**Research, conceptual design, and prototypical development of a tool with GUI for ADB-based automated testing of Android devices**

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 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 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 software 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 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 thoroughly 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 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 personal 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 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 [15] (after [Sarkar et al. 2019], p. 1). 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 [43] (after [Mayrhofer et al. 2019], p. 1). End users can install or uninstall apps by themselves which makes the overall system more dynamic and flexible [50] (after [Schmerl et al. 2016], p. 1). 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 [14] (after [Motan and Zein 2020], p. 1). Because of its customization properties and flexibility developers can easily deploy applications across multiple platforms without major code changes [15] (after [Sarkar et al. 2019], p. 3). 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 [107](after [Li et al. 2015], p. 1). 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 [108] (after [Li et al. 2017], p. 3, 4). 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 [15] (after [Sarkar et al. 2019], p. 1).

2.1.1 Android Framework Architecture

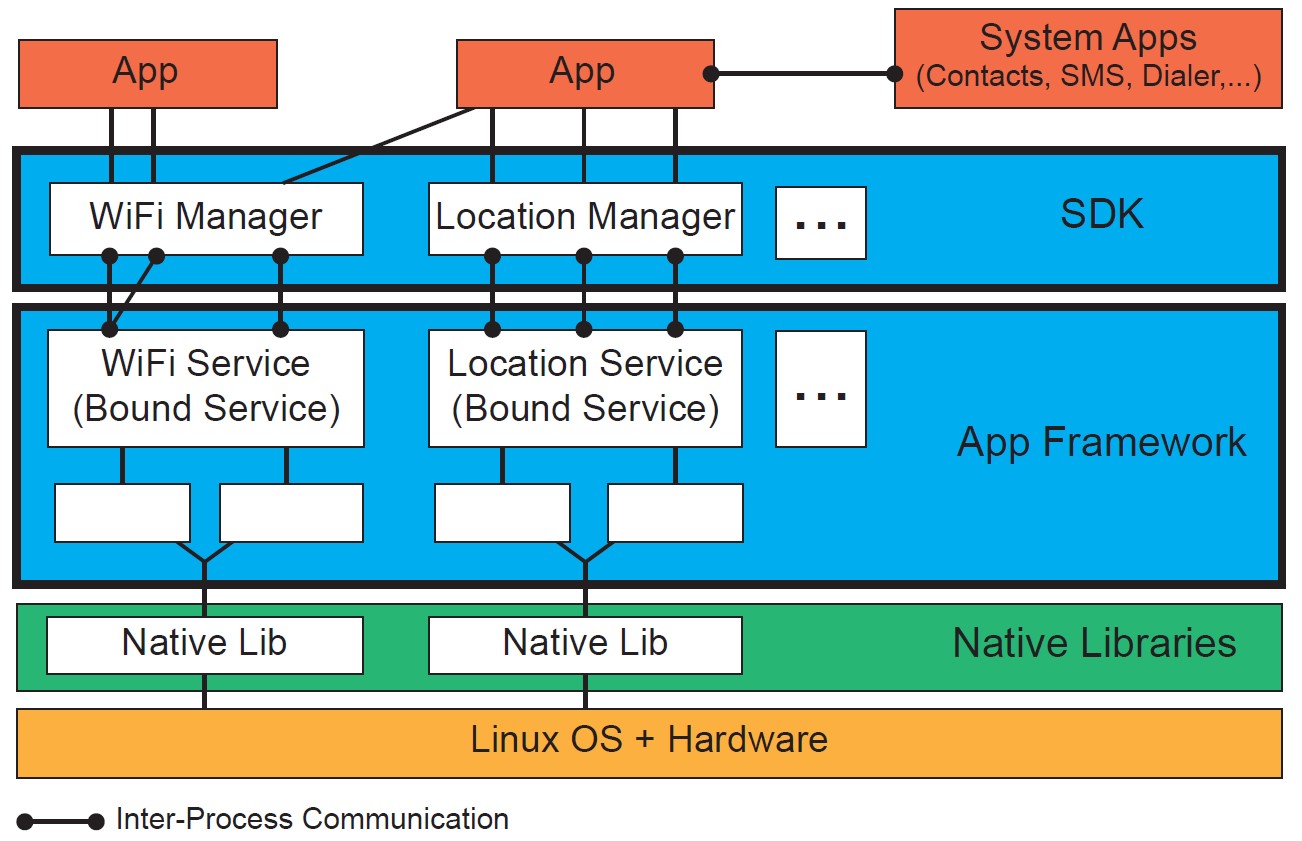
The Android Framework has the layered architecture shown in Figure 1.

Figure 1: Android Software Stack with abstract control and data flows. [47] (from [Backes et al. 2016])

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 [48]. This layer also includes various hardware components CPU, memory, and device-specific peripherals [47] (after [Backes et al. 2016], p. 1103).

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 [48]. An example is SSL for secure communication and the SQLite library for accessing the database [47] (after [Backes et al. 2016], p. 1102).

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 [47] (after [Backes et al. 2016], p. 1102, 1103).

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 [47] (after [Backes et al. 2016], p. 1103).

.

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. [48]

Apps communicate with each other and the framework services using Inter-Process Communication (IPC) mechanisms which is vital for maintaining security and resource management [47] (after [Backes et al. 2016], p. 1102).

.

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. [104]

Figure 2 displays various states an Android activity goes through from launch to shutdown. 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 [103]. onStart() is called when the activity becomes visible to the user [102]. onResume() is called when the activity starts interacting with the user. At this point, the activity is at the top of the activity stack [102]. onPause() is called when the system is about to start another activity [103]. onStop() is called when the activity is no longer visible to the user [102]. onRestart() is called when the activity restarts after being stopped [102]. onDestroy() is called before the activity is destroyed [102]. This is the final cleanup method [102].

There are different Activity Lifecycle States. The entire Lifetime is from onCreate() to onDestroy(). The activity is created and destroyed once during this period [49]. Visible Lifetime is from onStart() to onStop(). In this state, the activity is visible to the user but not necessarily in the foreground [49]. Foreground Lifetime is from onResume() to onPause(). The activity is in the foreground in this state, interacting with the user [49].



Figure 2: Android Activity Lifecycle [102]

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 [50] (after [Schmerl et al. 2016], p.6). For instance, playing music while navigating other programs or finding location data even when the app is not actively displayed on the screen [105].

