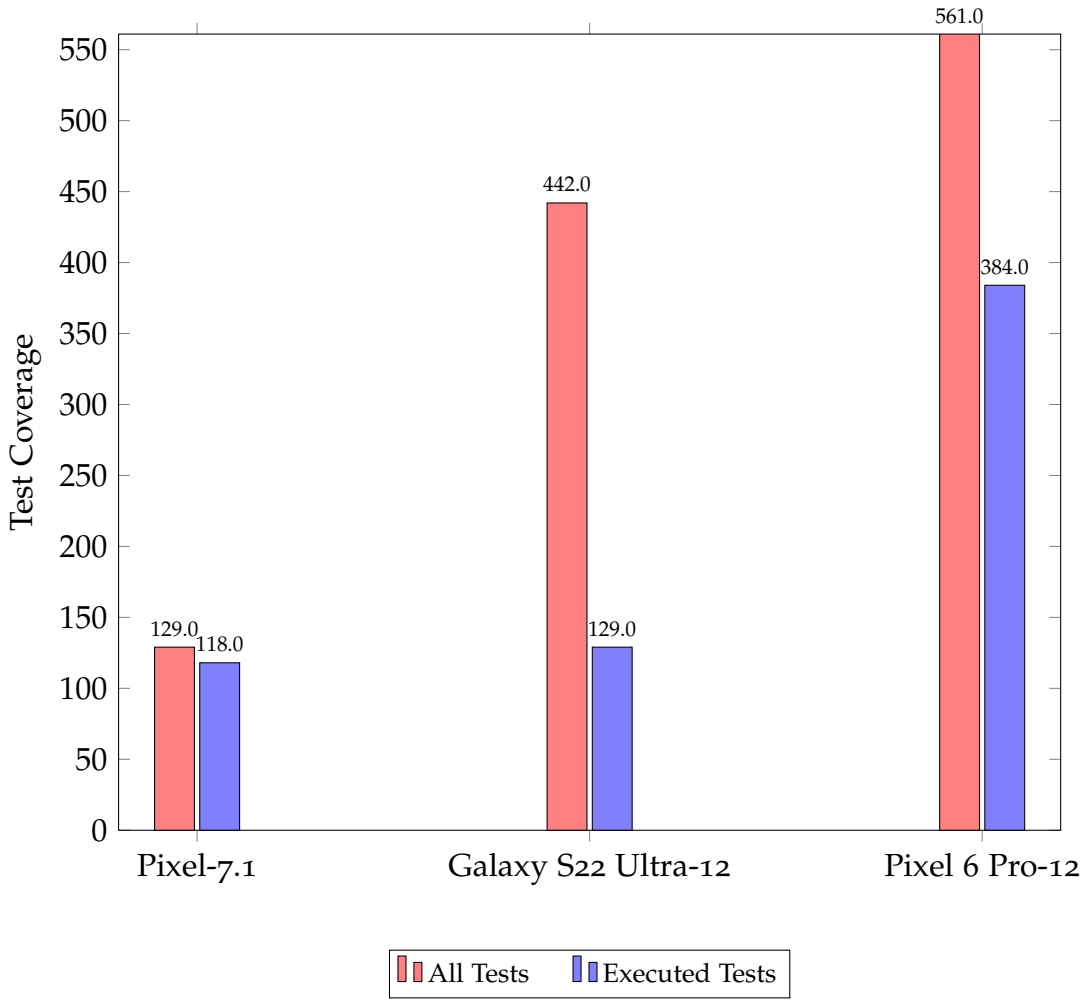

Figure 5.6.: Test Coverage on devices with time out for Pocket Paint

5.4. Device Specific Faults

Figures 5.7, 5.8, 5.9, 5.10, 5.11 show the Android application of Pocket Code (Catrobat., 2023b) and also explain device-specific errors. For this purpose, a Samsung Galaxy S8 with Android version 7 was used.

