Research, Conceptual Design, and Prototypical Development of a Tool with GUI for ADB-based Automated Testing of Android Devices

MASTER THESIS

by
Shaira Rahman Shuchi
(11153541)

Submitted to obtain the degree of

Master of Science (M.Sc.)

at

Cologne University of Applied Sciences

Faculty of Computer Science and Engineering Science

Course of Study

Digital Sciences

First supervisor: Prof. Dr. Mario Winter
Second supervisor: A.S.M Shafiul Alam

Gummersbach, September 2024

Technology
Arts Sciences
TH Köln

Contact details: Shaira Rahman Shuchi
shaira_rahman.shuchi@smail.th-koeln.de

Prof. Dr. Mario Winter
Cologne University of Applied Sciences
Faculty of Computer Science and Engineering
Steinmüllerallee 6
51643 Gummersbach
mario.winter@th-koeln.de

A.S.M Shafiul Alam
MSc. In Computer Science & Engineering
Secusmart GmbH
Heinrichstraße 155
40239 Düsseldorf
shafiul.alam@secusmart.de

Confidentiality clause

The following Master thesis contains confidential data and information disclosed by Secusmart GmbH | Blackberry, Heinrichstraße 155, 40239 Düsseldorf. It may not be disclosed, published or made known in any other manner, including in the form of extracts, before date or without the explicit permission of Secusmart GmbH. The Master thesis is made available to members of the Examination Board solely for the purpose of assessment.

Declaration of Authenticity

I guarantee that the work I am submitting here has been written by myself. I have suitably marked all texts which have been quoted, either word for word or in paraphrased form, from the published or unpublished work of others.

All sources and aids which I used for this work are listed. The work has not been submitted with the same contents or with any significant parts of this work to any other examination authorities.

Wuppertal, 27.08.2024

Place, Date

Authorized signature

Abstract

Android has become the most commonly used operating system for mobile devices. If any bug remains in the applications of any Android operating system, it can affect user experience and harm the company's reputation. As a result, all the applications need to be tested for every software update to ensure their functionality. Manually performing these tests can be both time consuming and prone to human error. To address this problem of Secusmart GmbH, a prototype of a graphical user interface (GUI) based tool for automated testing of Android devices has been designed and implemented using Android Debug Bridge (ADB). The main objective of this design was to increase the efficiency of testing, especially smoke testing. After implementation, several test cases were conducted and the results were analyzed. The experimental results of this thesis show improvement in testing accuracy and efficiency compared to using traditional manual testing.

Table of Contents

List of Figures

List of Tables

1 Introduction

With the rapid growth of mobile technology, the software development of mobile applications is going through immense changes. With these diverse changes, the testing method of mobile software needs to change accordingly. As mobile devices are getting more and more diverse and complicated, it is becoming very difficult to ensure the expected performance and quality of these devices. To overcome these challenges, automated testing has become an integral part of the software development lifecycle.

This thesis concentrates on researching and creating a conceptual design and the prototypical implementation of some tools where the user interface is based on Graphical User interface (GUI) for automated testing of Android devices using Android Debug Bridge (ADB). The main goal is to make the prototype of an efficient, reliable, and user-friendly automated testing tool for the developers and testers of the company.

1.1 Motivation

Nowadays for mobile application testing, test automation has become an important part of the software development lifecycle. Ensuring consistent performance and user experience for all Android devices has become a great challenge for developers because of the divergent hardware setups of different Android devices. Although manual testing has been used for a long time, it is no longer a practical choice because of its slow process and the chance of human error. As a result, automated testing has become a clear choice for testing Android devices.

Android Debug Bridge (ADB) is well suited for automated testing because of its versatile interface to communicate with different Android devices but it is not suitable for testers who do not have extensive background knowledge. Testers usually do not like using the complex command-line interface of ADB. An easier and user-friendly solution is necessary to reduce this complexity which can be achieved by integrating ADB with Graphical User Interface (GUI).

The initial problem addressed by this thesis is the complication and inefficiency of the currently used testing methods for smoke testing of Android devices. Manual testing of devices is not scalable and they tend to fail to provide comprehensive coverage. This leads to application performance and user experience issues. Although automated testing is more efficient, its effect is hampered by the technical complexity of tools like ADB, which requires high technical knowledge to use them.

Therefore, the primary motivation of this thesis is to solve these challenges by investigating how ADB-based automated testing can be integrated with a GUI and develop the design considerations along with implementing a prototype. This tool focuses on enhancing the quality, reliability and performance of automated smoke testing that eases the work of developers and testers of the company.

1.2 Research Questions and Tasks

The goal of this study is to find the most appropriate answers to the following research questions:

RQ1. How can a GUI-based tool be designed to simplify the use of ADB in case of automated testing of Android devices?

RQ2. What is the effect of implementing a GUI-based automated testing tool on the efficiency of testing compared to traditional manual testing method?

RQ3. How does the automated testing tool ensure the accuracy and reliability of test results for different Android devices of different configurations?

RQ4. How can the automated testing tool be scaled to support multiple devices effectively?

RQ5. What are the results of experiments performed comparing the automated testing tool with manual testing methods in terms of error detection and resource utilization?

The following tasks will be undertaken to address these research questions:

1. A thorough review of the existing literature on Android, automated testing, ADB and GUI will be conducted to learn about their structure and usage.

2. The typical problems experienced while using the current testing tools and methods will be identified, analyzed and documented.

3. A GUI-based prototype tool will be conceptualized and designed using ADB for automated smoke testing.

4. The designed tool will be implemented ensuring efficiency, user-friendliness and capability of handling a variety of testing scenarios.

5. Experimental studies will be carried out to compare the performance of the designed tool with traditional manual testing methods.

6. Necessary data will be collected to evaluate how practical the tool is based on efficiency, reliability, and performance.

1.3 Methodology Overview

A mixed method approach is adopted in this thesis to investigate the transition from manual testing to automated testing. The methodology consists of qualitative and quantitative components. This mixed methodology focuses on improving the efficiency, accuracy and scalability of Android application testing.

1.3.1 Qualitative Component

The qualitative component involves a detailed literature review and survey to obtain necessary information about current testing methods and their limitations.

An in-depth review of existing literature on the current manual testing and automated testing methods will be conducted. This will help to find out their methodologies and limitations. Then the structure of ADB and GUI will be discussed in detail to learn more about their usage and existing problems. Related academic papers and websites will be accessed to gain a comprehensive amount of information about these topics.

A survey will be conducted which will be filled out by the current employees of the company about different matters of Android application testing. It will help to know about their preferred testing tools and methods and which type of challenges they are facing while testing Android devices. Moreover, their opinion and advice will be gathered through this survey which will give a clear outline of the tasks that need to be performed.

1.3.2 Quantitative Component

The quantitative component involves designing, developing and evaluating a GUI-based prototype of the automated testing tool using ADB.

A detailed experimental plan will be designed based on test scenarios, metrics and data collection methods. The provided experimental design will ensure valid and reliable results. Then the prototype tool will be implemented based on the design. Key factors like executing ADB commands, managing test cases, and providing a user-friendly interface will be ensured while implementing the design. Finally, necessary data

about test execution time, defect detection and resource allocation will be collected after conducting experiments.

The methodology can be divided into three important phases:

The first phase is the preparatory phase. Here, the environment will be set up by configuring Android devices, installing necessary applications, and preparing test suites. A variety of Android devices will be set up to ensure comprehensive testing. The required software components such as ADB and Python libraries will be installed in respective devices.

The next phase is the execution phase. At first, the software testers will perform selected test cases manually and document each step to ensure comparison with the developed automated testing tool later. Then the software testers will conduct the same test cases using the designed automated testing tool and monitor the execution in real-time to ensure it mimics every detail when tests are run manually.

The final phase is the evaluation phase. In this phase, metrics such as execution times, defect logs, and resource utilization are compared to get the details about the performance improvement by using the automated testing tool.

1.3.3 Continuous Improvement

After implementation, evaluation and required improvement processes, it is important to ensure that the tool remains effective for upcoming evolving testing requirements. In order to ensure that, feedback will be collected from testers of the company to identify areas of improvement. Moreover, regular software updates will be released to enhance performance, fix bugs and add new features based on the evolving testing requirements. The tool's performance will be under continuous monitoring and necessary adjustments will be made to ensure optimum operation.

1.4 Structure of the Thesis

The thesis is structured into six chapters. The first chapter discusses the introduction to the thesis topic, the motivation behind the thesis, research questions and tasks to do and an overview of the whole methodology. The second chapter reviews the background and existing literature on the Android operating system, manual and automated testing methods along with ADB and GUI-based testing tools. This chapter discusses about basic concepts of these topics and some of the limitations. The third chapter describes a case study to illustrate the challenges in current testing methods and proposes a conceptual design for the

prototype tool. This chapter discusses the architecture of the designed tool for automated testing in detail. The fourth chapter describes the implementation of the prototype tool along with its performance, procedures and data collection methods. The fifth chapter presents the findings of the experimental study including data analysis and interpretations. This chapter also highlights the contributions and limitations of this research and suggests areas of future improvements for the prototype tool.

2 Background and Literature Review

With the sharp increase in usage and productivity of Android devices, Android has become a leading operating system for mobile devices. As a result, a comprehensive understanding of Android's architecture, components and the difficulties faced in developing and testing such a system has become crucial. This chapter discusses the fundamental aspects of the Android operating system. The architecture and core components of the Android system are illustrated in detail to get a good understanding of the Android ecosystem. In the next section, various aspects of mobile application testing including manual testing and automated testing are investigated deeply. This is followed by another in-depth discussion about Graphical User Interface (GUI) which plays an important role in user experience and application functionality. This chapter also reviews some of the commonly used automated testing tools and their limitations to match the expected results. Finally, a company overview of Secusmart GmbH is included in order to provide a better understanding of the practical implications of these technologies and methodologies within a real-world business environment. This detailed and comprehensive overview sets the stage for the case study and conceptual design discussed in the following chapter.

2.1 Android

Android is an open-source operating system based on the Linux kernel. It allows the development of applications using Java (after [Sarkar et al., 2019]). It is an end-user operating system that supports more than 3.9 billion active users (after [AppMySite, 2022]) including different devices like smartphones, tablets, wearables, TVs, IoT and more (after [Mayrhofer et al., 2019]). End users can install or uninstall apps by themselves which makes the overall system more dynamic and flexible (after [Schmerl et al., 2016]). These applications are created for different important domains such as health, education and the military and so it is necessary to have high-quality standards (after [Motan and Zein, 2020]). Because of its customization properties and flexibility developers can easily deploy applications across multiple platforms without major code changes (after [Sarkar et al., 2019]). The accessible development environment based on familiar Java

programming language and the availability of libraries implementing all sorts of different functions has made Android a popular choice among developers worldwide (after [Li et al., 2015]). The users can easily find their required Android apps in Google Play Store and some other alternative stores like Anzhi and AppChina. These store platforms have also made it very easy for organizations to market their apps (after [Li et al., 2017]). Although it has a lot of flexibility, Android's security vulnerabilities are major risks as a lot of applications do not undergo a necessary security check before becoming available on the Android market (after [Sarkar et al., 2019]).

2.1.1 Android Framework Architecture

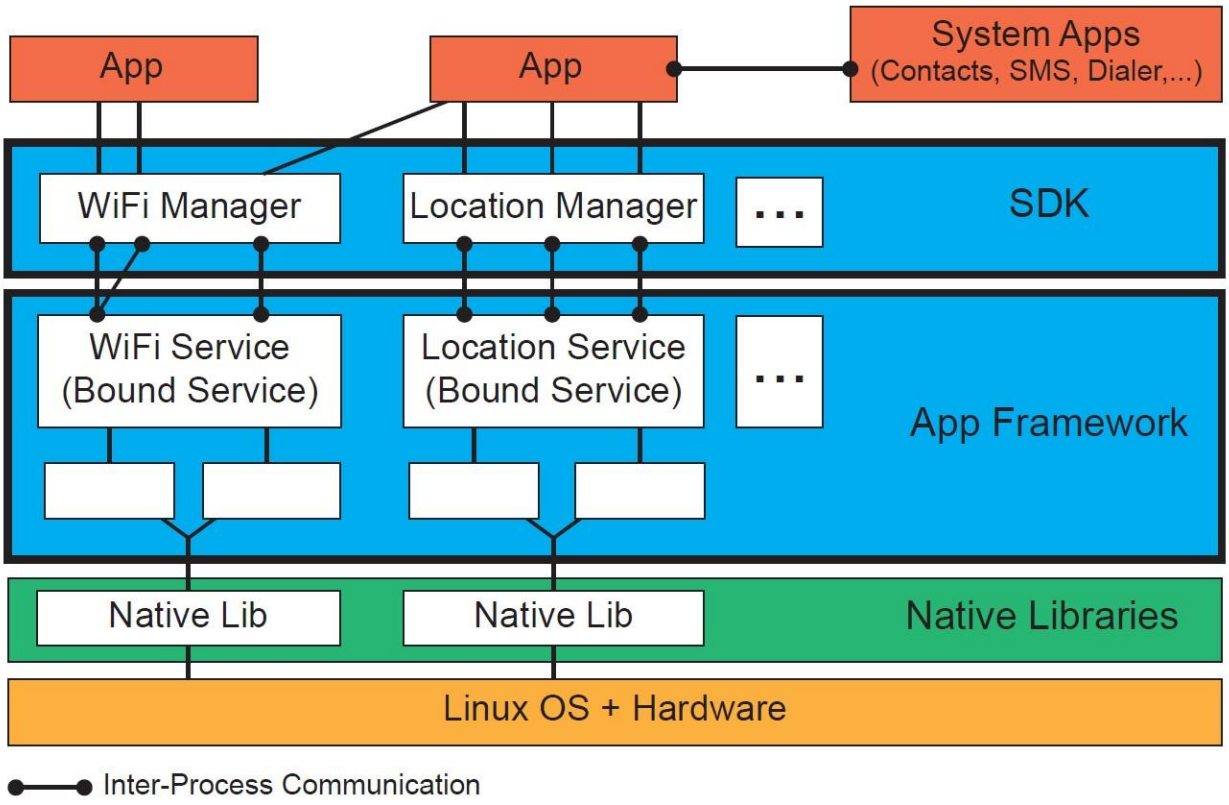

Figure 1: Android software stack with abstract control and data flows (from [Backes et al., 2016]).

The Android Framework has the layered architecture shown in Figure 1.

The bottom layer consists of Linux OS and Hardware. This layer is considered the foundation of the Android OS. This layer manages core system services such as memory management, process management, security, and networking. Android customizes the Linux kernel so that it can be used on mobile devices (cf. [Mazuera-Rozo et al., 2019]). This layer also includes various hardware components CPU, memory, and device-specific peripherals (after [Backes et al., 2016]).

The second layer consists of Native libraries written in C/C++ that provide low-level functionalities and computational services required for the Android Framework and Runtime (cf. [Mazuera-Rozo et al., 2019]). An example is SSL for secure communication and the SQLite library for accessing the database (after [Backes et al., 2016]).

The third level is the App Framework. This layer contains the Android application framework written in Java. It provides higher-level services to applications. The Android app API, such as getting location data or telephony functionalities is implemented at this level. WiFi Service and Location Service are implemented as bound services. Applications can bind to them and communicate with them via a defined interface. Classes like WiFiManager and LocationManager act as abstractions encapsulating the complex details of these services. This provides a simpler interface for applications to interact with them. This layer includes the Software Development Kit (SDK) which possesses tools and libraries that developers use to build Android applications (after [Backes et al., 2016]).

The fourth layer includes pre-installed System Apps like Contacts, SMS, and Dialer that interact with the app framework services. These apps are programmed using the same framework APIs as third-party apps and may be replaced or customized by the device vendors (after [Backes et al., 2016]).

The final layer is the App layer which contains all the applications installed on the device. These applications are written in Java and use Android APIs to implement their features. Some common apps are browser, calendar, telephony, messaging, and settings (after [Mazuera-Rozo et al., 2019])

Apps communicate with each other and the framework services using Inter-Process Communication (IPC) mechanisms which is vital for maintaining security and resource management (after [Backes et al., 2016]).

2.1.2 Components of Android Apps

The structure of an Android app is based on four basic components. They are Android Activities, Services, Content Provider, and Broadcast Receiver.

An activity in an Android app represents a single screen with a user interface. Activities are independent but they for a cohesive user experience to interact with other apps. The Android Activity lifecycle can be described as a series of states of activity to go through from creation to destruction, managed by lifecycle callback methods (after [Readthedocs.io., 2023]).

Figure 2 displays various states an Android activity goes through from launch to shut down. Each state has specific lifecycle callbacks. onCreate() is called when the activity is first created. It is used for initial setup such as inflating the layout (after [Activity, 2024]). onStart() is called when the activity becomes visible to

the user. onResume() is called when the activity starts interacting with the user. At this point, the activity is at the top of the activity stack (after [Readthedocs.io, 2023]). onPause() is called when the system is about to start another activity (after [Activity, 2024]). onStop() is called when the activity is no longer visible to the user. onRestart() is called when the activity restarts after being stopped. onDestroy() is called before the activity is destroyed. This is the final cleanup method (after [Readthedocs.io, 2023]).

There are different Activity Lifecycle States. The entire Lifetime is from onCreate() to onDestroy(). The activity is created and destroyed once during this period. Visible Lifetime is from onStart() to onStop().the activity is visible to the user but not necessarily in the foreground in this state. Foreground Lifetime is from onResume() to onPause(). The activity is in the foreground in this state, interacting with the user (after [Ghanem and Zein, 2020]).

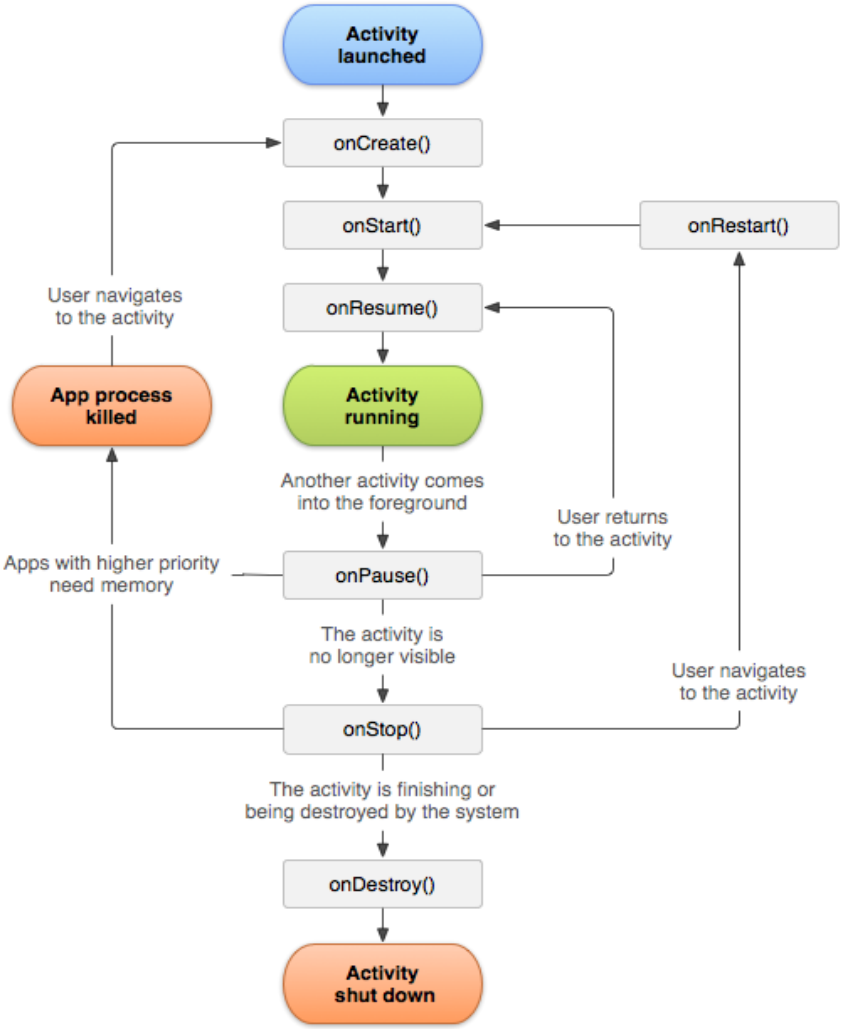

Figure 2: Android activity lifecycle (from [Readthedocs.io, 2023])

The services component type defines a service within an app. A service is a component that can perform long-running operations in the background without a user interface. Other elements within an app can start or connect to such a service for tasks like network transactions, playing music, and interacting with a content provider. Services specify the interfaces they offer and the services they use through specified ports (after [Schmerl et al. 2016]). For instance, playing music while navigating other programs or finding location data even when the app is not actively displayed on the screen (after [TechAhead, 2024]).

There are two types of services. They are Started Services and Bound Services. Started Services are initiated by a component using the startService() method and run indefinitely until they complete their task or are explicitly stopped. Bound Services allow components to bind to them using the bindService() method which provides a client-server interface to interact with the service (after [Android Developers, 2019]).

Content providers manage and encapsulate data. They provide security mechanisms like read and write permissions (after [Schmerl et al. 2016]). They allow applications to query and share data which is either stored privately within their own environment or shared across multiple applications. As a result, seamless data sharing and management of crucial app information is possible (after [TechAhead, 2024]). Architecturally, permissions to read and modify the data are differentiated within these components (after [Schmerl et al. 2016]). The functionality of content providers can be divided into CRUD Operations and Data Sharing types. In CRUD Operations, Content providers support Create, Read, Update, and Delete operations, allowing apps to perform these actions on the data. Data Sharing facilitates sharing of data such as contacts, images, or media files between applications (after [Android Developers, 2019]).

Broadcast receivers handle system-level events, such as completion of device boot or low battery level notifications (after [Schmerl et al., 2016]). They serve as gateways into users' apps and manage signals like low battery alerts or Wi-Fi connectivity status changes (after [TechAhead, 2024]). Differing from activities, broadcast receivers can exclusively obtain a specific subset of intent types known as standard broadcast actions (after [Schmerl et al., 2016]). There are two types of Broadcasts: System Broadcasts and Custom Broadcasts. System Broadcasts include announcements like device boot completion, low battery alerts, or changes in network connectivity. With Custom Broadcasts, applications can create and send their own broadcast messages to communicate with other apps or within the same app (after [Android Developers, 2019]).

2.1.3 Fundamentals of Android Application Testing

Testing is an important part of the software development cycle. It is a necessary process because it helps to improve app quality, increase user satisfaction level, and decrease overall time to fix bugs (after [Android Developers, n.d.]). The Android distribution ecosystem is very vulnerable because of poorly tested apps. It

significantly affects user experience and leads to poor app rating which harms the reputation of developers and organizations (after [Wang et al., 2018]). So, the purpose of Android app testing is to ensure the functionality, usability, and compatibility of apps across different Android devices (after [Kong et al., 2019]). Effective testing methods are required to improve the security and reliability of these applications (after [Imparato, 2015]).

There are a few main challenges in testing Android applications. The first of these is that traditional Java testing tools cannot be directly used for Android because of its event-based mechanisms like Inter-Component Communication (ICC) (after [Li et al., 2015]). The next challenge is that diverse OS versions and device types complicate testing strategies (after [Li et al., 2018]). The large number of apps require scalable testing approaches (after [Kong et al., 2019]). Another challenge is generating comprehensive test cases. It is challenging due to the event-driven nature and framework libraries of Android apps (after [Mirzaei et al., 2012]). The complex UI events are also challenging, for example, generating events like drag and hover is difficult, as these require precise conditions (after [Song, Qian and Huang, 2017]). Developers often face another unique challenge due to the specific components and lifecycle management required by Android applications which leads to potential bugs (after [Amalfitano et al., 2012]). Another challenge occurs for publishing in market place like Google Play, which requires additional approval cycles (c.f. [ISTQB (CT-MAT), 2019]).

Figure 3 shows the process of testing Android apps. There are five steps of this process. The first step is to install the Android application to a device. This device can be either a physical Android device or an emulator that simulates the Android environment. This installation is necessary to prepare the app for testing. It allows subsequent steps to interact with the app directly. The second step is the test case generation. The app is analyzed to generate test cases in this step. Test cases are a set of conditions for which a tester can determine whether an application is working correctly or not. The generation of test cases can be done automatically or manually depending on the testing approach. Automated test case generation is often preferred for efficiency. Some testing techniques, such as automated random testing, do not require pre-knowledge of the app for generating test cases (after [Kong et al., 2019]).