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 [106]. Bound Services allow components to bind to them using the bindService() method which provides a client-server interface to interact with the service [106].

Content providers manage and encapsulate data. They provide security mechanisms like read and write permissions [50] (after [Schmerl et al. 2016], p.6). They allow applications to query and share data which is either stored privately within their own environment or shared across multiple applications [105]. As a result, seamless data sharing and management of crucial app information is possible [105]. Architecturally, permissions to read and modify the data are differentiated within these components [50] (after [Schmerl et al. 2016], p.6). 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 [106]. Data Sharing facilitates sharing of data such as contacts, images, or media files between applications [106].

Broadcast receivers handle system-level events, such as completion of device boot or low battery level notifications [50] (after [Schmerl et al. 2016], p.6). They serve as gateways into users’ apps and manage signals like low battery alerts or Wi-Fi connectivity status changes [105]. Differing from activities, broadcast receivers can exclusively obtain a specific subset of intent types known as standard broadcast actions [50] (after [Schmerl et al. 2016], p.6). 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 [106]. With Custom Broadcasts, applications can create and send their own broadcast messages to communicate with other apps or within the same app [106].

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 [122]. 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 [109][110][111]. So, the purpose of Android app testing is to ensure the functionality, usability, and compatibility of apps across different Android devices [24]. Effective testing methods are required to improve the security and reliability of these applications [3].

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) [112]. The next challenge is that diverse OS versions and device types complicate testing strategies [113] [114]. The large number of apps require scalable testing approaches [24]. Another challenge is generating comprehensive test cases. It is challenging due to the event-driven nature and framework libraries of Android apps [115]. The complex UI events are also challenging, for example, generating events like drag and hover is difficult, as these require precise conditions [33]. Developers often face another unique challenge due to the specific components and lifecycle management required by Android applications which leads to potential bugs [1].

Figure 3 illustrates the typical process involved in testing Android apps. This process comprises five key steps. The first step is installing the target Android application onto a device. This device can be either a physical Android device or an emulator that simulates the Android environment [24]. This installation is necessary to prepare the app for testing. It allows subsequent steps to interact with the app directly [24]. The second step is test case generation. The app is analyzed to generate test cases in this step. Test cases are a set of conditions or variables under which a tester can determine whether an application or software system is working correctly or not [24]. Depending on the testing approach, the generation of test cases can be done automatically or manually. Automated test case generation is often preferred for efficiency [24]. Some testing techniques, such as automated random testing, do not require pre-knowledge of the app for generating test cases [24].

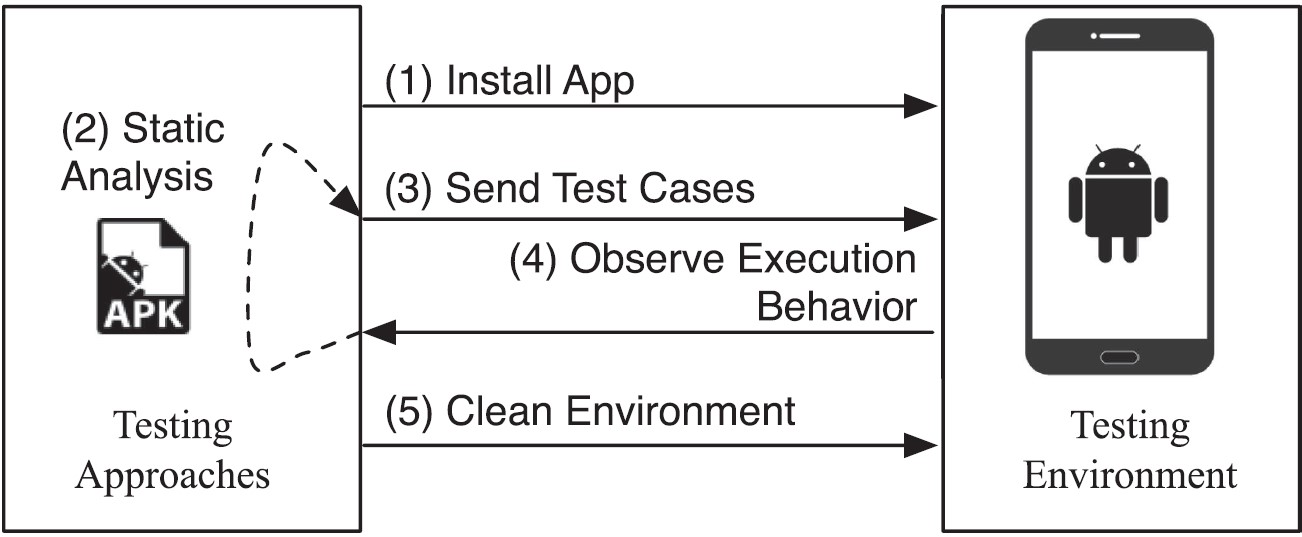


Figure 3 Testing process of android app [24]

The third step is about the execution of Test Cases. In this step, the generated test cases are sent to the Android device to execute them against the target app [24]. During this step, the app is examined with different inputs and scenarios specified by 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 [24]. The fourth step is about observation and collection of execution behavior. As the test cases are executed, the behaviors of the app are observed and recorded from different perspectives [24]. This observation stage involves monitoring the app's performance, responsiveness, stability, and correctness under different conditions [24]. Data collected stage during this step can include logs, screenshots, performance metrics, and error reports which are essential for identifying and diagnosing issues [24]. 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 [24]. 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 [24].

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

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 [24]. Takala et al. described their experiences of applying model-based GUI testing to Android apps, covering implementation, modeling, and execution aspects [116].

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 [117]. Mahmood et al. developed an evolutionary testing framework for Android apps that combines genetic and hill-climbing techniques to maximize code coverage [118].

Random Testing: This technique tests programs by generating random inputs and comparing the outputs to a specification to determine pass or fail [119].

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 [24].

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 [24].

Concolic Testing: This is a hybrid technique that combines symbolic execution and concrete execution paths to verify software [120].

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 [24].

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 [121]. It has a set of unique features which are specified in the Android permission system [39]. 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 [39].

ADB consists of three main components and they are Client, Daemon (adbd) and Server. Client sends commands from your development machine [40]. Daemon (adbd) runs commands as a background process on the Android device [40]. The server manages communication between the client and the daemon. It runs as a background process on the development machine [40].

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 [40].

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 [40].