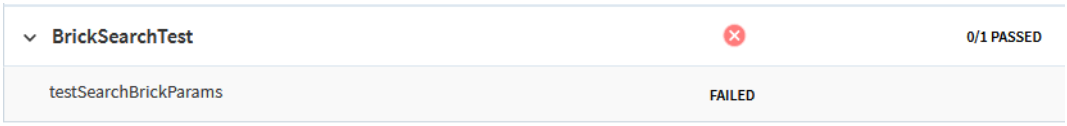

Figure 5.7.: BrowserStack results for BrickSearchTest for Galaxy S8-7.0

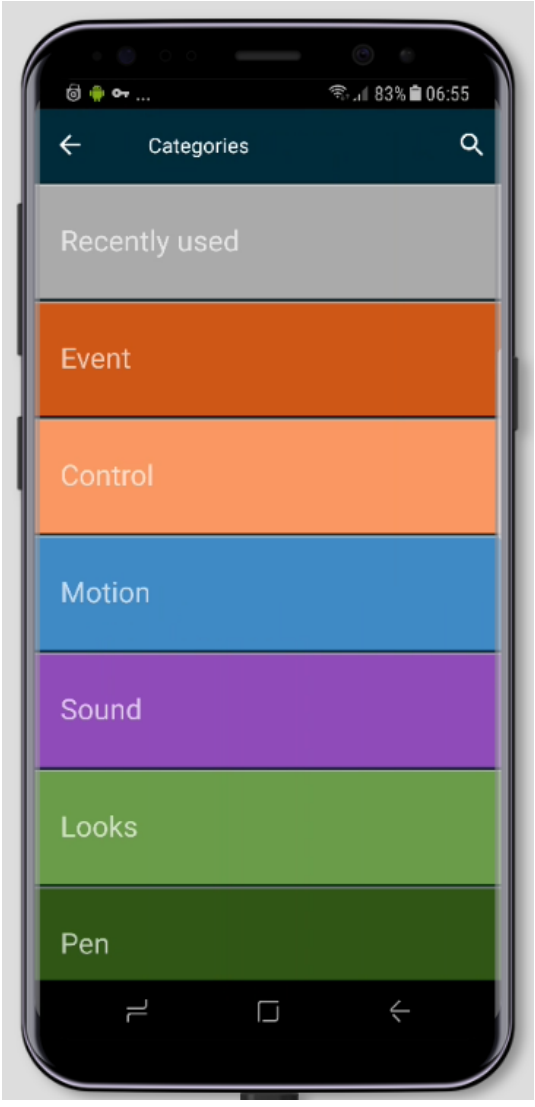

Figure 5.8.: Home screen for different bricks categories

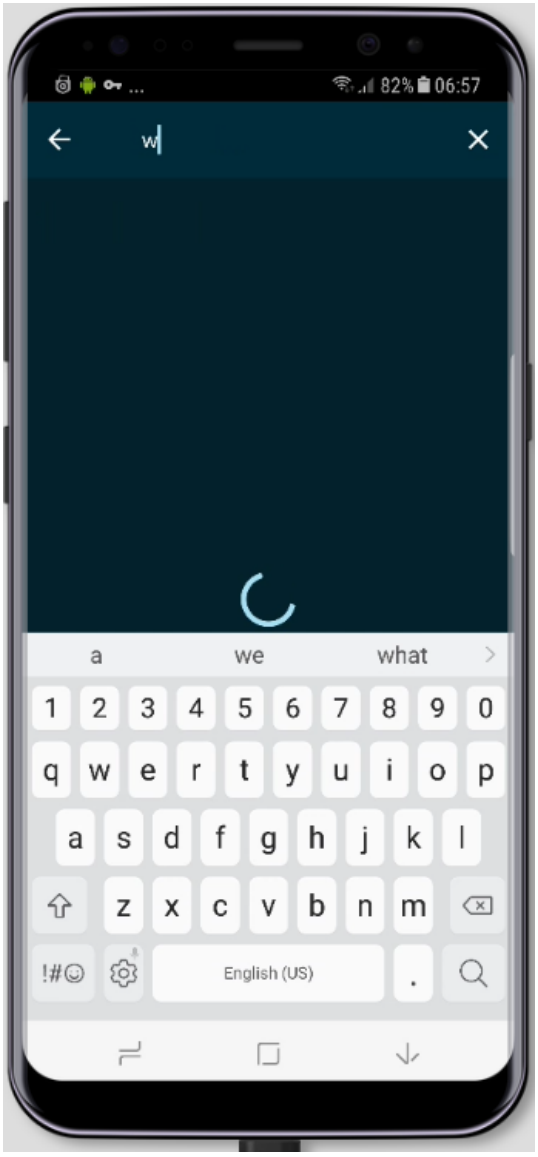

Figure 5.9.: Typing input to the search bar on Galaxy S8-7.0

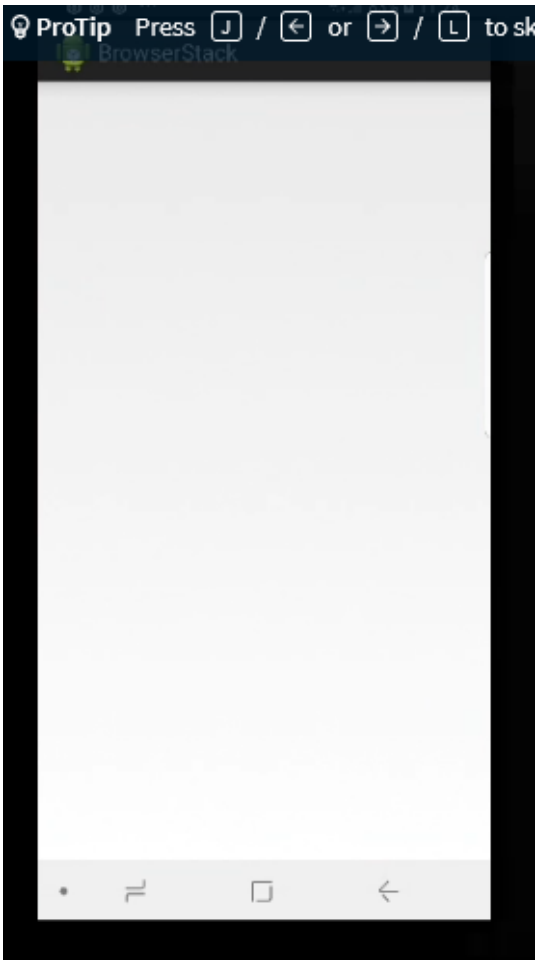

Figure 5.10.: Search input leads to a white screen and the app incorrectly crashes on Galaxy S8-7.0