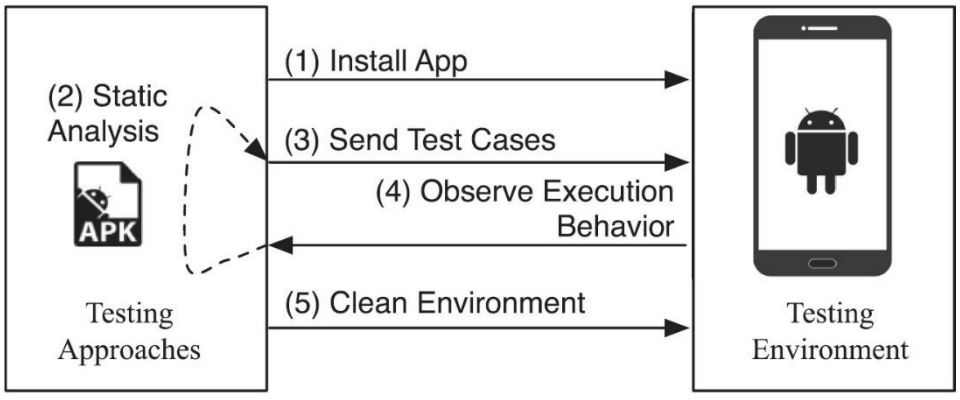

Figure 3: Testing process of android app (from [Kong et al., 2019])

The third step is about the execution of Test Cases. The generated test cases are sent to the Android device to execute them in this step. Then the app is examined with the help of different inputs mentioned in the test cases. The goal is to simulate user interactions and system events to find out any defects or issues in the app's behavior. The fourth step is about observation and collection of execution behavior. The behaviors of the app are observed from different points of views. This observation stage includes monitoring the app's performance, responsiveness, stability, and correctness under different conditions. Data collected stage during this step can include logs, screenshots, performance metrics, and error reports which are essential for identifying and diagnosing issues. The fifth and final step is uninstallation and data wiping. The target app is uninstalled from the device and any data related to the testing process is wiped after the testing is completed. This cleanup step is done to ensure that the device is reset to its original state and to make it ready for further testing with other apps or subsequent test iterations of the same app (after [Kong et al., 2019]).

There are different kinds of test types and methodologies available. The most used test types and methodologies are described below:

Smoke Testing: Smoke testing is conducted on an initial software built by developers. It ensures that the core functionalities are working correctly before proceeding to in-depth testing phases (after [G, P and G, 2022]).

Sanity Testing: Sanity testing is performed after passing smoke and other levels of testing. It verifies that new functionalities work as expected and no new issues have been introduced (after [G, P and G, 2022]).

Stress Testing: Stress testing assess app performance under high CPU usage, low memory, battery stress and poor bandwidth. Stress conditions can be simulated using tools like Monkey (c.f. [ISTQB (CT-MAT), 2019]).

Security Testing: Security testing focuses on data security. This includes testing for code injection, encryption of data and secure deletion (c.f. [ISTQB (CT-MAT), 2019]).

Performance Testing: This measures app performance and compares chronometry with similar apps (c.f. [ISTQB (CT-MAT), 2019]).

Usability Testing: This tests for user experience and guarantees that the app is intuitive, consistent and follows platform design guidelines (c.f. [ISTQB (CT-MAT), 2019]).

Database Testing: This validates data storage, synchronization, security and CRUD operations (c.f. [ISTQB (CT-MAT), 2019]).

Model-based Testing: This methodology involves automatically generating test cases based on a model that describes the functionality of the system. Although designing and creating the model requires significant effort, test cases can be comprehensively and automatically generated and executed (after [Kong et al., 2019]). Takala et al. described their experiences of applying model-based GUI testing to Android apps, covering implementation, modeling, and execution aspects (after [Takala, Katara and Harty, 2011]).

Search-based Testing: Metaheuristic search techniques are used in this method to generate test cases for finding as many errors as possible, especially critical errors (after [Afzal, Torkar and Feldt, 2009]). Mahmood et al. developed an evolutionary testing framework for Android apps that combines genetic and hill-climbing techniques to maximize code coverage (after [Mahmood, Mirzaei and Malek, 2014]).

Random Testing: This technique tests programs by generating random inputs and comparing the outputs to a specification to determine pass or fail (after [Hamlet, 2002]).

Fuzzing Testing: This testing method involves testing software with invalid, unexpected, or random data inputs. It is primarily used for security testing by finding exceptions such as crashes and memory leaks (after [Kong et al., 2019]).

A/B Testing: This method compares two variants of a testing object to determine which one is more effective. It is often used for statistical hypothesis testing (after [Kong et al., 2019]).

Concolic Testing: This is a hybrid technique that combines symbolic execution and concrete execution paths to verify software (after [Anand et al., 2012]).

Mutation Testing: This method evaluates the quality of existing software tests by applying mutation operators to the source program to create mutants and checking if the test suite can detect these changes (after [Kong et al., 2019]).

Accessibility Testing: This guarantees app usability for users with disabilities by testing device accessibility settings. It follows accessibility guidelines from platform vendors like Google (c.f. [ISTQB (CT-MAT), 2019]).

2.1.4 Android Debug Bridge (ADB)

ADB is a command line tool that allows developers to communicate with any Android device connected to a development system such as a PC (after [Lin et al., 2014]). It has a set of unique features which are specified in the Android permission system. Android uses a permission system to protect its system resources. Permissions are categorized into different levels such as normal, dangerous, and signature levels depending on the risk. ADB has access to some of these higher-level permissions that are otherwise

restricted. It allows developers to programmatically interact with critical system resources (after [Yang, Wang and Zhang, 2017]).

ADB consists of three main components and they are Client, Daemon (adbd) and Server. Client sends commands from your development machine. Daemon (adbd) runs commands as a background process on the Android device. The server manages communication between the client and the daemon. It runs as a background process on the development machine (after [Easttom and Sanders, 2019]).

When the ADB client starts, it checks for an existing ADB server process and launches one if it is not running. The server listens for commands on TCP port 5037. The first step in connecting a device is to list all connected devices by using the adb devices command. ADB allows users to use the adb shell command to enter a shell on the Android device where standard Linux commands can be run (after [Easttom and Sanders, 2019]).

ADB provides several commands that are useful for forensic examinations:

adb pull: This command extracts a file or directory from the device to the connected computer.

adb Push: This command is used to send a file to the device. The source location of the file must be specified in the source argument of the command. The destination where the file should be sent must also be specified.

adb restore: This command creates a backup of the device.

adb reboot: This command reboots the device in normal mode. It can be used when something is flashed on the device and it is necessary to reboot.

dumpsys: This command dumps system data for specific packages or activities.

adb install: This command installs an app directly from the computer to the device. It is necessary to specify the APK location in the command and it will install the selected app on the device.

pm list packages: Lists all packages on the device with various filtering options (after [Easttom and Sanders, 2019]).

ADB can be used for forensic analysis of rooted and non-rooted devices. For non-rooted devices, ADB can still be used to navigate and extract data from the external SD card. Commands such as adb pull /sdcard are useful for creating forensic images of the SD card. However, accessing internal storage directories may be restricted due to insufficient permissions. Rooting the device allows full access to the entire file system.

This includes accessing critical directories and extracting SQLite databases used by applications. This process involves using commands such as dd for imaging and adb pull for data extraction (after [Easttom and Sanders, 2019]).

2.2 Testing of Mobile Applications:

Mobile applications make everyday tasks easier and offer numerous conveniences in various aspects of people's lives. With the increase in user demands, mobile applications are becoming more complex. The biggest challenge is not to create an application but to ensure its effectiveness and stability (after [Kulkarni and Soumya A, 2016]). Many mobile applications are developed under some constraints such as budget, time, and resources. They are often released quickly due to market pressure (after [Merina, Anggraini and Hakiem, 2018]). As a result, it is common for applications to face issues and respond poorly to unexpected user actions. This can ruin user trust, reduce perceived usability and negatively impact the reputation of the company developing the application. These applications require extensive testing because they must correctly handle a variety of system and user actions (after [Pareek et al., 2015]). Therefore, conducting functional or acceptance testing during the development of mobile applications is essential to improve their security and reliability (after [Blundell et al., 2015]).

2.2.1 Testing Standard of Mobile Applications

There are some testing guidelines that need to be followed by an analyzer. These guidelines are discussed below:

Testing for Failure: The purpose of testing is to detect errors. The main goal is to show that the system is not free of bugs and to detect as many issues as possible to improve the effectiveness of the process (after [Halani, Kavita and Saxena, 2021]).

Proactive Testing: Testers should start testing early in the development cycle to quickly identify and fix faults. This approach is cost-effective because fixing errors later in the process becomes more expensive. For example, it will be costlier to discover a requirements-related error at the end of the process than at the beginning of the process (after [Halani, Kavita and Saxena, 2021]).

Context-Dependent Testing: Testing varies depending on the context. It means that different tests are performed under different conditions. For example, if two projects are based on different models, the testing methodology for both projects will also be different. Factors such as the type of product, user

specifications, documentation and risk influence the testing technique (after [Halani, Kavita and Saxena, 2021]).

Test Plan Definition: It outlines the scope, risks, objectives, methods, and tools of testing. The main goal here is to meet the needs of both the company and the client. A well-defined approach is required to ensure that all tests are conducted correctly and produce reliable results (after [Halani, Kavita and Saxena, 2021]).

Defining Effective Test Cases: The analyzer must be aware of the customer's requirements to verify them against the system and ensure that they are properly met. Effective test cases should be designed to detect the maximum number of errors in a short period. Each test case should contain input data, expected output data, and the actual output of the system (after [Halani, Kavita and Saxena, 2021]).

Condition Validity Testing: It is also beneficial to perform tests for erroneous conditions to see how the product behaves and to identify most problems (after [Halani, Kavita and Saxena, 2021]).

Regular Review of Test Cases: Test cases should be frequently evaluated because repeating the same tests will not reveal new errors or faults. Testers should modify test cases to get the best results (after [Halani, Kavita and Saxena, 2021]).

2.2.2 Mobile Application Test Types

Certified Tester Mobile Application Testing (CT-MAT) discusses various aspects of testing mobile applications. There can be 3 types of testing. They are Testing for Compatibility with Device Hardware, Testing for App Interactions with Device Software and Testing for Various Connectivity Methods. A description of different mobile application test types is provided below according to [ISTQB (CT-MAT), 2019].

1. Testing for Compatibility with Device Hardware:

Testing for Device Features: A wide range of devices with different features should be covered for compatibility testing. Devices can differ in features like navigation methods, keyboards and hardware such as radio, USB, Bluetooth, cameras, speakers, microphones and headphone access. Testing should guarantee the app function for all the different devices.

Testing for Different Displays: Testing must be performed for devices with various screen sizes, viewport sizes, aspect ratios, and resolutions (ppi/dpi). Images should not shrink problematically for high dpi/ppi.

Testing for Device Temperature: Battery charging or continuous use of cellular data and Wi-Fi can overheat devices. Overheating can reduce CPU frequency and turn off parts of the system. Testing should guarantee that the app behaves as expected under heat.

Testing for Device Input Sensors: Tests check app functionality for different motions, lighting conditions, sound inputs/outputs and accurate location positioning under different conditions.

Testing Various Input Methods: Testing should ensure that apps work with various soft keyboards and gestures (e.g., touch, swipe, pinch). Cameras should function correctly for capturing images, scanning codes and measuring distances.

Testing for Screen Orientation Change: Tests check correct usability, data retention and functional behavior during multiple orientation switches in mobile devices.

Testing for Typical Interrupts: Calls, messages, low memory and app switching are some of the common interruptions. Tests ensure the app handles interrupts without any negative impact.

Testing for Access Permissions to Device Features: Apps require access to folders and sensors (e.g., camera, microphone). Tests verify the app's functionality with reduced permissions.

Testing for Power Consumption and State: Tests evaluate battery power state, data integrity under low power conditions and power consumption during active and background usage.

2. Testing for App Interactions with Device Software:

Testing for Notifications: Testing ensures correct handling of notifications in both foreground and background, especially under low battery.

Testing for Quick-access Links: Tests must ensure that actions via quick-access links are accurately reflected when the app is opened.

Testing for User Preferences Provided by the Operating System: Tests guarantee that apps respect user-set preferences, such as sound, brightness and power save mode.

Testing for Different Types of Apps: Test conditions vary by app types like native apps, hybrid apps and web apps.

Testing for Interoperability with Multiple Platforms and OS Versions: Apps supported on multiple operating systems must be tested for handling interrupts and notifications.

Testing for Interoperability and Co-existence with Other Apps: Tests should ensure correct data transfer, no harm to user data and address potential conflicts such as conflicting GPS settings.

3. Testing for Various Connectivity Methods:

Mobile devices connect via various networks such as 2G, 3G, 4G, 5G, Wi-Fi, NFC and Bluetooth. Users may switch between connectivity modes like Wi-Fi to cellular or encounter no network like flight mode. Connectivity tests must ensure correct app functionality across different modes, smooth switching between modes and clear user information if functionality is restricted due to low or no connectivity.

Testers have two options for testing. They are manual testing and using test scripts for automation (after [Kulkarni and Soumya A, 2016]). These testing methods are further described in the following subchapters.

2.3 Manual Testing

Manual testing is the software testing process where testers execute test cases manually without the use of automation tools. This testing method checks all the functionality of the application according to the requirements and ensures that the software works as expected (after [Halani, Kavita and Saxena, 2021]).

2.3.1 Types of Manual Testing

There are several types of manual testing techniques:

Unit Testing: Individual components of the software are tested to ensure that they function properly. This technique is performed by developers to quickly identify problems but cannot verify component interactions (after [Hooda and Singh Chhillar, 2015]).

Regression Testing: This is performed mainly during the maintenance phase to ensure that new code does not affect existing functionalities. This technique can also be automated (after [Halani, Kavita and Saxena, 2021]).

Acceptance Testing: It is a test level that focuses on determining whether to accept the system (c.f. [Istqb.org, 2024]). It is done from the client's perspective. Here, the customers conduct tests to ensure that the software meets their standards. Feedback from these tests influences the final delivery of the product (after [Hsia et al., 1994]).

Alpha and Beta Testing: Alpha testing is a type of acceptance testing performed in the developer's test environment by roles outside the development organization (c.f. [Istqb.org, 2024]). It is performed internally by the development team before releasing the product to customers. Beta testing is conducted by real users in a real environment to identify remaining issues after release (after [Halani, Kavita and Saxena, 2021]).

Black Box Testing: It focuses on testing the functionality of the application without knowing the internal workings. Black Box Testing techniques include Boundary Value Analysis, Graph-Based Testing, Worst-Case Testing, and Robustness Testing (after [Jan et al., 2016]).

White Box Testing: It involves testing the internal structures or workings of an application. White Box Testing techniques include testing of all modules within the software (after [Nidhra et al. 2012]).

2.3.2 Advantages and Disadvantages of Manual Testing

Manual testing of software has both advantages and disadvantages. Manual testing is generally cheaper compared to automated testing. It provides more accurate user interface feedback and identifies issues that automated tests might miss. Manual testing techniques are suitable for scenarios where automation is not technically feasible or cost effective. It also allows testers to perform ad-hoc testing without pre-scripted test cases (after [Halani, Kavita and Saxena, 2021]).

On the other hand, manual testing is usually very time consuming, especially for large scale regression testing. It is also less reliable as it is prone to human error and may miss defects due to fatigue or oversight. Moreover, manual testing techniques are non-reusable as test cases are not recorded for future use. This makes repeating tests laborious. Manual testing is not suitable for some types of testing, such as performance or load testing, where automation is required for accuracy (after [Halani, Kavita and Saxena, 2021]).

2.4 Automated Testing

Automated testing is a software testing technique that uses specialized software to manage the execution of tests and compare actual results with expected or predicted results (after [Techopedia.com, 2019]). This process is performed automatically with minimal or no input from the test creator. The primary goal is to automate repetitive but necessary tasks in a formalized testing process already in place (after [Bezbaruah, Pratap and Hake, 2020]). Test automation is particularly beneficial for performing tests that are too complex or tedious to be done manually (after [Merina, Anggraini and Hakiem, 2018]).

2.4.1 Automated Testing Tools

Automated testing tools can be categorized based on how they work, their development phase and their functionality. The main types of automated testing tools are discussed below:

Load Testing Tools: Load testing is a method of to evaluate the performance of a system that allows the tester to analyze how well it performs under stress (after [Abbas, Sultan and Bhatti, 2017]). Automated load testing specifically aims to identify both functionality and performance issues when the system is under load. This type of testing is particularly applicable to websites and frameworks (after [Alferidah and Ahmed, 2020]). Some of the load testing tools are mentioned below:

- Apache JMeter: It is an open-source tool used for load testing and regression testing. It provides accurate results and supports GUI testing. It takes more time to set up as it requires many steps and configurations (after [Abbas, Sultan and Bhatti, 2017]).

- LoadRunner: It identifies performance bottlenecks and can simulate multiple users to check network performance. LoadRunner has a high licensing cost which can result in a huge expense for any organization (after [Abbas, Sultan and Bhatti, 2017]).

- Siege: It detects system performance under load and is faster to set up but it provides limited results. Siege can sometimes generate inaccurate results, which may affect the reliability of the performance data collected during testing (after [Abbas, Sultan and Bhatti, 2017]).

- Microsoft Visual Studio (TFS): It supports load testing alongside project and code management. The problem is that Microsoft Visual Studio (TFS) only supports Windows operating systems. This limits usability for teams working in multiple operating system environments (after [Abbas, Sultan and Bhatti, 2017]).

Acceptance Testing Tools: Acceptance testing is conducted to verify whether a software system meets the requirements and provides the required functionality. There are various tools available for acceptance testing but the most commonly used is FIT (Framework for Integrated Test) (after [Negara and Stroulia, 2012]).

- FIT (Framework for Integration Test): It is used for acceptance testing and allows customers to create test cases and classes that are automatically tested (after [Negara and Stroulia, 2012]).

Functional Testing Tools: Functional testing is used to verify whether a website or web application is correctly performing all necessary functions or not (after [Alferidah and Ahmed, 2020]). Some of the popular functional testing tools are mentioned below:

- Selenium: It is a popular automated functional testing tool that runs tests directly in the browsers and supports multiple platforms (after [Holmes and Kellogg, 2006]). This makes tests easy to set up, record, edit, and debug (after [Nagowah and Roopnah, 2010]). Selenium tests can be fragile in case of dealing with GUI changes (after [Holmes and Kellogg, 2006]).

- FitNesse: It combines a testing tool, wiki, and web server to validate functionality and acceptance. It provides automatic comparisons to expected outcomes. Keeping the tests organized can be challenging in it (after [Holmes and Kellogg, 2006]).

Regression Testing Tools: Automated regression testing is similar to automated functional testing which verifies the functionality of the system and ensures that newly added features of the system do not introduce errors or bugs into the existing system (after [Alferidah and Ahmed, 2020]). Some of the regression testing tools are listed below:

- IBM Rational Functional Tester: It is used for automated regression and functional testing. The tool uses Java as the scripting language. This means that non-IT employees or those without programming knowledge cannot use the tool effectively and it is less accessible for those without some technical expertise (after [Nagowah and Roopnah, 2010]).

- Quick Test Professional (QTP): It is an automated regression testing tool that uses Visual Basic (VB) and can be used for both manual and automated testing. QTP can require significant effort in maintaining test scripts (after [Nagowah and Roopnah, 2010]).

- Sahi: It is a web application testing tool that supports script recording and playback. It is developed in Java and JavaScript (after [Nagowah and Roopnah, 2010]).

Graphical User Interface (GUI) Testing Tools: Various Tools Including popular tools like QTP, Abbot, Selenium, Rational Functional Tester (RFT), WinRunner, SilkTest, and IBM Rational Robot are used for GUI testing (after [Miao and Yang, 2010]). These tools validate GUI functionality against system specifications. GUI will be discussed in more detail in the following subchapter.

End-to-End Testing Tools: Automated end-to-end testing helps detect regressions early in the development process and this lays a solid foundation for future system changes. This type of testing includes various methods to verify the correctness of the application from the user's perspective. Tools for end-to-end testing include Selenium and Robotium (after [Vasilyev et al., 2017]).

- Robotium: It is an end-to-end automated testing tool that provides guidance to developers on how to address testing tasks (after [Vasilyev et al., 2017]). Robotium is not able to handle different applications in one test (after [Root Info Solutions, 2017]).

2.4.2 Test Automation Frameworks

A test automation framework is a structured set of guidelines and processes for creating and designing test cases. It provides a systematic way to generate, execute, and manage tests (after [Merina, Anggraini and Hakiem, 2018]). Choosing the right framework is important to maximize the efficiency and effectiveness of testing (after [Bhargava, 2013]).

Four primary challenges must be addressed when designing an automated testing framework to detect bugs related to user interaction features. They are automating the exploration of mobile applications, automating the incorporation of user-interaction features during exploration, automating the analysis of bug identification, and utilizing historical bug information to enhance bug detection (after [Méndez-Porras et al., 2020]). Figure 4 provides an overview of a sample automated testing framework for mobile applications leveraging user interaction features and historical bug data.

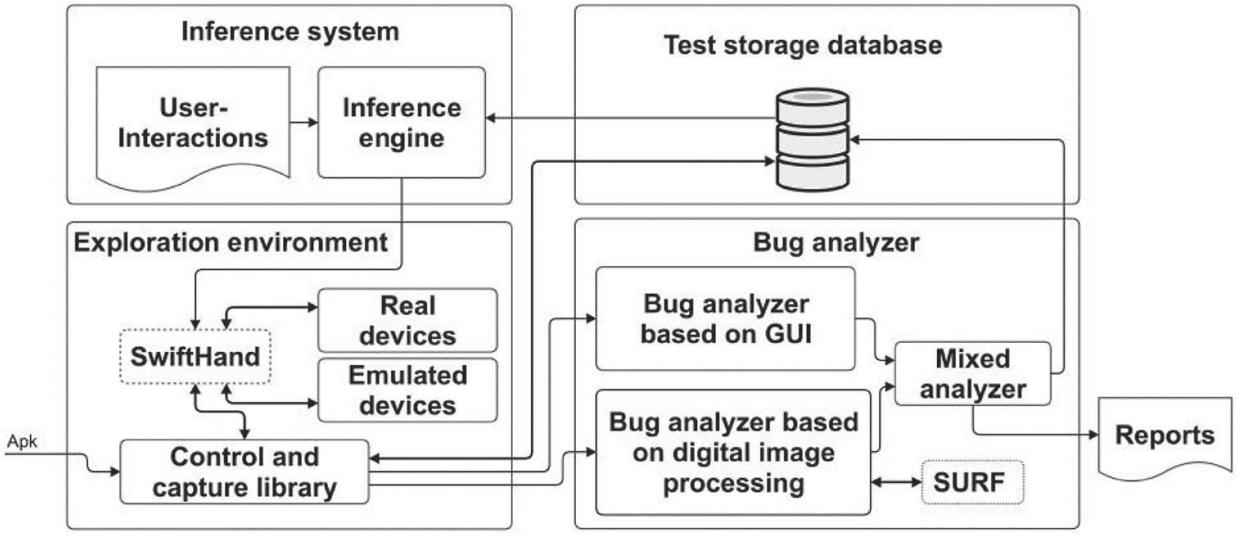

Figure 4: Components of the automated testing framework (from [Halani et al., 2021])

The exploration Environment component handles the automated exploration of the mobile application. It launches the app using the APK file and navigates through different screens to achieve maximum branch coverage (after [Méndez-Porras et al., 2020]). A model of the application is created for automated exploration using SwiftHand (after [Choi et al., 2013]). It was modified to include user-interaction features

and capture images and GUI information before and after interactions to detect potential bugs (after [Méndez-Porras et al., 2020]).

The inference engine introduces user interaction features during automated exploration. Two strategies are implemented here. The first one is the Random strategy where user interaction features are introduced randomly. The second one is the Frequency of Bugs by Widget strategy. Here, historical bug data is analyzed to identify widgets that are prone to bugs. Then corresponding user interaction features are introduced to find similar bugs in new applications (after [Méndez-Porras et al., 2020]).