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 [40]. 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 [40].

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 [123]. Many mobile applications are developed under some constraints such as budget, time, and resources. They are often released quickly due to the market pressure [4]. 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. [124]. Therefore, conducting functional or acceptance testing during the development of mobile applications is essential to improve their security and reliability [125].

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 [32].

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 [137].

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 [32].

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 [32].

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 [32].

Condition Validity Testing: It is also beneficial to perform tests for erroneous conditions to see how the product behaves and to identify most problems [32].

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 [32].

Testers have two options for testing. They are manual testing and using test scripts for automation [123]. 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 [32].

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 [167].

Regression Testing: It is performed mainly during the maintenance phase to ensure that new code does not affect existing functionalities. This technique can also be automated [32].

Acceptance Testing: 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 [168].

Alpha and Beta Testing: Alpha testing 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 [32].

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 [169].

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 [170].

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 [32]. It provides more accurate user interface feedback and identifies issues that automated tests might miss [32]. Manual testing techniques are suitable for scenarios where automation is not technically feasible or cost effective [32]. It also allows testers to perform ad-hoc testing without pre-scripted test cases [32].

On the other hand, manual testing is usually very time consuming, especially for large scale regression testing [32]. It is also less reliable as it is prone to human error and may miss defects due to fatigue or oversight [32]. Moreover, manual testing techniques are non-reusable as test cases are not recorded for future use. This makes repeating tests laborious [32]. Manual testing is not suitable for some types of testing, such as performance or load testing, where automation is required for accuracy [32].

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 [126]. 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 [28]. Test automation is particularly beneficial for performing tests that are too complex or tedious to be done manually [4].

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 [131]. 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 [22]. 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 [131]. It takes more time to set up as it requires many steps and configurations [131].
* LoadRunner: It identifies performance bottlenecks and can simulate multiple users to check network performance [131]. LoadRunner has a high licensing cost which can result in a huge expense for any organization [131].
* Siege: It detects system performance under load and is faster to set up but it provides limited results [131]. Siege can sometimes generate inaccurate results, which may affect the reliability of the performance data collected during testing [131].
* 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 [131].

Acceptance Testing Tools: Acceptance testing is conducted to verify whether a software system meets the requirements and provides the required functionality [132]. There are various tools available for acceptance testing but the most commonly used is FIT (Framework for Integrated Test) [132].

* FIT (Framework for Integration Test): It is used for acceptance testing and allows customers to create test cases and classes that are automatically tested [132].

Functional Testing Tools: Functional testing is used to verify whether a website or web application is correctly performing all necessary functions or not [22]. 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 [133]. This makes tests easy to set up, record, edit, and debug [134]. Selenium tests can be fragile in case of dealing with GUI changes [133].
* FitNesse: It combines a testing tool, wiki, and web server to validate functionality and acceptance. It provides automatic comparisons to expected outcomes [133]. Keeping the tests organized can be challenging in it [133].

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 [22]. 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 [134].
* 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 [134]. QTP can require significant effort in maintaining test scripts [134].
* Sahi: It is a web application testing tool that supports script recording and playback. It is developed in Java and JavaScript [134].

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 [135]. These tools validate GUI functionality against system specifications. GUI will be discussed in more details 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 [136]. This type of testing includes various methods to verify the correctness of the application from the user's perspective [136]. Tools for end-to-end testing include Selenium and Robotium [136].

* Robotium: It is an end-to-end automated testing tool that provides guidance to developers on how to address testing tasks [136]. Robotium is not able of handling different applications in one test [164].

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 [4]. Choosing the right framework is important to maximize the efficiency and effectiveness of testing [128].

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 [34]. Figure 4 provides an overview of a sample automated testing framework for mobile applications leveraging user interaction features and historical bug data.

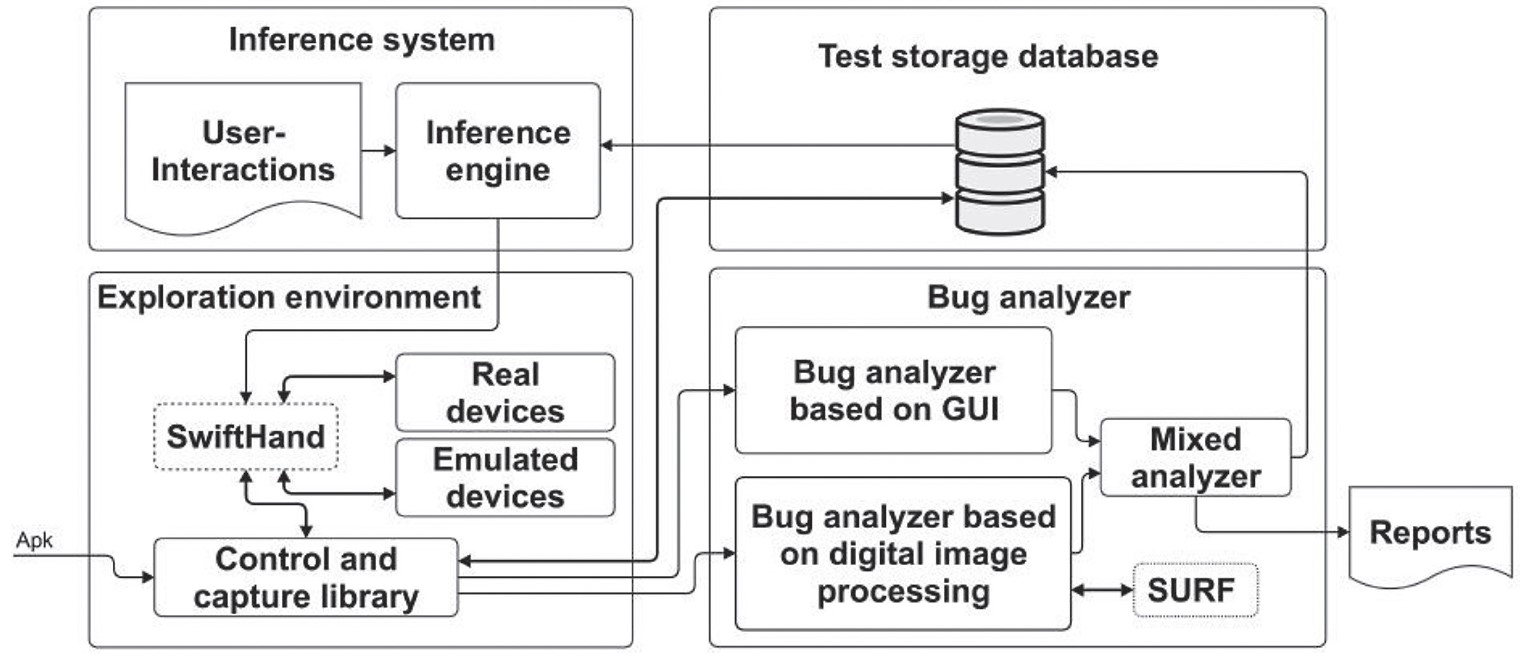


Figure 4: Components of the Automated Testing Framework [32]

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 [34]. A model of the application is created for automated exploration using SwiftHand [138]. It was modified to include user-interaction features and capture images and GUI information before and after interactions to detect potential bugs [34].