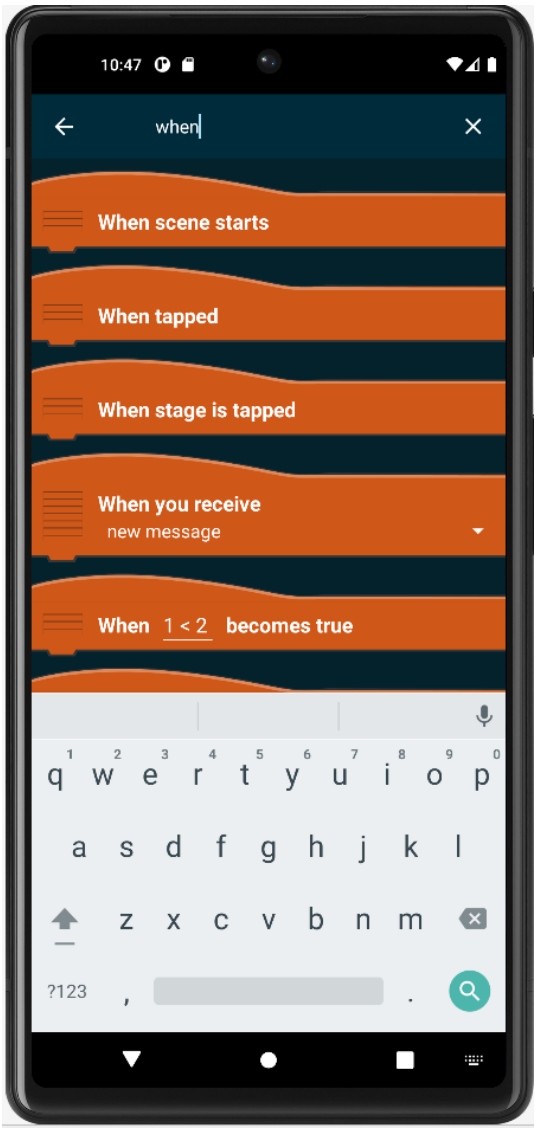

Figure 5.11.: A Pixel 6 with Android version 12 would show the right output

5.5. Summary

The collected data shows that Espresso's tests for the Android version of Pocket Code and Pocket Paint performed worse than XCUITest tests for the iOS version of Pocket Code on BrowserStack. Furthermore, it must be noted that, since the tests vary for each project, a direct comparison is not totally feasible. The findings of the individual tests do, however, yield some interesting numbers that at the very least make a comparison worthwhile.
The Catty tests all scored very well with a worst-case coverage score of 95.49% as listed in table 5.2. While Paintroid had a worst score of 29.23% as listed in table 5.5, Catroid had a score of 93.06% as listed in table 5.4, which is just marginally worse than Catty's. The test results for the Samsung Galaxy S7 running Android version 6 for the Android version of Pocket Code in table 5.4 and the Samsung Galaxy S8 running Android version 7 for Pocket Paint in table 5.5 are quite impressive. The total number of tests carried out in both instances is much less than in the corresponding comparison with other test devices. Unfortunately, BrowserStack does not provide any precise explanations of why this is the case. However, it can be assumed with a high degree of probability that in these circumstances many test cases are simply skipped by BrowserStack. This is known as an "idle timeout" and indicates that the test case has been inactive for a considerable amount of time, meaning that the application has not received a new command from the test. Performing this prevents a timeout of the entire test run (BrowserStack, 2023b). However, the reasons why the test execution on these two test devices is noticeably inferior to that on other test devices should be determined in more detail in the future. Nevertheless, there have been instances of test runs on devices that took so long that they ran into a time out, particularly with Catroid and also Paintroid. Figure 5.6 contains these devices with timeout for Paintroid. Additionally, based on the following presumptions, it was predicted that the Catty tests on iOS devices will perform better than the Espresso tests on Android devices using BrowserStack.
One reason for this is that the different iPhone devices are all very similar. Furthermore, there is the fact that they are all developed by one manufacturer, namely Apple. Additionally, there is less variation in OS compared to Android-based devices. For Android devices, there are many different manufacturers, so the hardware can differ quite a bit. Moreover, since many manufacturers make modifications to the standard Android version to varying degrees, the appearance of the operating system can differ greatly. Naturally, this discrepancy might also have a major impact. A problem for the Pocket Code and Pocket Paint applications could be that tests have so far only been performed on virtual devices or only occasionally on real devices. This could lead to some problems, since bugs could easily be overlooked or

could not be detected at all. Some versions of Android underwent more tests overall than others in some cases. Furthermore, several of these variations are also present in devices from different vendors running the same version of Android. This once again demonstrates how drastically different Android versions can be amongst vendors.
The run times for the Android version of Pocket Code, which last approximately 30 minutes, are accordingly similar to those of the emulators and are therefore unaffected. Naturally, this is only true when parallel testing is enabled and each device is tested with two shards at once. It is interesting to look at a few issues with Android devices in more detail. As already shown in this BrowserStack section, some issues affect only particular devices while working on others. This has been confirmed using BrowserStack Live, which could be used to demonstrate that the device actually crashes when manually replicating an automation test. Therefore, it will be necessary to carefully review these errors as time progresses and consider time outs for some devices. For Pocket Paint, it will be important to increase the coverage in general, since the tests run worse on BrowserStack than is the case for Pocket Code. Furthermore, table 5.5 also shows time outs, such as for the devices Google Pixel with Android 7.1 or Google Pixel 6 Pro with Android 12.
Finally, BrowserStack provides numerous different real test devices. Switching between individual test devices is considerably simpler or easier with BrowserStack than it would be with virtual devices. Of course, this only applies if BrowserStack is already set up. Therefore, BrowserStack testing can give a solid summary of the parts of a project where there are still significant issues and where to focus.

6. Lessons Learned

During this process, much attention was paid to BrowserStack and the best way to implement it for the Catrobat projects. At the time of this work, BrowserStack had not developed yet a Jenkins CI solution for the Espresso and XCUITest testing frameworks. Therefore, I made an effort to create one on my own. Moreover, I dealt extensively with Jenkins integration and Groovy programming in addition to BrowserStack itself. Due to the unique features of both Espresso and XCUITest, a foundational understanding of these two frameworks was established, including how they differ from each other and from Appium. Naturally, a foundational understanding of Jenkins and CI was created. I learnt more about Fastlane for the Catrobat Catty project. Both the app development and the creation of the test suites were carried out using it. In addition, an understanding was created in Gradle for the Catrobat Catroid project. In contrast to Paintroid, it was not possible to immediately create the test.apk without errors. In general, the discrepancies that exist when testing the Pocket Code and Pocket Paint applications were examined in more detail. Furthermore, the variations between virtual and real devices were explained. Particularly for testing on real devices, it was demonstrated that a project app's functionality cannot be determined solely by its operation on a particular device alone. Therefore, the focus was also on the selection of the right test devices.
In summary, the main points are listed below.

- Fundamental knowledge about CI and especially how to implement this in Jenkins.
- Understanding of Jenkins and developing your own local Jenkins server.
- Understanding about Jenkins plugins.
- Learning of the Groovy programming language.
- Learning Fastlane for Catrobat Catty and Gradle, especially for Catrobat Catroid.
- Differences between virtual and real devices.
- BrowserStack Automate and Live.
- What Software Testing means and different types of testing.
- Fundamental knowledge of the Espresso, XCUITest, and Appium testing frameworks.
- Create a custom browserStack integration solution in Jenkins for the XCUITest and Espresso testing frameworks.

7. Conclusion and Future Work

Research Question 1 How do virtual and real devices compare in terms of testing time, number of tests executed, and percent of tests passed for the Pocket Code and Pocket Paint applications?
There are some differences between individual Catrobat projects. On the one hand, between the Android version and the iOS version of Pocket Code. Moreover, within the Android applications, Pocket Code and Pocket Paint. These differences are shown in the BrowserStack section and are already briefly explained. The tests were conducted using BrowserStack with two shards per device. Therefore, it must be emphasised again that some tests were conducted simultaneously on a single device. Furthermore, the individual figures and tables in the BrowserStack section represent the average value for each device based on three test runs for each device. When making these comparisons, note that the test suites are different in each case.

- Time: For the iOS version of Pocket Code, the overall test device's average execution time is 21.625 minutes. This takes 29.74 minutes for the Android version of Pocket Code and only 15.6 minutes for Pocket Paint. It should be mentioned that test devices that ran out of time were not included. The fastest in terms of speed is Pocket Paint, followed by Pocket Code. If we also consider the number of tests conducted, the answer for the Android version of Pocket Code is 0.007 minutes/test. (average run time of all devices divided by the total number of tests executed). It is 0.1984 minutes for the iOS version of Pocket Code and 0.0367 minutes for Pocket Paint for each test. The three fastest in this scenario are the Android version of Pocket Code, Pocket Paint, and the iOS version of Pocket, in that order. One could argue that there is not much difference between the execution of tests with and without BrowserStack. Basically, it can be claimed that there has not really been any improvement. However, BrowserStack allows you to run up to five shards concurrently on a single device, which can significantly improve run time compared to virtual devices.
- Number of tests executed: The average number of tests run on all devices for the Android version of Pocket Code is 3286.04. For the Android version of Pocket Code, the average total number of tests is approximately 3342.52. A total of approximately 399.65 tests have typically been performed on all devices for Pocket Paint. Furthermore,

486.55 tests have been performed on average in general. The iOS version of Pocket Code was subjected to an average of approximately 107.75 tests. Typically, there are 111 tests conducted in total. This means that the Android version of Pocket Code is in the lead in both cases. It is followed by Pocket Paint, which is then followed by the iOS version of Pocket Code.

