POLITECNICO DI TORINO

Masters of Science in Computer Engineering

Tesi di Masters of Science

Multi-locator For GUI-Testing

Supervisor

Prof. Riccardo COPPOLA

Prof. Tommaso FULCINI

Candidate

Syed Faraz SHAH

April 2024

Sommario

Introduction: This thesis explores using multi-locator strategies to analyze widget attribute changes and their visual impact on mobile application graphical user interfaces (GUIs). Through a comprehensive examination of attribute changes such as bounds, resource ID, text, and content description, this research aims to demonstrate multi-locator techniques' effectiveness in simultaneously locating multiple widgets. The study encompasses a detailed analysis of attribute variations across consecutive application versions, highlighting their influence on widget changes and the resultant visual alterations in the GUI. This investigation provides valuable insights to inform software development practices and enhance the user experience.

Objective: This thesis encompasses mobile application development, specifically focusing on widgets and their attribute changes using graphical user interface (GUI) testing for Android applications. Within this scope, the thesis investigates the challenges and complexities developers face in ensuring the functionality, usability, and visual consistency of GUI elements across different versions of mobile applications. The research also explores strategies for analyzing layout-based properties and their impact on GUI testing processes. Overall, the project contributes to the field of software quality assurance, particularly in the context of mobile application development and GUI-based testing.

Methodology: For GUI testing, evaluating real applications from the market provides practical insights into GUI component composition and user experience. It enables the examination of interface design principles like layout, color scheme, typography, and iconography, identifying effective patterns and avoiding pitfalls. We selected multiple real mobile applications from various categories, focusing on those with multiple versions available. Applications were sourced from APKMirror, with the most recent versions downloaded for evaluation based on inclusion criteria. Visual mutation testing was initiated to identify components like buttons, text fields, and dropdowns, generating mutations by modifying colors, sizes, and positions and adding/removing elements.

This methodology involves installing two chosen versions of each mobile application on the Google Nexus 5X AVD and preparing them for evaluation by completing setup processes and examining their XML layouts using the UI Automator dump command. A script is then used to filter out irrelevant Android Views and parse potential characteristics and their values into a CSV file. Distinctive identifiers are added to facilitate widget identification, and screenshots of the home page are taken for reference. Related widgets are identified and manually paired to compare attribute changes between versions, with differences obtained from CSV files. This process allows for evaluating attribute suitability as a locator, providing insights into attribute evolution and its impact on application development. Using a generated CSV file, we initiated the analysis of each application to compile an Oracle table containing all attributes within the user interface. Our primary focus was on identifying changes by examining nodes such as content description, resource-id, and attribute progression. Additionally, for specific applications, we also sought to extract text associated with each attribute to enrich our analysis.

Results:

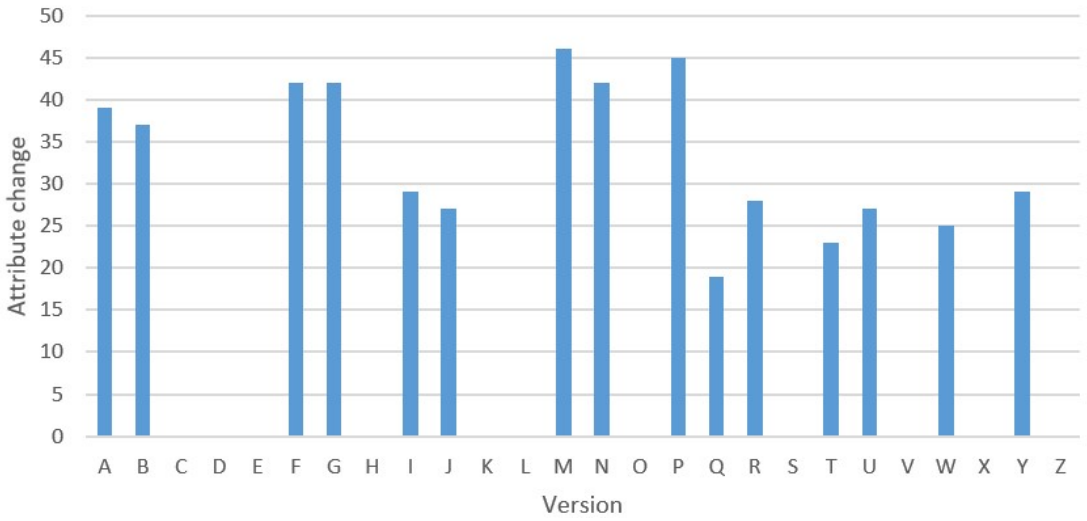

Figura 1: calm with all versions and attribute variability

Results obtained in the figures 4.6, 4.7 and 4.8 we can observe that the bounds attribute in calm application ranked as the most unstable attribute due to its high variability approximately 80 percent of instability in different releases of the application. Despite its frequent presence and variability in assumed values, it's deemed unsuitable as a locator due to instability. Changes in the "bounds" attribute often coincide with alterations in the "index" attribute, indicating shifts in both visual arrangement and tree structure. The attribute resource-id Ranked

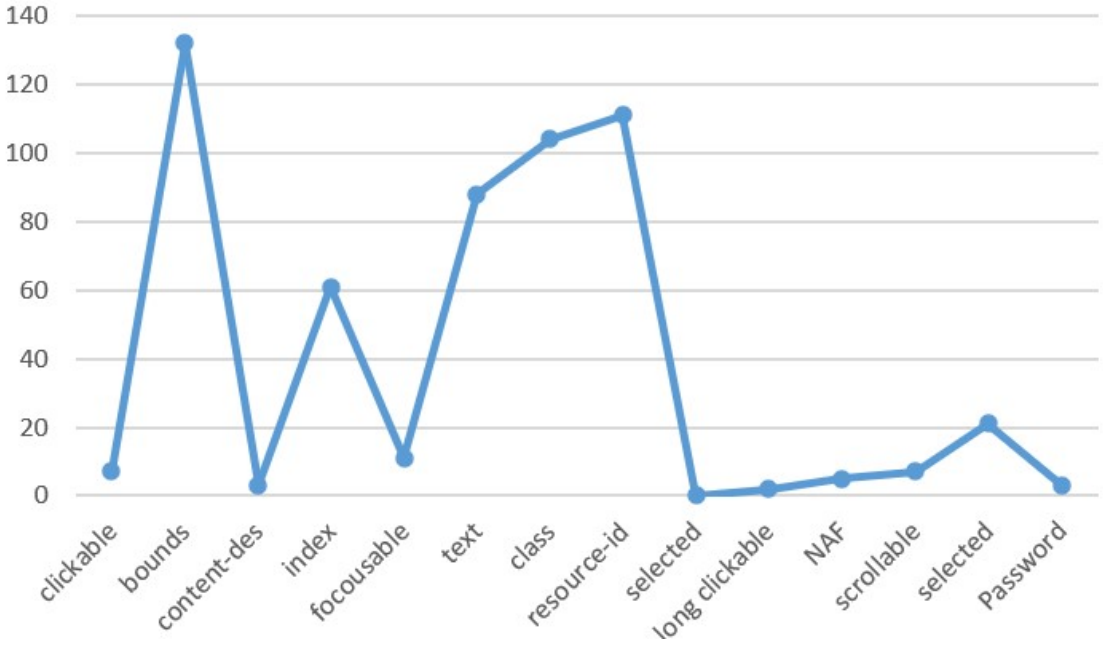

Figura 2: calm with all attribute changes in all releases

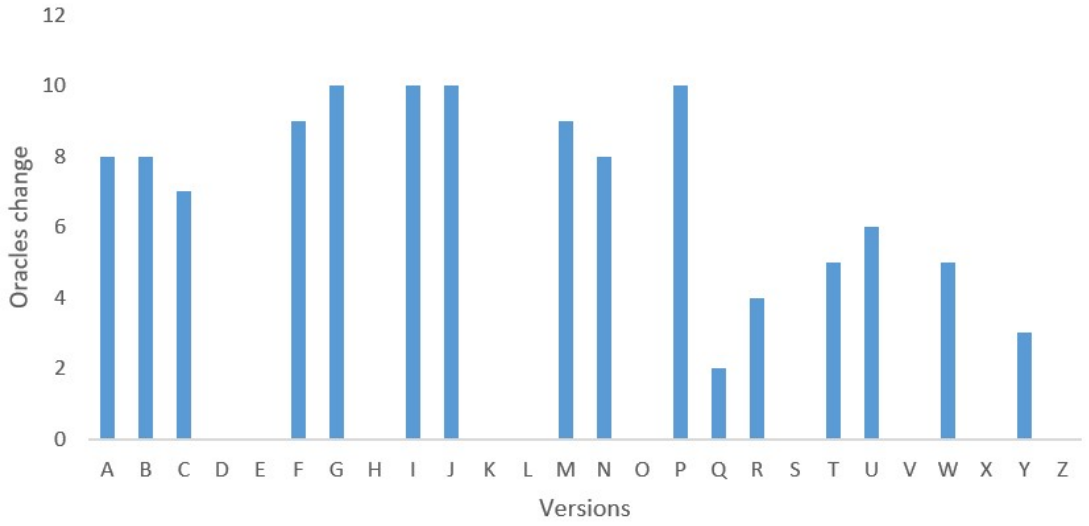

Figura 3: calm with all versions and oracle variability

second for instability with a variability of 70 percent. Despite Android Developers' recommendation for uniquely identifying resources using this attribute, half of the widget changes affect this identifier. Maintaining the "resource-id" updated across the project requires a significant effort, especially during GUI testing outsourcing.

Class and Text Attributes (Instability: 65 percent and 60 percent, respectively). These are considered quite unstable attributes with moderate variability. Frequent absence of values adds to their unreliability, contributing to their instability.

In our examination of visual mutations, mutation M11 emerged with the highest frequency of visual changes in the GUI, primarily attributed to alterations in the bound attribute. This attribute notably impacts the widget's positioning within the interface. Following closely is mutation M15, which involves modifications to the widget's graphical appearance. Notably, changes in the application's appearance often coincide with slight rearrangements of widgets within the interface.

Conclusion: Here are the essential findings and contributions of the research.

- We found that the five top attributes, Bounds, Resource-id, class, text, and content-description, are the most variable.
- The study revealed notable variabilities in attributes across consecutive versions, especially in attributes like Bound, resource-id, and content description, which directly influenced the widget changes such as layout, position, and identification of GUI elements.
- Attribute variation had a significant visual impact on the GUI, affecting the overall appearance and usability of the application. Variability in attributes such as text and content description influenced the clarity and accessibility of GUI elements.
- Using a multi-locator proved instrumental in facilitating efficient localization and attribute analysis.