The Bug Analyzer component analyzes the data from user interactions to identify bugs. It uses digital image processing and GUI information to detect inconsistencies before and after user interactions (after [Méndez-Porras et al., 2020]). The interest points detector (SURF) (after [Bay et al., 2008]) finds interest points in images and analyzes GUI changes to identify potential bugs. The information is stored in the historical bug repository if a bug is detected (after [Méndez-Porras et al., 2020]).

Test Storage Database stores all test cases, historical bug information, and GUI data before and after user interactions. It maintains a comprehensive record of event sequences and interactions to improve future bug detection and analysis (after [Méndez-Porras et al., 2020]).

Figure 5 illustrates the flowchart of Automated Testing Framework. The Application Explorer launches the app on actual or emulated devices using APK file and navigates through its screen. The Inference Engine adds user interaction features during exploration. The Analyzer then receives image pairs and GUI information which are captured before and after each interaction (after [Halani, Kavita and Saxena, 2021]).

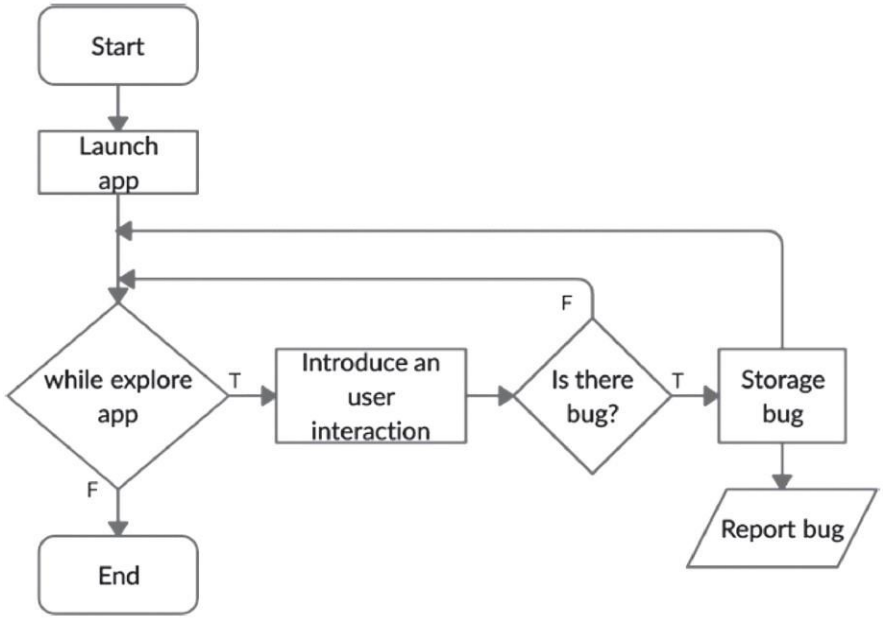

Figure 5: Flowchart of the automated testing framework (from [Halani, Kavita and Saxena, 2021])

There are many different types of test automation frameworks, each with its own advantages and disadvantages. A brief discussion of each framework is discussed below:

Linear Automation Framework: They use record and playback functionality to enable testers to record actions and replay them to repeat tests. Test scripts can be generated quickly, but these are not reusable and are difficult to maintain and extend due to hard-coded data (after [Hanna et al., 2014] and [smartbear.com, 2024]).

Modular-Based Automation Framework: Based on abstraction, independent scripts are created by dividing the application into logical units. Modules are used hierarchically to build large test cases. This makes maintenance and scaling easier, but requires programming skills and hard-coded data (after [Hanna et al., 2014] and [smartbear.com, 2024]).

Library Architecture Testing Framework: It is similar to modular-based framework but divides the application into procedures and functions stored in libraries. This increases modularization and makes maintenance easier and cost efficient. It still requires hard-coded data and technical expertise (after [Berihun et al., 2023]).

Data-Driven Testing Framework: Data-Driven testing is a scripting technique that uses data files to contain the test data and expected results needed to execute the test scripts (c.f. [Istqb.org, 2024]). This framework separates test script logic from test data and stores data externally in key-value pairs. This reduces the number of scripts required which saves time and increases flexibility. However, this requires skilled programmers to manage the links between scripts and data sources (after [smartbear.com, 2024] and [Hayes, 2004]).

Keyword-Driven Testing Framework: Keyword-Driven testing is a scripting technique in which test scripts contain high-level keywords and supporting files that contain low-level scripts that implement those keywords (c.f. [Istqb.org, 2024]). It uses keywords stored in external files to dictate actions, combining the benefits of data-driven testing with minimal scripting skills. Keywords offer high code reusability but the framework is complex and needs a good automation expert (after [Hanna et al., 2014], [smartbear.com, 2024] and [Jain and Sharma, 2012]).

Hybrid Test Automation Framework: It combines elements of the above frameworks to leverage their strengths and mitigate their weaknesses. It provides a flexible approach suitable for the application under test (after [Jain and Sharma, 2012]).

Some of the well-known frameworks for reliability are Espresso, Calabash, and Appium. Espresso is primarily developed for Android UI testing by Google (after [Merina, Anggraini and Hakiem, 2018]). It

supports parallel execution and uses Java for scripting. The problem with Espresso is that it is limited to Android and does not support iOS. Calabash supports both Android and iOS platforms (after [Pareek et al., 2015]). It uses Ruby for scripting (after [Pareek et al., 2015]) and demonstrated the best overall performance (after [Kulkarni and Soumya A, 2016]). However, it does not support parallel execution (after [Pareek et al., 2015]). Appium supports Android, iOS, and mobile web applications. It uses various programming languages for scripting (after [Anusha and Saravanan, 2017]). It also supports parallel execution (after [Pareek et al., 2015]). However, it is more complex to set up compared to the other frameworks (after [Merina, Anggraini and Hakiem, 2018]).

2.4.3 Taxonomy of Mobile Automation Testing

The taxonomy of testing is a systematic classification to organize different features of the testing process into distinct categories. This framework helps to better understand and manage different elements related to testing based on the idea of software and mobile application development. The details of the taxonomy of mobile automation testing are shown in Figure 6.

Test Objectives: The major concern in mobile automation testing is to satisfy both functional and non-functional requirements. Reusability avoids creating new test cases for each application. It emphasizes on unique functionality (after [Berihun et al., 2023]). Efficiency focuses on the appropriate utilization of test resources and aims to reduce the time and cost of testing. Independent test case generation, identifying screen compatibility issues and converting bug reports into test cases are the techniques to improve efficiency (after [Fazzini, 2018] and [Machiry et al.,2013]). Functionality makes sure that an application meets its user expectations, reduces errors and maintains quality (after [Mohammad et al., 2019]). Reliability is about ensuring that an application works without failure for a certain period of time. Automated testing frameworks are essential for creating reliable applications to effectively identify and resolve bugs (after [Lovreto et al., 2018] and [Chauhan and Singh, 2014]). Compatibility checks whether an application runs on different hardware and operating system platforms (after [Fazzini, 2018] and [Machiry et al.,2013]). Scalability measures the ability of an application to handle increases in user traffic, data size and transaction frequency (after [Mu et al., 2009] and [Tirodkar and Khandpur, 2019]). Performance testing evaluates the speed, response time and stability of mobile applications. This treats the issues which are important for user satisfaction like slow user interaction and poor responsiveness (after [Sinaga et al., 2018] and [Kim et al., 2009]).

Test Techniques: Test techniques in mobile automation testing deals with various approaches and test types. Regarding Test approaches, Linear testing is a testing technique that uses the record or playback method. Here, testers enable the recording mode on the testing tool while executing actions on the application being tested (after [Hanna et al., 2014] and [Kulkarni and Soumya A, 2016]). Data-driven testing

uses tables to store test data so that a single test script can run tests for all the data in the tables without hard-coding any environment settings. This approach is recommended for its reusable functionality and low maintenance costs. It is supported by frameworks that use the Appium library (after [Hanna et al., 2014] and [A. Alotaibi and J. Qureshi, 2017]). Keyword-driven testing expands data-driven testing technique by storing test data and keywords in external files. This makes test script creation and maintenance easier and more efficient. Hybrid testing combines different frameworks to merge the advantages and lessen the disadvantages of each (after [Berihun et al., 2023]). Model-driven testing automatically to generates code from a model. It also automates test case execution from a model that combines model-driven and domain-specific modeling language (DSML) concepts to improve mobile application quality (after [Marín et al., 2016] and [Ridene and Barbier, 2011]).

Regarding test types, Black-box testing is performed without knowledge of the internal code structure. It is used to assess system functionality by providing inputs and observing outputs and reveals system reactions to various situations (after [Tirodkar and Khandpur, 2019] and [Vajak et al., 2018]). White-box testing requires complete knowledge of the program structure. It is exhaustive, time-consuming and usually applied during unit testing (after [Bansal, 2014] and [Chauhan and Singh, 2014]). Grey-box testing combines aspects of both black-box and white-box testing. It uses detailed design documents and requirements information (after [Jamil et al., 2016] and [Choudhary et al., 2015]). Regression testing focuses on identifying negative side effects of code changes and re-executing affected tests (after [Berihun et al., 2023]).

Test Challenges: Mobile test automation faces several challenges when using current frameworks. As a result, researchers need to develop novel approaches. The main challenges are complexity, maintenance cost, time, and fragmentation (after [Berihun et al., 2023]). The increasing size and complexity of mobile applications are causing significant quality issues (after [Wu et al., 2013]). The requirements for high-quality apps complicate application design and testing. Although keyword-driven testing is known for reducing complexity by applying different levels of keyword abstraction, some research suggests that it actually increases complexity (after [Berihun et al., 2023] and [Hanna et al., 2014]). Mobile app maintenance costs account for 15-20% of overall app development costs (after [App, 2024]) but test automation frameworks can help to reduce these costs. Fixing defects, improving performance, upgrading and maintaining robustness are some of the maintenance works (after [Cherednichenko, 2021]). Despite available automation tools and approaches, mobile testing remains a time-consuming task (after [A. Alotaibi and J. Qureshi, 2017]). Test automation itself requires development effort and significant time (after [Berihun et al., 2023]). Proper use of automation tools can decrease testing time and make the process more efficient (after [Salam et al., 2022]). Fragmentation is another major challenge when users use different operating system versions of Android devices (after [Garousi and Elberzhager, 2017]). It is difficult to identify all the

issues that end-users might face. Therefore, fragmentation makes testing complicated (after [Berihun et al., 2023]).

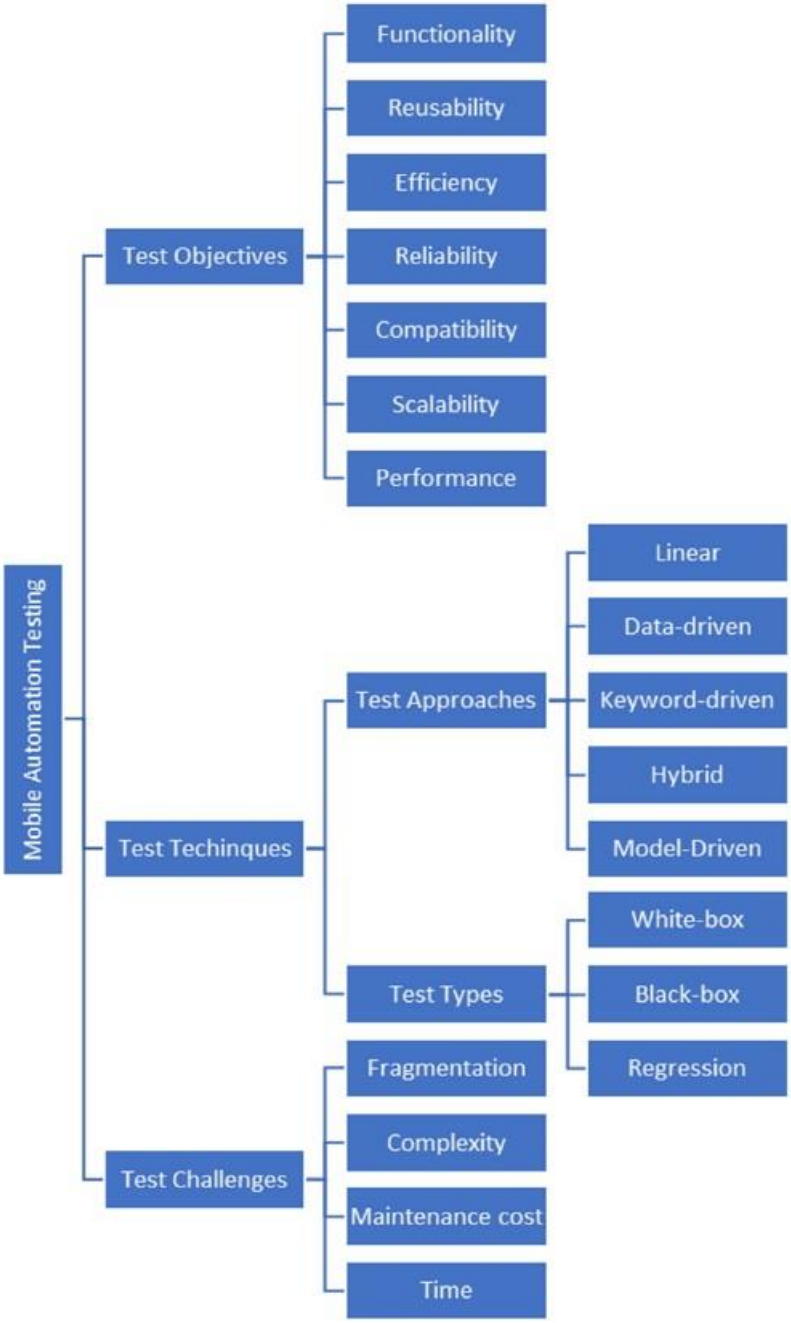

Figure 6: Taxonomy of mobile application testing (from [Berihun et al., 2023])

2.4.4 Benefits of Automated Testing

There are some significant benefits of using automated testing methods. Some of the main advantages are described below:

Repeatable Tests: Automated testing allows developers to run the same tests multiple times ensuring consistency and reducing the risk of unnoticed errors (after [Yalamanchili and Kumari, 2016]).

Reduced Time and Cost: Automation replaces time-consuming manual testing methods. It enables faster and more efficient application releases. The cost of resource allocation is also reduced significantly by using automated testing (after [Lavingia et al., 2024]).

Increased Test Coverage: Automation enables testing for a wider range of scenarios, environments and configurations. It is very crucial for the diverse shapes and sizes of Android applications (after [Lavingia et al., 2024]).

Improved Accuracy and Reliability: Automation ensures the quality, reliability, and performance of increasingly complex mobile applications by eliminating human errors (after [Silva et al., 2016]).

2.4.5 Recent Research in Automated Testing

Researchers are continuously trying to develop and improve automated testing methods. Some of the significant ongoing research areas of automated testing are described below:

Integration with Cloud-Based Testing Platforms: Cloud-based testing provides scalability and cost-effectiveness by allowing tests to be run on multiple devices simultaneously. This can significantly reduce testing time and expenses (after [Lavingia et al., 2024]).

Artificial Intelligence (AI) and Machine Learning (ML): Using AI and ML algorithms can improve automation by generating test scripts, optimizing test execution, detecting anomalies, modeling user behavior and creating realistic test scenarios (after [Lavingia et al., 2024]).

Integration with Continuous Integration/Continuous Deployment (CI/CD) Pipelines: Integrating automated testing with CI/CD pipelines can ensure early detection of faults in the development cycle. This can improve the overall efficiency of the development process (after [Lavingia et al., 2024]).

2.5 Graphical User Interface (GUI)

In the modern world, almost all mobile applications have Graphical User Interface (GUI) front end. A GUI can respond to user interactions such as mouse movements or menu selections. It provides an interface to the underlying application code and interacts with the code via messages or method calls (after [Memon, 2002]). A recognized approach to testing GUI-based applications is to perform a sequence of user input events on GUI widgets through the GUI. This process is referred to as GUI testing in the literature (after [Memon et al., 1999]).

With the rapid growth of mobile applications, it has become very important to ensure their quality through effective testing. One of the most widely used methods to test mobile applications is Graphical User Interface testing (GUI testing) (after [Hu and Neamtiu, 2011]). Mobile applications are event-driven systems that primarily interact with users through a GUI (after [Zhu et al., 2015]). Android applications have complex GUIs and require thorough testing to ensure high quality. GUI testing involves user interactions to achieve specific tasks within the app which makes it essential for app success in stores like Google Play (after [Gu and Rojas, 2023]). GUI-based testing on smartphones is crucial because the primary input is user gestures. Developers look for fast and automated GUI testing methods to ensure software quality across various platforms (after [Hsu et al., 2017]). The GUI of an Android application is tightly integrated with its business logic. This makes GUI testing crucial to verify application functionality and ensure quality. Automated GUI testing is preferred because it requires less time and effort than manual testing (after [Khan and Bryce, 2022]).

An important aspect of any GUI testing technique is the approach used to generate the test data. There are several test data generation techniques for GUI testing. Some of the popular GUI testing techniques are Capture/Replay, Model-based and Random testing. There are also some less commonly used methods such as symbolic execution, formal methods and statistical analysis (after [Banerjee et al., 2013]). Capture/replay techniques involve recording user interactions and converting these user application interactions into test scripts. These test scripts can be automatically replayed (after [Hicinbothom and Zachary, 1993]). These techniques are commonly used for regression testing. They are highly popular in various fields like web application and desktop application testing (after [Elbaum et al., 2005]). Model-based techniques offer an alternative approach to generating test data. It depends on a model of the GUI to generate test cases. The most widely used GUI models for GUI testing are event flow graphs (EFG) and finite state machines (FSM). Usually, these models should be abstracted from the application through resource intensive activities (after [Banerjee et al., 2013]). Another well-known method for test data generation is Random testing. It is the simplest technique for selecting test cases. They are randomly chosen from the input domain based on certain distributions. Random testing is widely used in various fields like hardware testing and protocol testing (after [Hamlet, 2002]).

2.5.1 Automated GUI Testing Tools and Frameworks

Automated GUI testing tools help ensure that mobile applications display correctly on different types of devices and screen sizes. These tools simulate user gestures and validate the GUI changes through a test oracle (after [Chu and Lin, 2018]). Some of the Automated GUI Testing Tools are mentioned below:

Sikuli: It is an open source automated GUI testing tool that uses screenshots for verification. It redirects the screens of the tested app to the PC. The test oracle compares the redirected screen to prerecorded screenshots to verify correctness (after [Yeh et al., 2009]).

SPAG-C: It uses image comparison methods to verify GUI display. At first, a screenshot is taken with an external camera. Then three image comparison methods (SURF, image histogram and template comparison) are used to compare the captured screenshot with prerecorded screenshots (after [Lin et al., 2014]).

GUICOP: It uses GUI layout information instead of screen images for verification. It depends on the specifications of GUI components to determine correctness. It compares the actual GUI layout with the expected layout based on specifications (after [Zaraket et al., 2012]).

FLAG: It automates test case generation, user gesture simulation and screenshot verification. It uses UI Automator viewer to analyze GUI components and generates test cases for all possible GUI operations. It simulates user gestures based on the test cases and saves the screenshots generated from each simulated gesture. It then reproduces gestures on the Device Under Test (DUT) and compares new screenshots with stored images. It uses Optical Character Recognition (OCR) for text and SURF for images (after [Chu and Lin, 2018]).

Espresso and UI Automator are two of the most popular GUI Testing Frameworks. Espresso is recommended for smaller projects due to its simple API and synchronization capabilities. Espresso can only interact within the current application under test (AUT) (after [Gu and Rojas, 2023]). UI Automator is suitable for cross app interactions and system level operations. It needs more lines of test code and manual handling of wait actions for window transitions (after [Coppola et al., 2017]). UI Automator is often combined with Espresso to cover a broader range of test cases (after [Zelenchuk, 2019]).

2.5.2 Challenges in Android GUI Testing

There are some challenges for using Android GUI testing. Android applications have a large number of possible events and event sequences. This makes comprehensive testing very difficult. A large event space exponentially increases the number of event combinations (after [Khan and Bryce, 2022]).

Another challenge occurs because of the diversity of screen sizes and OS versions. Android applications must function for various devices with different screen sizes and OS versions which makes GUI testing complex (after [Khan and Bryce, 2022]). Traditional tools like Sikuli (after [Yeh et al., 2009]) and SPAG-C (after [Lin et al., 2014]) depend on static screenshots. They are limited by screen size differences which creates problem in accurately determining test results (after [Khan and Bryce, 2022]).

Another challenge is caused by dynamic GUIs. Modern Android apps contain numerous dynamically constructed GUIs. This makes accurate behavior modeling challenging. Moreover, state explosion problems can occur for handling non-deterministically changing GUIs. This also makes the testing process complicated (after [Baek and Bae, 2016]).

2.6 Company Overview: Secusmart GmbH

Secusmart GmbH is a subsidiary of BlackBerry Limited. It specializes in secure mobile communications. It was established to protect confidential information. Secusmart's main mission is to ensure secure and efficient communication for businesses and governmental organizations. The company's expertise in encryption technology provides robust security solutions customized to the needs of the modern mobile work environment.

Secusmart GmbH was founded to meet the growing demand for secure communication solutions. Secusmart recognized that advanced security measures are needed for growing technological advances and cyber threats. A global leader in secure communications BlackBerry Limited acquired Secusmart in 2014 to further enhance its capabilities and reach.

Secusmart provides a wide range of solutions for protecting voice and data communications. Key offerings are described below:

SecuSUITE for Government: It is a complete solution that can secure mobile communication for governmental agencies. It provides protection against espionage and cyber-attacks.

SecuVOICE: It is a highly secured solution for voice encryption. It provides privacy for confidential conversations.

SecuSUITE for Samsung Knox: Samsung Knox's secure platform is integrated with Secusmart's encryption technology. This can provide a reliable mobile environment for businesses.

These solutions use advanced encryption technology to provide security for voice, message and data communications. Secusmart integrates with existing mobile platforms so that users can remain productive without compromising security.

Some of the applications which need continuous testing are SecuStore, SecuConnect, SecuFox, Contacts, Notes, Tasks, Inbox, SecuDrive, SecuOffice, Adobe Acrobat, SecuServices, SecuGallery, SecuSuite, SecuVoice, ESRI Fieldmaps, Netcloud, Horizon, Skype, Citrix, Wire, BWmessenger, SE-Netz etc. Among them, SecuSuite, SecuVoice and SecuStore are under test automation. Other applications are still being tested manually.

Secusmart serves a wide range of clients including government agencies and enterprises. The solutions designed by Secusmart can meet the strict security requirements of these sectors. It also ensures compliance with global security standards. Multiple industries of different sectors use Secusmart’s solutions. It demonstrates the company’s versatility and reliability in providing secure communication solutions.

Secusmart invests heavily in research and development to stay ahead of emerging threats. Secusmart ensures that its solutions can prevent all kinds of possible threats.

Secusmart is committed to protecting the confidentiality of its clients' communications. Secusmart’s aim is to provide secure, reliable, and user-friendly communication solutions. This enables clients to operate confidently in the digital world. Secusmart's values include a commitment to safety, innovation and customer satisfaction. This drives its business operations and strategic decision making.

Secusmart GmbH is one of the leaders in secure mobile communications and it is backed by the robust support of BlackBerry Limited. Secusmart continues to set the standard in the industry with its suite of security solutions and focus on client needs. The company’s ability to provide tailored and cutting-edge security solutions ensures that it remains a trusted partner for organizations worldwide.

For more detailed information about Secusmart GmbH and its offerings, please visit their official website at [Secusmart](https://www.secusmart.com/de).

3 Case Study and Conceptual Design

This chapter presents an in-depth case study and conceptual design of an automated testing tool. This tool is developed to improve the efficiency and reliability of smoke testing of Android devices. This tool was created to solve some significant challenges faced by a company that provides security features for Android devices. This chapter is structured to provide a detailed case study of the transition from manual to automated testing and then provide a thorough description of the conceptual design of the automated testing tool initially for smoke testing. This study includes the objectives and requirements of the design, the system architecture, the user interface design and the technical specifications.

3.1 Case Study

In our company, some employees need to conduct the testing daily. A business device is provided by Secusmart. The Secusmart phone, which is collaborated with Samsung Organization, is used as a testing device and also for meeting, sending emails, opening documents and other daily official activities.

People who are not dedicated testers are also using this device. A survey was conducted in July 2024 to see how frequently employees are performing testing on their business Android devices. Here in this Figure 7, it can be seen that most of the employees have to perform testing at least once a week. The dedicated testers need to perform testing daily. This is the reason why testing Android devices is considered as an important task in Secusmart.