- Percent of tests passed: For the iOS version of Pocket Code, the average number of tests that worked is approximately 97.07%. For the Android version of Pocket Code, it is about 97.93% and for Pocket Paint it is approximately 81.85%. This shows that the Android version of Pocket Code performs the best, closely followed by the iOS version of Pocket Code, and finally Pocket Paint. In particular, Pocket Paint indicates that a large number of executed tests fail in BrowserStack. On the other hand, a very high percentage of tests have already been passed for Pocket Code applications.
  All of the above information is summarised in table 7.1.

Table 7.1.: Average test values of Pocket Code and Pocket Paint applications on BrowserStack

Project	**Average execution time**	**Average time per test**	**Average number of executed tests**	**Average total number of tests**	**Average percent of tests passed**
Pocket Code (iOS version)	21.62m	0.198m	107.75	111	97.07%
Pocket Code (Android version)	29.74m	0.007m	3286.04	3342.52	97.87%
Pocket Paint	15.6m	0.0367m	399.65	486.55	81.85%

Research Question 2 How strong are the differences between individual device versions and between device vendors for real devices?
In contrast to the Android versions of Pocket Code, the picture for the iOS version of Pocket Code is rather homogeneous. All devices are very similar in terms of run time, total number of tests, and number of tests executed. This fact is probably due to the fact that the test devices come from one vendor, namely Apple. Furthermore, the test devices differ less from each

other in terms of both hardware and software, as is the case with Android devices. The average run time of 21.62 minutes also differs less than that of the simulators, which need about 24 minutes. For Android devices, the picture is sometimes very different and errors could also be found that are not on all devices, but device-specific errors. There are partly big differences in the run-times for the Android version of Pocket Code. The Vivo Y50 with Android version 10, has a run time of 47 minutes. However, the Samsung Galaxy S7 with Android version 6.0 has a run time of only 5 minutes. In addition to that, some devices even had timeouts, as the diagram 5.6 shows. Moreover, the average test run duration for the Android version of Pocket Code on emulators is 41 minutes. This is a significant improvement for the most part. In addition, the total number of tests performed on each device varies significantly. For example, the Samsung Galaxy running Android version 6 only performs 89 tests in total. However, the Samsung Galaxy S10 running Android version 9 performs 4337 tests in total. This fact is even less surprising than the fact that some tests probably do not run on certain Android versions. The fact that the total number of tests per device also differs between devices that have the same Android version, but the device vendor differs between these devices is more surprising. You can see this very well in the examples of the Motorola Moto G7 Play and Google Pixel 3, both with Android version 9. The Pixel has 4314 and the Motorola test device just 957 total test cases. This difference refers to the three test runs for each individual device on BrowserStack. However, running these devices multiple times also shows similar fluctuations as is already the case with three runs. One reason for this could be that BrowserStack simply skips some tests on certain devices to avoid timeouts. It shows that testing can be significantly affected by the manufacturer and potentially device-specific differences. As all devices can demonstrate roughly the same high coverage, there is less difference in the successful execution of tests per device than there is in the total number of tests per device. For the Android version of Pocket Code, it can therefore be summarised that execution on several devices makes a lot of sense, since in some cases large differences can be recognised with regard to the individual devices. The fact that some tests fail on certain devices is an especially exciting circumstance for which a closer look is necessary for the reasons. Significant differences between the various test devices were partially unanticipated. Especially the strong differences within the same Android versions come to mind here. But the run times of the test execution between the individual test devices sometimes also vary a lot.

For Pocket Paint, the picture is similar to the Android version of Pocket Code, but there are some peculiarities. The fact that the test coverage is much worse catches the eye. The Google Pixel 6 Pro with Android version 12 only achieves 29.23% of successful tests. The average coverage rate is also much

worse compared to Pocket Code, which was already shown in **Research Question 1**. Furthermore, certain test devices had timeouts, just as in the case of the Android version of Pocket Code. Although the overall run time per test is faster than the iOS version of Pocket Code. However, Pocket Paint falls behind the Android version of Pocket Code when compared directly. Moreover, the running time is noticeably better here than for emulators, with about 24 minutes even with 2 shards. But what distinguishes Pocket Paint from the Android version of Pocket Code is that the overall number of tests carried out on each piece of testing equipment varies less noticeably. Test devices from different manufacturers often have the same number of tests when using the same Android version. However, compared to the Android version of Pocket Code, coverage varies more from device to device. In conclusion, the partially poor coverage is particularly evident for Pocket Paint. It seems that Android versions 10 through 13 perform poorly in this case on average. Therefore, the focus should clearly be on increasing coverage.
As shown, there are some differences between testing on virtual or real devices. Generally speaking, these can sometimes be less, as in the case of the iOS version of Pocket Code. Sometimes, it could be stronger, as is the case for the Android version of Pocket Code and Pocket Paint. Especially for the last mentioned projects, a closer look should be taken in the future to find out why some tests do not work at all or why some tests fail on certain real devices with certain OS versions. This circumstance should be quite interesting to discover and deal with in detail. It has also been shown through this work that testing on virtual devices alone can never be sufficient to make a meaningful statement about the error proneness of software projects. In this situation, the straightforward selection and execution of numerous test devices is crucial. As has already been demonstrated numerous times in this work, this situation primarily affects the Android version of Pocket Code and Pocket Paint more than the iOS version of Pocket Code. However, the choice of various test devices is even more crucial, since BrowserStack could not be used at the time of this study without limitations. Given this restriction, it seems reasonable to only frequently check the status of each individual Catrobat project on BrowserStack. The possibility of choosing from different devices and selecting the right test devices was also a part of this work and was appropriately implemented for each Catrobat project.

8. Acknowledgment

This is my chance to express my gratitude to Univ.-Prof. Dipl.-Ing. Dr.techn. Slany for offering the subject and for his time. I also want to thank Professor Slany for taking over my supervision. A particular appreciation goes out to Dipl.-Ing., BSc. Patrick Ratschiller as well. He kept an eye on me closely all the time, and his advice and criticism helped me a lot. He proofread my writing and I also want to thank him for always finding time for me.