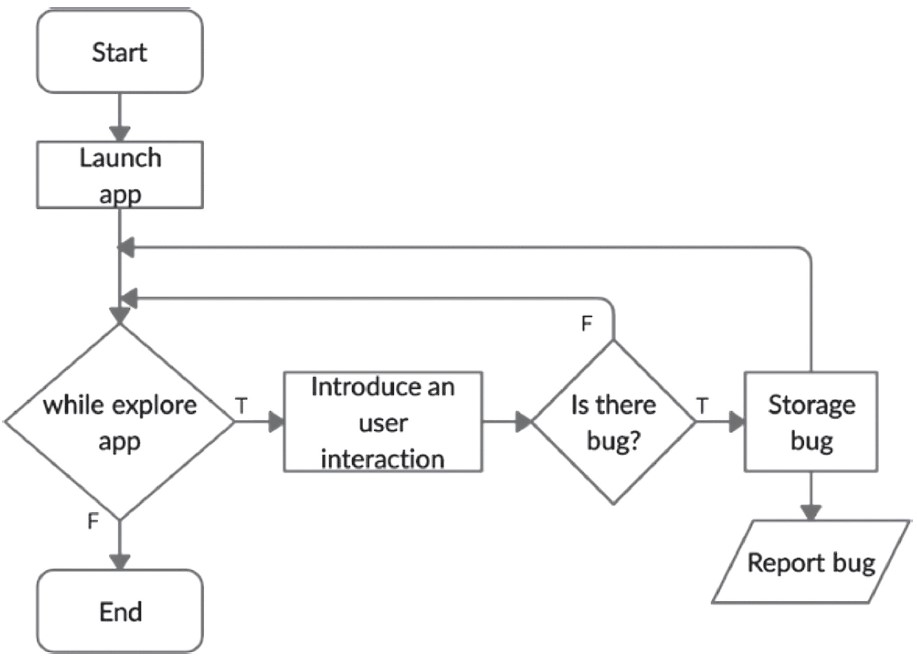
The inference engine introduces user interaction features during automated exploration. Two strategies are implemented here. The first one is the Random strategy where user interaction features are introduced randomly [34]. The second one is Frequency of Bugs by Widget strategy. Here, historical bug data is analyzed to identify widgets which are prone to bugs. Then corresponding user interaction features are introduced to find similar bugs in new applications [34].

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 [34]. The interest points detector (SURF) [139] 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 [34].

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 [34].

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 [32].

Figure 5: Flowchart of the automated testing framework [32]



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 [140, 141].

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 [140, 141].

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 [44].

Data-Driven Testing Framework: 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 [141, 142].

Keyword-Driven Testing Framework: 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 [140, 141, 143].

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 [143].

Some of the well-known frameworks for reliability are Espresso, Calabash, and Appium [4]. Espresso is primarily developed for Android UI testing by Google [4]. It supports parallel execution and uses Java for scripting [124]. The problem with Espresso is that it is limited to Android and does not support iOS [124]. Calabash supports both Android and iOS platforms [124]. It uses Ruby for scripting [124] and demonstrated the best overall performance [123]. However, it does not support parallel execution [124]. Appium supports Android, iOS, and mobile web applications [129]. It uses various programming languages for scripting [129]. It also supports parallel execution [124]. However, it is more complex to setup compared to the other frameworks [4].

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 is 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 [44]. 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 [144,145]. Functionality makes sure that an application meets its user expectations, reduces errors and maintains quality [146]. 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 [147, 148]. Compatibility checks whether an application runs on different hardware and operating system platforms [144, 145]. Scalability measures the ability of an application to handle increases in user traffic, data size and transaction frequency [149, 150]. 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 [151, 152].

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 [140, 123]. 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 [140, 155]. 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 [44]. Hybrid testing combines different frameworks to merge the advantages and lessen the disadvantages of each [44]. Model-driven testing automatically to generates code from a model. It also automates test case execution from a model which combines model-driven and domain-specific modeling language (DSML) concepts to improve mobile application quality [156, 157, 158].

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 [150,159]. White-box testing requires complete knowledge of the program structure. It is exhaustive, time-consuming and usually applied during unit testing [160, 148]. Grey-box testing combines aspects of both black-box and white-box testing. It uses detailed design documents and requirements information [161, 162]. Regression testing focuses on identifying negative side effects of code changes and re-executing affected tests [44].

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 [44]. The increasing size and complexity of mobile applications are causing significant quality issues [163]. 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 [44, 140]. Mobile app maintenance costs account for 15-20% of overall app development costs [74] but test automation frameworks can help to reduce these costs. Fixing defects, improving performance, upgrading and maintaining robustness are some of the maintenance works [153]. Despite available automation tools and approaches, mobile testing remains a time-consuming task [155]. Test automation itself requires development effort and significant time [44]. Proper use of automation tools can decrease testing time and make the process more efficient [10]. Fragmentation is another major challenge where users use different operating system versions of Android devices [166]. It is difficult to identify all issues that end-users might face. Therefore, fragmentation makes testing complicated [44].