Indice

Elenco delle tabelle VIII

Elenco delle figure IX

Acronimi XI

1 Introduction 1
1.1 Motivation . . . 2
1.2 Objective . . . 3

2 Background and state of the art 5
2.1 The Android Application Architecture . . . 5
2.1.1 Categories of Mobile Applications . . . 7
2.2 Introduction to Mobile Testing . . . 8
2.2.1 Types of Mobile App Testing . . . 8
2.2.2 Automated GUI Testing . . . 9
2.2.3 Approaches for Automated GUI Testing . . . 11
2.2.4 Challenges in Mobile Application Testing . . . 13
2.2.5 Framework's and Tools for Automated Testing . . . 16

3 Methodology 22
3.1 Research Methodology . . . 22
3.1.1 DataSet Selection . . . 24
3.1.2 Visual Mutation . . . 28
3.1.3 Analysis Process . . . 31

4 Results 39
4.1 . . . 39

5 Conclusion 58

Bibliografia 62

Elenco delle tabelle

3.1 List of selected applications with their release version and date . . . 27
3.2 Mutation operators . . . 30
3.3 All Oracles . . . 33
3.4 Oracles . . . 36

4.1 Attribute Variability . . . 55
4.2 Widget Variability . . . 56

Elenco delle figure

1 calm with all versions and attribute variability iii
2 calm with all attribute changes in all releases iv
3 calm with all versions and oracle variability iv

2.1 Espresso Framework 17
2.2 UIautomator Framework 19

3.1 Visual Mutation Flow chart 29
3.2 Difference between old release and latest release 32
3.3 Visual Difference between the consecutive releases 37

4.1 distribution of empty and valued attributes in the selected app . . 40
4.2 distribution of boolean attributes in the selected app 41
4.3 cdisplay with all versions and attribute variability 43
4.4 cdisplay with all Attributes change 44
4.5 cdisplay with all versions and oracles variability 45
4.6 calm with all versions and attribute variability 46
4.7 calm with all attribute changes in all releases 46
4.8 calm with all versions and oracle variability 47
4.9 youVersion with all versions and attribute variability 48
4.10 youVersion with all attribute changes in all releases 48
4.11 youVersion with all versions and oracle variability 49
4.12 Mirror with all versions and attribute variability 49
4.13 Mirror with all attribute changes in all releases 50
4.14 Mirror with all versions and oracle variability 50
4.15 Sketchbook with all versions and attribute variability 51
4.16 Sketchbook with all attribute changes in all releases 52
4.17 Sketchbook with all versions and oracle variability 53
4.18 Cdisplay all versions mutation 54
4.19 youVersion all versions mutation 54

Acronimi

GUI
: Graphical User Interface

SUT
: System Under Test

MVC
: Model View Controller

IPO
: Input Process Output

JSON
: JavaScript Object Notation

VCS
: Version Control System

Capitolo 1

Introduction

A product has a life cycle starts with its conception and concludes with its distribution and release to the market. Each product must complete several tests before being allowed on the market to comply with the law and ensure that it does so without producing any negative impacts. The software must also be tested because its life cycle resembles a physical object. Because it requires a lot of time and resources—sometimes more than the design itself—and is, therefore, very expensive, software vendors frequently undervalue this stage.

Every stage of software development should include testing. Each line of code should pass one or more tests that confirm its accuracy, and each time a change is made, the test is updated. However, the testing phase includes more than just functionality verification; performance, usability, compliance with requirements, security, and more are also examined. Mobile apps are described as mobile software (i.e., programs that run on mobile devices) that incorporates context-sensitive features, such as contextual sensing and adaption and context-triggered activities, into its design. They can be divided into two categories: native apps, which are created specifically to run on a given mobile platform by its design principles, and web-based apps, which are built from web pages and only partially or not at all make use of the unique capabilities of the mobile device.

Following a Statista report [1], The world's population of mobile phone users has grown during the last ten years. From 3.6 billion users in 2016 to 6.25 billion in 2021 and then to 7.6 billion users by 2027, smartphone usage is anticipated to increase gradually. Numerous applications are available on mobile devices that can now perform functions previously only available on high-end desktop computers.

One feature that has contributed to Android's success is the availability of marketplaces (like the Play Store) where developers may sell—or release for free—their applications. Due to the overwhelming amount of software on those platforms and the resulting competition, applications must fulfill their promises to users. Mobile applications, often known as apps, are becoming increasingly important in our

personal and professional lives due to the increasing number of mobile devices. In this context, GUI testing is crucial since it enables the simulation of direct user interaction. Automated GUI testing, in particular, makes it feasible to quickly and consistently show that the Application Under Test (AUT) is functionally valid. The ability to quickly run the same tests on demand on the target AUT is another benefit of having a set of automated tests. If the tests are successful, this ensures that the application problems won't recur. The structure and workflow of the application GUI are susceptible to changes, and GUI test scripts often refer to precise sequences of operations to be performed on particular GUI widgets.

An automated test suite is created to ensure that all of an app's components function properly; if a test case fails, it should indicate improper application behavior. Even though this is generally true when testing the application at a lower level, such as with unit testing and integration testing, when testing apps through their GUI, tests may also fail because of locator changes. Locators help testers pinpoint specific elements on screens and access widget functionality.

It is common for an app's GUI, internal description, or properties to change over time as it develops, either with minor attribute adjustments to some widgets or a whole new visual design on some screens. In both situations, tests may fail if the widget locators change between releases, which requires fixing tests with a new, accurate locator. These test failures are due to a failure with a widget's location procedure rather than an issue with the app itself.

1.1 Motivation

This research addresses the challenges posed by attribute changes in mobile application development. Attributes such as bounds, resource ID, text, and content description play a crucial role in defining the appearance and behavior of UI elements within mobile applications. However, managing attribute variations can be complex and time-consuming, often leading to unexpected widget changes and visual inconsistencies in the graphical user interface (GUI).

Understanding the impact of attribute changes on widget modifications and GUI visual appearance is essential for ensuring the usability and effectiveness of mobile applications. By investigating how attribute variations influence widget changes and visual alterations in the GUI, we can gain valuable insights into optimizing software development processes and enhancing user experience.

Furthermore, utilizing multi-locator strategies presents an opportunity to streamline attribute analysis and widget localization. Multilocator techniques enable the simultaneous identification of multiple widgets based on various attributes, offering a more efficient and effective approach to GUI testing in mobile applications.

1.2 Objective

This thesis explores using multi-locator strategies to analyze widget attribute changes and their visual impact on mobile application graphical user interfaces (GUIs). Through a comprehensive examination of attribute changes such as bounds, resource ID, text, and content description, this research aims to demonstrate multi-locator techniques' effectiveness in simultaneously locating multiple widgets. The study encompasses a detailed analysis of attribute variations across consecutive application versions, highlighting their influence on widget changes and the resultant visual alterations in the GUI. This investigation provides valuable insights to inform software development practices and enhance the user experience. This thesis investigates how to increase location robustness by combining different locator variables. We present the findings of an experimental investigation that looked at the distributions of locator values in well-known Android applications to determine which qualities are more stable and less subject to change when used as locators.

Capitolo 2

Background and state of the art

2.1 The Android Application Architecture

Android is a mobile operating system based on the Linux operating system kernel. An android comprises an operating system and a software platform where the operating system allows the application to run, which is somewhat different from the traditional applications by default. An Android application distribution comes with a set of minimal applications like (browser, email client) setting applications, which any third-party application can replace.

The Linux kernel acts as a hardware abstraction layer between the device's physical layer and the Android software stack. The kernel manages the permission of who can do what, which files are readable and which files are executable which operations to invoke and it also manages the security.

The kernel manages the memory allocation to spread the physical memory among the Android monitors. It also manages the processes and the threads of the network layer whenever you need to open a network connection. Basically, it is the operating system that is in charge of returning you to a socket, and it also manages all the hardware and peripherals such as display, keyboard, camera and flash memory that contains the file system.

Each Android system has its own process, and inside the process is an instance of the Android runtime library, the ART(Dalvik) virtual machine. The Android application framework provides some basic components that respond to different user experience needs. Android apps declare in the AndroidManifest.xml file their main components, which can be of four different types:

Activities

The elements in charge of an app's user interface are called activities. Each activity is a window with different UI components, like buttons and text fields. By providing the proper callbacks for each life-cycle phase (i.e., generated, paused, resumed, and destroyed), developers may regulate the behavior of each activity. Because they respond to user input events like clicks, activities are the main focus of Android testing tools.

Services

Services are parts of a program that can perform time-consuming tasks in the background. Because they do not have a user interface, testing tools for Android generally do not directly target them; however, they may be indirectly tested through some activities.

Broadcast receivers

Interprocess communication is made possible by broadcast receivers and intents. Apps can register broadcast receivers to get notifications about particular system events via intents. As a result, apps can respond. For example, if a new SMS is received, a new connection becomes accessible, or a new call is placed. The manifest file or the app's code can both specify broadcast receivers at runtime. Testing tools must be aware of the relevant broadcast receivers to fully examine an app's behavior and set the appropriate intents.

Content providers

Content providers provide a structured interface to shared data storage, such as calendars and contact databases. In addition to making them available to other apps, apps may have their content suppliers. Like all software, an app's behavior may be influenced by the condition of these content providers (such as whether or not a list of contacts is empty or contains duplicates). Because of this, testing tools should "mock" content providers to make tests more deterministic and comprehensive of an app's behavior. [2]

Fragments

Fragments In Android development, a fragment is a modular and reusable component representing a portion of a user interface (UI) and its associated behavior. Fragments are used to create flexible and dynamic UI designs that can be combined and reused across different activities or within a single activity.

Android introduced Fragments to address the challenges of creating responsive UIs that adapt to different screen sizes and orientations. Using fragments, developers can build UI components that can be added, removed, or replaced dynamically within an activity based on user interactions or device configurations.

2.1.1 Categories of mobile applications

App development is driven by one of three types of mobile apps: native, web-based, and hybrid. And even though they all have unique structures and approaches to writing code, they also have several similarities. We'll get into each category's specifics in this section.

Native apps

The creation of native apps is specific to one platform or operating system. Additionally, they employ a programming language unique to that platform or operating system. Usually, users can use iOS, Android, or Windows Phone. In practice, that means a native app will depend on a platform's ecosystem in addition to using the platform's native API to support features like app distribution. For instance, a native Android app would use the codebase for the Google Play store. However, native apps can provide better user experiences by adapting to the features of particular operating systems.

Native apps are directly executed by hardware. Because of this, they can support robust performance, high levels of security, and cutting-edge features that are "native" to the particular operating system. This implies improved performance capabilities and possibly fewer bugs to fix. For various kinds of mobile apps, the following are some of the codebases that are most frequently used across the three operating systems:

- **iOS**: Programming languages include Swift, Python, and Objective-C
- **Android**: Programming languages include Kotlin and Java
- **Windows**: Programming languages include C and .NET

Web-based apps

Mobile apps that use web-based technology are known as web apps. Avoid downloading or installing them onto a device because they may be accessed using a mobile web browser. Additionally, they are internet-capable, giving them more flexibility and a responsive design that works with any mobile device or operating system.