Bibliography

Alejandro, G.-M., Hayretdinm, B., & Sven, N. (2019). Differences in android behavior between real device and emulator: A malware detection perspective. *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, 399–404. https://doi.org/10.1109/IOTSMS48152.2019.8939268 (cit. on p. 3)

Appium. (2023). *Appium*. Retrieved July 7, 2023, from https://appium.io/docs/en/2.0/. (Cit. on p. 20)

Bose, S. (2023). *App & browser testing made easy*. Retrieved July 6, 2023, from https://www.browserstack.com/guide/ui-testing-guide. (Cit. on pp. 19, 20)

BrowserStack. (2023a). *Automated mobile app testing*. Retrieved July 2, 2023, from https://www.browserstack.com/app-automate. (Cit. on p. 19)

BrowserStack. (2023b). *Documentation*. Retrieved May 17, 2023, from https://www.browserstack.com/docs/. (Cit. on pp. 1, 14, 17, 23, 31, 46, 65)

BrowserStack. (2023c). *Interactive mobile app testing*. Retrieved July 2, 2023, from https://www.browserstack.com/app-live?utm_source=google&utm_medium=cpc&utm_platform=paidads&utm_content=458645050284&utm_campaign=Search-RLSA-Brand-Automate-Audiences-EMEA-CL&utm_campaigncode=Live+1000900&utm_term=p+browserstack%20live. (Cit. on p. 18)

BrowserStack. (2023d). *Our vision is to be the testing infrastructure for the internet*. Retrieved July 28, 2023, from https://www.browserstack.com/company#:~:text=It%20started%20in%202011%20when,%2C%20operating%20systems%2C%20and%20browsers.. (Cit. on p. 18)

BrowserStack. (2023e). *Test on the right mobile devices*. Retrieved April 27, 2023, from https://www.browserstack.com/test-on-the-right-mobile-devices. (Cit. on p. 34)

Catrobat. (2023a). *Catrobat*. Retrieved July 28, 2023, from https://catrobat.org/. (Cit. on p. 4)

Catrobat., I. C. A. (2023b). Retrieved April 27, 2023, from https://www.catrobat.org/. (Cit. on p. 60)

Collins, T. (2023). *Why is manual testing not sufficient for continuous delivery?* Retrieved June 24, 2023, from https://www.browserstack.com/guide/why-is-manual-testing-not-sufficient-for-continuous-delivery#:~:text=Manual%20testing%20is%20not%20the%20best%20approach%20for%20testing%20apps,enhancements%20from%20time%20to%

20time.&text=Humans%20make%20errors.,and%20cause%20a%20quality%20drop.. (Cit. on p. 15)

Docker. (2023). *Develop faster. run anywhere.* Retrieved June 23, 2023, from https://www.docker.com/. (Cit. on p. 6)

Doshi, K. (2023). *Types of testing: Different types of software testing in detail*. Retrieved July 11, 2023, from https://www.browserstack.com/guide/types-of-testing#:~:text=Manual%20testing%20can%20be%20time,tools%20to%20execute%20test%20cases.. (Cit. on pp. 7–13)

El-Morabea, K., & El-Garem, H. (2021). Ui tests. In *Modularizing legacy projects using TDD* (pp. 45–64). Apress. https://doi.org/10.1007/978-1-4842-7428-6_3. (Cit. on pp. 19, 20)

Enoiu, E., Sundmark, D., Čaušević, A., & Pettersson, P. (2017). A comparative study of manual and automated testing for industrial control software. *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 412–417. https://doi.org/10.1109/ICST.2017.44 (cit. on pp. 13, 14)

Fantini, G. (2022). *Continuous integration for end-to-end testing of mobile applications* (Master's thesis) [Rel. Luca Ardito, Maurizio Morisio, Marco Torchiano]. Politecnico di Torino. Turin, Italy. (Cit. on p. 3).

Fastlane. (2023). *Fastlane*. Retrieved May 29, 2023, from https://docs.fastlane.tools/. (Cit. on pp. 6, 36)

Gao, J., Bai, X., Tsai, W.-T., & Uehara, T. (2014). Mobile application testing: A tutorial. *Computer*, *47*(2), 46–55. https://doi.org/10.1109/MC.2013.445 (cit. on pp. 23–26)

Gillis, A. S. (2023). *Apk file (android package kit file format)*. Retrieved July 17, 2023, from https://www.techtarget.com/whatis/definition/APK-file-Android-Package-Kit-file-format. (Cit. on p. 7)

GitHub. (2023). *Paintroid*. Retrieved June 23, 2023, from https://github.com/Catrobat/Paintroid. (Cit. on p. 4)

Gizmochina. (2023). Retrieved April 27, 2023, from https://www.gizmochina.com. (Cit. on p. 52)

Goebelbecker, E. (2023). *How to install and run jenkins with docker compose*. Retrieved May 21, 2023, from https://www.cloudbees.com/blog/how-to-install-and-run-jenkins-with-docker-compose. (Cit. on p. 28)

Google. (2023). *Gradle plugin user guide*. Retrieved May 27, 2023, from https://sites.google.com/a/android.com/tools/tech-docs/new-build-system/user-guide. (Cit. on p. 31)

Gradle. (2023). *Command-line interface*. Retrieved May 27, 2023, from https://docs.gradle.org/current/userguide/command_line_interface.html. (Cit. on p. 31)

Harrold, M. J., Gupta, R., & Soffa, M. L. (1993). A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, *2*(3), 270–285. https://doi.org/10.1145/152388.152391 (cit. on p. 14)

Hayes, L. G. (2004). *The automated testing handbook* (2nd). Software Testing Inst. (Cit. on p. 15).

Hooda, I., & Chhillar, R. (2015). Software test process, testing types and techniques. *International Journal of Computer Applications*, *111*, 10–14. https://doi.org/10.5120/19597-1433 (cit. on pp. 8–12)

Jackson, M. (2023). *On jenkins terminology updates*. Retrieved July 30, 2023, from https://www.jenkins.io/blog/2020/06/18/terminology-update/. (Cit. on p. 17)

Jenkins. (2023a). *Credentials binding plugin*. Retrieved May 28, 2023, from https://www.jenkins.io/doc/pipeline/steps/credentials-binding/. (Cit. on p. 32)

Jenkins. (2023b). *Docker*. Retrieved June 23, 2023, from https://www.jenkins.io/doc/book/installing/docker/. (Cit. on p. 6)

Jenkins. (2023c). *Extending with shared libraries*. https://www.jenkins.io/doc/book/pipeline/shared-libraries/. (Cit. on p. 39)

Jenkins. (2023d). *Plot*. Retrieved June 25, 2023, from https://plugins.jenkins.io/plot/. (Cit. on p. 31)

Jethva, H. (2023). *Jenkins architecture with diagrams tutorial (explained)*. Retrieved July 30, 2023, from https://cloudinfrastructureservices.co.uk/jenkins-architecture-with-diagrams-tutorial-explained/. (Cit. on p. 17)

JOHNSTON, C. (2023). *Android fragmentation: One developer encounters 3,997 devices*. Retrieved June 24, 2023, from https://arstechnica.com/gadgets/2012/05/android-fragmentation-one-developer-encounters-3997-devices/. (Cit. on p. 15)

Khan, M. E., & Khan, F. (2012). A comparative study of white box, black box and grey box testing techniques. *International Journal of Advanced Computer Science and Applications*, *3*(6). https://doi.org/10.14569/IJACSA.2012.030603 (cit. on p. 13)

Kitakabee. (2023). *What is a test suite & test case? (with examples)*. Retrieved July 2, 2023, from https://www.browserstack.com/guide/what-is-test-suite-and-test-case. (Cit. on p. 14)

Lämsä, T. (2017). Comparison of gui testing tools for android applications (cit. on p. 21).