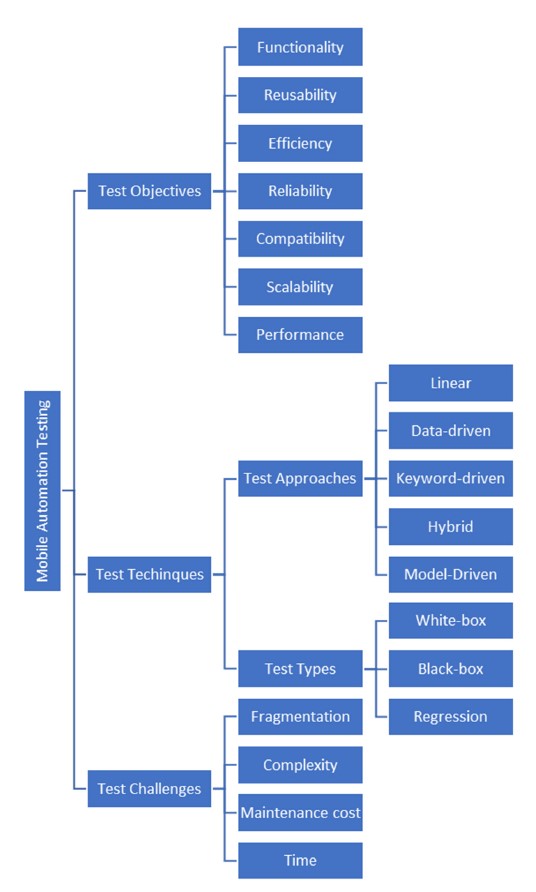


Figure 6: Taxonomy of Mobile Application Testing

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 [130].

Reduced Time and Cost: Automation replaces time-consuming manual testing method. It enables faster and more efficient application releases [11]. The cost of resource allocation is also reduced significantly by using automated testing.

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 [11].

Improved Accuracy and Reliability: Automation ensures the quality, reliability, and performance of increasingly complex mobile applications by eliminating human errors [12].

2.4.5 Recent Research in Automated Testing

Researchers are continuously trying 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.

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 [11].

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 [11].

2.5 Graphical User Interface (GUI)

In modern world almost all the 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 [177]. A recognized approach of 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 [178].

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) [176]. Mobile applications are event-driven systems that primarily interact with users through a GUI [6]. Android applications have complex GUIs and require thorough testing to ensure high quality [13]. GUI testing involves user interactions to achieve specific tasks within the app which makes it essential for app success in stores like Google Play [13]. GUI-based testing on smartphones is crucial because the primary input is user gestures [9]. Developers look for fast and automated GUI testing methods to ensure software quality across various platforms [9]. 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 [16]. Automated GUI testing is preferred because it requires less time and effort than manual testing [16].

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 [179]. Capture/replay techniques involve recording user interactions and converting these user application interactions into test scripts. These test scripts can be automatically replayed [180]. These techniques are commonly used for regression testing. They are highly popular in various fields like web application and desktop application testing [181]. Model-based techniques offer an alternative approach to generate 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 [179]. 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 [119].

2.5.1 Automated GUI Testing Tools and Frameworks

Automated GUI testing tools help ensure that mobile applications display correctly on different types devices and screen sizes. These tools simulate user gestures and validate the GUI changes through a test oracle [19]. 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 [173].

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 [174].

GUICOP: It uses GUI layout information instead of screen images for verification. It depends on specifications of GUI components to determine correctness. It compares the actual GUI layout with the expected layout based on specifications [175].

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 [19].

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) [13]. 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 [171]. UI Automator is often combined with Espresso to cover a broader range of test cases [172].

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 [16].

Another challenge occurs because of the diversity screen sizes and OS versions. Android applications must function for various devices with different screen sizes and OS versions which makes GUI testing complex [16]. Traditional tools like Sikuli [173] and SPAG-C [174] depend on static screenshots. They are limited by screen size differences which creates problem in accurately determining test results [16].

Another challenge is caused by dynamic GUIs. Modern Android apps contain numerous dynamically constructed GUIs. This makes accurate behavior modeling challenging [21]. Moreover, state explosion problems can occur for handling non-deterministically changing GUIs. This also makes the testing process complicated [21].

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 customizing with 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's provides a wide range of solutions for protecting voice and data communications. Key offerings are described below:

SecuSUITE for Government: It is 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 of 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.

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 protect 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 makings.

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 testing Android devices. This tool was created to solve some significant challenges faced by a company which 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. 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

The initial state of testing within the company was entirely manual. Testers had to manually execute predefined test cases on various devices. After executing the test case they needed to document each step and observing the results. This process was time consuming and prone to human error. High resource consumption was a major issue as manual testing required a lot of human resources. Multiple testers often worked on different devices at the same time. Inconsistency was another major challenge with manual testing. Human error used to lead to variations in test execution and results. As a result, reproducing and verifying issues became more difficult. Another persistent problem was scalability. Testing on multiple devices and configurations was not only difficult but also highly inefficient. This increased delays and potential gaps in coverage.

Figure 7: Steps of Manual Testing

There are several steps for manual testing from preparing the test environment to documenting results. The steps of manual testing are shown in figure 7. In preparation phase, the testers needed to identify test objectives, define test cases, set up test environment and install applications. In execution Phase, testers had to execute test cases, document gained results and retest if it was necessary. Compiling results, documenting issues and preparing test report were the tasks of reporting phase. Finally, reports were reviewed and finalized after considering feedback in review phase. This entire process was not only time consuming but also prone human error.

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

Automation also created the potential to perform tests that were impractical manually. For example, large scale regression tests for multiple devices could be automated to ensure that new changes do not introduce bugs. Automated tools could execute these tests overnight or during off hours. This maximized resource utilization and accelerated the development cycle. Automation has also enabled more advanced testing techniques like performance and load testing which were difficult to perform manually.

There are already a lot of Automated testing tools available as mentioned in chapter 2 but they have limitations like high set up 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 testing requirements and standards which are not readily available in the existing testing tools. Considering these limitations and company requirements, it was decided to build a new testing tool dedicated only for the use of company employees.

3.2 Conceptual Design of the Automated Testing Tool

3.2.1 Design Objectives and Requirements

Some specific objectives and requirements like enhancing efficiency and accuracy were considered for designing the automated testing tool. The primary goals of the tool were to automate the execution of test cases and provide a user-friendly interface to ensure comprehensive logging for analysis and debugging. The tool was designed to meet the needs of the testers of the company by simplifying the testing process and providing adequate support for various testing scenarios.

The design of the tool was focused on solving key problems identified during the manual testing phase. The design also aimed to create an intuitive interface that would be easy for testers to use. This is essential for minimizing learning effort and maximizing productivity. The system architecture was designed consisting of a client server model. For example, the GUI worked as an interface to clients while the back-end accomplished runs of ADB commands and logged results.