Web apps have a single codebase and use languages like HTML5, CSS, Javascript, and Ruby. Additionally, they employ the developer's preferred web application

frameworks or server-side languages, such as PHP, Rails, or Python. A ready-made SDK is unavailable for web mobile applications, which only employ a small subset of native functionalities.

Hybrid apps

A hybrid app combines a native app and a web app, making it one of the many varieties of mobile apps. In reality, it's made as a web app that runs inside a native app container. Hybrid apps provide the advantages of a native experience while adapting to non-native situations by utilizing specific native platform capabilities and device hardware. However, hybrid apps use front-end development tools like React, JavaScript, HTML5, Ionic, Cordova, and CSS to power this cross-platform functionality. Additionally, users can download hybrid apps from their app store because they offer better access to native device APIs and hardware than non-hybrid alternatives like Ionic, Cordova, Flutter, Electron, Swift, and Sencha Touch.

2.2 Introduction to Mobile Testing

Mobile testing refers to testing mobile applications and ensuring their functionality, usability, and compatibility on various mobile devices. With the rapid growth of the mobile industry and the increasing number of mobile applications, it has become crucial to test mobile apps to deliver a seamless user experience.

Mobile testing involves comprehensively examining mobile applications across different platforms, such as iOS, Android, and Windows, considering the diverse screen sizes, resolutions, hardware capabilities, and operating systems. The primary goal of mobile testing is to identify any issues or bugs that may affect the performance, stability, and security of the application

2.2.1 Types of Mobile App Testing

To address all the above technical aspects, the following types of testing are performed on mobile applications.

- *Usability testing* – To ensure that the mobile app is easy to use and provides a satisfactory user experience to the customers.
- *Compatibility testing* – To ensure that the Testing of applications in different mobile devices, browsers, screen sizes, and OS versions is according to the requirements.
- *Performance testing* – To ensure that the application does not exploit any available resources since they can be limited.

- *Security testing* – To ensure that an application is tested to validate if the information system protects data.

- *GUI Testing* – To ensure correct behavior and state of the GUI. This includes verification of data handling, control flows, states, and display of windows and dialogs.

Testing mobile GUI applications raises unique challenges: the character is event-driven, which makes GUI applications non-deterministic; the user can click anywhere on the screen. The growth platform is usually distinctive from the target platform for mobile applications. Testing the nearly unlimited variety of feasible states a GUI might have is impossible. This makes mobile GUI testing more difficult compared to pure functional testing.

GUI testing can be done in two ways:

Manual GUI testing - Manual testing of Android applications involves testing an Android app's functionality, usability, and performance using human testers. It typically requires testers to interact with the app on real Android devices or emulators to simulate real-world usage scenarios. Manual testing can be time-consuming and prone to human error. Consider complementing it with automated testing techniques to increase test coverage, improve efficiency, and ensure consistent results.

Automated GUI testing - Automated testing in mobile applications involves using specialized tools, frameworks, and scripts to automate test cases and verify mobile apps' functionality, performance, and usability. Automated testing offers several benefits, including increased test coverage, faster execution, improved reliability, and the ability to run tests on multiple devices and configurations.

2.2.2 Automated GUI Testing

The advantages of automated software testing over manual testing include increased test frequency. However, there are drawbacks, such as the fact that most methodologies only function at low levels of system abstraction. Several automated test methods work against or through the GUI to test the SUT at a higher level. These GUI-based testing methods have been divided into three chronological generations to illustrate their distinctions better. How they interact with the SUT, whether through precise coordinates, GUI hooks, or image recognition, varies depending on the generation. In this section, we present the key properties of the three generations.[3]

First Generation: Coordinate-Based

The first generation of automated GUI testing refers to testing graphical user interfaces (GUIs) by specifying and interacting with GUI elements based on their screen coordinates. In this approach, testing tools or scripts are programmed to simulate user interactions by providing X and Y coordinates to locate and interact with GUI elements such as buttons, input fields, or dropdown menus.

In coordinate-based testing, the tool uses predefined coordinates to click on specific areas of the screen or move the cursor to perform actions. For example, if there is a login button on a GUI, the testing script would be programmed to click on the coordinates corresponding to the position of that button.

This approach has its limitations. Since it relies on fixed coordinates, any changes to the GUI layout, screen resolution, or device size can cause the test script to fail. Even small changes in the GUI can lead to significant issues in maintaining and updating the tests. Moreover, coordinate-based testing is highly dependent on the GUI structure and may not be suitable for complex or dynamically changing interfaces.

Due to these limitations, coordinate-based testing has been largely replaced by more advanced and flexible approaches such as user identifiers (e.g., IDs, classes, or XPath) or visual recognition techniques. These newer methods allow for more robust and maintainable automated GUI testing, adapting to changes in the UI and reducing the fragility associated with coordinate-based testing.[3]

Second Generation: Component/Widget-Based

The second generation of automated GUI testing refers to testing graphical user interfaces (GUIs) by using hooks into the SUT's GUI. The testing tool can access and alter GUI events and data using hooks. Additionally, most tools from the second generation offer record and replay, which reduces the cost of test development. Additionally, most tools assist the user by maintaining the property data for GUI components, such as ID numbers, labels, and component types. These properties must have this functionality because they are confusing.

Without technical or subject knowledge, a human tester cannot intuitively recognize a component from its ID number or component type. However, groupings of attributes, when combined, allow the tester to differentiate between components. Indirect interaction with GUI controls, information retrieval, and answer validation are all capabilities of test scripts or tools. Compared to coordinate-based testing, second-generation testing enables higher-level testing by interacting with the SUT more abstractly. It provides more flexibility and maintainability than the prior generation.[3]

Third Generation:

The third generation of automated GUI testing refers to a method of testing graphical user interfaces (GUIs) by Visual GUI Testing, which involves utilizing image recognition techniques to interact with the System Under Test (SUT) based on visual elements. Visual GUI testing uses image recognition techniques to recognize and interact with GUI elements. The testing tool takes screenshots or images of the SUT's GUI and analyzes them rather than utilizing coordinates or hooks. The testing tool uses image recognition to recognize GUI elements including buttons, icons, text fields, and labels based on their outward look. This makes it possible to interact with the GUI without relying on certain coordinates or hooks.

Compared to earlier versions, Visual GUI Testing is more robust and flexible. It is more resistant to minor visual alterations since it can handle changes in the GUI layout, size, or position of elements. It is ideal for testing complicated or dynamically changing interfaces to use visual GUI testing. It can deal with situations in which GUI elements are dynamically generated or changed, which makes it simpler to automate the testing of such interfaces. Testing based on image recognition can be computationally expensive since it needs complex algorithms to evaluate and compare images. This might affect how quickly and effectively tests are run.[3]

2.2.3 Approaches for Automated GUI Testing

Random/Fuzzy Testing

Software applications are tested using the random testing technique, which generates random, independent inputs. The output results are compared to the program requirements to determine whether the test was successful or unsuccessful. Program exceptions are used to identify test case failures without specifications. Almost all other test suite-generating approaches incorporate random testing as a core component.

Using erroneous, unexpected, or random data as inputs to a testing object is called "fuzzing testing." It is frequently used to check for security flaws in computer systems or applications. The primary attention then switches to monitoring the application for errors like crashes, failing built-in code assertions, or discovering potential memory leaks. Fuzzing testing differs from random testing in that it primarily accepts unexpected, erroneous inputs—often on purpose—and focuses on tracking crashes and other exceptions of the tested apps. In contrast, random testing is not required to adhere to any such software criteria. [4]

Model-based Testing

Model-based testing advances traditional testing approaches by automatically producing test cases based on a model representing the system's functioning under test. The final test technique is frequently extensive, even if such a methodology necessitates a significant, typically manual, effort to develop and build the model since test cases can be automatically generated and run. According to [4], the most prevalent methodology used in Android testing literature is model-based testing: In 63 percent of papers, model-based testing procedures are used. A thorough documentation of Takala et al.'s [5] experiences using model-based GUI testing on Android apps were presented. They frequently review how apps are modeled, model-based testing and test automation are deployed, and how tests are created and carried out.[4]

Capture and Replay Testing

Each user interface element in Android is a subclass of the View class. A view is in charge of drawing and takes up a rectangle area on the screen. The Android platform offers a collection of pre-built views that can be utilized to create a Button, CheckBox, or TextView in a user interface. Furthermore, a view is a component of a user interface screen. In a hierarchy, child views are placed on the tree's branches, and a root view is placed at the top. A root view is typically a layout view (i.e., container view). The layout, a subclass of ViewGroup (a subclass of View), is utilized to manage the placement of child views on the screen. A perspective is also in charge of managing unexpected events inside of the view. User events in Android are received by the top view in the hierarchy and passed down until they reach the right place (the view currently in focus). As a result, in our method, the source code of the Android application under test (AUT) is instrumented, and a mock layout called InterceptLayout.

. The InterceptLayout will analyze the user events it has recorded and related test scripts will be generated automatically. The suggested approach will capture two different kinds of events: key and touch events. A key event is set off when users push a physical key on a device, like the Home or Menu button. The produced script will essentially send a key event with a key code corresponding with the physical key being pushed to the device for such a key event.

A touch event is set off when users contact a touch-enabled device's screen. Any movement gesture on the screen counts as a press, a release, or other action. The suggested solution will first get the screen coordinates of the event and the intended target UI component (i.e., widget) to construct the appropriate test scripts for touch events. The appropriate test script is then built to send the events and actions to the target UI component, depending on the target UI component and event action code information.

The proposed method enables users to enter assertions that may be used to evaluate whether the runtime outputs of UI components are accurate to verify the execution outcome of an Android application. When capturing user interactions immediately following the occurrence of the target UI component for the assertion, the assertion may be added.

The AUT and the test script are assembled and packaged into an.apk file to replay the recorded test script. This file is uploaded to the target device through Android Debugging Bridge (ADB), a toolset of the Android SDK that enables connection between the desktop computer and the target device. The execution results of this apk file, including the conclusions of the assertions, are logged and delivered back to the desktop for additional analysis and presentation.[6]

Scripted and White-Box Testing

In a white-box testing scenario, the program is evaluated while being aware of the specifics of its implementation. Developers typically use unit testing in the early stages of software development. Once all the software components have been put together, comprehensive testing—also referred to as regression testing—is a typical usage case. When a strategy in this SLR calls for knowledge of the app source (or byte) code, whether obtained directly or by reverse engineering, we refer to it as a white-box approach [4]

2.2.4 Challenges in Mobile Application Testing

According to a report published by Statista [1] Over the past ten years, the number of mobile phone users around the globe has increased. Smartphone usage is expected to rise steadily, from 3.6 billion users in 2016 to 6.25 billion users in 2021 and then to 7.6 billion users by 2027.