Lefterov, D., & Enkov, S. (2020). Automated testing framework with browserstack integration. *Proceedings of the International Scientific Conference "Machines, Technologies, Materials"*, *1*, 105–108 (cit. on pp. x, 3, 18).

Manohar, P. (2023). *Emulator vs simulator vs real device testing: Key differences*. Retrieved April 27, 2023, from https://www.lambdatest.com/blog/emulator-vs-simulator-vs-real-device/. (Cit. on pp. 23–26)

Manoj, H. (2015). *Appium essentials* (English Edition). Packt Publishing. (Cit. on p. 20).

McKenzie, C. (2023). *Master-slave terminology alternatives you can use right now*. Retrieved July 30, 2023, from https://www.theserverside.com/opinion/Master-slave-terminology-alternatives-you-can-use-right-now. (Cit. on p. 17)

Meiliana, Septian, I., Alianto, R., & Daniel. (2018). Comparison analysis of android gui testing frameworks by using an experimental study. *Procedia Computer Science*, *135*, 736–748. https://doi.org/10.1016/j.procs.2018.08.211 (cit. on pp. 21, 22)

Meyer, M. (2014). Continuous integration and its tools. *IEEE Software*, *31*(3), 14–16. https://doi.org/10.1109/MS.2014.58 (cit. on pp. 15, 16)

Min, Y., & Cai, S. (2018). *Comparing different approaches of gui testing for mobile applications on android platform* (Master's thesis) [Faculty of Computing, Department of Software Engineering]. Blekinge Institute of Technology. Karlskrona, Sweden. (Cit. on pp. 19, 20).

Muccini, H., Di Francesco, A., & Esposito, P. (2012). Software testing of mobile applications: Challenges and future research directions. *2012 7th International Workshop on Automation of Software Test (AST)*, 29–35. https://doi.org/10.1109/IWAST.2012.6228987 (cit. on pp. 7, 8)

Nishant, V. (2017). *Mobile test automation with appium: Mobile application testing made easy*. Packt Publishing. (Cit. on p. 20).

Nuriel, R. (2023). *Appium vs. espresso vs. xcuitest automation - key differences*. Retrieved July 7, 2023, from https://www.perfecto.io/blog/rise-espresso-xcuitest-fall-appium. (Cit. on p. 22)

Pai, A. (2023a). *App & browser testing made easy*. Retrieved April 27, 2023, from https://www.browserstack.com/guide/testing-on-emulators-simulators-real-devices-comparison. (Cit. on pp. 23–26)

Pai, A. (2023b). *Manual testing vs automation testing*. Retrieved July 9, 2023, from https://www.browserstack.com/guide/manual-vs-automated-testing-differences#:~:text=Manual%20Testing%20shows%20lower%20accuracy,eliminating%20the%20chances%20of%20errors.&text=Manual%20Testing%20needs%20time%20when%20testing%20is%20needed%20at%20a%20large%20scale.. (Cit. on pp. 13, 14)

Preeti Wadhwani, P. S. (2023). *Automation testing market*. Retrieved July 9, 2023, from https://www.gminsights.com/industry-analysis/automation-testing-market. (Cit. on p. 13)

Ram, K. J. (2021). Continuous integration and automation [Available at SSRN: https://ssrn.com/abstract=3966011]. *International Journal for Innovative Engineering and Management Research* (cit. on p. 15).

Scratch. (2023). *Über scratch*. Retrieved June 23, 2023, from https://scratch.mit.edu/about. (Cit. on p. 4)

Sense. (2023). *Market share of jenkins*. Retrieved June 24, 2023, from https://6sense.com/tech/continuos-integration/jenkins-market-share#:

~:text=Jenkins%20has%20market%20share%20of,CircleCI%20with%205.40%25%20market%20share.. (Cit. on p. 16)

Slany, W. (2014). Tinkering with pocket code, a scratch-like programming app for your smartphone. *Proceedings of Constructionism* (cit. on p. 4).

Smart, J. F. (2011). *Jenkins: The definitive guide: Continuous integration for the masses*. O'Reilly Media, Inc. (Cit. on pp. 15, 17).

Solutions, D. (2023). *Advantages of hosting jenkins on cloud(aws) over on-premise – chapter – 2*. Retrieved August 4, 2023, from https://www.deftboxsolutions.com/blog/advantages-of-hosting-jenkins-on-cloudaws-over-on-premise-chapter-2/. (Cit. on pp. x, 17)

Specifications, D. (2023). Retrieved April 27, 2023, from https://www.devicespecifications.com/en. (Cit. on pp. 50, 52)

Store, A. (2023). *Pocket code*. Retrieved July 21, 2023, from https://apps.apple.com/app/pocket-code/id1117935892. (Cit. on pp. x, 5)

Tutorialandexample. (2023). *Advantages and disadvantages of jenkins*. Retrieved June 28, 2023, from https://www.tutorialandexample.com/advantages-and-disadvantages-of-jenkins. (Cit. on p. 16)

ubuntuusers. (2023). *Jq*. https://wiki.ubuntuusers.de/jq/. (Cit. on p. 43)

Unadkat, J. (2023a). *Appium vs espresso: Key differences*. Retrieved July 7, 2023, from https://www.browserstack.com/guide/appium-vs-espresso. (Cit. on pp. 1, 20–22)

Unadkat, J. (2023b). *Appium vs xcuitest : Key differences*. Retrieved July 7, 2023, from https://www.browserstack.com/guide/appium-vs-xcuitest. (Cit. on p. 22)

Vashishtha, V., Singla, T., & Singh, S. (2014). Software testing. *International Journal of Research*, *1*(8), 1258–1264 (cit. on pp. 7, 8, 10).

Vilkomir, S., Marszalkowski, K., Perry, C., & Mahendrakar, S. (2015). Effectiveness of multi-device testing mobile applications. *2015 2nd ACM International Conference on Mobile Software Engineering and Systems*, 44–47. https://doi.org/10.1109/MobileSoft.2015.12 (cit. on p. 34)

Wilson, G., Aruliah, D. A., Brown, C. T., Chue Hong, N. P., Davis, M., Guy, R. T., Haddock, S. H. D., Huff, K. D., Mitchell, I. M., Plumbley, M. D., Waugh, B., White, E. P., & Wilson, P. (n.d.). Best practices for scientific computing. *PLoS Biology*. https://doi.org/10.1371/journal.pbio.1001745 (cit. on p. 6)

Appendix

Appendix A.