3.2.2 System Architecture

The system was divided into several main components. Each of them had a specific task. The GUI component was planned to develop using Tkinter. It would provide an interface for selecting devices, executing commands and viewing results. It was designed to be user friendly and responsive. It would also generate real time feedback to users. The command processor was planned to manage the execution of ADB commands. This would ensure accurate and efficient test execution. It could also handle the interaction with connected devices and processed the commands selected by the user. The logger was designed to be responsible for capturing and storing detailed logs of test execution which would aid in analysis and debugging. The logger could ensure that all outputs were documented to provide a detailed record of each test run.

3.2.3 GUI Design and User Interaction

The design of the GUI was based on the simplicity and responsiveness. It was planned to keep the interface simple and intuitive to reduce the learning time for new users. Clear prompts and instructions were set up to guide users through each step of the testing process and real time feedback was planned to keep them informed about the status of test execution.

The main 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 which ones to test. 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 provide real time updates on the status of test execution. This would ensure that users were always aware of the current state of the process. Users could view and export detailed logs of test execution. This functionality would prove to be very important for analyzing test results and diagnosing issues.

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.

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

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

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

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

Refresh Devices: It refreshes the list of connected devices.

\*\*Add more commands here if necessary.

\*\*Figures These figures show the screenshots of the GUI interface for demonstrating the various options available to testers after running the code.

The user interaction flow was designed to be easy to understand so that the users can easily navigate the tool and complete their tasks efficiently. The flow design included steps for selecting devices, choosing commands, executing tests and reviewing results. This clear flow would allow the tester 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 versatility and ease of use. Its rich collection of libraries and tools made it an ideal choice for developing the automated testing tool. Tkinter was chosen to use for creating a user-friendly interface as it is a standard GUI toolkit for Python. ADB, which is a versatile command line tool for managing Android devices, was selected for interacting with the tested devices.

The proposed development environment included tools like PyCharm or Visual Studio Code for writing and debugging the script and Git for version control and collaboration. These tools would provide a robust and efficient environment for developing the automated testing tool. PyCharm has its powerful code analysis and debugging features. This would be particularly useful for identifying and fixing issues during development. Visual Studio Code could provide extensive library of extensions. It could also offer flexibility and customization options that might enhance the development process. Using Git should facilitate collaboration among team members and ensure that code changes could be tracked and managed effectively.

3.2.5 Security and Compliance

Data security was a top priority while designing the automated testing tool. Necessary measures were determined to be implemented to ensure secure storage of logs and test results. Any unauthorized access could be prevented. Access control mechanisms were suggested to apply to ensure that only authorized personnel could access sensitive data. This would be done through a combination of user authentication and role-based access control which could restrict access based on the user's role within the organization.

Compliance with relevant industry standards of software testing and 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 to ensure systematic development and deployment. 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. 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. An error message is displayed and the program terminates if the device list fails to update. The selected ADB command is executed if the list is successfully updated. This involves 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. The program then displays the details of the executed command in the UI and continuously updates the output text with the command results. This makes sure that the most recent output is displayed. The algorithm checks if the command is still running. If not, it completes the execution by marking the command as not executed. the program proceeds 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 | : | Check if Device List is Updated   1. If the device list is not updated:    * Display an error message.    * Go to End. 2. If the device list is updated:    * Proceed to execute the ADB command. |
| 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 Text   1. Continuously check the command output. 2. Display the output in the text widget. 3. Scroll the text widget to the end to show the latest output. |
| 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 (after canceling the command)   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 12 | : | End |

Figure 8 Flowchart of ADB Command UI

The flowchart of ADB Command UI in figure 8 clearly shows the sequence of operations. It ensures that each step is performed in the correct order to provide a visual representation of the process. This structured approach not only made the development of the tool easier but also ensured that it met the desired goals of efficiency and accuracy.

3.2.7 Detailed Component Design

The system architecture was divided into several major components, each with a specific task. This modular design would ensure that each component could be developed and tested independently to improve the overall system.

The GUI component would be developed using Tkinter. This would provide a user-friendly interface for selecting devices, executing commands and viewing results. It was planned to build the tool in such a way that it would be easy to use and quick to respond which might allow users to interact with it effortlessly. The interface was designed to have elements like device selection dropdown menus, command execution buttons, and result and log 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 precise and effective test execution. It was planned to handle the communication with linked devices and carry out the user's chosen instructions. This component was designed to be robust and reliable. As a result, it would prove to be capable of handling multiple commands simultaneously and provide instant updates about the status of each command.

The logger was programmed to be in charge of recording and saving comprehensive logs of test execution. This would help in analysis and debugging. The logger was structured to make sure to document all outputs and create a detailed record of every test conducted. This component would be essential for diagnosing issues and ensuring that tests are carried out accurately. Logs would contained detailed information about every command executed, the tested device and the results of the execution. This information would be crucial for troubleshooting and confirming the accuracy of the tests.

3.2.8 Proper Optimization and Improvement

Error detection and handling was necessary for proper optimization of the design. Robust error handling mechanisms were designed to be applied to ensure that the tool could handle unexpected issues. This would enable handling device disconnections, command failures and other potential issues. The tool was designed to recover from errors automatically wherever possible (Can it actually do that?). This would minimize the impact on the testing process. Detailed error logs were also designed to provide information about the nature of the error. The error logs would also provide information about the steps taken to recover from the error. This information might help to identify recurring errors and improve the overall performance of the tool.

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 would enable parallel testing across several devices. This was planned to be done by effectively utilizing resources and managing connections between devices. The performance of the tool was planned to be optimized to ensure that tests could be executed efficiently. This would reduce the overall testing time. Performance metrics would be monitored in order to find areas that needed improvement. This continuous optimization process would 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 which is compatible with these systems. This would ensure that the results could be easily incorporated into existing workflows. This integration could make testing more efficient by gathering all necessary information in one place and thus improve overall effectiveness.

A thorough training program was planned to be created in order to guarantee that testers could utilize the new tool successfully. Multiple training sessions and ongoing support was proposed to be arranged for the testers. Training sessions would be conducted to familiarize testers with the tool and provide hands-on experience. Ongoing support would provide the solution of the issues faced by the testers. This extensive program for training and support is planned to guarantee that testers could smoothly switch to the new automated testing procedure.