Mobile technologies, such as smartphones, have integrated seamlessly into our lives and will continue to do so. Only mobile programs, or "Apps," can perform all of the capabilities of a smartphone. Due to the rise in the number of mobile OS and browser combinations. To be at the top of their game, developers and testers must overcome several significant obstacles. Adopting methods to overcome difficulties in mobile testing is essential, particularly in light of the end-user's current hyper-awareness of what they desire.

Mobile apps, as opposed to desktop or web applications, are much more event-driven, accepting user input and adjusting to environmental changes due to a wide range of sensors and hardware components that provide interfaces for touch-based gestures, temperature measurements, GPS locations, orientation, and other functions. Developers find simulating these various input possibilities challenging in safe testing settings. Additionally, due to time and financial restrictions, testing apps

against a broad range of configurations indicative of "in-the-wild" circumstances is less likely to be done by startups, small development teams, or even major corporations. A list of problems and a possible solution is presented by Linares-Vasquez et al [7]

Fragmentation

The numerous mobile application icons are what immediately come to mind when we talk about mobile applications. The mobile device's operating system is its heart. Each operating system has a different set of rules and ways of operation that assist the programs running on the device. Most manufacturers use the size of the mobile device's screen as a smanytice customers to buy larger phones. So, numerous configurations have been created using various devices, versions, and operating systems. It is quite challenging to test the application in all settings because there are so many of them, not to mention how expensive that would be.

Moving the testing step to cloud/crowd-based services, which give developers the capacity to test apps on a great number of virtual and physical devices or with a multitude of users using various devices, maybe a potential solution to this problem. Anyhow, this kind of service is typically more expensive and time-consuming than what Agile and DevOps practices would allow.[7]

Test Flakiness

Many modern mobile apps frequently utilize back-end services. The greater an app's reliance on back-end servers and services, the more susceptible it is to race-condition scenarios, loss of service brought on by a lack of connectivity, response time outs, and data-integrity problems during large-scale data transfers to and from a device. These situations have the potential to generate nondeterminism in-app behavior and results, which has an immediate influence on testing: test cases may fail (or pass) as a result of non-deterministic outcomes that change assertions. This phenomenon is known as test flakiness

Flaky tests frequently appear while testing complex apps because of assumptions codified in test scripts and test sequences (such as the inter-arrival time between events) and a lack of techniques for spotting unexpected circumstances when contacting back-end servers/services in both the app and the tests. Other "flakiness" sources include different run-time device states that rely on the resources available when running the tests.

Espresso only performs GUI events in test cases when the GUI is not in use, lowering the risk of tests failing due to longer-than-expected inter-arrival delays. However, both from the GUI front-end and services back-end perspectives, test flakiness continues to be a significant challenge for automated approaches.[7]

Lack of History Awareness in Test Cases

In test cases for mobile apps, history awareness has mostly been analyzed in two ways:

- **Event-Flow-Graphs**- Using different GUI states (like windows or screens) and transitions (between the states) defined as input events (like clicking the OK button), EFGs simulate GUI behavior as a finite state machine; EFGs are extracted from apps automatically while they are running, manually, using static analysis tools like GATOR.
- **Language Model**- In testing, the words are GUI events, and the distributions are created by examining execution traces gathered during the execution of the software. Language models are probabilistic distributions computed over sequences of tokens or words.

Although EFGs lack a specific memory mechanism, we can still execute in the past by navigating the graph. Instead, language-based models explicitly implement memory by creating the subsequent event in the sequence based on the previous ones using a conditional probability distribution.[7]

Difficulties in evolving and maintaining GUI scripts

The process of creating test scripts takes time since a practitioner must record or create the test for each target device. However, these scripts are frequently either related to the locations of the displays or affected by reactive GUI layout changes on devices of various sizes. When modifications affect the GUI (or GUI behavior) as expected in the scripts, test scripts must be updated as the application progresses. Although scripts are coupled to component ids that are subject to change, automation APIs like Espresso allow for the partial decoupling of GUI events from device characteristics. A current method for automatically evolving scripts created or recorded utilizing Automation APIs does not exist as of yet.

Models, like those used in some AIG techniques, could be one potential solution to this issue. Theoretically, a model could co-evolve with an app's modifications, and test cases might be produced using the model's ongoing updates. Although this could currently be achieved by automated GUI-ripping approaches (and tools based on systematic exploration) that extract the model at run time, this approach wastes the potentially useful knowledge embedded in previously generated models because the model is generated "just-in-time".[7]

Absence of mobile-specific testing oracles

Several options have been investigated for the deployment of mobile application oracles. Some of those depend on the state of the graphical user interface (GUI)

or the raising of exceptions and errors to determine whether or not a test has failed. The lack of specialized oracles is still a problem; methods that rely on app-raised exceptions can help identify crashes and unexpected problems but cannot distinguish between errors occurring on the GUI and vice versa.

Oracles are still manually implemented and updated as a result, making them a very expensive voice in testing processes.[7]

Missing support for multi-goal automated testing

Most testing operations and currently applied methods concentrate on destructive testing intended to elicit failures from mobile applications. However, this is only a small portion of the various testing methods that mobile app developers must use to guarantee that their creations are of the highest caliber and function as intended. Regression, functional, security, localization, energy, performance, and play-testing are other crucial forms of testing.[7]

2.2.5 Framework's and Tools for Automated Testing

For developers, the Android Testing Support Library [8] provides a flexible testing framework that supports the testing of single and multiple activities running on the same device. The framework is built on the AndroidJUnitRunner Java class, which enables the execution of JUnit3 and JUnit4 tests on Android applications. UI Automator and Espresso use it to run their tests.

There are further testing tools available in the literature that can build interface test cases automatically using either random numbers or a model.

Espresso

Espresso is an open-source automation framework that uses a gray-box methodology to test the GUI of a single application. To actuate the most pertinent input simulations of the framework, the internal arrangement of the parts inside the view tree of the application must be known. The name of the class that is instantiated by the application's first action must be provided by the programmer.

Espresso's onView() method makes finding particular UI elements in an application possible. The method accepts a Matcher as a parameter, which enables selecting the appropriate view based on certain criteria. For example, views can be chosen based on their class names or IDs, current state, or textual content. Similar functionalities are offered by the onData() method, which is made to operate with AdapterViews. By using ViewInteraction.perform() and DataInteraction.perform(), Espresso enables the execution of operations on selected views (such as clicking, typing, pressing buttons, and swiping).

To match and discover UI elements/views in the view hierarchy of an Android activity screen, Espresso offers a wide variety of view matcher classes (in the androidx.test.espresso.matcher.ViewMatchers package). The onView method in Espresso accepts a single Matcher (View matchers) parameter, locates the relevant UI view, and returns a ViewInteraction object. The ViewInteraction object produced by the onView method can also be used to assert the matched view or to trigger actions like clicking on it.

When a view is selected or matched, Espresso offers various view action classes (under android.test.espresso.action.ViewActions). Any action can be called by calling the "perform" method of the ViewInteraction object and passing it the appropriate view actions once onView matches and returns the ViewInteraction object.

Espresso offers various view assertions (in the androidx.test.espresso.assertion. ViewAssertions package) to guarantee the matched view is what we anticipated, much like view matchers and view actions. Any assert can be checked using the check function of ViewInteraction by feeding it the appropriate view assertion once onView matches and returns the ViewInteraction object.

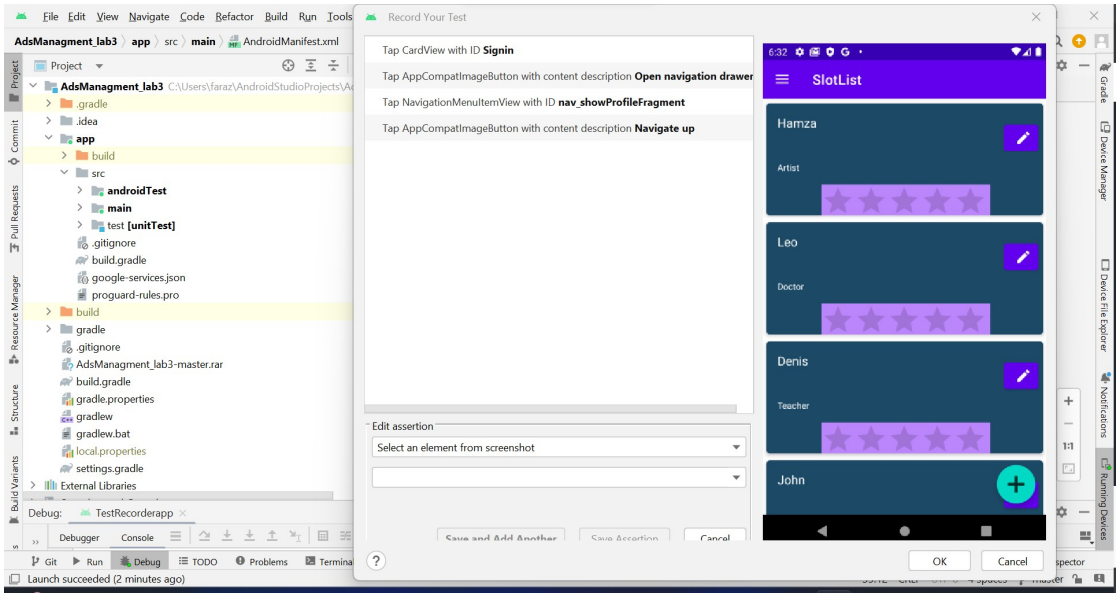

Figura 2.1: Espresso Framework

UIAtomator

UI Automator is a functional cross-app UI testing framework for installed and system apps. Depending on which activity has the focus, the UI Automator APIs

allow you to interact with visible items on a device, enabling you to perform actions like opening the Settings menu or the app launcher on a test device. Useful descriptors like the text displayed in a UI component or its content description can be used in your test to look for that component quickly. UI Automator allows access to the device state in addition to the features offered by Espresso. It is possible to find out the device's orientation, the size, and the resolution of the screen, as well as to carry out actions like rotating the device and touching buttons.

The following are some of the main characteristics of the UI Automator testing framework:

- **UiDevice**- The main method for accessing and controlling a device's state is through the UiDevice object. UiDevice methods are used in tests to examine the status of different attributes, such as the display size or current orientation. The UiDevice object can also be used in the test to carry out device-level operations by rotating the device in a certain direction and pressing the Home and Menu buttons and the D-pad hardware buttons.