BrowserStack Android Devices

```
parameters {
    ...
    booleanParam name: 'DEVICE_TESTING', defaultValue: true, description: 'When selected UI-testing runs locally'
    ...
    separator(name: "BROWSERSTACK", sectionHeader: "BrowserStack configuration",
        separatorStyle: "border-width: 0",
        sectionHeaderStyle: """
            background-color: #ffff00;
            text-align: center;
            padding: 4px;
            color: #000000;
            font-size: 20px;
            font-weight: normal;
            font-family: 'Orienta', sans-serif;
            letter-spacing: 1px;
            font-style: italic;
        """)
    choice choices: [
        'AndroidDevices',
        'Samsung Galaxy S23-13.0',
        'Samsung Galaxy S23 Ultra-13.0',
        'Samsung Galaxy S22 Ultra-12.0',
        'Samsung Galaxy S22 Plus-12.0',
        'Samsung Galaxy S22-12.0',
        'Samsung Galaxy S21-12.0',
        'Samsung Galaxy S21 Ultra-11.0',
        'Samsung Galaxy S21-11.0',
        'Samsung Galaxy S21 Plus-11.0',
        'Samsung Galaxy S20-10.0',
        'Samsung Galaxy S20 Plus-10.0',
```

```
31          'Samsung Galaxy S20 Ultra-10.0',
32          'Samsung Galaxy M52-11.0',
33          'Samsung Galaxy M32-11.0',
34          'Samsung Galaxy A52-11.0',
35          'Samsung Galaxy Note 20 Ultra-10.0',
36          'Samsung Galaxy Note 20-10.0',
37          'Samsung Galaxy A51-10.0',
38          'Samsung Galaxy A11-10.0',
39          'Samsung Galaxy S10e-9.0',
40          'Samsung Galaxy S10 Plus-9.0',
41          'Samsung Galaxy S10-9.0',
42          'Samsung Galaxy Note 10 Plus-9.0',
43          'Samsung Galaxy Note 10-9.0',
44          'Samsung Galaxy A10-9.0',
45          'Samsung Galaxy Note 9-8.1',
46          'Samsung Galaxy J7 Prime-8.1',
47          'Samsung Galaxy S9 Plus-9.0',
48          'Samsung Galaxy S9 Plus-8.0',
49          'Samsung Galaxy S9-8.0',
50          'Samsung Galaxy Note 8-7.1',
51          'Samsung Galaxy A8-7.1',
52          'Samsung Galaxy S8 Plus-7.0',
53          'Samsung Galaxy S8-7.0',
54          'Samsung Galaxy S7-6.0',
55          'Samsung Galaxy S6-5.0',
56          'Samsung Galaxy Tab S8-12.0',
57          'Samsung Galaxy Tab S7-11.0',
58          'Samsung Galaxy Tab S7-10.0',
59          'Samsung Galaxy Tab S6-9.0',
60          'Samsung Galaxy Tab S5e-9.0',
61          'Samsung Galaxy Tab S4-8.1',
62          'Google Pixel 7 Pro-13.0',
63          'Google Pixel 7-13.0',
64          'Google Pixel 6 Pro-13.0',
65          'Google Pixel 6 Pro-12.0',
66          'Google Pixel 6-12.0',
67          'Google Pixel 5-12.0',
68          'Google Pixel 5-11.0',
69          'Google Pixel 4-11.0',
70          'Google Pixel 4 XL-10.0',
71          'Google Pixel 4-10.0',
72          'Google Pixel 3-10.0',
73          'Google Pixel 3a XL-9.0',
```

```
74          'Google Pixel 3a-9.0',
75          'Google Pixel 3 XL-9.0',
76          'Google Pixel 3-9.0',
77          'Google Pixel 2-9.0',
78          'Google Pixel 2-8.0',
79          'Google Pixel-7.1',
80          'Google Nexus 5-4.4',
81          'OnePlus 9-11.0',
82          'OnePlus 8-10.0',
83          'OnePlus 7T-10.0',
84          'OnePlus 7-9.0',
85          'OnePlus 6T-9.0',
86          'Xiaomi Redmi Note 11-11.0',
87          'Xiaomi Redmi Note 9-10.0',
88          'Xiaomi Redmi Note 8-9.0',
89          'Xiaomi Redmi Note 7-9.0',
90          'Motorola Moto G71 5G-11.0',
91          'Motorola Moto G9 Play-10.0',
92          'Motorola Moto G7 Play-9.0',
93          'Vivo Y21-11.0',
94          'Vivo Y50-10.0',
95          'Oppo Reno 6-11.0',
96          'Oppo A96-11.0',
97          'Oppo Reno 3 Pro-10.0',
98          'Huawei P30-9.0'
99      ]
100     booleanParam name: 'BROWSERSTACK_TESTING',
    defaultValue: false, description: 'When selected
    testing runs over Browserstack'
101     choice choices: ['1', '2', '3', '4',' 5'],
    description: 'Number of Shards for running tests on
     BrowserStack. <a href="https://app-automate.
    browserstack.com/dashboard/v2/builds/">BrowserStack
     Dashboard</a>', name: 'BROWSERSTACK_SHARDS'
102 }
```

Appendix B.

BrowserStack iOS Devices

```
parameters {
    ...
    choice choices: [
       'IosDevices',
        'iPhone 14 Pro-16',
        'iPhone 14 Pro Max-16',
        'iPhone 14 Plus-16',
        'iPhone 14-16',
        'iPhone 12 Pro Max-16',
        'iPhone 12 Pro-16',
        'iPhone 12 Mini-16',
        'iPhone 11 Pro Max-16',
        'iPhone XS-15',
        'iPhone 13 Pro Max-15',
        'iPhone 13 Pro-15',
        'iPhone 13 Mini-15',
        'iPhone 13-15',
        'iPhone 11 Pro-15',
        'iPhone 11-15',
        'iPhone XS-14',
        'iPhone 12 Pro Max-14',
        'iPhone 12 Pro-14',
        'iPhone 12 Mini-14',
        'iPhone 12-14',
        'iPhone 11 Pro Max-14',
        'iPhone 11-14',
        'iPhone XS-13',
        'iPhone 11 Pro Max-13',
        'iPhone 11 Pro-13',
        'iPhone 11-13',
        'iPhone XR-15',
        'iPhone 8-15',
        'iPhone 8-13',
```

```
34          'iPhone SE 2020-16',
35          'iPhone SE 2022-15',
36          'iPhone SE 2020-13',
37          'iPad Air 4-14',
38          'iPad 9th-15',
39          'iPad Pro 12.9 2022-16',
40          'iPad Pro 12.9 2020-16',
41          'iPad Pro 11 2022-16',
42          'iPad 10th-16',
43          'iPad Air 5-15',
44          'iPad Pro 12.9 2021-14',
45          'iPad Pro 12.9 2020-14',
46          'iPad Pro 11 2021-14',
47          'iPad Pro 12.9 2020-13',
48          'iPad 8th-16',
49          'iPad Pro 12.9 2018-15',
50          'iPad Mini 2021-15',
51          'iPad 8th-14',
52          'iPad Mini 2019-13',
53          'iPad Air 2019-13'
54      ]
55      ...
56  }
```

Appendix C.