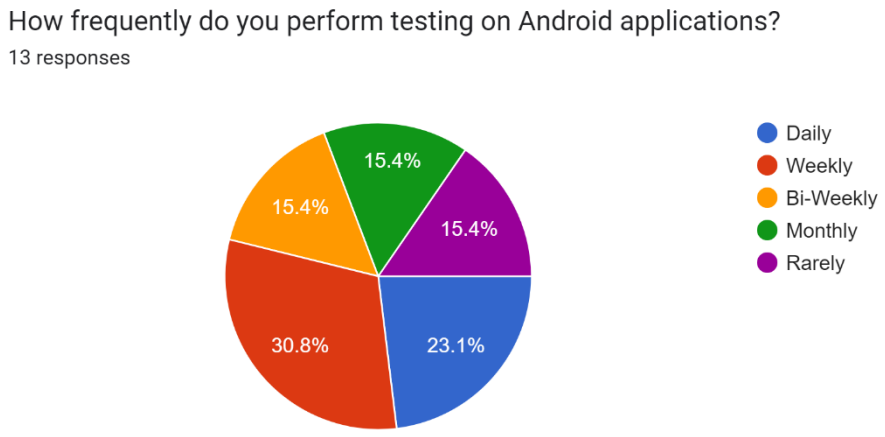

Figure 7: Frequency of testing Android devices

The initial state of testing within the company is mostly manual. The survey illustrated in Figure 8 shows that only 23.1% of testing is performed using test automation methods. The rest of 76.9% of testing still fully or partially depends on manual testing methods.

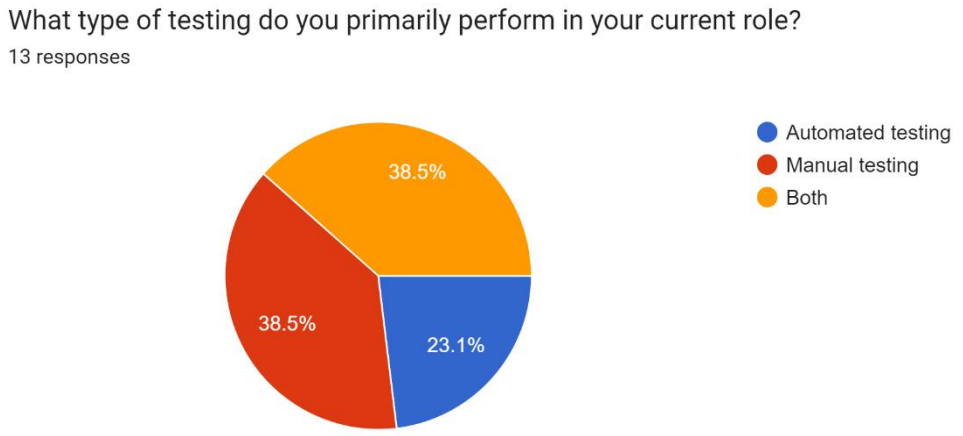

Figure 8: Type of testing performed in Secusmart

Currently at Secusmart, only two application are tested using test automation. For the other 37 applications, still manual testing is being used. Testers often face several challenges while performing these manual testings. The survey of Figure 9 shows some of the common challenges faced by the testers.

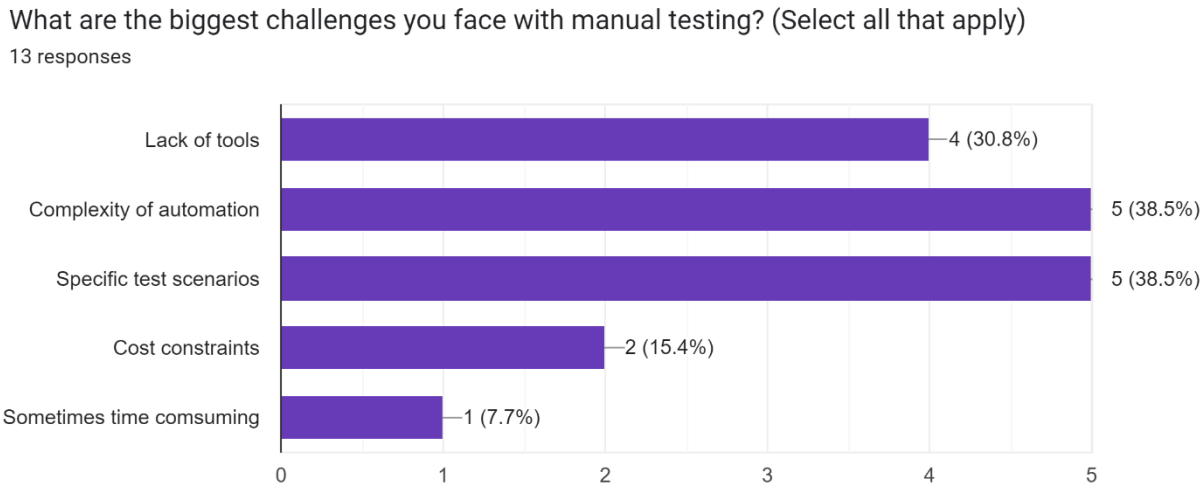

Figure 9: Challenges faced with manual testing

One of the major challenges of using manual testing is the lack of tools available. Another problem faced by the testers is when they want to merge a part of manual testing with some automation. This makes the process complicated. Testers have to manually execute predefined test cases on various devices. After executing the test case they need to document each step and observe the results. This process is time consuming and prone to human error. High resource consumption is a major issue as manual testing requires a lot of human resources. Multiple testers often work on different devices at the same time. Inconsistency is another major challenge with manual testing. Human error leads to variations in test

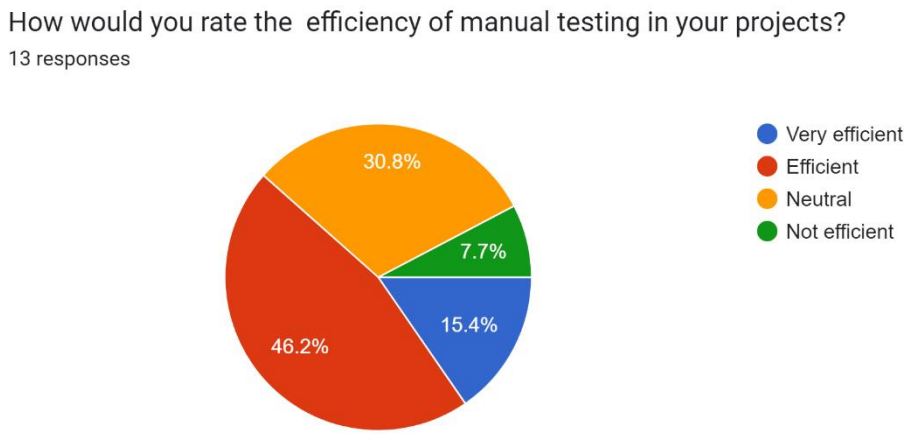

Figure 10: Efficiency of manual testing

execution and results. As a result, reproducing and verifying issues become more difficult. Another persistent problem is scalability. Testing on multiple devices and configurations is not only difficult but also highly inefficient. This increases delays and potential gaps in coverage. For these reasons, the efficiency of manual testing is considered to be very low by the testers as shown in the survey illustrated in Figure 10.

There are several steps for the current manual testing from preparing the test environment to documenting results. The steps of current manual testing of the 37 applications are shown in Figure 11. In the preparation phase, the testers need to identify test objectives, define test cases and set up a test environment. In the execution phase, testers have to execute test cases, document gained results and retest if it is necessary. Compiling results, documenting issues, creating tickets and preparing test reports are the tasks of the reporting phase. Finally, reports are reviewed and finalized after considering feedback in the review phase. This entire process is not only time consuming but also prone to human error.

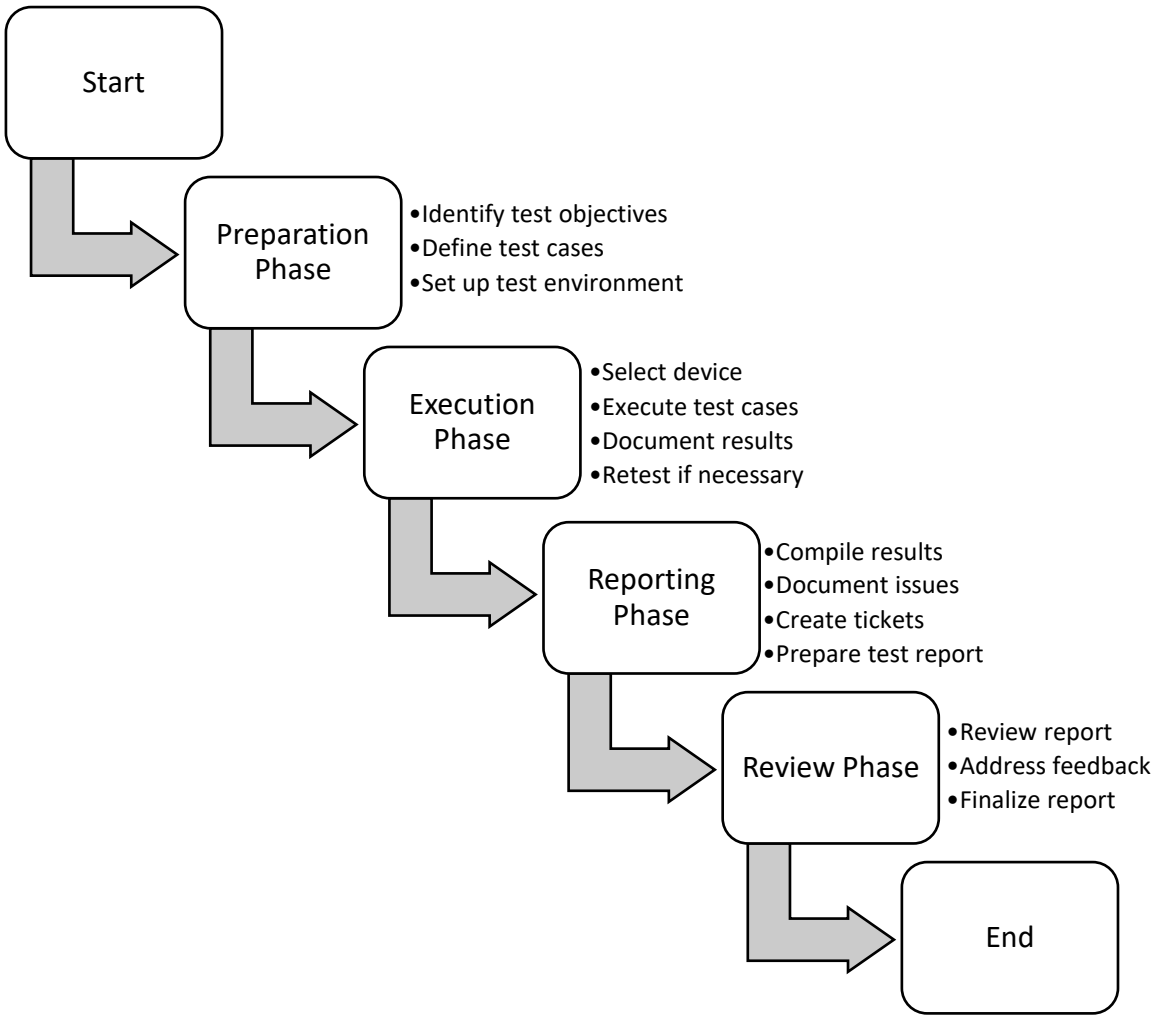

Figure 11: Steps of current manual testing

The need for a more efficient testing approach is evident after considering these limitations. Test automation is identified as the solution to solve these problems. The primary motivations for automation include the need for increased efficiency and automated testing can perform repetitive tasks much faster and with greater accuracy than manual testing. Test automation also should also minimize the risk of human error to provide consistent results. The ability to run tests on multiple devices simultaneously is also a major advantage. This improves coverage and ensures more effective testing.

Automation also should also create the potential to perform tests that are impractical manually. For example, large scale regression tests for multiple devices can be automated to ensure that new changes do not introduce bugs. Automated tools can execute these tests overnight or during off hours. This maximizes resource utilization and accelerates the development cycle. Automation also enables more advanced testing techniques like performance and load testing which are difficult to perform manually. As a result, Automated testing is identified to be more efficient in identifying bugs from the survey conducted with the testers. This can be seen from the survey illustrated in Figure 12.

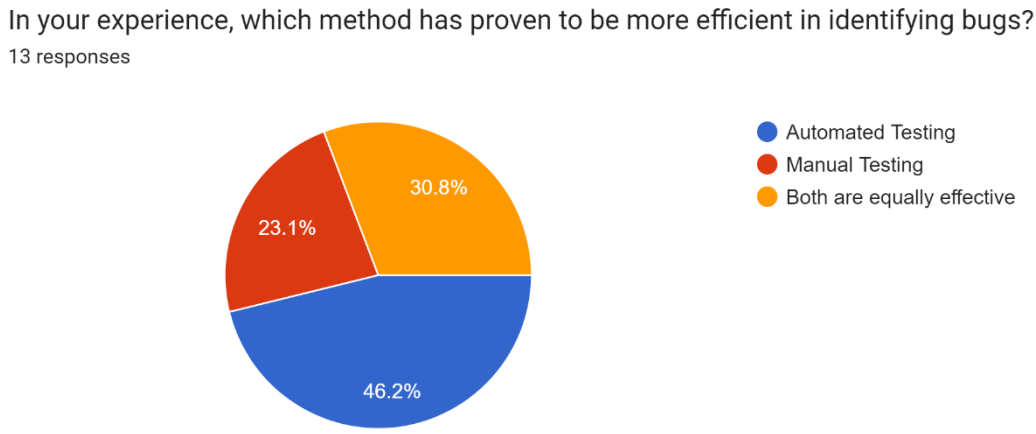

Figure 12: Efficiency comparison between manual and automated testing

There are already a lot of Automated testing tools available as mentioned in chapter 2. Some of them are already being used for testing two applications (SecuSuite and SecuVoice)) in Secusmart but these automated tools cannot be used for all other 37 apps because of their limitations like high setup time, significant licensing cost, inaccurate result generation, limitation to work in multi-OS environments, being fragile dealing with GUI changes, inability to handle different applications in one test and so on. Moreover, the company also has some specific acceptance criteria and standards that are not readily available in the existing testing tools. Considering these limitations and company requirements, it has been decided to build a new smoke testing tool dedicated only to the use the rest 37 apps of the company.

For example, Jenkins is one of the testing tools used for testing SecuSuite and SecuVoice. The component diagram of the current Jenkins tool is shown in Figure 13. Programmers upload codes the source code repository Git. Then Jenkins server pull the latest code from GIT and generates APK with the help of build system Gradle. After that, it deploys APK and runs the tests using Espresso test framework to the Android device. The results of the tests are forwarded back to the Jenkins and it generates test reports with the help of reporting tool Junit. Finally, Jenkins sends the notification via email to the tester.

This tool has several advantages like it can be easily integrated with CI/CD pipelines, it has automated build and deployment features and it has wide plugin system for various tools. Although it has many advantages, this tool cannot be used for testing the other 37 apps for some important limitations of this app regarding the company requirements. Jenkins needs a complex setup. It needs configuring the server, setting up build jobs, managing credentials and integrating with version control systems. This setup can be time-consuming. Moreover, Jenkins consume a lot of resources like CPU, memory and storage. Also, Jenkins cannot be exactly customized to specific testing of company apps. Finally, using Jenkins for 37 apps can be very costly

for the company. This is why we have decided to build a simpler and more effective tool for testing the remaining 37 apps.

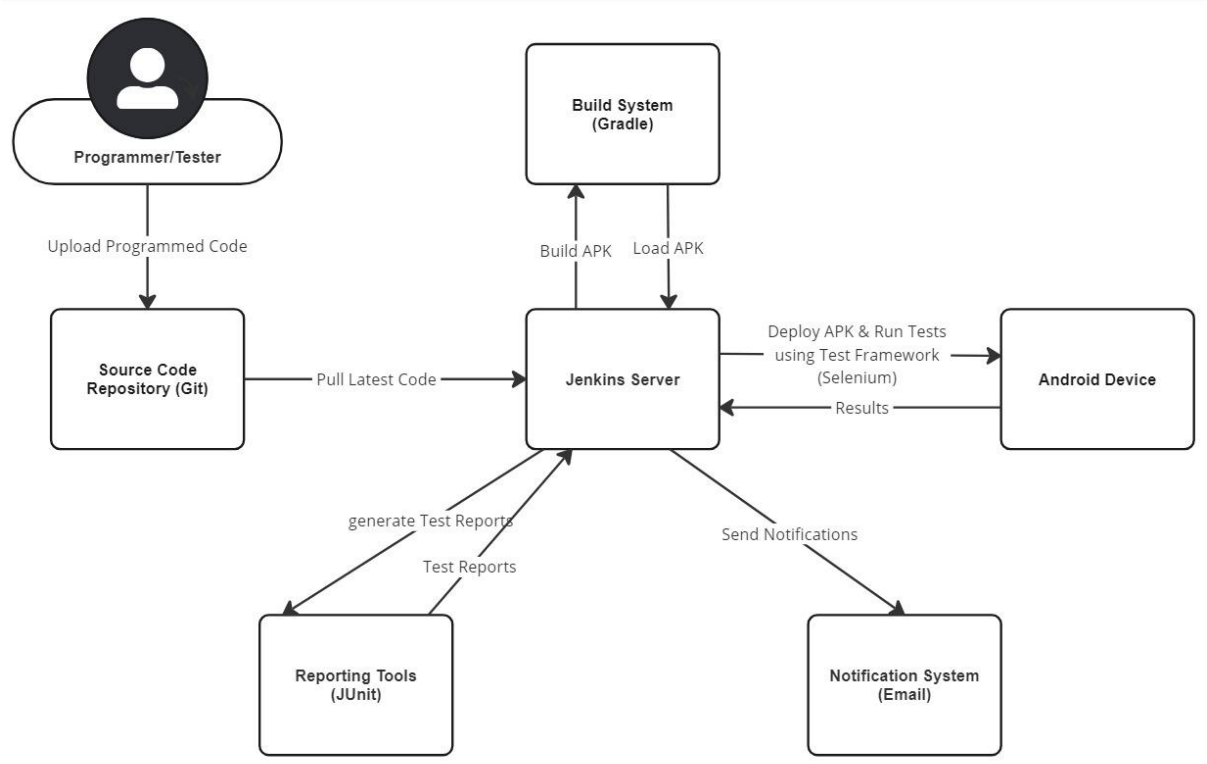

Figure 13: Component diagram for using Jenkins tool in Secusmart

3.2 Conceptual Design of the Automated Testing Tool

3.2.1 Design Objectives and Requirements

Improving efficiency and accuracy were the main design objectives of the automated testing tool. There were also some functional and non-functional requirements for the design based on research questions, Taxonomy of Mobile Application Testing described in Figure 6 and survey conducted on employees.

Functional Requirements: The prototype tool must be able to handle different Android devices of different configurations. These refer to the scalability and combability of the testing tool. The connected list of devices also needed to be dynamically updated. The tool must execute of ADB commands directly from the GUI. Also, it must provide real-time updates on the status of test execution. The tool must log all test executions including successful and failed tests. These should be used later for review and debugging. The

tool must allow users to select and execute predefined test suites. The tool also needed to have the quality of reusability. Finally, the tool must export logs and results in a format compatible with existing test case management systems.

Non-Functional Requirements: The tool must be user-friendly and designed to be usable by testers without technical expertise. This is essential for reducing learning time. This will also increase productivity. It must be robust and capable of handling unexpected errors such as device disconnections or command failures without crashing. It should comply with relevant industry standards and data security. The tool must provide secure storage of logs to prevent unauthorized access.

3.2.2 System Architecture

The system was planned to be divided into some components. They are described below:

GUI Component: The GUI component was planned to be developed using Tkinter. It would provide an interface for selecting devices and executing commands. It was designed to be user friendly. It would also generate real-time feedback to the users. This GUI component can be mapped to the User Interaction unit of the components of the automated testing framework showed in Figure 4.

Command Processor: The command processor was planned to manage the execution of ADB commands. It could handle the interaction with connected devices and process the commands selected by the user. This command processor component can be mapped to the Inference engine unit of the components of the automated testing framework showed in Figure 4.

Logger: The logger was designed to store logs of test execution. Analyzing these logs would help to detect bugs. This logger component can be mapped to the Bug Analyzer component and Test Storage Database unit of the components of the automated testing framework showed in Figure 4.

3.2.3 GUI Design and User Interaction

The design of the GUI was planned to keep the interface simple to reduce the learning time for the new users. Clear prompts were planned to set up to guide the users in each step of the testing process. Real time feedback was planned to keep them informed about the status of test execution.

The plotted functionalities of the GUI were device management, command execution, real time status updates and log viewing. It was designed in such a way that the users could select and manage connected devices. The interface would display a list of available devices to allow users to choose one of them. Users could execute predefined ADB commands and custom scripts. The GUI was structured to provide an easy way to select commands and start their execution. The GUI was intended to provide real time updates on

the status of test execution. This was intended so that the users were always aware of the current state of the process. Users could view and export detailed logs of test execution. This functionality would be very important for analyzing test results and diagnosing issues.

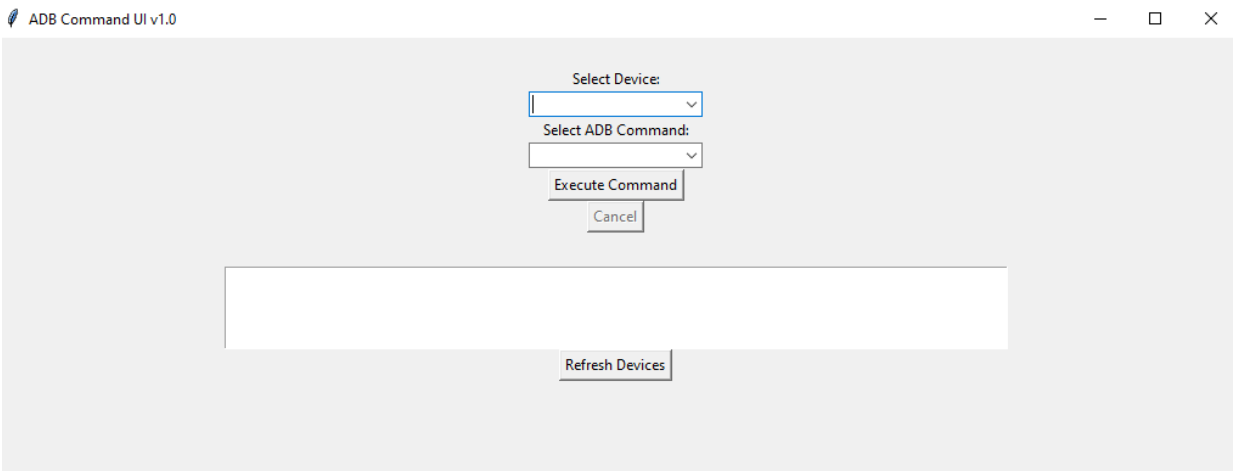

Figure 14: Designed GUI interface

The GUI would include several drop-down menus and buttons. Some of the menus planned to be implemented are discussed below with their intended functionality:

Select Device: It allows testers to choose the connected device to run tests.