- **UiCollection**- Lists all of the UI elements in a container so that they may be counted or targeted based on their visible text or content-description properties. If you want to recreate user interactions with a collection of things (like songs in a music album or a list of emails in an inbox), the UiCollection class is used. Specify a UiSelector that looks for a UI container or wrapper of other child UI components, such as a layout view containing child UI elements, to generate a UiCollection object.

- **UiObject**- A UI element displayed on the device is represented by a UiObject. Finding a UiObject that represents a view that fits a selector criterion can be done using the findObject() method. As required, the UiObject instances are used to build in other phases of app testing. It should be noted that each time the test utilizes a UiObject instance to click on a UI element or query a property, the UI Automator test framework looks for a match in the current display.

- **UiScrollable**- Searching for items in a scrollable UI container is supported by UiScrollable. The UiScrollable class emulates vertical or horizontal scrolling across a display. This method is useful when you need to scroll to see an off-screen UI element.

- **UiSelector**- A UiSelector represents a query for one or more target UI components on a device. The first matching element in the layout hierarchy is returned as the target UiObject if multiple matching elements are discovered. The search can be narrowed by chaining together several attributes when building a UiSelector. A UiAutomatorObjectNotFoundException is raised if

no matching UI element is located. To nest several UiSelector instances, we can use the childSelector() function. Using a Resource ID rather than a text element or content descriptor is suggested when specifying a selector. Not all items include text (icons in a toolbar, for instance). Because text selectors are fragile, even little UI changes can cause tests to fail.

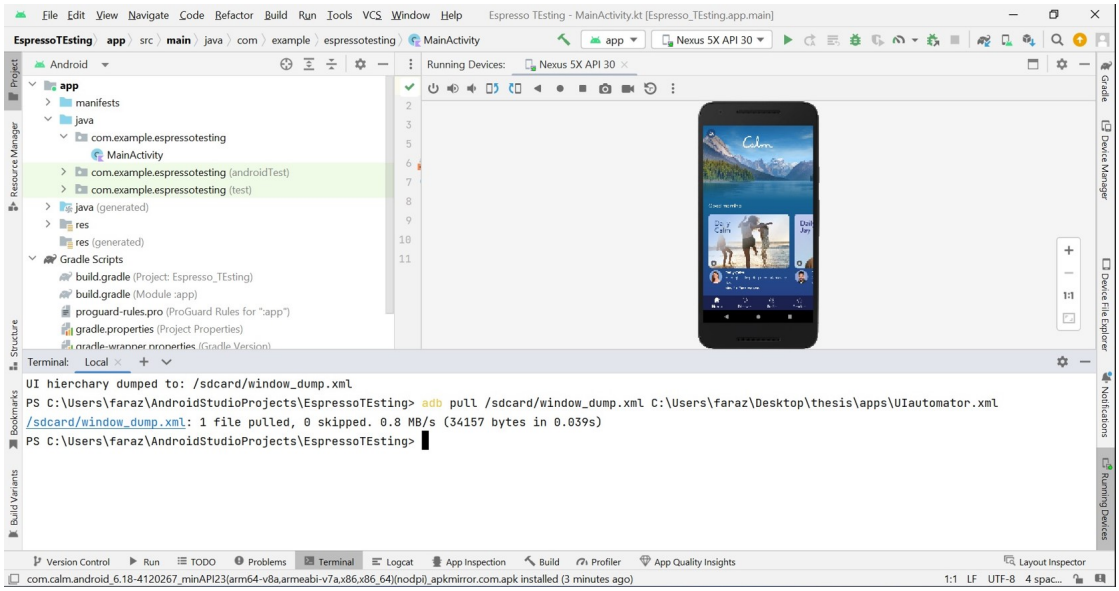

Figura 2.2: UIautomator Framework

Appium

Appium is an open-source tool that is widely used for automating mobile application testing, including Android testing. It provides a framework and APIs for testing native, hybrid, and mobile web applications on Android devices. Here are some key features and concepts related to using Appium for Android testing:

- **Cross-platform Capability**: Appium is designed to support both Android and iOS platforms, allowing you to write test scripts that can be used across different mobile platforms with minimal changes.

- **WebDriver Protocol**: Appium implements the WebDriver protocol, which enables you to write tests in various programming languages such as Java, Python, Ruby, etc. You can use WebDriver commands to interact with the application's user interface elements and perform actions like tapping buttons, entering text, swiping, and verifying element states.

- **Real Devices and Emulators**: Appium supports testing on real Android devices and emulators/simulators. You can specify the desired device or emulator configuration in your test scripts, allowing you to test your app on various devices with different screen sizes, resolutions, and Android versions.
- **Appium Server**: The Appium server bridges your test scripts and the mobile device/emulator. It receives commands from your test scripts, translates them into the appropriate actions, and executes them on the target device.
- **Desired Capabilities**: When initiating a test session, you need to define desired capabilities, which are a set of key-value pairs that provide information about the test environment, such as the device name, platform version, app package name, app activity, and more. These capabilities help Appium identify the device and application to test.
- **Inspector Tools**: Appium provides inspector tools, such as Appium Desktop and Appium Inspector, which allow you to inspect the UI hierarchy of your application and identify the unique identifiers (e.g., resource ID, XPath, accessibility ID) for the elements you want to interact with in your test scripts.
- **Integration with Test Frameworks**: Appium can be integrated with popular test frameworks like JUnit, TestNG, NUnit, and others, allowing you to organize and manage your test scripts effectively.

These frameworks provide features for test case management, reporting, and running tests in parallel. Appium's extensive documentation, community support, and active development make it a popular choice for Android testing. It provides a flexible and powerful platform for automating mobile app testing, allowing you to write reliable and scalable test scripts to ensure the quality of your Android applications.

Capitolo 3

Methodology

3.1 Research Methodology

The goal of this chapter is the widget analysis of Android application and their attribute usage. Widget analysis of an application is the process of examining the various widgets or user interface elements present in a mobile application. It involves assessing their functionality, design, placement, and overall user experience. To understand what changes are made in the application's graphical user interface with respect to time and with a new release of the application. Here are some key aspects to be considered during the widget analysis of a mobile application

- **Widget Types**: Identify the different widgets used in the application, such as buttons, text fields, checkboxes, radio buttons, drop-down menus, sliders, lists, tabs, etc. Understand their purpose and how they contribute to the app's overall functionality.

- **Widget Placement**: Evaluate the placement of widgets within the application's screens. Assess if the placement follows standard design principles and if it facilitates easy access and interaction for users. Ensure that widgets are logically grouped and located where users expect to find them.

- **Widget Styling and Visual Design**: Analyze the visual design of widgets, including their color, shape, size, font, and overall aesthetics. Ensure the styling is consistent throughout the application and aligns with the app's branding guidelines. Pay attention to the contrast, readability, and visual hierarchy of widgets.

- **Widget Labels and Instructions**: Review the labels and instructions associated with each widget. Check if they are clear, concise, and meaningful

to users. Labels should provide users with a clear understanding of the purpose and functionality of the widget.

- **Widget Interactions and Responsiveness**: Evaluate how users interact with widgets and assess their responsiveness. Test the behavior of widgets when tapped, swiped, or interacted with using gestures. Ensure that widgets provide appropriate feedback, such as visual cues or haptic feedback, to indicate user actions and system responses.

- **Widget Usability and Accessibility**: Consider the usability of widgets by assessing their ease of use and intuitiveness. Check if widgets adhere to accessibility guidelines, such as providing proper labeling, support for screen readers, and accommodating users with disabilities. Ensure that widgets are accessible to a wide range of users.

- **Widget Performance**: Assess the performance of widgets in terms of responsiveness, load times, and resource consumption. Verify that widgets do not significantly impact the application's overall performance, leading to slow response times or increased battery usage.

- **Widget Compatibility**: Test the application on different devices, screen sizes, and resolutions to ensure that widgets adapt and display correctly across various configurations. Verify that widgets are responsive in both portrait and landscape orientations.

- **Widget Consistency**: Ensure consistency in widget design, behavior, and functionality across different screens and sections of the application. Inconsistencies may lead to confusion and a fragmented user experience.

GUI testing assesses the visual elements, user interactions, and overall usability of an application's interface. For this reason, we choose to evaluate real applications from the market to examine the composition of various applications. By evaluating real applications, we can gain practical insights into the composition of GUI components and their effectiveness in delivering a seamless user experience.

when conducting GUI testing, it is essential to examine how different applications handle interface design principles such as layout, color scheme, typography, and iconography. Real-world applications provide us with diverse examples to study and compare, enabling us to identify effective design patterns and avoid potential pitfalls. Through this evaluation process, we can determine the composition of GUI elements that best suit users' preferences and expectations.

This chapter focuses on the research methodology, subject pool identification procedure, and widget analysis procedure employed in the study. These components are critical in ensuring the validity and reliability of the research findings.

Several of the qualities are connected to the widget's functionality (such as clickable, focused, scrollable, etc.). These characteristics are frequently boolean and are always valued (by default, false). A different category of widgets displays the widget's textual content, if any, or offers a description that helps users recognize the widget (e.g., id, content-desc,...). We are interested in learning how frequently a value might be possible in these widget categories and how diverse the values anticipated across apps are. An attribute is more likely to be used as a locator if it is commonly valued (i.e., not empty) and sufficiently diverse throughout the collection of applications, according to the logic underpinning this research. In reality, attributes with little variability won't effectively differentiate between several widgets on the same screen or application. The total number of distinct values an attribute has taken over its whole range of values is what we refer to as variability. It shows the potential for a characteristic to take on distinct values for various widgets.

In this analysis, we compare the attribute values between two releases. For each application, we manually define a set of corresponding widgets or widgets with the same functionality in both program versions. To find this set, we manually reviewed each app's two versions to check which widgets were related to the same function (regardless of how they looked). We then labeled them as corresponding widgets and used their metadata in the analysis. To determine if and how the relevant widgets' attributes changed over time, we compared the attribute values of the two versions' related widgets. By calculating these statistics, we may determine which attributes are less suited for use as locators by estimating how unstable the attributes are. We measure instability as the ratio of the number of times a certain widget attribute has changed value between the chosen two versions. Instead, stability can be considered the polar opposite of stability, meaning that the less stable it is, the more likely it is to break.

3.1.1 DataSet Selection

When selecting an application for GUI testing, it is essential to consider both variability and the existence of important qualities.

Variability: To check an application's functionality, usability, and responsiveness, GUI testing involves interacting with the graphical user interface. Many different platforms, operating systems, browsers, screen sizes, resolutions, and application localization options exist. To test GUIs effectively, we must choose an application that supports the range of variability required. This makes sure that the tests we run are thorough and accurate to the target environment.

Existence of important Qualities: When conducting GUI testing, it is vital to validate the application's existence of important qualities such as functionality,

usability, performance, and security. The selection of an application for GUI testing should be based on the specific qualities critical for testing objectives.