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 evaluation and improvement process would ensure that the tool remained aligned 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 highlights the significant benefits of automation in improving testing efficiency and accuracy. The conceptual design of the automated testing tool demonstrates how to plan and design an effective and user-friendly automated testing tool. Overall, the automated testing tool represents a significant improvement in the company's testing capabilities. It addresses the limitations of manual testing and provides a scalable and efficient solution for testing Android devices.

4 Implementation

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: the preparatory phase, the execution phase and the evaluation phase.

The preparatory phase consisted of establishing the environment and choosing the test cases. Ensuring the availability of essential hardware and software components was the main concern of this phase. 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 along with necessary dependencies like ADB and Tkinter were installed on the test system. Making sure that the software and hardware were compatible was an essential step during this phase. 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 are performed. The classification helped in analyzing how the tool performed in various test categories.

During the execution stage, the chosen test cases were executed through manual and automated testing methods. Gathering data during this stage was essential for obtaining information on performance metrics like execution time and defect detection. At first, experienced testers performed the selected test cases manually and documented the results. 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.

During the evaluation phase, the data collected during the execution phase was analyzed. Execution times, defect logs and resource utilization metrics were compared. This comparison provided quantitative insights into the performance improvements by using the automated testing tool. Statistical methods were also employed to determine the significance of the differences between manual and automated testing. These methods included t-tests and ANOVA to ensure that the findings were statistically correct (Need to check if we can really apply t-test and ANOVA!!).

The steps of implementation methodology are shown in Figure 9.

Figure 9: Steps of Implementation Methodology

4.2 Setup and Configuration

Setting up and configuring the automated testing tool was the first step in the implementation process. This included setting up the required software, configuring the testing environment and verifying that all dependencies were resolved properly.

The automated testing tool was developed using Python, with Tkinter for the graphical user interface and ADB for device communication. Python was set up on the test system to make sure it works well with the operating system. Version management was handled using virtual environments to avoid conflicts with other installed software. Essential Python libraries such as Tkinter for GUI development and subprocess for executing ADB commands were installed through pip, the package installer for Python. Dependencies were managed using a `requirements.txt` file to ensure that all necessary packages were installed correctly.

Setting up the testing environment required connecting Android devices to the testing system and configuring ADB to detect them. Multiple Android devices were connected via USB and configured for debugging mode. This step ensured that the devices were ready to interact with the testing tool. ADB was set up to display and communicate with connected devices. This included setting up udev rules on Linux systems to manage device permissions. Consistent device recognition was crucial for reliable test execution.

Dependency Resolution is a process that finds and resolves dependencies in a system. Resolving dependencies was essential for ensuring that the automated testing tool could interact with the devices and execute the required commands. All dependencies were installed and verified. This included device drivers and additional Python libraries. The installation process was documented to facilitate replication. Initial tests were conducted to verify that the tool could detect and communicate with the connected devices. These tests included basic ADB commands to ensure that the setup was correct.

Figures relevant for this subsection could include screenshots of the software installation process, diagrams showing the environment setup, and tables listing the dependencies and their versions.

4.3 Execution Workflow

The execution workflow of the automated testing tool was designed to be intuitive and efficient. This allowed testers to select devices, commands and tests seamlessly.

The tool's user interface was designed to offer a simple and easy-to-use 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 to reduce the number of steps required 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 to make it easier to choose. When a command was selected, the tool instantly changed the user interface to show appropriate test suites and extra test choices. This continuous updating made sure that testers constantly had the right choices available.

Executing commands involved sending ADB commands to the selected device and capturing the output for real-time display and analysis. The tool sent ADB commands to the selected device using Python’s subprocess module. This method ensured that the UI stayed responsive by enabling asynchronous execution. The output from the command execution was captured and displayed in the GUI. This included both standard output (stdout) and standard error (stderr). Real-time capturing provided immediate feedback to testers to improve the debugging process.

Providing real-time feedback was essential for monitoring the progress of test executions. The tool displayed real-time status updates, including command execution progress and completion notifications. Color-coded messages indicated the status of each command(?). All errors that occurred during command execution were captured and shown to the user. This enabled prompt troubleshooting. Detailed error messages helped in identifying and resolving issues quickly.

Figures relevant for this subsection could include screenshots of the user interface at different stages of the execution workflow.

4.4 Data Logging and Analysis

Data recording and analysis were essential parts of the project. This offered valuable information on how well the automated testing tool was performing.

The tool implemented 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 messages encountered. These logs offered a precise record of the testing activities. Logs were then stored in a structured format. This ensured simple retrieval and analysis of the logs. The storage system made sure that logs were sorted based on test case and device.

Analyzing the logged data provided valuable important information on how well the automated testing tool performed and identified 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.

4.5 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 and the strategies undertaken to mitigate them.

4.5.1 Technical Challenges and Mitigations

Technical challenges mainly involved integrating the automated testing tool with different Android devices and ensuring consistent performance in various environments.

Ensuring compatibility with a wide range of Android devices was a significant challenge because of their diverse each hardware configurations and OS versions. Extensive compatibility testing was conducted to ensure that the tool could interact with all targeted devices. Devices were categorized according to their hardware and software characteristics to identify any specific compatibility issues. Fallback mechanisms were implemented to handle devices that did not fully support certain ADB commands. These mechanisms included alternative commands and error handling routines to ensure that testing could continue without interruption.

Ensuring reliable execution of ADB commands on various devices and network conditions needed strong error handling and recovery mechanisms. Error handling mechanisms were implemented to capture and report any issues encountered during command execution. This involved implementing retry logic for temporary errors and thorough logging for ongoing problems. Strategies were created and included in the tool to address common problems like device disconnects or command timeouts. These strategies ensured that tests could resume or restart without any significant intervention.

4.5.2 Operational Challenges and Mitigations

Operational challenges included guaranteeing testers to efficiently utilize the automated testing tool and incorporate it into current workflows.

It was essential to train testers effectively in order to ensure successful adoption of the new tool. Extensive training programs were created to teach testers about the features and usage of the automated testing tool. These programs featured interactive sessions and step-by-step tutorials. Testers were given thorough documentation such as user manuals and troubleshooting guides to assist them. The documentation provided information from installing to utilizing advanced capabilities of the tool.

Integrating the automated testing tool into existing testing workflows required planning and coordination. 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. Steps were taken to guarantee that the tool could be smoothly integrated without causing any disruption to current workflows. This involved creating necessary APIs and scripts.