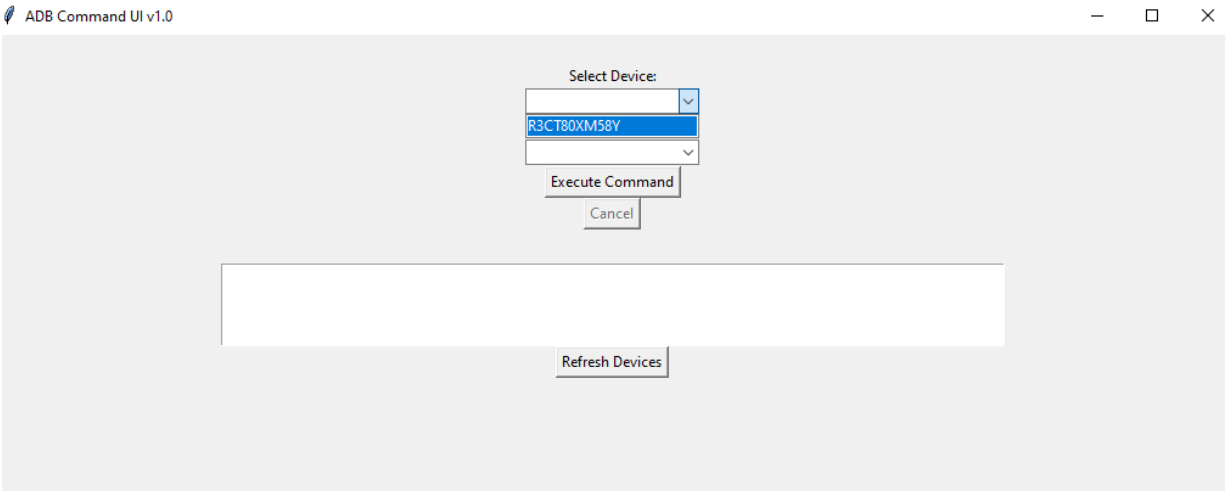

Figure 15: Select Device option interface

Select ADB Command: It provides options to run specific tests, sanity suites, test suites and other command executions.

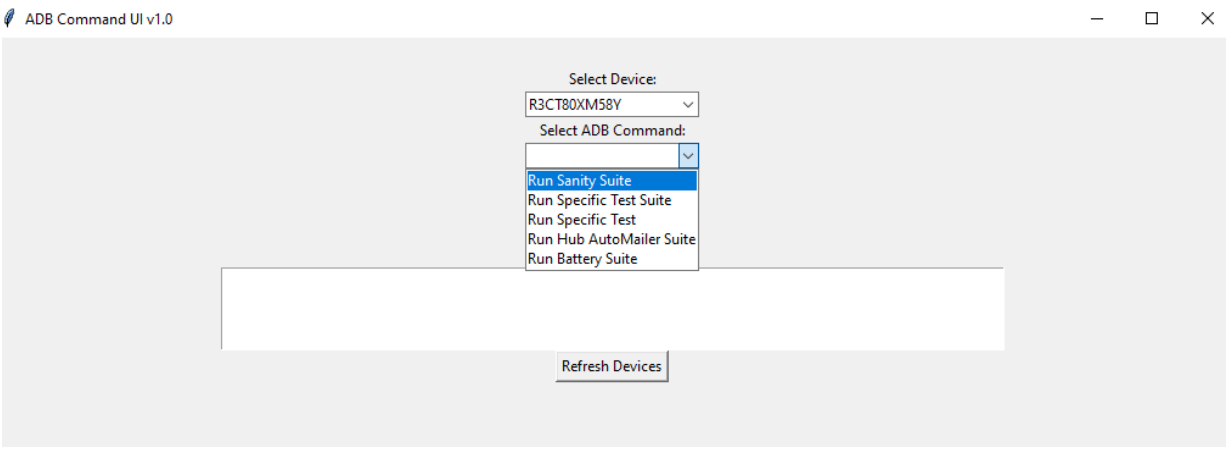

Figure 16: Select ADB command option interface

Select Test Suite: It offers a list of predefined test suites such as Hub, WebexTest, WireTest and more.

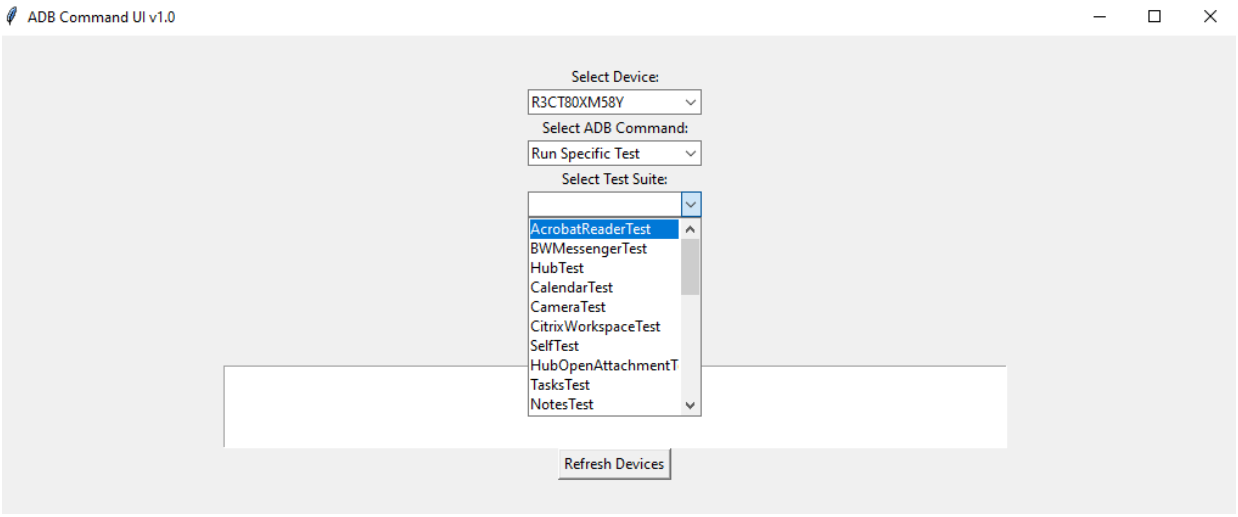

Figure 17: Select test suite interface

Select Additional Test: It allows the selection of specific tests within a suite such as testSendReceive and testSendReceive_duration.

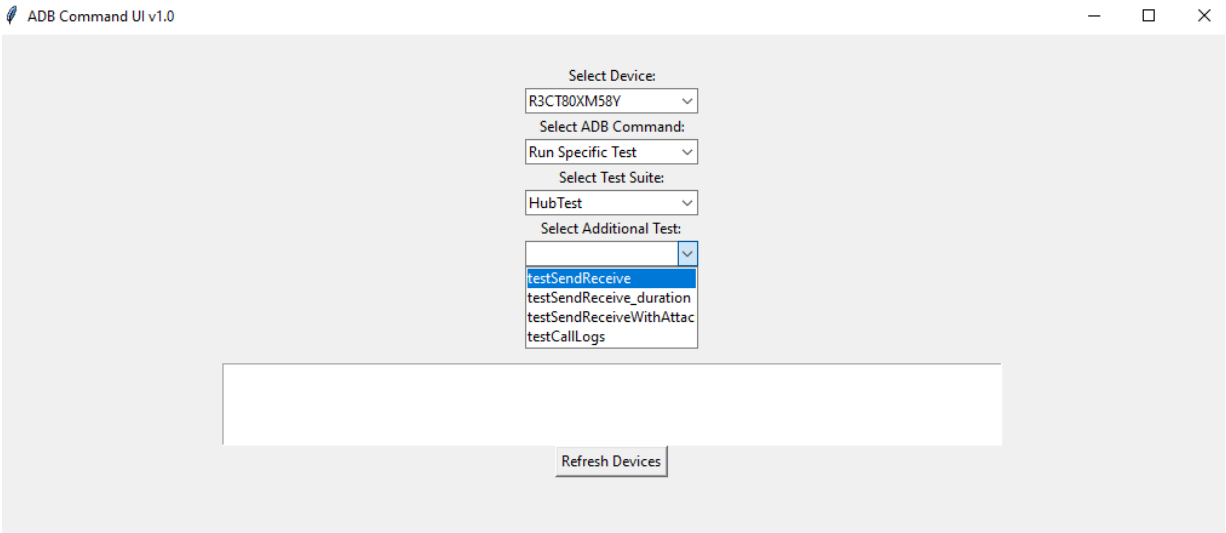

Figure 18: Select additional test interface

Execute Command: It initiates the selected test on the chosen device.

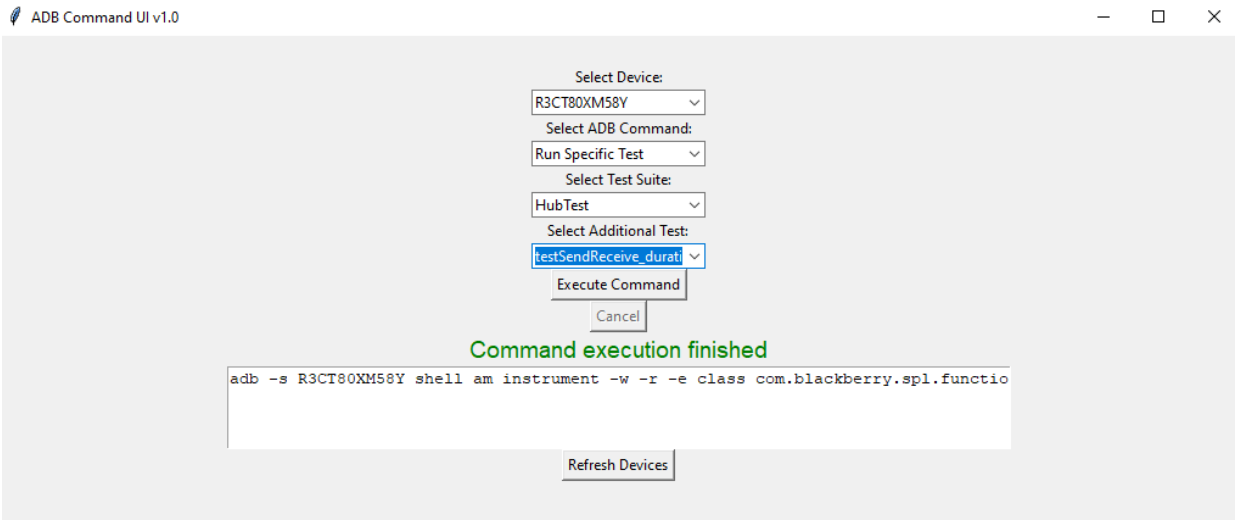

Figure 19: Execute command interface

Refresh Devices: It refreshes the list of connected devices.

The user interaction flow was designed to be easy to understand so that the users can easily navigate the tool. This flow would allow the testers to focus on the core task without being distracted by a complex interface.

3.2.4 Technical Specifications

Technical specifications for the tool included the development environment along with the hardware and software requirements. The tool would require a development system with sufficient processing power and memory. They were needed for handling multiple device connections and test executions. Software requirements included Python, ADB and Tkinter.

The hardware requirements for the development system were aimed to be relatively low. Modern computers usually have enough resources to handle the demands of automated testing. A typical setup would require a multi-core processor, at least 8 GB of RAM and sufficient storage for logs and test results. These specifications could ensure that the system runs multiple cases of the testing tool at the same time to enable parallel testing on multiple devices.

Software requirements were planned to be more specific. Python was chosen for its ease of use. Its rich collection of libraries and tools also made it an ideal choice for developing the automated testing prototype. Tkinter was chosen to use for creating a user-friendly interface as it is a standard GUI toolkit for Python. ADB, which is a command line tool for Android devices, was selected for interacting with the tested devices.

Table 1: Technical specifications for the tool.

Category	**Specification**
Hardware Requirements	
Processor	Multi-core processor
Memory (RAM)	At least 8 GB
Storage	Sufficient storage for logs and test results
Software Requirements	
Programming Language	Python
GUI Toolkit	Tkinter
Device Management Tool	Android Debug Bridge (ADB)
Development Environment	
Code Analysis & Debugging	IDLE
Customization & Flexibility	Visual Studio Code
Collaboration	Git

IDLE and Visual Studio Code were suggested for developing the script. The collaboration task was planned to be done with the help of Git. IDLE possesses good code analysis features. It is also good for debugging. This can help to identify and fix code related issues. Visual Studio was chosen for its great library of extensions. It could give an opportunity for customization. Git could be used for collaborating among team members. Any code change could be tracked by other team members with the help of it.

All the technical specifications of the designed prototype tool are shown in Table 1.

3.2.5 Security and Compliance

Data security was a top priority while designing the automated testing tool. Any unauthorized access was planned to be prevented. Access control mechanisms were suggested to ensure that only authorized personnel could access data. This would be done through a combination of user authentication which could restrict access based on the user's role within the organization. Compliance with relevant industry standards of data security was also an important consideration. The tool was designed to meet these standards and ensure that it could be used in environments with strict security requirements.

3.2.6 Algorithm and Flowchart Description

The automated testing tool was designed following a structured algorithm. The algorithm is developed considering Company requirements and the flowchart of the automated testing framework shown in Figure 5. The key steps in the algorithm are shown in Algorithm 1.

The algorithm begins with the initialization of the user interface. Here the main window is created and UI components like labels, comboboxes, buttons and status labels are added. This step is similar to the Launch App step of Figure 5 where Application Explorer launches the app and interacts with the device. The program then tries to build the device list by executing an ADB command to get connected devices and updates the combobox with this list. Then the selection options appear on the screen. The selected ADB command is then executed. Some of the commands are retrieving the selected device and command, acquiring any additional parameters, starting a new thread to execute command and updating the status to reflect the running command. This is similar to the Inference Engine in Figure 5 which introduces user interaction features and executes actions on the device. The program then displays the details of the executed command in the UI and continuously updates the output text with the command results. This is similar to the Analyzer of the Flowchart of the automated testing framework which stores and processes data. The algorithm checks if the command is still running. If it is not running then it completes the execution by marking the command as not executed. The program moves to cancel the ADB command by terminating the command process if the command is still running. The program finalizes the execution after canceling the command.

Algorithm 1: ADB Command UI Algorithm

Step 1 : Start

Step 2 : Initialize UI

1. Create the main window.
2. Set the window title and size.
3. Add UI components such as labels, comboboxes, buttons, and status labels.

Step 3 : Populate Device List

1. Execute ADB command to list connected devices.
2. Update the device combobox with the list of connected devices.

Step 4 : Select Test Options

1. Display option to select devices
2. Display option to select ADB command
3. Display option to select test suite
4. Display option to select additional test

Step 5 : Execute ADB Command

1. Get the selected device from the combobox.
2. Get the selected command from the combobox.
3. Get any additional parameters needed for the command.
4. Start a new thread to run the command.
5. Update the status to indicate the command is running.

Step 6 : Display Executed Command

1. Show the details of the executed command in the UI.

Step 7 : Update Output

1. Store the results in the folder where ADB is installed.
2. Display the output in a .txt file.

Step 8 : Check if Command is Running

1. If the command is not running:
    - Finalize the execution.
2. If the command is still running:
    - Proceed to cancel the ADB command.

Step 9 : Finalize Execution (if command is not running)

1. Mark the command as not running.
2. Enable the execute button.
3. Disable the cancel button.
4. Update the status to indicate the command has finished.

Step 10 : Cancel ADB Command (if command is still running)

1. Kill the command process.
2. Update the status to indicate the command was canceled.
3. Reset the UI elements to their initial state.

Step 11 : Finalize Execution

Step 12 : End

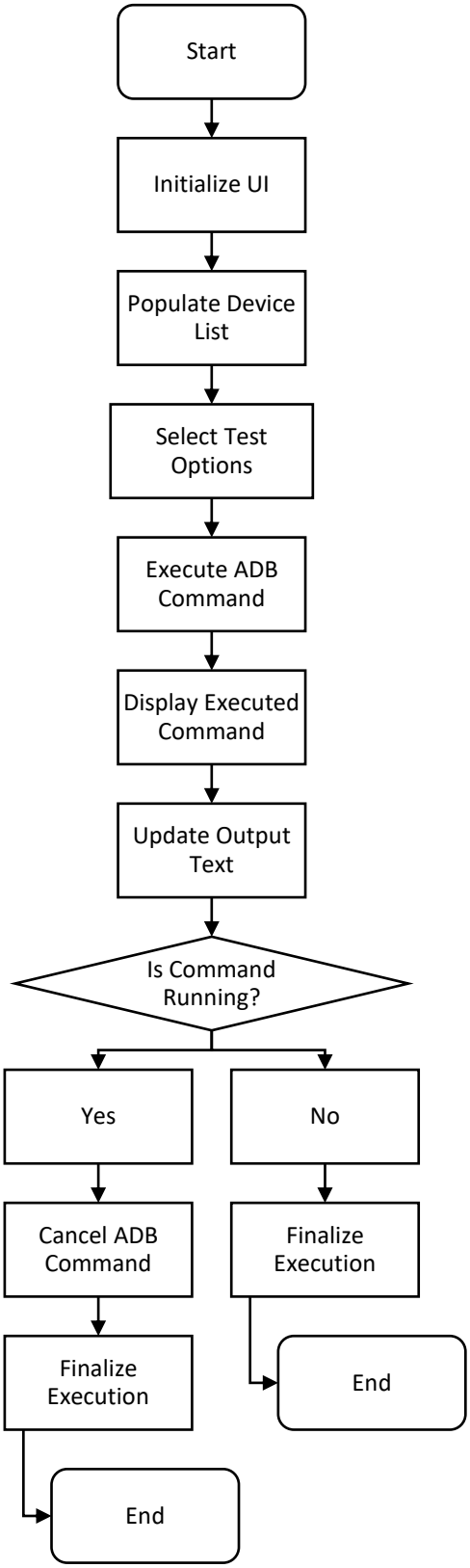

Figure 20: Flowchart of ADB command UI

The flowchart of ADB Command UI in Figure 20 shows the sequence of operations. Each step of the ADB command UI is shown here to get a better understanding of the step by step process.

3.2.7 Detailed Component Design

The system architecture was divided into various components. This modular design would ensure that each component could be developed and tested independently to improve the overall system.

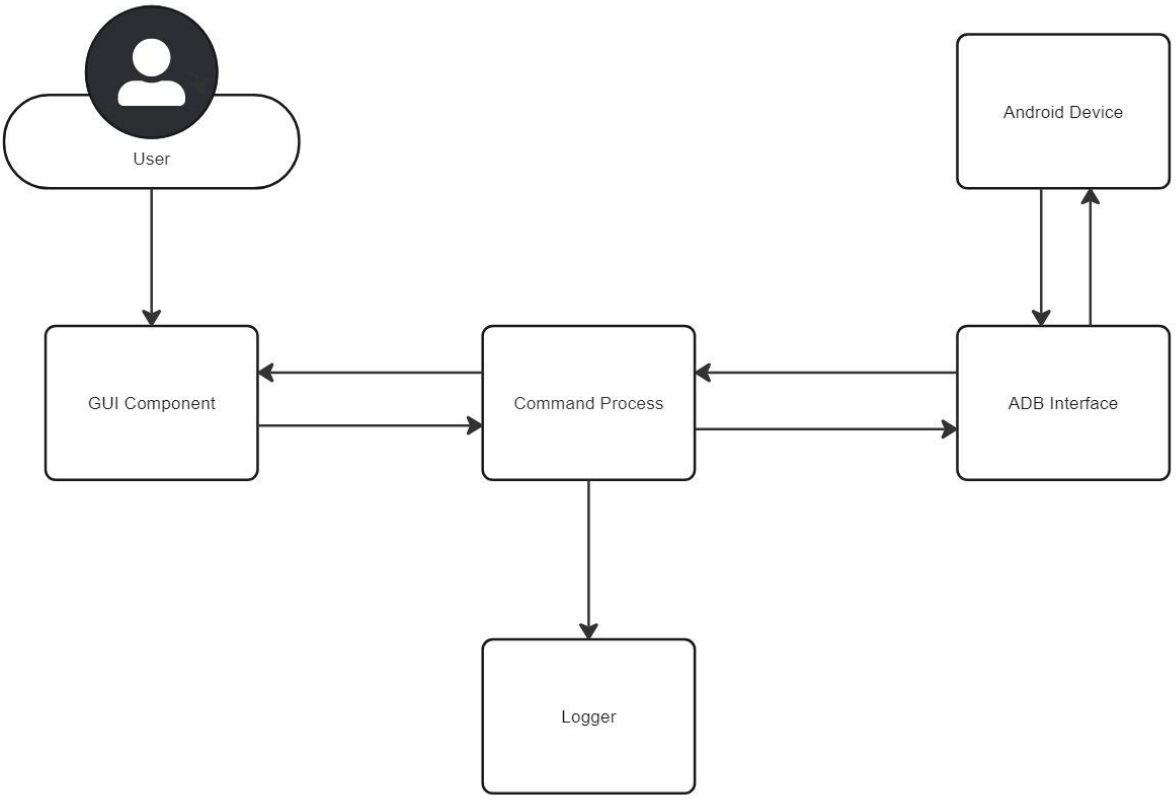

Figure 21: Component diagram of the designed prototype

Figure 21 shows how the components interact with each other. User initiates a test through the GUI Component in the beginning. Then GUI Component sends the selected command to the Command Processor. Then Command Processor creates the corresponding ADB command and the ADB Interface sends the command to the Android Device. Android Device executes the command successfully and returns the result. Then ADB Interface relays the result back to the Command Processor and Command Processor updates the GUI Component with the result. Finally, the results are sent to the Logger and it records the successful execution for later analysis.

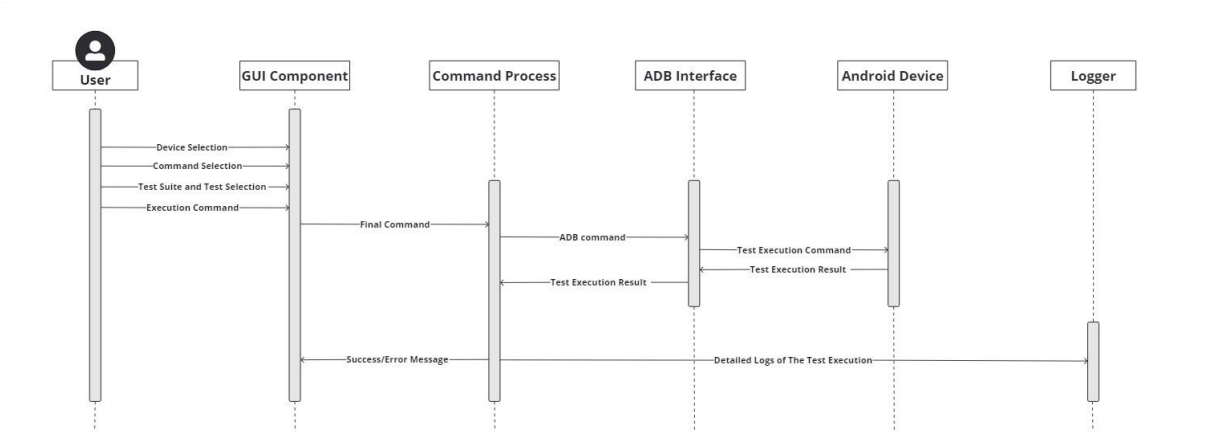

Figure 22: Sequence diagram of the designed prototype

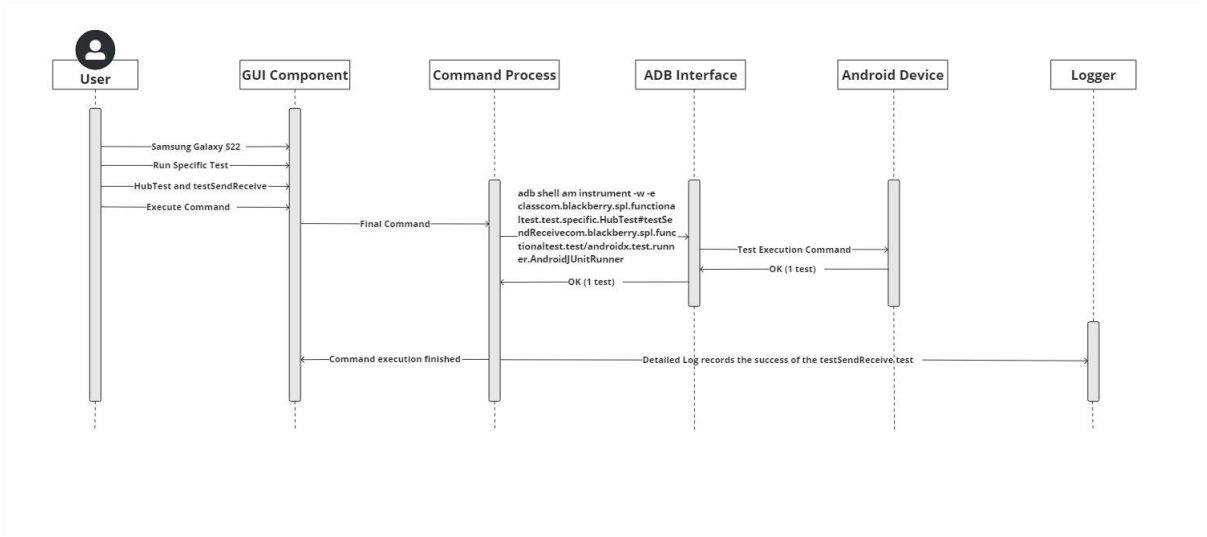

Figure 23: Sequence diagram of the designed prototype for a specific test case

Figure 22 shows the sequence diagram of the designed prototype and Figure 23 shows the sequence diagram of the designed prototype for a specific test case. At first, the user selects "Samsung Galaxy S22" from the "Select Device" dropdown in the GUI Component. Then he or she selects "Run Specific Test" from the "Select ADB Command" dropdown. After that, the user selects HubTest from the "Select Test Suite" dropdown and testSendReceive from the "Select Additional Test" dropdown. Then he or she clicks the "Execute Command" button and the GUI Component sends a final command to the Command Processor to execute the testSendReceive test on HubTest. The Command Processor constructs the ADB command to execute the testSendReceive test on the HubTest suite. The command might look something like: adb shell am instrument -w -e class com.blackberry.spl.functionaltest.test.specific.HubTest#testSendReceive

com.blackberry.spl.functionaltest.test/androidx.test.runner.AndroidJUnitRunner. The Command Processor sends the generated ADB command to the ADB Interface. The ADB Interface connects to the " Samsung Galaxy S22" device and sends the command to execute the testSendReceive test. The Android Device runs the testSendReceive test as part of the HubTest suite. The test executes successfully and the Android device returns the result: "OK (1 test)" indicating success. The ADB Interface receives the success result and sends it back to the Command Processor. The Command Processor updates the GUI Component to show the successful execution of the testSendReceive test. The Command Processor sends detailed logs of the test execution to the Logger. The Logger records the success of the testSendReceive test for future reference and analysis.