To do this, we chose many real mobile applications from various categories. We searched various online resources, including APKMirror, AppBrain, and fdroid, to select the application process. In this thesis work, we have selected all the applications from APKMirror. An APKMirror website gives users access to Android application package (APK) files for downloading programs not offered on the Google Play Store. The Android operating system distributes and installs apps using APK files, a native format for that platform. For an app to run properly on an Android smartphone, they are equipped with all of the essential code, resources, and metadata. There are numerous instances in which APKMirror is beneficial. We might be able to download and install programs from an APKMirror if, for instance, the Android device we are using does not have access to the Google Play Store. In addition, there is a way to install an older version of an application that is no longer available through the Google Play Store. An APKMirror has all the older versions available for download.

Application Selection Criteria

In our selection process, we focused on applications with multiple versions available. We specifically downloaded the most recent version of each application and applied a set of inclusion criteria to determine which versions to evaluate. These criteria were established to ensure the relevance and suitability of the chosen versions for our analysis.

The following inclusion criteria were employed:

- **IC0**: The application must have multiple versions available on APKMirror, which hosts various Android application packages (APKs). This criterion allowed us to select applications with a history of updates and improvements.

- **IC1**: We considered the most recent version of each application to ensure that our analysis focused on the latest developments and features. This criterion enabled us to evaluate applications that were up-to-date and relevant to users.

- **IC2**: We assessed whether the application met our minimum standards for functionality and usability. This criterion ensured that the selected applications were viable candidates for evaluation and provided meaningful insights into the composition and characteristics of their GUI components.

- **IC3**: The application's availability on APKMirror was crucial to ensure access to different versions for comparison. This criterion ensured that the selected applications could be reliably sourced and evaluated.

- **IC4**: The application's home screen must be accessible without registering or logging in explicitly. The use of a login based on an external API (such as login via Google, Facebook, or Twitter), deemed an acceptable example, is excluded from this criterion.

- **IC5**: The previous version must be installed on the utilized Google Nexus 5X Android Virtual Device (AVD), which must be running Android SDK 29.

We initiated an iterative evaluation process after satisfying the first four inclusion criteria. We examined each version of the application available on APKMirror, starting from the earliest release and progressing to the most recent. This iterative approach allowed us to track the evolution of the application's GUI components and assess any changes made over time.

We compared each version's GUI components, functionalities, and user experiences during the iterative evaluation. We aimed to identify notable differences, improvements, or regressions in the GUI composition. This iterative process enabled us to gain insights into the evolution of the application's GUI elements and understand the progression of its design and usability.

The evaluation continued until we encountered a version that satisfied the fifth inclusion criterion, IC5. The details of IC5 were not provided, but it likely represented a specific requirement related to the application's GUI components, functionality, or user experience. Once a version meeting IC5 was identified, it was considered the final version for our analysis.

By employing these inclusion criteria and the iterative evaluation process, we ensured that our analysis encompassed the applications' most recent and relevant versions, enabling a comprehensive examination of their GUI components and their evolution over time.

Tabella 3.1, we present a complete list of applications used in the study. Each application is described with its corresponding category, package name, release name, and publication dates of the two versions under consideration. Each row represents an application, and the corresponding columns provide the necessary information about the application. The breakdown of columns is as follows:

- **Category**: The category or classification of the application (i-e, productivity, Art, Books, Comics, etc.)

- **Application**: The name or identifier of the application.

- **OldV**: The specific version or release name of the application (i-e version3.7.2)

- **Release**: The publication dates of the two versions under consideration. It's associated with the release name of each application.

Category	Application	OldV	Release	NewV	Release
Art	Sketchbook	3.7.2	18-10-2019	5.3.1	27-06-2022
Auto	CarMax	2.47.1	30-08-2018	3.31.0	13-02-2023
Beauty	Mirror Plus	2.9.1	24-09-2016	4.2.2	24-03-2023
Books	YouVersion	6.4.2	21-12-2016	9.20.0	27-02-2023
Business	UPS mobile	4.5.0	07-08-2016	9.9.3	14-03-2023
Comics	Cdisplay	1.1.70	08-04-2023	1.3.57	08-03-2023
Communication	Firefox	65.0.1	13-02-2019	111.0.0	14-03-2023
Education	Google Classroom	4.5.212	12-06-2018	8.0.341	02-11-2022
Entertainment	Tubi	3.7.0	01-12-2020	4.43.2	10-03-2023
Events	Gametime	11.2.15	28-03-2019	2023.3.0	03-03-2023
Finance	wise	7.29.1	13-10-2021	8.3.2	23-03-2023
Food	Burger king	6.2.0	14-05-2019	6.25.25	20-03-2023
Health	Calm	3.11	03-01-2018	6.18	08-03-2023
House	Angi	21.0.18	26-11-2021	23.10.0	15-03-2023
Libraries	Allinone toolbox	6.4.3	08-08-2016	8.3.0	30-03-2023
Lifestyle	Pinterest	8.39.0	24-10-2020	11.9.0	16-03-2023
Maps	Transit	4.3.1	16-11-2017	5.13.5	24-02-2023
Media	Video downloader	1.1.98	21-9-2020	2.4.6	21-03-2023
Medical	Nevada COVID trace	1.2.11	05-10-2020	1.4.0	14-09-2022
Music	Soundcloud	2017.12.14	14-12-2017	2023.03.03	08-03-2023
News	NewsBreak	4.6.4	27-11-2019	23.11.0	22-03-2023
Personalization	Backgrounds hd	4.8.25	17-01-2017	5.0.064	16-02-2023
Photography	Lightroom	4.4	13-08-2019	8.2.2	16-03-2023
Productivity	HP smart	4.7.104	01-05-2018	11.0.0	22-03-2023
Shopping	Walmart	22.1.1	15-01-2022	23.09.0	14-03-2023
Social	Reddit	2020.30.0	13-08-2020	2023.11.0	21-03-2023
Sports	FOX Sports	5.0.0	20-07-2020	5.71.0	27-03-2023
Tools	Google translate	5.12.0	08-09-2017	7.0.22	15-03-2023
Travel	Booking.com	11.9	28-01-2017	35.9	13-03-2023
Weather	AccuWeather	4.8.2	06-07-2017	8.9.1	24-03-2023

Tabella 3.1: List of selected applications with their release version and date

- **NewV**: The latest version or release name of the application (i-e, version 5.3.2)

To ensure complete coverage and efficient testing, it's important to consider several aspects while choosing applications for GUI testing. For data set selection, we choose to test 30 applications for the following reasons: A broad and representative selection of 30 apps represents the software landscape. Applications from various categories, including media players, browsers, productivity tools, and more, are included in this list. It guarantees that a wide range of GUI components, functions, and interactions are tested. Give top programs with a sizable user base priority. we can meet the demands and expectations of a large audience by testing

frequently utilized applications. This method makes it easier to spot frequent problems, guarantees platform compatibility and provides a better user experience.

Applications are chosen with varied degrees of sophistication. This method allows us to evaluate the GUI testing technique for straightforward, moderately complicated, and extremely complex interfaces. Testing apps with varying degrees of complexity enables the discovery of potential usability problems and guarantees a thorough testing procedure.

3.1.2 Visual Mutation

Mutation testing is a fault-based testing technique that assesses the effectiveness of a test set by creating a set of faulty programs called mutants. These mutants are derived from the original program by introducing simple syntactic changes, representing common mistakes programmers make.

The goal of mutation testing is to evaluate the ability of the test set to detect these seeded faults. Each mutant is executed using the input test set. If the output of a mutant differs from the output of the original program for any test case, it indicates that the seeded fault has been detected.

The mutation score is calculated by determining the ratio of the number of detected faults to the total number of seeded faults. It represents the quality of the input test set and indicates how effective the tests are in identifying the seeded faults. A higher mutation score suggests a more thorough and effective test suite, while a lower score implies that the test suite may lack fault detection capability.

By measuring the mutation score, developers and testers can gain insights into the adequacy of their test suite and identify areas where additional testing or improvements are needed. The ultimate goal is to achieve a high mutation score, indicating a strong test suite that can effectively detect faults and enhance the overall quality of the software.[9]

we started the visual mutation testing process for the GUI of the application to identify the different components, such as buttons, text fields, and dropdowns within the application. Generate visual mutations by applying various modifications to the GUI components, such as changing colors, size, and positions and adding/ removing elements. Then, we captured screenshots and events (user interactions) while executing test cases on both the original and mutated versions of the app. To execute test cases on the mutated app, we run test cases on the mutated app to test its behavior and capture relevant screenshots, and then we will compare the captured screenshots between the original and the mutated versions to detect any visual differences. After the comparison, we will analyze the comparison results to detect visual differences, such as changes in color, layouts, missing or added components