4.6 Continuous Improvement and Maintenance

Continuous improvement and maintenance activities were carried out after implementing the automated testing tool to guarantee long term performance.

Collecting feedback from testers was essential for identifying areas for improvement and ensuring that the tool met their needs. Mechanisms for collecting feedback like surveys and feedback forms were implemented. These offered organized methods for testers to provide their experiences and feedback. Moreover, regular reviews of feedback were conducted to identify common issues and areas for improvement.

Updates and improvements were done to the automated testing tool based on the feedback. Regular software updates were released to fix bugs and improve performance. New features and enhancements were added based on tester feedback and evolving requirements.

Continuous maintenance activities were essential for keeping the automated testing tool reliable. Regular maintenance activities like updating device drivers and libraries were routinely conducted. Constantly monitoring the tool's performance allowed for potential issues to be detected and dealt with before affecting testing activities. Performance measures were monitored and examined to guarantee efficient functioning of the tool.

4.7 Case Studies

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

4.7.1 Case Study 1: Large-Scale Regression Testing

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

More than 200 test cases were part of the regression test suite. This covered various functionalities across multiple applications. The automated testing tool was employed to perform these test cases on various devices. This significantly reduced the time required for testing. The automated tool allowed regression testing to be finished in half the 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.

Figures of screenshots of the regression testing logs.

4.7.2 Case Study 2: Compatibility Testing

Another case study involved using the automated testing tool for compatibility testing across different Android OS versions and device models. The goal was to ensure that 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 simultaneously 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.

Figures relevant for this subsection could include tables listing the tested devices and OS versions and screenshots of compatibility test results.

4.7.3 Case Study 3: Stress Testing

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

Stress testing included running 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 and improving their durability under heavy load.

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

5 Results, Discussion and Future Work

This chapter presents the results of the experimental study and implementation of the automated testing tool. This is followed by a discussion of the findings and potential future work. The results demonstrate the effectiveness and efficiency of the automated testing tool compared to traditional manual testing. The discussion provides insights into the significance of these results. Finally, potential areas for future work to enhance the testing framework further will be explored and any limitations identified during the study will be addressed. This chapter also aims to provide comprehensive answers to the research questions outlined in Chapter 1.

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 in comparison to manual testing. The findings show a notable increase in efficiency.

Automated testing consistently reduced the execution time of test cases. For example, the test called `HubTest\_testSendReceive\_duration` that usually took about 159 seconds when done manually now takes between 90 and 160 seconds with the automated tool. This signifies a significant reduction in time.

Figures/table/chart showing comparative execution times for various tests

5.1.2 Accuracy Evaluation

Another critical metric was the accuracy of defect detection in automated testing. The automated tests successfully identified issues that manual testing occasionally missed.

Automated testing detected issues in UI elements and scenarios that caused failures like the absence of UI elements (`testSendReceiveWithAttachment`). These were not always reliably identified in manual testing because of human error.

The automated tool provided detailed logs for each test execution and captured all relevant information and errors. This facilitated accurate defect monitoring and fixing.

Figures/table showing number of defects detected by automated versus manual testing

5.1.3 Scalability Assessment

The scalability of the automated testing tool was assessed by testing its capability to manage various devices and run tests concurrently.

The tool efficiently managed parallel executions on multiple devices. This improving the throughput of test cases. For instance, running the `HubTest\_testSendReceive\_duration` test concurrently on several devices demonstrated the tool's capability to scale across various environments.

The tool was tested on various Android devices with different OS versions. This extensive compatibility testing confirmed that the tool could handle the diversity present in the Android ecosystem. This ensured that the tool can support devices from various manufacturers and with diverse hardware configurations.

A table summarizing the devices tested, their OS versions, and the results can be included here.

5.1.4 Failure Analysis

It is crucial to analyze the failures to understand the tool’s limitations and areas for improvement in addition to successful test executions.

Several tests did not pass because the tool could not identify certain UI elements, like in the `testSendReceiveWithAttachment` where the “Open with option: SmartOffice” was not detected. A number of tests, including those related to `TasksTest` and `NotesTest`, were not executed. This suggested potential problems with test configuration or how tests are being called. This consistent lack of execution across multiple test files indicates that there is a need to improve the test configurations and the conditions that activate them.

These observations are crucial for enhancing the reliability and efficiency of the automated testing tool.

Figure of screenshots of these failures.

5.2 Discussion

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

5.2.1 Benefits of Automation

There were some significant benefits found for using the automated testing tool instead of traditional manual testing. Automated testing reduces the time required for test execution. This allows for more frequent and thorough testing cycles. Automated tests also ensure reliable outcomes by removing the inconsistency and mistakes caused by manual testing. This consistency is crucial for regression testing to prevent new changes from causing bugs in current features. The tool’s thorough logging and error reporting capabilities improve the ability to diagnose and fix issues quickly. Automated reporting ensures that all relevant information is captured systematically which helps to detect defects faster.

5.3.2 Challenges Encountered

Some challenges were identified despite the positive results. Certain tests failed due to 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 significant challenges that need to be solved. Several tests, such as `TasksTest` and `NotesTest`, did not execute any cases. This suggests possible problems with test configuration or execution logic. This requires further investigation of the test configurations to ensure that all intended tests are executed correctly.

5.2.3 Comparative Analysis of User Experience

Comparing the experience of the testers for using the automated and manual testing approaches provides important insights.

Even though manual testing offers for more flexible and intuitive exploration of the application, it is time-consuming and prone to human error. It is often challenging to reproduce the same testing conditions consistently and this leads to variability in results.

The automated tool is beneficial in executing repetitive and well-defined test cases quickly and accurately. This makes it ideal for regression testing and routine validation tasks. However, it requires ongoing maintenance to ensure that the tests remain effective as the application evolves.

A figure comparing the advantages and disadvantages of manual versus automated testing

5.4 Future Work

Several areas for future work have been identified based on the results of this study to enhance the automated testing tool further.

5.4.1 Enhanced UI Object Recognition

Improving the tool's capacity to identify and interact with different UI components is essential. Machine learning algorithms can be incorporated to improve the tool’s ability to detect and interact with dynamic UI elements. Machine learning models can be trained on a diverse set of UI designs to improve their adaptability.

References:

[Sarkar et al. 2019] 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] 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 et al. 2019] 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 et al. 2016] 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.

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.

‌

‌

‌

‌

‌

‌