This testing process described in the component diagram of Figure 21 and the sequence diagram of Figure 22 can be mapped to the testing process of android apps shown in Figure 3. The ‘Device Selection’ and ‘Command Selection’ steps represent the preparation of the test environment of Figure 3. The ‘Execution Command’ step in the sequence diagram maps to the ‘Execution of Test Cases’ step in Figure 3. The ‘Test Execution Result’ and the logging activities in the designed tool map to the ‘Observation and Documentation’ step in Figure 3.

The GUI component was plotted to be developed using Tkinter. This would provide a user-friendly interface for selecting devices and executing commands. It was planned to build the prototype in such a way that it would be easy to use and quick to respond. The interface was designed to have elements like device selection dropdown menus, command execution buttons and result display panels. This design could guarantee that all essential features can be easily reached. It would also minimize the learning curve for new users.

The command processor was designed to manage the implementation of ADB commands to guarantee effective test execution. It was planned to handle the communication with linked devices and carry out the user's chosen instructions. Some of the important and complicated functionalities in the code are discussed below with relevant code snippets.

The part of the code given in Figure 24 executes ADB commands in a separate subprocess. It uses subprocess.Popen to start the command with stdout and stderr piped for real-time feedback. Handling a subprocess correctly was difficult. Issues such as deadlocks (when reading from stdout and stderr simultaneously) and managing subprocess termination caused problems. Buffering is managed using ‘bufsize=1’ to handle line-buffered output. Reading stdout and stderr in real-time without blocking the UI thread was complex. ‘universal_newlines=True’ was used to enable text mode. It was combined with ‘self.root.after()’ to ensure non-blocking and timely output handling.

```
def execute_adb_subprocess(self, adb_command):
    try:
        self.command_running = True
        self.subprocess_obj = subprocess.Popen(
            adb_command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, bufsize=1,
            universal_newlines=True  # Enable text mode for real-time output
        )

        self.root.after(100, self.check_for_updates)

    except Exception as e:
        self.handle_execution_error(e)
        self.finalize_execution()
```

Figure 24: Code snippet of executing ADB commands in a subprocess

```
def update_additional_params(self, event=None):
    # Clear existing widgets in the frame
    for widget in self.additional_params_frame.winfo_children():
        widget.destroy()

    selected_command = self.command_combobox.get()

    if selected_command in ["Run Sanity Suite", "Run Smoke Suite"]:
        # Provide options for suite.SanitySuite, suite.SmokeSuite, and suite.BatterySuite
        self.test_suite_label = tk.Label(self.additional_params_frame, text="Select Test Suite:")
        self.test_suite_label.pack()

        self.test_suite_combobox = ttk.Combobox(self.additional_params_frame, values=[
            "SanitySuite",
            "SmokeSuite"
        ])
        self.test_suite_combobox.pack()
```

Figure 25: Code snippet of dynamically updating the GUI based on user input

This part of the code given in Figure 25 updates the GUI dynamically when the user selects a different ADB command from the combobox. Handling complex UI updates based on user input was prone to error. It was difficult correctly initialize and update the widgets without overlap or redundancy. An existing widget is first cleared using ‘widget.destroy()’ before adding new ones for a clean update. Ensuring that UI elements update correctly in response to events needs careful event binding.

```
def cancel_adb_command(self):
    if self.command_running:
        try:
            # Kill the adb process and update UI
            self.subprocess_obj.kill()
            self.subprocess_obj.wait()  # Wait for the process to finish
            self.status_label.config(text="Command canceled", fg="orange")
        except Exception as e:
            self.status_label.config(text=f"Error canceling command:\n{str(e)}", fg="red")
        finally:
            self.command_running = False
            self.execute_button.config(state="normal")
            self.cancel_button.config(state="disabled")
    else:
        self.status_label.config(text="No command is currently running", fg="orange")
```

Figure 26: Code snippet of handling command cancellation

This part of the code given in Figure 26 allows the user to cancel an ADB command. It stops the subprocess by calling 'self.subprocess_obj.kill()'. Killing a subprocess abruptly could generate incomplete output files. The use of 'self.subprocess_obj.kill()' and 'self.subprocess_obj.wait()' helped to terminate the process and clean up the process resources properly. It was crucial to manage the state of the application to prevent multiple commands from running at the same time. A flag ('self.command_running') was used to track the execution state and appropriately enable or disable buttons to prevent conflicts.

```
def update_output_text(self):
    while self.command_running:
        try:
            stream_type, line = self.queue.get_nowait()
            if stream_type == 'stdout':
                self.adb_command_text.insert(tk.END, f"[STDOUT]: {line}")
            elif stream_type == 'stderr':
                self.adb_command_text.insert(tk.END, f"[STDERR]: {line}")
            elif stream_type == 'done':
                self.adb_command_text.insert(tk.END, "Command execution finished.\n")
        except queue.Empty:
            pass

        # Scroll to the end of the Text widget
        self.adb_command_text.yview(tk.END)

        # Update the UI after a short delay
        self.root.after(100)
```

Figure 27: Code snippet of managing output streams

This part of the code given in Figure 27 updates the Text widget with the output from the ADB command's stdout and stderr streams. Updating the GUI from a thread other than the main thread could cause crashes in Tkinter. A queue was used to manage output from the subprocess using self.root.after(100). Large amount of output could overwhelm the GUI or cause memory issues. 'yview(tk.END)' was used to guarantee that the Text widget scrolls appropriately and prevent memory overflow or slow performance.

The logger was planned to record logs of test execution. This would help in detecting the errors. The logger was structured to create a detailed record of every test conducted. Logs would contain detailed information about every command executed and the results of the execution. This information would be crucial for confirming the accuracy of the tests. Threading and asynchronous callbacks were used to periodically check on the subprocess's status.

3.2.8 Proper Optimization and Improvement

Error detection was necessary for proper optimization of the design. Error detecting mechanisms would help detect device disconnections, command failures and other potential issues. Detailed error logs were also designed to provide information about the nature of the error. The error logs would also provide information about how to recover from the error.

Scalability was another key consideration in the design of the automated testing tool. The tool was designed to handle multiple devices at the same time. This was planned to be done by effectively managing connections between devices. Performance metrics would be monitored in order to find areas that needed improvement. This continuous optimization process could ensure that the tool remained efficient with the evolving testing requirements.

The automated testing tool was designed to integrate with the company's existing testing infrastructure. The tool was planned to integrate with test case management systems and bug tracking tools. The tool was designed to export test results in a format that is compatible with these systems. This would ensure that the results could be easily incorporated into existing workflows.

Some important steps were agreed to be taken for continuous evaluation and improvement of the tool. Feedback would be collected from testers to identify areas for improvement. Regular updates would be made to the tool to solve the faults and add new features. This continuous improvement process would ensure that the tool remains up to date with the evolving needs of the testers.

This chapter discussed about the case study and conceptual design of the automated testing tool. The case study of transitioning from manual to automated testing within the company shows the benefits of

automation in improving testing efficiency and accuracy. The conceptual design of the automated testing tool demonstrates how to plan and design the automated testing tool.

4 Implementation and Evaluation

The shift from manual to automated testing is a major development in the field of Android applications testing. This chapter presents a detailed overview of the implementation process that supports the transition to automated testing in the company. This section goes into detail about the technical aspects of implementing the automated testing tool. It also focuses on setup, execution and maintenance stages.

4.1 Implementation Methodology

The methodology of the implementation was divided into three phases. They are the preparatory phase, the execution phase and the evaluation phase.

The preparatory phase consisted of establishing the environment and choosing the test cases. The main concern of this phase was to ensure the availability of hardware and software components. This included setting up Android devices and creating test suites. Various Android devices, including both older and newer models, were selected for testing. This variety helped in understanding how different hardware configurations affect the performance of the automated testing tool. The Python script including necessary dependencies like ADB and Tkinter were installed on the test system. A typical sample of test cases was chosen from the existing manual test suite. The test cases were sorted by complexity and how often they were performed. The classification helped in analyzing how the tool performed in various test categories.

The chosen test cases were executed through manual and automated testing methods during the execution stage. Data was collected to obtain information on performance metrics like execution time and defect detection. At first, the selected test cases were executed manually and the results were documented. Then the same test cases were executed using the automated testing tool. Real-time monitoring was conducted to ensure that the automated execution closely mimicked the manual process.

The data collected during the execution phase was analyzed. Execution times and defect logs were compared. This comparison provided quantitative information about the performance improvements for using the automated testing tool.

The steps of implementation methodology are shown in Figure 28.

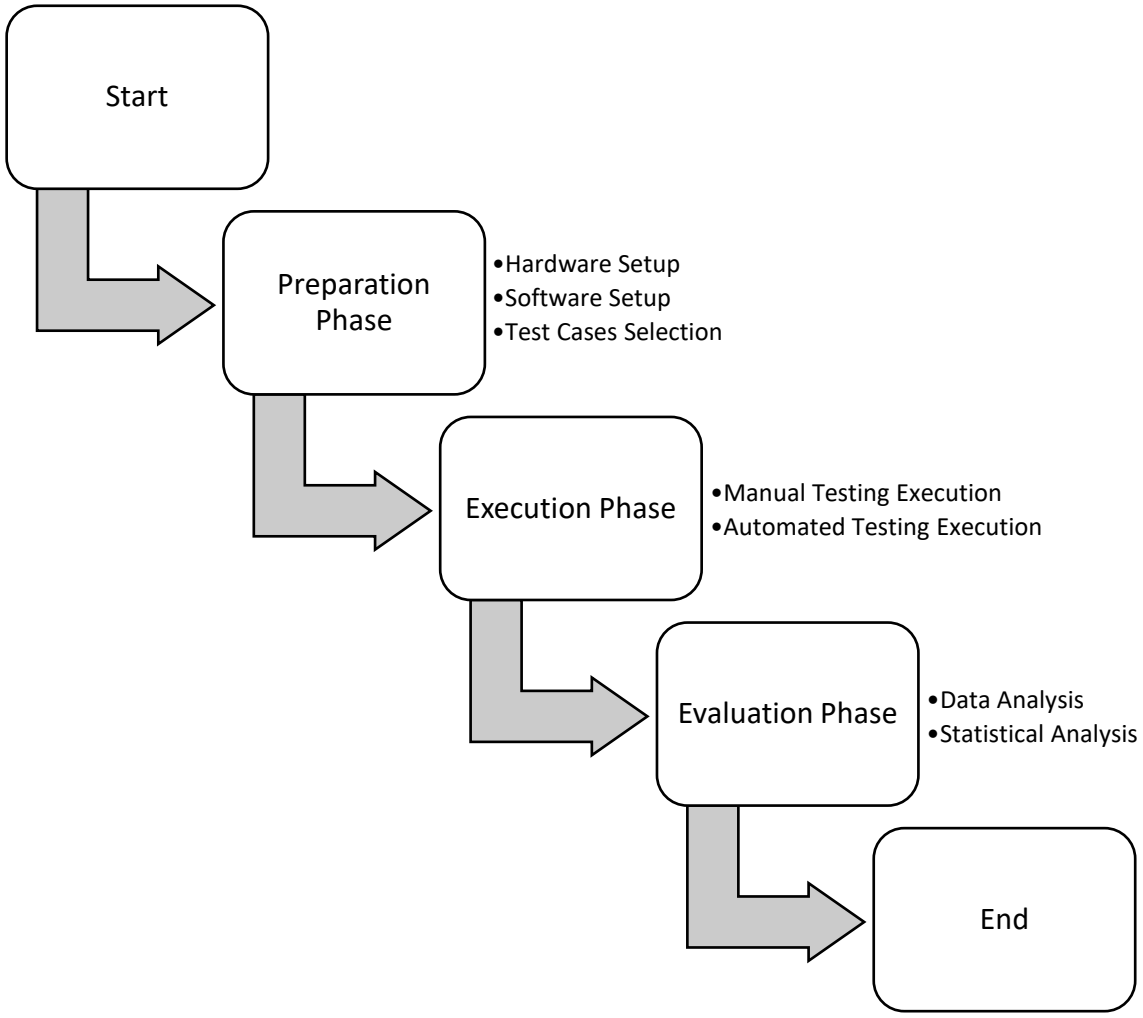

Figure 28: Steps of implementation methodology

4.2 Setup and Configuration

The initial step of the implementation process was to configure the automated testing tool. The tasks for this step can be divided into three categories. They are setting up software, configuring the testing environment and verifying dependencies.

The test tool being researched is written in Python. The graphical interface with this test device uses Tkinter. ADB (Android Debug Bridge) performs communication between the device and Automation Machines. Python was set up on the test system to develop the tool. Version management was handled using virtual environments. It was done to avoid conflicts with other installed software. Essential Python libraries and subprocesses for executing ADB commands were installed through pip. Pip is the package

installer for Python. A `requirements.txt` file was used to manage Dependencies. It was done to ensure that all necessary packages were installed correctly.

There are several prerequisites for executing the test. First, it should be ensured that the testing device is enrolled in the Samsung Knox portal and provides a specific profile for getting the secured applications.

Secondly, the applications that must be installed in the testing device are:

1. testsupporter-debug_V1.2.1
2. functionaltest-debug-androidTest.apk
3. functionaltest-debug.apk

Finally, it must be checked that the “USB debugging” of the device is turned on and ready for testing.

Android devices were connected to the testing system and ADB was configured to set up the testing environment. The Android device was connected via USB and configured for debugging mode. This step got the device ready to interact with the testing tool. ADB was set up to display and communicate with connected devices.

Dependency Resolution is a process to find and resolve dependencies in a system. For resolving dependency, the automated testing tool was able to interact with devices and execute required commands. All required dependencies were installed and verified. Device drivers and additional Python libraries were among the dependencies. The installation process was documented to facilitate replication. Initial tests were conducted to verify the tool’s ability to detect and communicate with the connected devices. These tests used basic ADB commands to make sure that the setup was correct.

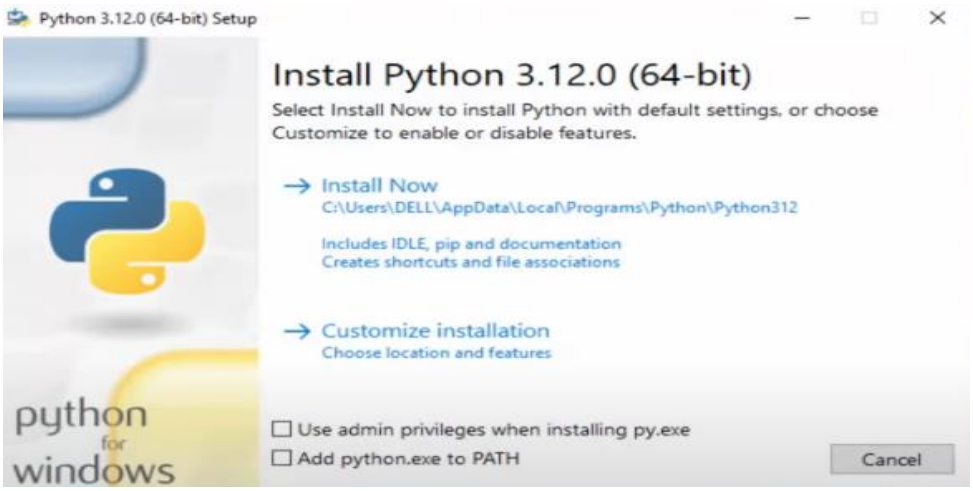

Figure 29: Installing python for writing script

4.3 Programming Implementation

A detailed description about how the programming was implemented for the prototype automated testing tool is given below.

4.3.1 Setting Up the Environment and Importing Required Modules

It was necessary to set up the environment by importing all required Python modules.

Implementation:

1. The ‘subprocess’ module was imported for running system commands from Python. It was used to execute ADB commands to interact with connected Android devices.

2. The ‘threading’ module was imported to allow the tool to run multiple threads at the same time.

3. ‘tkinter’ is the standard Python interface to the GUI toolkit. ‘ttk’ is a submodule to provide themed widgets. These were imported to create and manage the graphical user interface (GUI).

4. The ‘Queue’ class from the ‘queue’ module was imported for communication between the main GUI thread and the background threads that handle ADB command execution.

5. The ‘datetime’ module was imported to handle dates and times.

4.3.2 Initializing the Main GUI Window

The main window of the GUI needed to be created and essential UI components needed to be initialized.

Implementation:

1. ‘root = tk.Tk()’ was used to create the main window and ‘self.root’ was used to refer to this window throughout the class.

2. The window title was set by using self.root.title(f"ADB Command UI {self.VERSION_CODE}"). It would help users to identify the application.

3. ‘self.root.geometry("1024x786")’ was used to set the geometry (width and height) of the window.

4. Labels were created using ‘tk.Label’. They were added to the window using the pack() method. ‘ttk.Combobox’ was used to create comboboxes. They would help the users to select devices and commands. Some examples of the main code are given in Figure 30.

```
self.devices_label = tk.Label(root, text="Select Device:")
self.devices_label.pack()

self.device_combobox = ttk.Combobox(root, values=self.get_device_list())
self.device_combobox.pack()

self.command_label = tk.Label(root, text="Select ADB Command:")
self.command_label.pack()

self.command_combobox = ttk.Combobox(root, values=[
    "Run Sanity Suite",
    "Run Specific Test Suite",
    "Run Specific Test",
    "Run Hub AutoMailer Suite",
    "Run Battery Suite"
])
self.command_combobox.pack()
```

Figure 30: Creating comboxes code for device and command selection

4.3.3 Populating and Managing the Device List

It was necessary to display a list of connected Android devices in a dropdown so that the user could select the target device for ADB commands.

Implementation:

1. ADB command ‘adb devices’ could be used to fetch the list of connected devices. The command was run using ‘subprocess.run’.

```
def get_device_list(self):
    # Use adb devices to get the list of connected devices
    result = subprocess.run(["adb", "devices"], capture_output=True, text=True)
    devices = [line.split('\t')[0] for line in result.stdout.splitlines()[1:] if line.strip()]
    return devices
```

Figure 31: Code for retrieving device list

2. ‘self.device_combobox['values'] = self.get_device_list()’ was used to populate the combobox with the retrieved device IDs.

3. A refresh button was created and bounded to the ‘refresh_device_list()’ method. ‘self.root.after(30000, self.refresh_device_list)’ was used to refresh the list every 30 seconds. It was done to detect connected or disconnected devices during testing.

```
def refresh_device_list(self):
    # Update the values in the device combobox
    self.device_combobox['values'] = self.get_device_list()
    self.root.after(30000, self.refresh_device_list)  # Schedule the next refresh
```

Figure 32: Code for setting up Refresh functionality

4.3.4 Implementing Command Selection and Parameter Handling

It was necessary to create a user interface for selection of ADB commands and dynamically update additional parameters based on the selected command.

Implementation:

1. Creatibd command selection combobox would allow the user to select from predefined ADB commands. A combobox was initialized with command options.

```
def update_additional_params(self, event=None):
    # Clear existing widgets in the frame
    for widget in self.additional_params_frame.winfo_children():
        widget.destroy()

    selected_command = self.command_combobox.get()

    if selected_command in ["Run Sanity Suite", "Run Smoke Suite"]:
        # Provide options for suite.SanitySuite, suite.SmokeSuite, and suite.BatterySuite
        self.test_suite_label = tk.Label(self.additional_params_frame, text="Select Test Suite:")
        self.test_suite_label.pack()

        self.test_suite_combobox = ttk.Combobox(self.additional_params_frame, values=[
            "SanitySuite",
            "SmokeSuite"
        ])
        self.test_suite_combobox.pack()
```

Figure 33: Code showing a part of implementing command selection and parameter handling

2. ‘update_additional_params()’ was used to dynamically create and display relevant input fields.

3. ‘self.command_combobox.bind("<<ComboboxSelected>>", self.update_additional_params)’ was used to bind the combobox selection.

4.3.5 Executing ADB Commands

It was required to construct the appropriate ADB command based on the user’s selections and execute it in a subprocess.

Implementation:

1. Values from the comboboxes and entry fields were retrieved to collect the selected device, command and any additional parameter.

2. String formatting was used to construct the command.

3. 'subprocess.Popen' was used to start the command. stdout and stderr were captured to provide real-time feedback to the user.

```
def execute_adb_command(self):
    if self.command_running:
        self.status_label.config(text="Command is already running", fg="orange")
        return

    try:
        self.update_ui_before_execution()

        selected_device = self.device_combobox.get()
        selected_command = self.command_combobox.get()
        additional_params = self.get_additional_params(selected_command)

        if selected_command == "Run Specific Test Suite" or selected_command == "Run Specific Test":
            test_suite = self.test_suite_combobox.get()
            test_name = f"{test_suite}_{self.additional_tests_combobox.get()}" if selected_command == "Run Specific Test" els
        else:
            test_name = self.test_suite_combobox.get()

        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        output_file_name = f"adb_output_{timestamp}_{selected_device}_{test_name}.txt"

        if selected_command == "Run Specific Test Suite" or selected_command == "Run Specific Test":
            adb_command = f"adb -s {selected_device} shell am instrument -w -r -e class {additional_params} com.blackberry.sp
        else:
            adb_command = f"adb -s {selected_device} shell am instrument -w -r -e class com.blackberry.spl.functionaltest.sui

        self.adb_command_text.delete(1.0, tk.END)  # Clear previous content
        self.adb_command_text.insert(tk.END, adb_command)

        threading.Thread(target=self.execute_adb_subprocess, args=(adb_command,), daemon=True).start()
```

Figure 34: Part of the code showing ADB commands execution

4.3.6 Managing Real-time Output and Feedback

It was required to display the output of the ADB command (stdout and stderr) in real-time within the GUI.

Implementation:

1. 'subprocess.PIPE' was used to capture the output. Threading was used to read and display the output without blocking the main GUI thread. This allowed to begin the ADB command execution in a subprocess and capture the output streams (stdout and stderr).

2. The Text widget was needed to be updated with new lines of output as they were produced by the subprocess. A loop was used to read lines from stdout and stderr. The lines were inserted into the Text widget and scrolled to the end to keep the latest output visible.

3. 'self.subprocess_obj.poll()' was used to periodically poll the subprocess. The execution was finalized and the UI was updated after the command was complete.

```
def execute_adb_subprocess(self, adb_command):
    try:
        self.command_running = True
        self.subprocess_obj = subprocess.Popen(
            adb_command, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE, bufsize=1,
            universal_newlines=True  # Enable text mode for real-time output
        )

        self.root.after(100, self.check_for_updates)

    except Exception as e:
        self.handle_execution_error(e)
        self.finalize_execution()

def check_for_updates(self):
    return_code = self.subprocess_obj.poll()
    if return_code is None:
        stdout = self.subprocess_obj.stdout.read()
        stderr = self.subprocess_obj.stderr.read()

        self.queue_output(stdout, 'stdout')
        self.queue_output(stderr, 'stderr')

        self.root.after(100, self.check_for_updates)
    else:
        self.queue_output('', 'done')
        self.finalize_execution()
```

Figure 35: Part of the code for managing real-time output and feedback

4.3.7 Handling Command Cancellation

It was needed to provide the user with the access to cancel an ongoing ADB command.