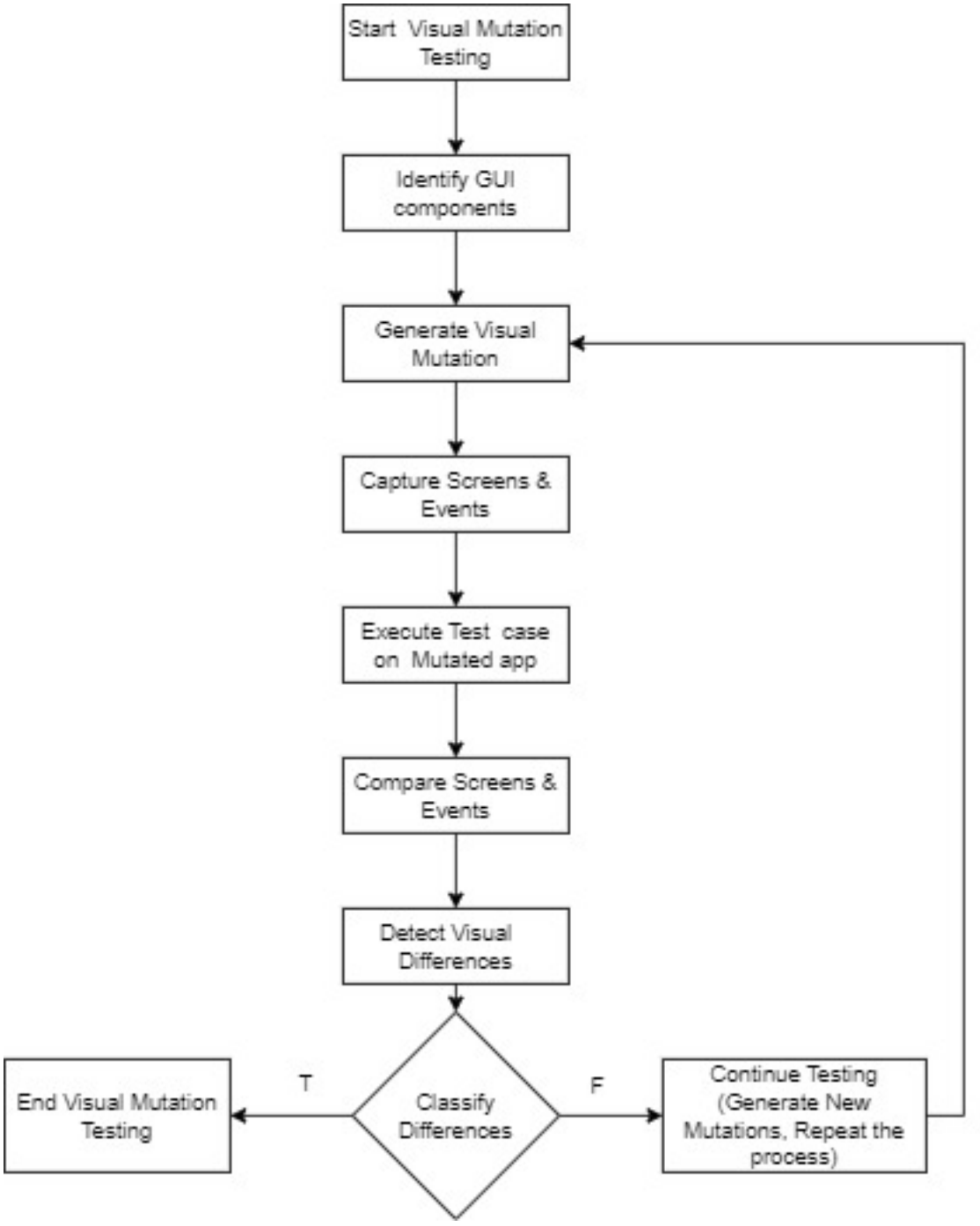

Figura 3.1: Visual Mutation Flow chart

At last, we will classify the detected visual differences based on their severity or

impact on GUI functionality by allowing for prioritization and further analysis. If the visual mutation testing process is successful, we will end the visual mutation testing, and if necessary, we will generate new visual mutation by repeating steps 3-8 and continue the testing process.

Mutation Operators

According to the visual mutation characterization provided by Alegroth et al. [10] in order to link each appropriate "widget" with its modification, if any. We adopted a portion of the full characterization (given in Tabella 3.2) for our labeling approach because some mutant operators did not fit our needs. We identified certain mutant operators that were more relevant or aligned with their specific requirements or objectives. The labeling approach we adopted would involve associating each widget with its respective modification based on the chosen subset of mutation operators.

we started our analysis from [11] as the baseline in which the first part of the analysis was the visual comparison of two versions of the application. We considered the mutant operator **M0** to be used when nothing has changed. Mutant operator pairs **M11-M13** and **M15-M16** were regarded as being mutually exclusive. The first reason is that both mutants change the original coordinates by assuming a different generic value or overlapping other widgets. The pair **M15-M16** is fundamentally associated with a change in the widget's visual appearance, either due to a change in the widget type or with a general variation. During the labeling process, we saw that each change in a widget's type reflected a change in its look. We chose to designate the condition as the M16 mutant operator to achieve a more precise description.

No.	Type	Mutation operators
M0	Static	No changes affected the widget
M1	Remove	Remove Completely
M2	Remove	Invisible
M3	Duplicate	Add Identical widget
M4	Duplicate	Add similar widget
M5	Duplicate	Add different widget
M10	Modified	Reduced the size o window to hide widgets
M11	Modified	Modify location of a widget to a proper location
M13	Modified	Modify location of a widget to overlap with another
M14	Modified	Modify size of widgets
M15	Modified	Modify appearance of widgets
M16	Modified	Modify the type of widgets (Button changed to TextField)

Tabella 3.2: Mutation operators

In the second part, we performed a longitudinal analysis. During the longitudinal analysis (i-e, analysis of the application in consecutive releases), we found some useful mutant operators like **M1** as some widgets are completely removed from the app GUI and never used in the new releases. For some of the widgets partially removed from the GUI and invisible in the app's GUI, we used the mutant operator **M2**. During the longitudinal analysis in the consecutive releases, some widgets are duplicated by adding identical widgets, similar widgets, or different widgets for these mutations. We considered the mutation operators **M3**-**M5**. Mutant operator pairs **M11**-**M13** were thought to be mutually exclusive. The first is that both mutants alter the original coordinates by assuming a generically different value or overlapping other widgets. The pair **M15**-**M16** is fundamentally associated with a change in the widget's visual aspect, either due to a change in the widget type or with a general variation. If no changes affected the widget, we considered it as mutant **M0**.

3.1.3 Analysis Process

After downloading and installing the two chosen versions of each app on the Google Nexus 5X AVD, a setup process was required to prepare each app for the home screen. This setup process involved logging in with external APIs as needed and closing all one-time screens or wizards that appear when an app is first opened on a device. After that, we examined its XML layout, which was collected via the UI Automator dump command. Once we had the dump file, we made a script to filter out any Android Views that were only ever used as containers and never appeared on the screen. The names of all potential characteristics and the corresponding presumed values for each widget were then parsed from the remaining widgets to produce a CSV file. Reviewing the downloaded files, we discovered that UIAutomator had not provided any unique identifiers, such as XPath. As a result, we decided to include a distinctive identifier in the form of a progressive number along the UI hierarchy in the created CSV file. This will enable us to identify between all of the widgets. We connected the matching widgets in the two versions of the program using the said identification. The old and updated versions of the CSV file containing the attribute values were collected for each application. For each version, we took a screenshot of the home page to quickly access the actual visual representation if necessary for the study. We had to identify all the pairs of related widgets—that is, widgets that play the same role in both the old and new versions of the program—to compare how the attributes for the same widget changed between two different iterations of the same application. The widgets with the same conceptual significance utilized to carry out the same application actions were carefully inspected in all the pairings of versions of the 30 chosen apps to obtain this list of similar widget pairs. We created matching widget

pairings by manually identifying the widget correspondence. The differences in the characteristics associated with the same conceptual widget in the program were then obtained using a lookup in the CSV files. With the help of this knowledge, it is feasible to evaluate if an attribute is suitable from the standpoint of attribute evolution as a locator. The main standard is that an attribute can be deemed less reliable as a locator if it is more likely to change in future application releases.

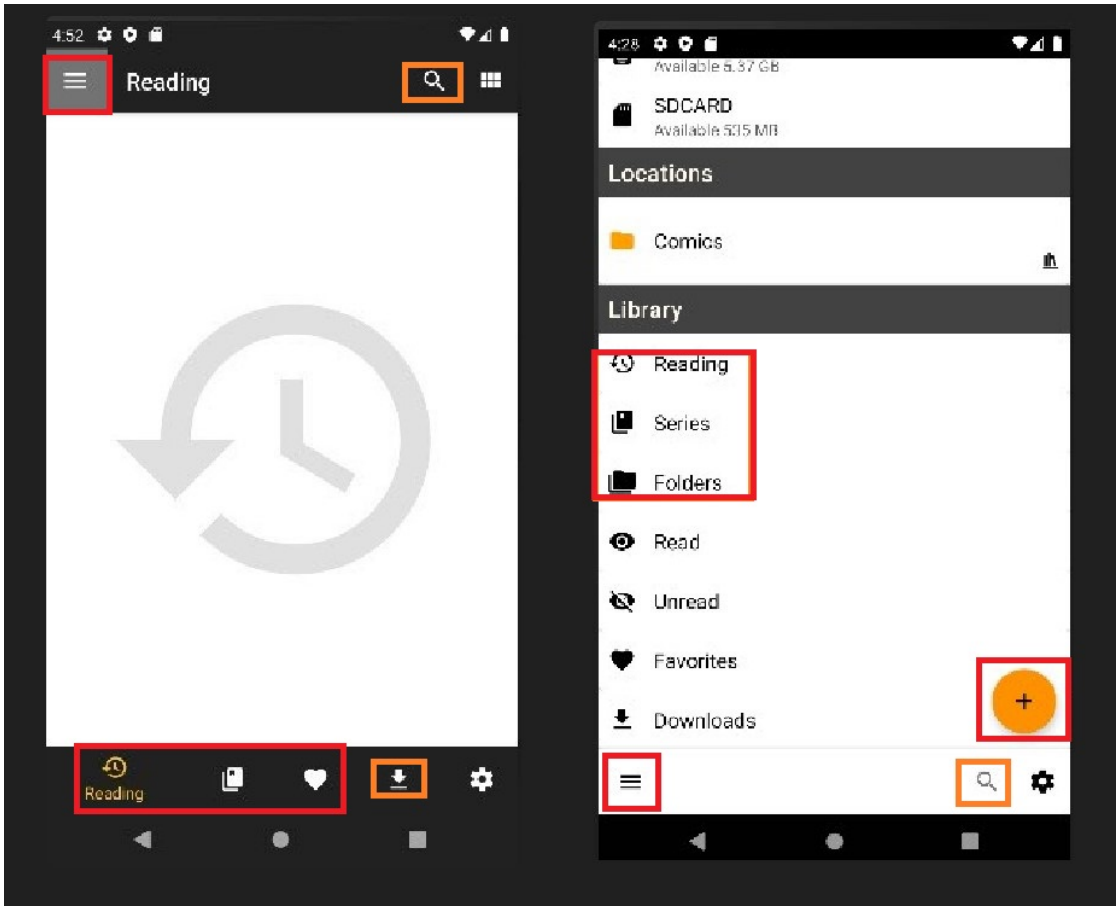

Figura 3.2: Difference between old release and latest release

With the help of a generated CSV file, we started to analyze each application from the list to get Oracle's table containing all the attributes in the user interface. Our focus is to identify the changes by considering the nodes, i-e, content-description, resource-id, and the progress of the attribute and also for some applications if we get the text for that attribute. In Tabella 3.3, we reported all the attributes with the possible change in the new version of the application. These changes are referred to as oracles.

Once we finished the analysis of widget attributes in the old and new versions,

App Name	Comment	Old Node	New Node
sketchbook	main menu	1	1
sketchbook	full screen	2	8
sketchbook	tools menu	3	4
sketchbook	brush	4	5
sketchbook	color	5	6
sketchbook	layer	6	7
Mirror	menu	0	6
Mirror	light button	4	9
Mirror	freeze button	5	
Mirror	zoom bar	6	11
Mirror	exposure bar	7	12
Mirror	MIRROR		16
Mirror	3D		17
cdisplay	Menu	2	40
cdisplay	Search	4	41
cdisplay	Display Mood	5	
cdisplay	Reading icon	9	18
cdisplay	Series icon	11	21
cdisplay	Folder icon		24
cdisplay	Fevorites icon	12	33
cdisplay	Download icon	13	36
cdisplay	Settings icon	14	42
cdisplay	Floating button		38
cdisplay	Read		28
cdisplay	unread		31
youVersion	Menu	2	
youVersion	signin	4	7
youVersion	HOME	8	36
youVersion	READ	10	41
youVersion	PLANS	12	45
youVersion	VIDEOS	14	
youVersion	LIVE	16	
youVersion	Search		49
youVersion	verse of the day	21	25
youVersion	share verse	28	27

Tabella 3.3: All Oracles

then, we extended our analysis to see what are the possible changes that occur in the GUI of the application in each version. For this reason, we randomly selected five applications from the list and started the longitudinal analysis of each of the five applications. We started the analysis from the first release of the application and continued to collect data for each release. The longitudinal analysis process is done manually.