BrowserStack Upload of Apks

```
1  final def browserStackCatOrPaintroid(final String
       credentials, final String path_to_app,
       final String path_to_test_suits, final String
       project) {
2      final String brow_app_url = '"https://api-cloud.
       browserstack.com/app-automate/espresso/v2/app"'
3      final String brow_test_url = '"https://api-cloud.
       browserstack.com/app-automate/espresso/v2/test-
       suite"'
4      final String build_url = '"https://api-cloud.
       browserstack.com/app-automate/espresso/v2/build"'
5      final String device = getDevice(params.
       BROWSERSTACK_ANDROID_DEVICES)
6      final String app_name = sh(script: "find ${
       path_to_app} -name \"*.apk\"", returnStdout: true)
7      final String test_suits_name = sh(script: "find $
       {path_to_test_suits} -name \"*.apk\"", returnStdout
       : true)
8      final String shards = params.
       BROWSERSTACK_ANDROID_DEVICES.equals('AndroidDevices
       ') ? "1" : params.BROWSERSTACK_SHARDS
9      final String framework = 'espresso'
10
11     final def response_app = sh(script: """ \
12                                 curl -u "$credentials
       " \
13                                 -X POST $brow_app_url
        -F "file=@${app_name}" \
14                                 """, returnStdout:
       true).trim()
15
16     echo("Repsonse from upload of ${app_name} is:
       $response_app")
```

```
17
18      final String app_url = sh(script: "echo '
    $response_app' | jq -r '.app_url'", returnStdout:
    true).trim()
19      final String app_id = sh(script: "echo '
    $response_app' | jq -r '.app_id'", returnStdout:
    true).trim()
20
21      final def response_test = sh(script: """ \
22                                  curl -u "$credentials
    " \
23                                  -X POST
    $brow_test_url -F "file=@${test_suits_name}" \
24                                  """, returnStdout:
    true).trim()
25
26      echo("Repsonse from upload of ${test_suits_name}
    is: ${response_test}")
27
28      final String test_suite_url = sh(script: "echo '
    $response_test' | jq -r '.test_suite_url'",
    returnStdout: true).trim()
29      final String test_suite_id = sh(script: "echo '
    $response_test' | jq -r '.test_suite_id'",
    returnStdout: true).trim()
30
31      checkIfBrowserStackIsReady(credentials, params.
    BROWSERSTACK_ANDROID_DEVICES.equals('AndroidDevices
    '))
32
33      final String config = """ \
34                              '{"shards": {"
    numberOfShards": ${shards}, \
35                              "deviceSelection": "all"
    }, "singleRunnerInvocation": "true", \
36                              "app": "${app_url}", "
    testSuite": \
37                              "${test_suite_url}", "
    devices":$device, "project": "Catrobat/${project}"
    }' \
38                          """
39
40      final def response = sh(script: """ \
```

```
41                                curl -u "$credentials" \
42                                -X POST $build_url -d
    $config \
43                                -H "Content-Type:
    application/json" | jq -r '.message, .build_id' \
44                                """, returnStdout: true).
    trim().split("\n")
45
46      final def message = [build_id: response[1],
    app_id: app_id, test_suite_id: test_suite_id,
    framework: framework]
47
48      return message
49  }
```

Appendix D.

Polling of BrowserStack Status for specific Test Run

```
1  void browserStackPolling(final String credentials,
       final def message) {
2      try {
3          String response, code
4          (response, code) = sh(script: """ \
5                                curl -w' \\n%{response_code}'
       \
6                                -u "$credentials" \
7                                -X GET "https://api-cloud.
       browserstack.com/app-automate/${message.framework}/
       v2/builds/${message.build_id}" \
8                                """, returnStdout: true).trim
       ().tokenize("\n")
9          while (code.toInteger() == 200) {
10             status = sh(script: "echo '$response' |
       jq -r '.status'", returnStdout: true).trim()
11             echo "Current status is: $status"
12             if (status.equals('queued') || status.
       equals('running')) {
13                 sleep 60
14                 (response, code) = sh(script: """ \
15                                       curl -w' \\n%{
       response_code}' \
16                                       -u "$credentials" \
17                                       -X GET "https://api-
       cloud.browserstack.com/app-automate/${message.
       framework}/v2/builds/${message.build_id}" \
18                                       """, returnStdout:
       true).trim().tokenize("\n")
19                 continue
20             } else if (status.equals('passed') ||
```

```
    status.equals('failed') || status.equals('error')
    || status.equals('time out') || status.equals('
    skipped')) {
21                  getBrowserstackEndMessage(credentials
    , status, message)
22                  return
23              } else {
24                  getBrowserstackEndMessage(credentials
    , status, message)
25                  return
26              }
27          }
28          getBrowserstackEndMessage(credentials, status
    , message)
29      } catch (exception) {
30          echo "BrowserStack testing failed: ${
    exception}"
31          deleteUpload(credentials, message)
32      }
33  }
```

Appendix E.

Fetching Test Reports

```
1  void getXmlFile(final String credentials, final def
       message) {
2      if (!fileExists('BrowserstackReports')) {
3          sh('mkdir BrowserstackReports')
4      }
5      sh('rm -rf BrowserstackReports/*')
6
7      final def sessions = sh(script: """ \
8                              curl -u "$credentials" \
9                              -X GET "https://api-cloud
       .browserstack.com/app-automate/espresso/v2/builds/{
       $message.build_id}" \
10                             | jq -r ".devices[].
       sessions[].id"\
11                             """, returnStdout: true).
       trim().split("\n")
12     for (session in sessions) {
13         sh """ \
14             curl -u "$credentials" \
15             -X GET "https://api-cloud.browserstack.
       com/app-automate/espresso/v2/builds/$message.
       build_id/sessions/$session/report" \
16             -o BrowserstackReports/${session}.xml
17         """
18     }
19     deleteUpload(credentials, message)
20     sh('junit-merge -o browserstack_reports.xml -d
       BrowserstackReports')
21     junitAndCoverage "$reports/jacoco/
       jacocoTestDebugUnitTestReport/jacoco.xml", 'unit',
       javaSrc
22 }
```