Implementation:

1. 'self.command_running' flag was used to check whether a command is currently running and if it can be safely canceled.

2. Subprocess that is executing the ADB command needs to be terminated. 'self.subprocess_obj.kill()' was used to send a termination signal to the subprocess. 'self.subprocess_obj.wait()' was used to guarantee that the process is terminated before proceeding.

3. The status label was changed to indicate the cancellation. The "Execute" button was re-enable and the "Cancel" button was disabled.

```
def cancel_adb_command(self):
    if self.command_running:
        try:
            # Kill the adb process and update UI
            self.subprocess_obj.kill()
            self.subprocess_obj.wait()  # Wait for the process to finish
            self.status_label.config(text="Command canceled", fg="orange")
        except Exception as e:
            self.status_label.config(text=f"Error canceling command:\n{str(e)}", fg="red")
        finally:
            self.command_running = False
            self.execute_button.config(state="normal")
            self.cancel_button.config(state="disabled")
```

Figure 36: Part of the code for handling command cancellation

4.3.8 Logging and Output Management

It was required to log the output of the ADB command to a file with a timestamp and generate a record of the command execution for later analysis.

Implementation:

1. ‘datetime.now().strftime()’ was used to format the timestamp and combine it with the device ID and test name.

2. ADB command string was constructed to include redirection to the output file. Both stdout and stderr were captured and written to the file.

```
timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
output_file_name = f"adb_output_{timestamp}_{selected_device}_{test_name}.txt"

if selected_command == "Run Specific Test Suite" or selected_command == "Run Specific Test":
    adb_command = f"adb -s {selected_device} shell am instrument -w -r -e class {additional_params} com.blackberry.s
else:
    adb_command = f"adb -s {selected_device} shell am instrument -w -r -e class com.blackberry.spl.functionaltest.su

self.adb_command_text.delete(1.0, tk.END)  # Clear previous content
self.adb_command_text.insert(tk.END, adb_command)

threading.Thread(target=self.execute_adb_subprocess, args=(adb_command,), daemon=True).start()
```

Figure 37: Part of the code for logging and output management

4.3.9 Updating the GUI Before and After Command Execution

It was necessary to prepare the GUI before the execution of a command and reset it after the command is completed.

Implementation

1. The user needed to be Indicated about a command is being executed. ‘Execute’ button was disabled to prevent multiple commands from being run simultaneously. ‘Cancel’ button was enabled to allow the user to stop the command if needed.

2. ‘self.adb_command_text.delete(1.0, tk.END)’ was used to clear the Text widget.

3. The user needed to be informed that the command has finished executing. The "Execute" button was re-enabled. The "Cancel" button was disabled.

```
def update_ui_before_execution(self):
    self.execute_button.config(state="disabled")
    self.cancel_button.config(state="normal")
    self.status_label.config(text="Executing command...", fg="blue")

    # Clear previous content in the Text widget
    self.adb_command_text.delete(1.0, tk.END)

    # Display the executed ADB command in the Text widget
    self.adb_command_text.insert(tk.END, "Command will be displayed here after execution...\n")

    # Clear the queue
    self.queue.queue.clear()

    # Start a new thread to continuously update the Text widget
    threading.Thread(target=self.update_output_text, daemon=True).start()


def finalize_execution(self):
    self.command_running = False
    self.execute_button.config(state="normal")
    self.cancel_button.config(state="disabled")
    self.status_label.config(text="Command execution finished", fg="green")
```

Figure 38: Part of the code for updating the GUI before and after command execution

4.3.10 Starting the Application

It was needed to initialize the Tkinter main loop in order to keep the GUI responsive and handle all user interactions until the application is closed.

Implementation:

1. The ‘Tk()’ class was used to create the root window.

2. An instance of the ‘ADBUI’ class was created to set up all the UI components.

3. ‘root.mainloop()’ was called to start the loop and keep the GUI running.

```
if __name__ == "__main__":
    root = tk.Tk()
    adb_ui = ADBUI(root)
    root.mainloop()
```

Figure 39: Part of the code to start the application

4.4 Execution Workflow

The execution workflow of the automated testing tool was designed to be efficient. This allowed testers to select devices, commands and tests without any major problem.

The tool's user interface was designed for a simple experience for the user. Testers had the option to pick devices, select ADB commands and specify test suites by using dropdown menus and buttons. Testers could select from a list of connected devices. The tool automatically updated the list upon connecting or disconnecting devices. The UI was designed for reducing the required number of steps to start a test. Testers were able to choose predefined ADB commands like "Run Specific Test" or "Run Sanity Suite" from a dropdown menu. The choices were designed to fit typical testing situations. This made it easier for the users to choose. The tool instantly changed the user interface to show appropriate test suites and extra test choices when a command was selected. This continuous updating made sure that testers constantly had the right choices available.

Running the commands required sending ADB commands to a chosen device and capturing results for displaying and analyzing them in real time. The tool sent ADB commands to the selected device using Python’s subprocess module. The output from the command execution was captured and displayed in the GUI. Real-time capturing provided immediate feedback to the testers for improving the debugging process.

It was necessary to provide real-time feedback for monitoring the progress of test executions. The tool displayed real-time status updates like completion notifications. Color-coded messages indicated the

status of completed command. All the errors occurred during command execution were captured and shown to the user in the log file. As a result, quick troubleshooting was possible.

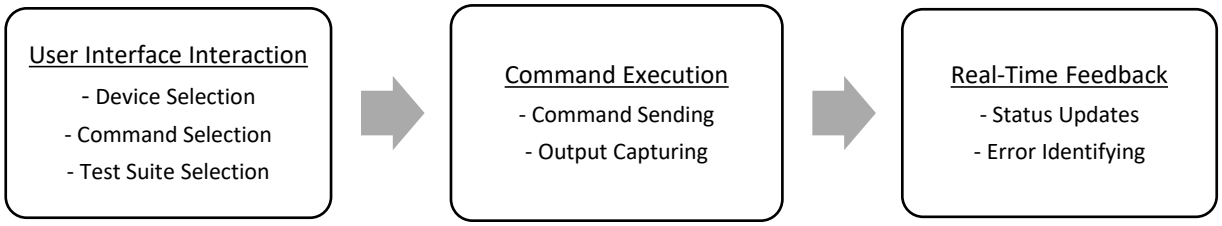

Figure 40: Steps for workflow execution

4.5 Data Logging and Analysis

Data recording and analysis were some of the most important parts of the project. This gave valuable information on how well the automated testing tool was performing.

The tool used a strong logging system to record specific details about every test execution. Detailed records of every test execution were documented. These included capturing timestamps, output commands and any error encountered. These logs offered a precise record of the testing activities. Logs were then stored in a structured format including device ID. This ensured simple retrieval and analysis of the logs. The storage system made sure that logs were sorted based on test case and device.

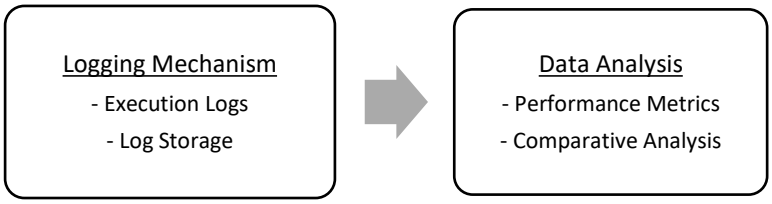

Figure 41: Steps for data logging and analysis

Analyzing the logged data delivered important information on how well the automated testing tool performed. It also pointed out the areas for improvement. Key performance metrics such as execution time and defect detection rate were examined. These metrics helped in measuring the advantages of automation. Then the performance of the automated testing tool was compared with manual testing to evaluate its effectiveness. This comparative analysis highlighted the strengths and areas for improvement of the automated approach of initial smoke testing.

4.6 Challenges and Mitigations

There were some challenges encountered during the implementation of the automated testing tool. This section discusses the key challenges encountered during the implementation followed by the strategies undertaken to mitigate them.

4.6.1 Technical Challenges and Mitigations

Some of the main technical challenges faced were integrating the automated testing tool with different Android devices and ensuring consistent performance in various environments. The challenges are related to the Test Challenges mention in the Taxonomy of mobile application testing described in Figure 6.

Android ecosystem has diverse hardware configurations and OS versions. As a result, it created complexity to ensure combability with this diverse range of android devices. The tool had to be able to communicate with all the targeted devices. This condition required extensive compatibility testing. Devices were categorized according to their hardware and software characteristics. This was done to identify any specific compatibility issues.

Strong error handling and recovery mechanisms were needed to ensure reliable execution of ADB commands on various devices. Mechanisms for error handling were added in order to identify and report any errors when executing the commands. Implementing retry logic for temporary errors and thorough logging for ongoing problems were some of the mechanisms. Strategies were created and included in the tool to solve common problems like device disconnects or command timeouts. These strategies guaranteed that tests could resume or restart without any significant intervention.

4.6.2 Operational Challenges and Mitigations

There were some operational challenges faced by the testers while using the automated testing tool. It was necessary to mitigate them to incorporate it into current workflows.

The tool needed to perform costing very less resources and the maintenance cost needed to be minimum. Also, it should be built in as less time as possible. These refer to the test challenges of the Taxonomy of mobile application testing described in Figure 6.

It took some thought and coordination to add the automation testing tool into our current test suite. Existing testing processes were mapped to identify areas where the automated tool could be integrated. This mapping helped in understanding the potential impact on current workflows. Scripts were created to smoothly integrate it without causing any disruption to current workflows.

4.7 Continuous Improvement and Maintenance

Upon the automated testing tool's integration, improvement and maintenance activities were held to guarantee long term performance.

It was necessary to collect feedback from testers for identifying areas for improvement and ensuring that the tool met their needs. Surveys and feedback forms were implemented to collect feedback. As a result, the testers were able to provide their experiences. Regular reviews of feedback were done to identify common issues among the testers. The areas for improvement were found from this method. Updates and improvements were done to the automated testing tool based on the feedback.

Continuous maintenance activities were needed for keeping the automated testing tool reliable. Maintenance activities like updating device drivers and libraries were routinely conducted. Potential issues could be detected and dealt with because of the constant monitoring of the tool's performance. Performance measures were monitored and examined to guarantee efficient functioning of the tool.

4.8 Case Studies

This section presents case studies of the automated testing tool’s implementation in real-world scenarios. This will demonstrate the practical application and benefits of it.

4.8.1 Case Study 1: Smoke Testing

The automated testing tool was used to perform smoke testing for a major update of the company’s Android application suite. The objective was to guarantee that new changes did not introduce any fault and that all critical functionalities continued to work as expected.

More than 200 test cases were part of the smoke test suite. This covered various functionalities across multiple applications. The automated testing tool was employed to perform these test cases on various devices. This reduced the time required for testing. Smoke testing required significantly less time compared to manual testing. It also achieved a higher rate of defect detection. The tool’s logging and reporting features helped to quickly identify and resolve issues. An example test cases is given below:

Test Case:

Step 1: The GUI component of the testing tool was launched.

Step 2: ‘Samsung Galaxy S22’ was selected from the ‘Select Device’ dropdown in the GUI Component.

Step 3: ‘Run Specific Test’ was selected from the ‘Select ADB Command’ dropdown.

Step 4: ‘HubTest’ was selected from the "Select Test Suite" dropdown.

Step 5: ‘testSendReceive’ was selected from the "Select Additional Test" dropdown.

Step 6: ‘Execute Command’ button was clicked.

Step 7: Test execution process was observed until the message “Command execution finished” appeared.

Step 8: The detailed logs of the test execution were verified.

Results: The test case testSendReceive executed without errors. The GUI displayed a success message indicating that the test ran successfully. The logger contained detailed logs of the test execution including 70.287 second time taken.

4.8.2 Case Study 2: Compatibility Testing

Another case study was conducted using the automated testing tool for compatibility testing across different Android OS versions and device models. The goal was to ensure that the company’s applications functioned properly on a diverse range of devices.

Compatibility testing consists of executing an identical set of test cases on devices that have different OS versions and hardware configurations. The automated testing tool made it possible to run test cases at the same time on different devices. The automated tool identified several compatibility issues that were not detected during manual testing. This shows its effectiveness in ensuring application robustness. The Table 2 shows the list of the tested devices along with their configurations.

Table 2: List of tested devices

Model	Android OS Version	RAM	Internal Memory	UI Version
Samsung Galaxy S24	Android 14	8GB	128 GB	One UI 6.0
Samsung Galaxy S23	Android 14	8 GB	128 GB	One UI 6.0
Samsung Galaxy S22	Android 14	8 GB	128 GB	One UI 6.0
Samsung Galaxy S20	Android 13	8 GB	128 GB	One UI 5.1
Samsung Galaxy Tab S9 Ultra	Android 13	12 GB	256 GB	One UI 5.1.1
Samsung Galaxy Tab S8	Android 12	8 GB	128 GB	One UI 4.1

4.8.3 Case Study 3: Stress Testing

Another case study was conducted utilizing the automated testing tool for stress testing to assess how well applications perform and remain stable under extreme conditions.

Stress testing included executing intensive test cases created to push the boundaries of the applications. It included scenarios with high user activity and resource usage. The automated testing tool executed these stress test cases repeatedly over extended periods and monitored the applications' performance. The stress tests revealed performance bottlenecks and stability issues that were not noticeable during normal testing. The results from stress testing helped in optimizing the applications under heavy load.

This chapter presented an overview of the process of implementing the automated testing for Android applications in the company.

5 Results, Discussion and Future Work

This chapter shows the results of the experimental study and implementation of the automated testing tool. Then a detailed discussion about the results is discussed for potential future works. The results demonstrate the efficiency of the automated testing tool compared to traditional manual testing. The discussion provides important information about these results. It shows the benefits and limitations of this designed tool. Finally, potential areas for future work to improve the automated testing are suggested.

5.1 Results

5.1.1 Efficiency Analysis

The main goal of designing the automated testing tool was to decrease the time needed to perform test cases compared to manual testing. The findings show a notable increase in efficiency.

Use of Automated testing reduced the execution time of test cases. For example, the test called `HubTest_testSendReceive_duration` that usually took about 159 seconds when it was done manually. Now it takes about 72 seconds with the automated tool. This shows a notable reduction in time.

The detailed efficiency analysis between manual and automated testing can be seen from Table 3.

Table 3: Efficiency analysis between manual and automated testing

Test name	Device	Time required for manual testing (in Seconds)	Time required for automated testing (in Seconds)
HubTest_testSendReceive_duration	S22_Android:14	151.4	70.287
HubTest_testSendReceive_duration	S20_Android:13	163.7	71.369
HubTest_testSendReceiveWithAttachment	S22_Android:14	177.3	78.008
HubTest_testSendReceiveWithAttachment	S20_Android:13	176.8	27.347

5.1.2 Accuracy Evaluation

Another critical objective was the accuracy of error detection in automated testing. The automated tests could successfully identify issues that were occasionally missed by manual testing.

Automated testing was able to detect issues in UI elements and scenarios that created failures like the absence of UI elements (`testSendReceiveWithAttachment`). These were not always identified in manual testing because of human error.

The automated tool provided detailed logs for each test execution. It also captured all relevant information to detect errors.

```
INSTRUMENTATION_STATUS:
class=com.blackberry.spl.functionaltest.test.specific.HubTest
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS: stream=
com.blackberry.spl.functionaltest.test.specific.HubTest:
INSTRUMENTATION_STATUS: test=testSendReceive_duration
INSTRUMENTATION_STATUS_CODE: 1
INSTRUMENTATION_STATUS:
class=com.blackberry.spl.functionaltest.test.specific.HubTest
INSTRUMENTATION_STATUS: current=1
INSTRUMENTATION_STATUS: id=AndroidJUnitRunner
INSTRUMENTATION_STATUS: numtests=1
INSTRUMENTATION_STATUS:
stack=com.blackberry.spl.functionaltest.support.VerificationError:
Inbox's main page did not open in expected shape
com.android.systemui|android.widget.FrameLayout||
. com.android.systemui:id/backdrop|android.widget.FrameLayout||
.   com.android.systemui|android.widget.FrameLayout||
.     com.android.systemui|android.view.View||
. com.android.systemui:id/captured_blur_container|android.view.View||
. com.android.systemui:id/scrim_behind|android.view.View||
. com.android.systemui:id/light_reveal_scrim|android.view.View||
. com.android.systemui:id/scrim_in_front|android.view.View||
. com.android.systemui:id/notification_panel|android.widget.FrameLayout||
.   com.android.systemui:id/keyguard_long_press|android.view.View||
.   com.android.systemui|android.widget.FrameLayout||
.   com.android.systemui|android.widget.FrameLayout||
.
```

Figure 42: Screenshot of error detection case

5.1.3 Scalability Assessment

The next important metric was the scalability of the automated testing tool. It was assessed by testing its capability to run tests on various devices.

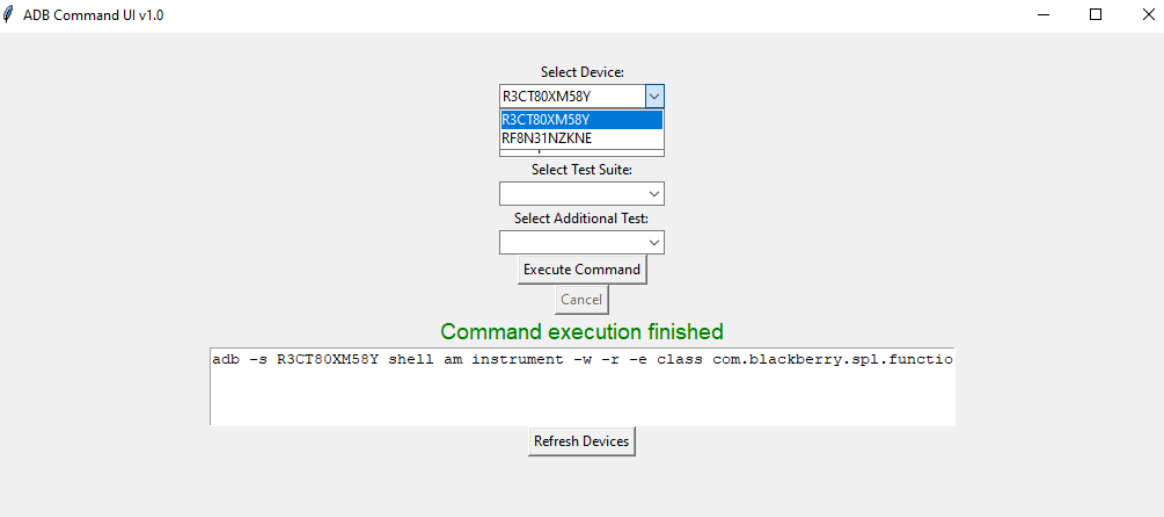

Figure 43: Screenshot of using multiple devices for testing

The designed tool was tested on many Android devices with different OS versions. This compatibility testing confirmed that the tool could handle the diverse range of Android OS. It also showed the tool's ability to support different hardware architectures of different devices.

5.1.4 Failure Analysis

Analyzing the failures was important to understand the designed prototype's limitations. It would also help to find future areas of improvement.

```
INSTRUMENTATION_STATUS: test=testSendReceiveWithAttachment
INSTRUMENTATION_STATUS_CODE: -2
INSTRUMENTATION_RESULT: stream=

Time: 78.008
There was 1 failure:
1)
testSendReceiveWithAttachment(com.blackberry.spl.functionaltest.test.spec
ific.HubTest)
com.blackberry.spl.functionaltest.support.VerificationError: Unrecognized
file selector.
com.android.settings|android.widget.FrameLayout||
. com.android.settings|android.widget.LinearLayout||
.   com.android.settings|android.widget.FrameLayout||
```

Figure 44: Screenshot of a failure case

Several tests did not pass because the tool could not identify certain UI elements. For example, the `testSendReceiveWithAttachment` where the “Open with option: SmartOffice” was not detected. This suggested potential problems with how tests are being called. This also indicated that there was a need to improve the test configurations.

5.2 Discussion

The results suggest that the designed automated testing tool prototype provides improvement compared to manual testing in terms of efficiency, accuracy and scalability. However, the results also indicate some limitations of the tool.

5.2.1 Benefits of Automation

There are some notable benefits found in using the automated testing tool instead of traditional manual smoke testing. Automated smoke testing reduces the time required for test execution. As a result, more frequent testing cycles are possible. Automated tests can also provide more accurate error detection by removing mistakes caused by manual testing. This is important to identify bugs for new system updates. The tool’s automated logging system improves the ability to detect errors faster.

5.2.2 Challenges Encountered

Although most of the results were positive, some challenges were encountered. Certain tests because of the tool’s inability to recognize specific UI elements (`testSendReceiveWithAttachment`). This emphasizes the need for improved object recognition algorithms. The variability in UI designs in Android applications causes challenges. This suggests possible problems with test configuration or execution logic. So, further improvement of the test configuration should be done to execute all the tests successfully.

5.2.3 Comparative Analysis of User Experience

The testers provided their important opinion after using both manual and automated testing methods.

Although manual testing offers more flexibility, it is time-consuming and prone to human error. It is often difficult to reproduce the same testing conditions in manual testing. As a result, variability in results is seen for using manual testing.

The designed tool is ideal for routine validation testing because of its successful repetitive test execution ability. However, it requires ongoing maintenance to keep the tests effective with the evolving applications.

5.3 Future Work

Several areas for future work have been identified to improve this designed smoke testing prototype.

5.3.1 Enhanced UI Object Recognition

The prototype should be able to better detect different types of UI components. This can be achieved by using machine learning algorithms. A larger data set containing various types of UI elements can be used to train the prototype. This should result in a better detection of UI elements.

5.3.2 Comprehensive Test Suite Development

A more extensive test suite can be created to include a wider variety of scenarios. It will improve the tool's efficiency.

Less common but critical scenario tests can be added so that the tool can handle all possible use cases. These cases can detect hidden bugs. Detecting these bugs can improve the performance of the prototype.

Performance and load testing capabilities can be added to evaluate the application’s behavior under stress conditions. Performance tests can measure stability during heavy usage.

5.3.3 Continuous Integration and Deployment

A CI/CD pipeline can be incorporated with the automated testing prototype. This will improve the efficiency of the testing process. This will also help to detect errors early. Integration with CI/CD tools such as Jenkins, Travis CI or GitLab CI can automate the testing workflow to provide immediate feedback.

Automated reporting and alerting mechanisms can be implemented to provide real-time feedback. These reports can be integrated with development tools. This will ensure that developers are quickly informed about any issues.

5.3.4 User Feedback Incorporation

User feedback should be collected to improve the prototype and resolve any emerging problem. Feedback from testers on the tool’s performance can be gathered through surveys and direct interactions with them. The prototype should be regularly updated based on the collected feedback.

This chapter provided a comprehensive overview of the results and discussions of the automated testing tool along with a plan for future improvements.

6 Conclusion

In the beginning, this thesis aimed to research and develop a GUI-based automated smoke testing tool for Android devices using ADB. The aim of designing this prototype was to enhance efficiency and improve the accuracy of test results. The primary objectives were mentioned in Chapter 1. The subsequent chapters discussed about the literature review, case study, conceptual design, implementation and evaluation of this prototype. This final chapter will discuss about the overall process and achievements. This chapter will also try to answer the primary research questions based on the whole content of this thesis.

The thesis was successful in achieving its primary objectives. The implemented GUI-based automated testing tool simplified the testing process for testers and made it more user-friendly.

The initial project plan was executed with minor adjustments. Some adjustments were required to solve unexpected technical challenges like compatibility issues with different Android devices.

Previously acquired knowledge about software development and Android application testing was required for building the tool. The project also required learning new skills. Integrating ADB with a user-friendly GUI required new knowledge. The combination of skills already possessed and those recently acquired played a vital role in the project's achievement.

The achievement of goals can be discussed by answering the research questions of the first chapter. The first research question asked how can a GUI-based tool be designed to simplify the use of ADB in automated testing of Android devices. The answer can be found from the design of the automated testing prototype discussed in Chapter 3. Python programming language and Tkinter library were used to create a graphical user interface (GUI) to simplify the use of ADB commands. As a result, testers can select devices and execute tests without needing to manually input complex commands. The second research question asked how the implementation of a GUI-based automated testing tool affects the efficiency of executing test cases compared to traditional manual testing methods. From Chapters 4 and 5, we can observe that the implementation of the prototype improved testing efficiency. The automated tool reduced execution time and minimized human error. That made the testing process faster and more reliable compared to manual methods. The third research question asked how the automated testing tool ensures the accuracy and reliability of test results for different Android devices with different configurations. To answer this question, the tool was tested on various Android devices with different configurations. This ensures its accuracy and reliability. Effective error detecting and detailed logging mechanisms were implemented to get accurate test results. The fourth research question asked how can the automated testing tool be scaled to support multiple devices effectively. This question was answered by successfully performing testing across multiple devices. This capability was crucial for handling numerous types of configurations of

Android devices. The final research question asked what are the results of experiments performed comparing the automated testing tool with manual testing methods in terms of error detection and resource utilization. From the experimental results, it is clear that the automated tool outperformed manual testing methods in error detection and resource utilization. The automated tool detected more errors in less time and utilized resources more efficiently.

In summary, the designed prototype of a GUI-based automated testing tool for Android devices using ADB can be a major upgrade from traditional manual testing methods. Implementing this tool can show an improved software development lifecycle of the company through its user-friendly interface and efficiency.

References:

Sarkar, A., Goyal, A., Hicks, D., Sarkar, D. and Hazra, S. (2019). Android Application Development: A Brief Overview of Android Platforms and Evolution of Security Systems. [online] IEEE Xplore. doi:https://doi.org/10.1109/I-SMAC47947.2019.9032440.

AppMySite (2022). Android vs iOS: Mobile Operating System market share statistics you must know. [online] AppMySite. Available at: https://www.appmysite.com/blog/android-vs-ios-mobile-operating-system-market-share-statistics-you-must-know/#:~:text=How%20many%20Android%20users%20are.

Mayrhofer, R., Stoep, J.V., Brubaker, C. and Kralevich, N. (2019). The Android Platform Security Model (2023). arXiv (Cornell University). doi:https://doi.org/10.48550/arxiv.1904.05572.

Schmerl, B.R., Gennari, J., Sadeghi, A., Bagheri, H., Malek, S., Cámara, J. and Garlan, D. (2016). Architecture Modeling and Analysis of Security in Android Systems. Lecture Notes in Computer Science, pp.274–290. doi:https://doi.org/10.1007/978-3-319-48992-6_21.

Motan, M. and Zein, S. (2020). Android App Testing: A Model for Generating Automated Lifecycle Tests. 2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). doi:https://doi.org/10.1109/ismsit50672.2020.9254285.

Li, L., Bissyandé, T.F., Klein, J. and Traon, Y.L. (2015). An Investigation into the Use of Common Libraries in Android Apps. arXiv (Cornell University). doi:https://doi.org/10.48550/arxiv.1511.06554.

Li, L., Gao, J., Hurier, M., Kong, P., Bissyandé, T.F., Bartel, A., Klein, J. and Traon, Y.L. (2017). AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. arXiv (Cornell University). doi:https://doi.org/10.48550/arxiv.1709.05281.

Backes, M., Sven Bugiel, Derr, E., McDaniel, P.D., Octeau, D. and Weisgerber, S. (2016). On Demystifying the Android Application Framework: Re-Visiting Android Permission Specification Analysis. USENIX Security Symposium, pp.1101–1118.

Mazuera-Rozo, A., Bautista-Mora, J., Linares-Vásquez, M., Rueda, S. and Bavota, G. (2019). The Android OS stack and its vulnerabilities: an empirical study. Empirical Software Engineering, 24(4), pp.2056–2101. doi:https://doi.org/10.1007/s10664-019-09689-7.

Readthedocs.io. (2023). 9. Activities and intents — Android App Development Documentation 1 documentation. [online] Available at: https://android-app-development-documentation.readthedocs.io/en/latest/intents.html [Accessed 4 Jul. 2024].

Activity, an (2024). Head First Android Development. [online] O'Reilly Online Learning. Available at: https://www.oreilly.com/library/view/head-first-android/9781449362171/ch04.html [Accessed 14 Jul. 2024].

Readthedocs.io. (2023). 10. Activity Lifecycle — Android App Development Documentation 1 documentation. [online] Available at: https://android-app-development-documentation.readthedocs.io/en/latest/activitylifecycle.html [Accessed 10 Jul. 2024].

Ghanem, T. and Zein, S. (2020). A Model-based approach to assist Android Activity Lifecycle Development. 2020 4th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT). doi:https://doi.org/10.1109/ismsit50672.2020.9254687.

TechAhead. (2024). Exploring Android App Components: Creating Dynamic User Experiences | TechAhead. [online] Available at: https://www.techaheadcorp.com/blog/exploring-android-app-components-creating-dynamic-user-experiences/#servicesthebackgroundworkers [Accessed 3 Jul. 2024].

Android Developers (2019). Application Fundamentals | Android Developers. [online] Android Developers. Available at: https://developer.android.com/guide/components/fundamentals.

Android Developers. (n.d.). Test your app. [online] Available at: https://developer.android.com/studio/test.

Wang, H., Li, H., Li, L., Guo, Y. and Xu, G. (2018). Why are Android apps removed from Google Play? doi:https://doi.org/10.1145/3196398.3196412.

Kong, P., Li, L., Gao, J., Liu, K., Bissyande, T.F. and Klein, J. (2019). Automated Testing of Android Apps: A Systematic Literature Review. IEEE Transactions on Reliability, [online] 68(1), pp.45–66. doi:https://doi.org/10.1109/tr.2018.2865733.

Imparato, G. (2015). A Combined Technique of GUI Ripping and Input Perturbation Testing for Android Apps. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. doi:https://doi.org/10.1109/icse.2015.241.

Li, L., Bartel, A., Bissyande, T.F., Klein, J., Le Traon, Y., Arzt, S., Rasthofer, S., Bodden, E., Octeau, D. and McDaniel, P. (2015). IccTA: Detecting Inter-Component Privacy Leaks in Android Apps. 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering. doi:https://doi.org/10.1109/icse.2015.48.

Li, L., Bissyandé, T.F., Wang, H. and Klein, J. (2018). CiD: automating the detection of API-related compatibility issues in Android apps. Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. doi:https://doi.org/10.1145/3213846.3213857.

Mirzaei, N., Malek, S., Păsăreanu, C.S., Esfahani, N. and Mahmood, R. (2012). Testing android apps through symbolic execution. ACM SIGSOFT Software Engineering Notes, 37(6), pp.1–5. doi:https://doi.org/10.1145/2382756.2382798.

Song, W., Qian, X. and Huang, J. (2017). EHBDroid: Beyond GUI testing for Android applications. [online] IEEE Xplore. doi:https://doi.org/10.1109/ASE.2017.8115615.

Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S. and Memon, A.M. (2012). Using GUI ripping for automated testing of Android applications. Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering - ASE 2012. doi:https://doi.org/10.1145/2351676.2351717.

Takala, T., Katara, M. and Harty, J. (2011). Experiences of System-Level Model-Based GUI Testing of an Android Application. [online] IEEE Xplore. doi:https://doi.org/10.1109/ICST.2011.11.

Afzal, W., Torkar, R. and Feldt, R. (2009). A systematic review of search-based testing for non-functional system properties. Information and Software Technology, [online] 51(6), pp.957–976. doi:https://doi.org/10.1016/j.infsof.2008.12.005.

Mahmood, R., Mirzaei, N. and Malek, S. (2014). EvoDroid: segmented evolutionary testing of Android apps. Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. doi:https://doi.org/10.1145/2635868.2635896.

Hamlet, R. (2002). Random Testing. Encyclopedia of Software Engineering. doi:https://doi.org/10.1002/0471028959.sof268.

Anand, S., Naik, M., Mary Jean Harrold and Yang, H. (2012). Automated concolic testing of smartphone apps. Foundations of Software Engineering. doi:https://doi.org/10.1145/2393596.2393666.

Lin, C.-C., Li, H., Zhou, X. and Wang, X.F. (2014). Screenmilker: How to Milk Your Android Screen for Secrets. doi:https://doi.org/10.14722/ndss.2014.23049.

Yang, L., Wang, L. and Zhang, D. (2017). Malicious Behavior Analysis of Android GUI Based on ADB. doi:https://doi.org/10.1109/cse-euc.2017.211.

Easttom, C. and Sanders, W. (2019). On the Efficacy of Using Android Debugging Bridge for Android Device Forensics. 2019 IEEE 10th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON). doi:https://doi.org/10.1109/uemcon47517.2019.8992948.

Madhuri Kishan Kulkarni and Soumya A (2016). Deployment of Calabash Automation Framework to Analyze the Performance of an Android Application. IJSRD : international journal for scientific research and development, 2(3), pp.70–75.

Calysta Merina, Nenny Anggraini and Nashrul Hakiem (2018). A Comparative Analysis of Test Automation Frameworks Performance for Functional Testing in Android-Based Applications using the Distance to the Ideal Alternative Method. 2018 Third International Conference on Informatics and Computing (ICIC). doi:https://doi.org/10.1109/iac.2018.8780548.

Blundell, P., Milano, Diego Torres and Blundell, Paul (2015). Learning Android Application Testing. Packt Publishing.

Halani, K.R., Kavita and Saxena, R. (2021). Critical Analysis of Manual Versus Automation Testing. [online] IEEE Xplore. doi:https://doi.org/10.1109/ComPE53109.2021.9752388.

Hsia, P., Gao, J., Samuel, J., David Chenho Kung, Toyoshima, Y. and Chen, C. (1994). Behavior-based acceptance testing of software systems: a formal scenario approach. doi:https://doi.org/10.1109/cmpsac.1994.342789.

Hooda, I. and Singh Chhillar, R. (2015). Software Test Process, Testing Types and Techniques. International Journal of Computer Applications, [online] 111(13), pp.10–14. doi:https://doi.org/10.5120/19597-1433.

Syed Umer Jan, Shah, A., Zia Ullah Johar, Shah, Y. and Khan, F. (2016). An Innovative Approach to Investigate Various Software Testing Techniques and Strategies. 2(2), pp.682–689. doi:https://doi.org/10.32628/ijsrset1622210.

Nidhra, S. and Dondeti, J. (2012). Black Box and White Box Testing Techniques - A Literature Review. International Journal of Embedded Systems and Applications, 2(2), pp.29–50. doi:https://doi.org/10.5121/ijesa.2012.2204.

Techopedia.com. (2019). What is Automated Testing? - Definition from Techopedia. [online] Available at: https://www.techopedia.com/definition/17785/automated-testing.

Bezbaruah, A., Pratap, B. and Hake, S.B. (2020). Automation of Tests and Comparative Analysis between Manual and Automated testing. 2020 IEEE Students Conference on Engineering & Systems (SCES). doi:https://doi.org/10.1109/sces50439.2020.9236748.

Abbas, R., Sultan, Z. and Bhatti , S.N. (2017). Comparative Study of Load Testing Tools: Apache JMeter, HP LoadRunner, Microsoft Visual Studio (TFS), Siege. Sukkur IBA Journal of Computing and Mathematical Sciences, 1(2), p.102. doi:https://doi.org/10.30537/sjcms.v1i2.24.

Alferidah, S.K. and Ahmed, S. (2020). Automated Software Testing Tools. [online] IEEE Xplore. doi:https://doi.org/10.1109/ICCIT-144147971.2020.9213735.

Negara, N. and Stroulia, E. (2012). Automated Acceptance Testing of JavaScript Web Applications. [online] IEEE Xplore. doi:https://doi.org/10.1109/WCRE.2012.41.

Holmes, A. and Kellogg, M. (2006). Automating functional tests using Selenium. [online] IEEE Xplore. doi:https://doi.org/10.1109/AGILE.2006.19.

Leckraj Nagowah and Purmanand Roopnah (2010). AsT-A simple automated system testing tool. doi:https://doi.org/10.1109/iccsit.2010.5563986.

Miao, Y. and Yang, X. (2010). An FSM based GUI test automation model. doi:https://doi.org/10.1109/icarcv.2010.5707766.

Vasilyev, A., Ilya Paramonov and Sergey Averkiev (2017). Method and tools for automated end-to-end testing of applications for sailfish OS. DOAJ (DOAJ: Directory of Open Access Journals). doi:https://doi.org/10.23919/fruct.2017.8071350.

Root Info Solutions. (2017). Robotium Testing for Android App Development - Rootinfosol. [online] Available at: https://rootinfosol.com/how-to-use-robotium-testing-tool-for-android-apps [Accessed 11 Jul. 2024].

Ashish Bhargava (2013). Designing and implementing test automation frameworks with qtp. Packt Publishing Limited.

Méndez-Porras, A., Alfaro-Velasco, J. and Martínez, A. (2020). Evaluation of the Automated Testing Framework: A Case Study. Revista Tecnología en Marcha. doi:https://doi.org/10.18845/tm.v33i3.4372.

Choi, W., Necula, G.C. and Sen, K. (2013). Guided GUI testing of android apps with minimal restart and approximate learning. Conference on Object-Oriented Programming Systems, Languages, and Applications. doi:https://doi.org/10.1145/2509136.2509552.

Bay, H., Ess, A., Tuytelaars, T. and Van Gool, L. (2008). Speeded-Up Robust Features (SURF). Computer Vision and Image Understanding, [online] 110(3), pp.346–359. doi:https://doi.org/10.1016/j.cviu.2007.09.014.

Hanna, M., El-Haggar, N. and Sami, M. (2014). A Review of Scripting Techniques Used in Automated Software Testing. International Journal of Advanced Computer Science and Applications, 5(1). doi:https://doi.org/10.14569/ijacsa.2014.050128.

smartbear.com. (2024). Test Automation Frameworks. [online] Available at: https://smartbear.com/learn/automated-testing/test-automation-frameworks [Accessed 12 Jul. 2024].

Berihun, N.G., Dongmo, C. and Van der Poll, J.A.V. der (2023). The Applicability of Automated Testing Frameworks for Mobile Application Testing: A Systematic Literature Review. Computers, 12(5), p.97. doi:https://doi.org/10.3390/computers12050097.

Hayes, L.G. (2004). The automated testing handbook. Richardson, Tx: Software Testing Institute.

Jain, A.K., & Sharma, S. (2012). AN EFFICIENT KEYWORD DRIVEN TEST AUTOMATION FRAMEWORK FOR WEB APPLICATIONS.

Pareek, P., Chaturvedi, R., and Bhargava, H. (2015). A Comparative Study of Mobile Application Testing Frameworks.

Anusha, M. and Saravanan, K.N. (2017) 'Comparative Study on Different Mobile Application Frameworks', International Research Journal of Engineering and Technology (IRJET), 4(3), pp. 1299-1300.

Fazzini, M. (2018). Automated support for mobile application testing and maintenance. doi:https://doi.org/10.1145/3236024.3275425.

Machiry, A., Tahiliani, R. and Naik, M. (2013). Dynodroid: an input generation system for Android apps. Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013. doi:https://doi.org/10.1145/2491411.2491450.

Mohammad, D.R., Al-Momani, S., Tashtoush, Y.M. and Alsmirat, M. (2019). A Comparative Analysis of Quality Assurance Automated Testing Tools for Windows Mobile Applications. 2019 IEEE 9th Annual

Computing and Communication Workshop and Conference (CCWC). doi:https://doi.org/10.1109/ccwc.2019.8666463.

Lovreto, G., Endo, A.T., Nardi, P. and Vinicius H. S. Durelli (2018). Automated Tests for Mobile Games: An Experience Report. doi:https://doi.org/10.1109/sbgames.2018.00015.

Chauhan, R.K. and Singh, I. (2014). Latest Research and Development on Software Testing Techniques and Tools.

Bansal, A. (2014). A Comparative Study of Software Testing Techniques.

Mu, B., Zhan, M. and Hu, L.-F. (2009). Design and Implementation of GUI Automated Testing Framework Based on XML. doi:https://doi.org/10.1109/wcse.2009.91.

Tirodkar, A.A. and Khandpur, S.S. (2019). EarlGrey: iOS UI Automation Testing Framework. 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft). [online] doi:https://doi.org/10.1109/mobilesoft.2019.00010.

Sinaga, A.M., Wibowo, P.A., Silalahi, A. and Yolanda, N. (2018). Performance of Automation Testing Tools for Android Applications. [online] IEEE Xplore. doi:https://doi.org/10.1109/ICITEED.2018.8534756.

Kim, H., Choi, B. and Yoon, S. (2009). Performance testing based on test-driven development for mobile applications. doi:https://doi.org/10.1145/1516241.1516349.

Marín, B., Gallardo, C., Quiroga, D., Giachetti, G. and Serral, E. (2016). Testing of model-driven development applications. Software Quality Journal, 25(2), pp.407–435. doi:https://doi.org/10.1007/s11219-016-9308-8.

Ridene, Y. and Barbier, F. (2011). A model-driven approach for automating mobile applications testing. HAL (Le Centre pour la Communication Scientifique Directe). doi:https://doi.org/10.1145/2031759.2031770.

Vajak, D., Grbic, R., Vranjes, M. and Stefanovic, D. (2018). Environment for Automated Functional Testing of Mobile Applications. doi:https://doi.org/10.1109/sst.2018.8564626.

Jamil, M.A., Arif, M., Abubakar, N.S.A. and Ahmad, A. (2016). Software Testing Techniques: A Literature Review. 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M). [online] doi:https://doi.org/10.1109/ict4m.2016.045.

Choudhary, S.R., Gorla, A. and Orso, A. (2015). Automated Test Input Generation for Android: Are We There Yet? (E). [online] IEEE Xplore. doi:https://doi.org/10.1109/ASE.2015.89.

App (2024). App Development Costs. [online] Business of Apps. Available at: https://www.businessofapps.com/app-developers/research/app-development-cost/#:~:text=The%20testing%20stage%2C%20which%20is [Accessed 16 Jul. 2024].

Cherednichenko, S. (2021). What's The Cost To Maintain and Support An App in 2021? [online] Medium. Available at: https://medium.com/mobindustry/whats-the-cost-to-maintain-and-support-an-app-in-2021-dd4c2ea867eb [Accessed 17 Jul. 2024].

A. Alotaibi, A. and J. Qureshi, R. (2017). Novel Framework for Automation Testing of Mobile Applications using Appium. International Journal of Modern Education and Computer Science, 9(2), pp.34–40. doi:https://doi.org/10.5815/ijmecs.2017.02.04.

Wu, Z., Liu, S., Li, J. and Liao, Z. (2013). Keyword-Driven Testing Framework For Android Applications. Proceedings of the 2nd International Conference on Computer Science and Electronics Engineering (ICCSEE 2013). doi:https://doi.org/10.2991/iccsee.2013.275.

Salam, M.A., Taha, S. and Hamed, M.G. (2022). Advanced Framework for Automated Testing of Mobile Applications. [online] IEEE Xplore. doi:https://doi.org/10.1109/NILES56402.2022.9942374.

Garousi, V. and Elberzhager, F. (2017). Test Automation: Not Just for Test Execution. IEEE Software, 34(2), pp.90–96. doi:https://doi.org/10.1109/ms.2017.34.

Yalamanchili, S. and K. Sitha Kumari (2016). Comparison of manual and automatic testing using genetic algorithm for information handling system. doi:https://doi.org/10.1109/scopes.2016.7955752.

Lavingia K., Purohit, P., Dutta, V. and Lavingia, A. (2024). Advancements in automated testing tools for Android set-top boxes: a comprehensive evaluation and integration approach. International Journal of Systems Assurance Engineering and Management. doi:https://doi.org/10.1007/s13198-024-02335-6.

Davi Bernardo Silva, Andre Takeshi Endo, Marcelo Medeiros Eler and Vinicius H. S. Durelli (2016). An analysis of automated tests for mobile Android applications. doi:https://doi.org/10.1109/clei.2016.7833334.

Memon, A.M. (2002). GUI testing: pitfalls and process. Computer, 35(8), pp.87–88. doi:https://doi.org/10.1109/mc.2002.1023795.

Memon, A.M., Pollack, M.E. and Mary Lou Soffa (1999). Using a goal-driven approach to generate test cases for GUIs. International Conference on Software Engineering. doi:https://doi.org/10.1145/302405.302632.

Hu, C. and Neamtiu, I. (2011). Automating gui testing for android applications. Automation of Software Test. doi:https://doi.org/10.1145/1982595.1982612.

Zhu, H., Ye, X., Zhang, X. and Shen, K. (2015). A Context-Aware Approach for Dynamic GUI Testing of Android Applications. doi:https://doi.org/10.1109/compsac.2015.77.

Gu, R. and Rojas, J.M. (2023). An Empirical Study on the Adoption of Scripted GUI Testing for Android Apps. doi:https://doi.org/10.1109/asew60602.2023.00030.

Hsu, C.-W., Lee, S.-H. and Shieh S.W. (2017). Adaptive Virtual Gestures for GUI Testing on Smartphones. IEEE Software, 34(5), pp.22–29. doi:https://doi.org/10.1109/ms.2017.3641115.

Khan, M.K. and Bryce, R. (2022). Android GUI Test Generation with SARSA. 2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC). doi:https://doi.org/10.1109/ccwc54503.2022.9720807.

Banerjee, I., Nguyen, B., Garousi, V. and Memon, A. (2013). Graphical user interface (GUI) testing: Systematic mapping and repository. Information and Software Technology, 55(10), pp.1679–1694. doi:https://doi.org/10.1016/j.infsof.2013.03.004.

Hicinbothom, J.H. and Zachary, W.W. (1993). A Tool for Automatically Generating Transcripts of Human-Computer Interaction. Proceedings of the Human Factors and Ergonomics Society Annual Meeting, 37(15), pp.1042–1042. doi:https://doi.org/10.1177/154193129303701514.

Elbaum, S., Rothermel, G., Karre, S. and Fisher II, M. (2005). Leveraging user-session data to support Web application testing. IEEE Transactions on Software Engineering, 31(3), pp.187–202. doi:https://doi.org/10.1109/tse.2005.36.

Chu E. T.-H. and Lin, J.-Y. (2018). Automated GUI Testing for Android News Applications. doi:https://doi.org/10.1109/is3c.2018.00013.

Yeh, T., Chang, T.-H. and Miller, R.C. (2009). Sikuli: Using GUI screenshots for search and automation.

Lin, Y.-D., Rojas, J.F., Edward T.-H. Chu and Lai, Y.-C. (2014). On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices. IEEE Transactions on Software Engineering, 40(10), pp.957–970. doi:https://doi.org/10.1109/tse.2014.2331982.

Zaraket, F., Masri, W., Adam, M., Hammoud, D., Hamzeh, R., Farhat, R., Khamissi, E. and Noujaim, J. (2012). GUICOP: Specification-Based GUI Testing. doi:https://doi.org/10.1109/icst.2012.168.

Coppola, R., Morisio, M., and Torchiano, M. (2017). Scripted GUI Testing of Android Apps: A Study on Diffusion, Evolution and Fragility. arXiv (Cornell University). doi:https://doi.org/10.48550/arxiv.1711.03565.

Zelenchuk, D. (2019). Android Espresso Revealed. Apress.

Baek, Y. M. and Bae, D.-H. (2016). Automated model-based Android GUI testing using multi-level GUI comparison criteria. doi:https://doi.org/10.1145/2970276.2970313.

G, R.S., P, M.K.H. and G, M.A. (2022). Smoke Test Execution in Software Application Testing. [online] IEEE Xplore. doi:https://doi.org/10.1109/ICERECT56837.2022.10059686.

ISTQB not-for-profit association. (2019). Mobile Application Testing. [online] Available at: https://www.istqb.org/certifications/mobile-tester.

Istqb.org. (2024). ISTQB Glossary. [online] Available at: https://glossary.istqb.org/en_US/search?term=&exact_matches_first=true&page=30 [Accessed 23 Aug. 2024].

Appendix

This appendix provides a directory listing of all related artifacts in ZIP file uploaded to Sciebo and Git. These artifacts consist of the source code, sample output files, necessary diagrams of the prototype and the final copy of this thesis.

Location of ZIP file:

Sciebo: https://th-koeln.sciebo.de/s/1umooXUrwQjhsd7

Github Private Repository: https://github.com/shaira22/ADBbasedGUItool