During this process, we downloaded every application release from APK Mirror. We installed them on the Google Nexus 5X AVD to get the XML by using the command UIAutomator dump to get the dump file, and then we got the XML file for each release. For the visual verification of widgets in each version we took a screenshot of that version, we continued the process until we get the data for the final release that was available in the APK mirror. For each application version, we have the XML file, the CSV file, and the screenshot.

Once we had all the data available for longitudinal analysis of that application, we started the comparisons of the CSV file and the corresponding visual mutations, as well as the screenshot for consecutive versions of the application. Starting from the first release of the application, we examined the widgets used within the app and analyzed their attributes, such as their appearance, behavior, and functionality. It helps in identifying any changes made to the widget in the new release of the application, widget analysis, and the potential changes that may occur in the widget attribute after a new release.

The overall process is categorized into two steps: identifying the widgets in consecutive releases and the changes occurring in the new release.

- **Identifying widgets**: start identifying the widgets used in the application, which are the UI components that provide specific functionality and user interaction on the home screen or with the app.

- **Understanding Attributes**: Analyze the attributes of each widget, including size, layout, appearance (colors, fonts, icons), behavior (interactivity, animations), and data handling (data binding, updates).

- **Assessing Functionality**: Evaluate how each widget functions within the application, such as button actions, input fields, progress indicators, or data display.

- **Experience**: Consider the overall user experience provided by the widgets, including ease of use, intuitiveness, and responsiveness.

When a new release of an application occurs, several changes can be made to widget attributes based on user feedback, design improvements, or feature enhancements. Here are some common changes that occurred.

- **Visual Enhancements**: Widgets might receive visual updates, such as color changes, fonts, icons, or overall styling to align with updated design guidelines.
- **Layout Modifications**: Widgets could undergo layout changes to improve their positioning, spacing, or alignment within the app's user interface.
- **Added/Removed Functionality**: New widgets might be introduced, while existing ones may be removed or modified to accommodate additional features or improve existing functionality.
- **Behavior Improvements**: Widgets can have changes in their behavior, including enhanced interactivity, smoother animations, or better responsiveness to user input.
- **Performance Optimizations**: Changes to widget attributes might aim to improve performance, reduce resource usage, or optimize battery consumption.

After having all the data available for each application version, we started the Oracle analysis of the app's attributes with the help of a CSV file. The nodes of attributes that we considered are the content description, resource-id, and progress. The change in attribute values is reported in Tabella 3.4 where the first column is the attribute itself considered for the longitudinal analysis and the columns from A-T are the consecutive versions of the application attribute change. In the Tabella 3.4, we reported the oracles of two different applications chosen for the analysis process. We can see a complete detail of the attribute changes across the consecutive releases.

CDisplay																				
Attribute	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**
Menu	2	2	2	2	2	2	2	2	2	2	40	40	40	40	40	40	40	40	40	40
Search	4	4	6	6	4	4	4	4	4	4	41	41	41	41	41	41	41	41	41	41
Display Mood	5	5	7	7	5	5	5	5	5	5	5									
Sort		6	8	8	6	6	6	6	6	6										
Reading icon	9	10	12	12	10	10	10	10	10	10	18	18	18	18	18	18	18	18	18	18
Reading title	10	11	13	13	11	11	11	11	11	11	19	19	19	19	19	19	19	19	19	19
Series icon	11	12	14	14	12	12	12	12	12	12	21	21	21	21	21	21	21	21	21	21
Series title											22	22	22	22	22	22	22	22	22	22
Folder icon					13	13	13	13	13	13	24	24	24	24	24	24	24	24	24	24
Folder title											25	25	25	25	25	25	25	25	25	25
Favorites icon	12	13	15	15	14	14	14	14	14	14	33	33	33	33	33	33	33	33	33	33
Favorites title											34	34	34	34	34	34	34	34	34	34
Download icon	13	14	16	16	15	15	15	15	15	15	36	36	36	36	36	36	36	36	36	36
Download title											37	37	37	37	37	37	37	37	37	37
Settings icon	14	15	17	17							42	42	42	42	42	42	42	42	42	42
Floating button											38	38	38	38	38	38	38	38	38	38
Read											28	28	28	28	28	28	28	28	28	28
unread											31	31	31	31	31	31	31	31	31	31
Sketchbook																				
main menu	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1				
fullscreen 2	2	6	6	6	6	6	6	6	6	6	6	6	6	6	8					
tools menu	3	3	2	2	2	2	2	2	2	2	2	2	2	2	2	4				
brush	4	4	3	3	3	3	3	3	3	3	3	3	3	3	3	5				
color	5	5	4	4	4	4	4	4	4	4	4	4	4	4	4	6				
layer	6	6	5	5	5	5	5	5	5	5	5	5	5	5	5	7				
undo																2				
redo																3				

Tabella 3.4: Oracles

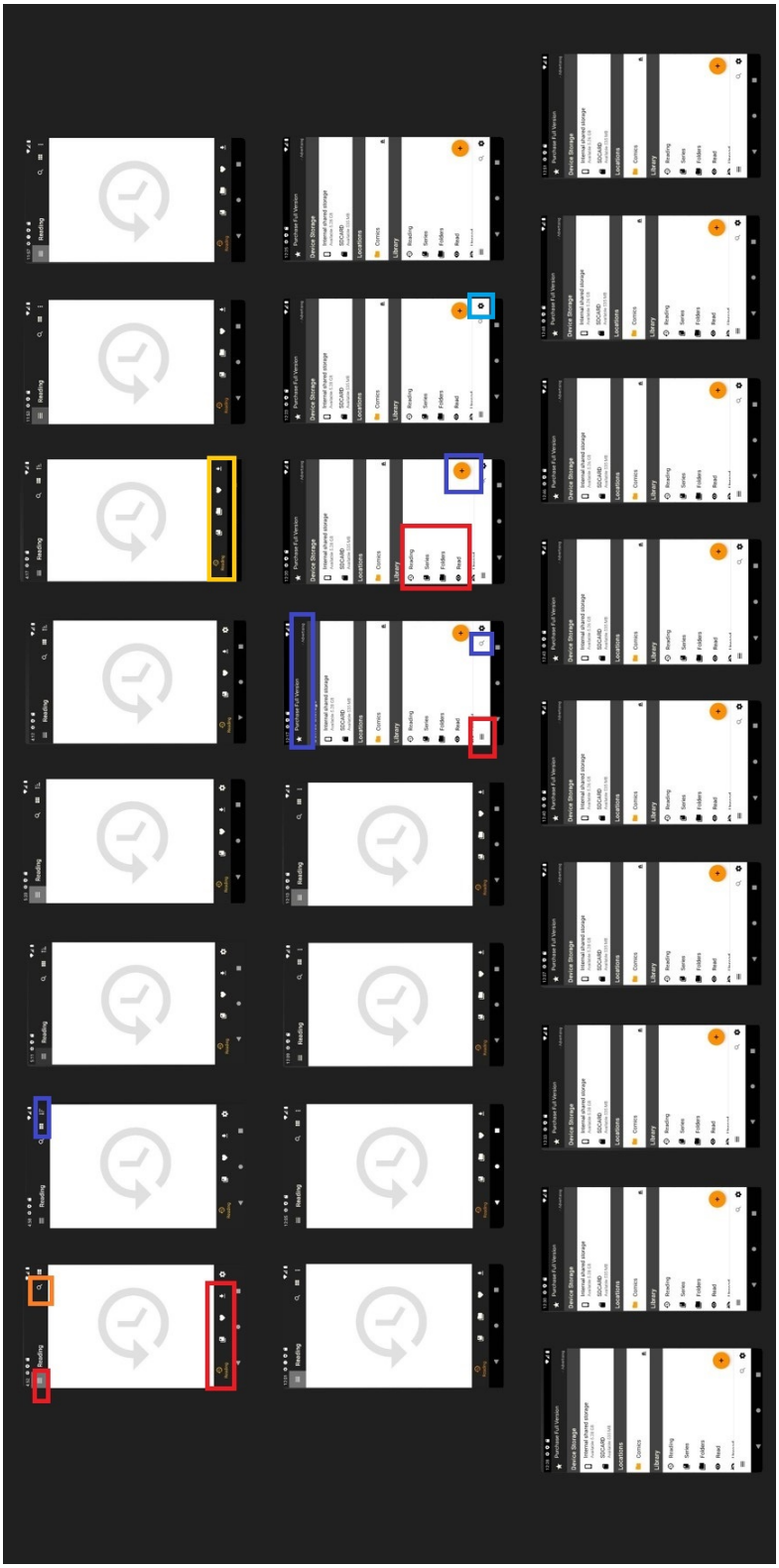

Figura 3.3: Visual Difference between the consecutive releases

Capitolo 4

Results

4.1

Collecting, analyzing, and interpreting data on GUI widget attributes in Android applications involved data processing techniques, statistical analysis, and visualization methods to gain insights into widgets' characteristics and properties. The first analysis is based on the total number of valued attributes.UI Automator helped us extract this information, including details about various properties (like focused, enabled, selected, etc) of the app. These properties can be either true or false. The information is split into two parts to clarify the analysis, each focusing on a different aspect of the valued properties.

In figure 4.1, Our analysis involves presenting the distribution of values that are either empty (meaning that an empty string is found in the UI Automator dump corresponding to that attribute) or valued (meaning that any string is found in the dump as a value corresponding to that attribute). Based on the provided data, we have a breakdown of the percentage of empty and present values for various attributes in GUI widgets and the distribution of empty and valued attributes in selected apps. Based on the graph, it can be observed that just the attributes (text, resource-id, and content-desc) do not always have values associated with them. Aspects generated by UIAutomator or Android Studio itself are the ones with 100 percent valued instances. Specifically, the bounds attribute relates to the upper-left and lower-right corners of the widgets on the screen, and the class attribute always corresponds to the Java class of the widget. Rather, the widget's reference in the app's hierarchical tree structure is reported via the index attribute. Lastly, Android Studio automatically sets the package attribute when a new View is created within the context of an application. NAF is essentially a boolean attribute, but it is typically not supplied (in the form of a NULL Value).

The distribution of values for the boolean characteristics that the UI Automator

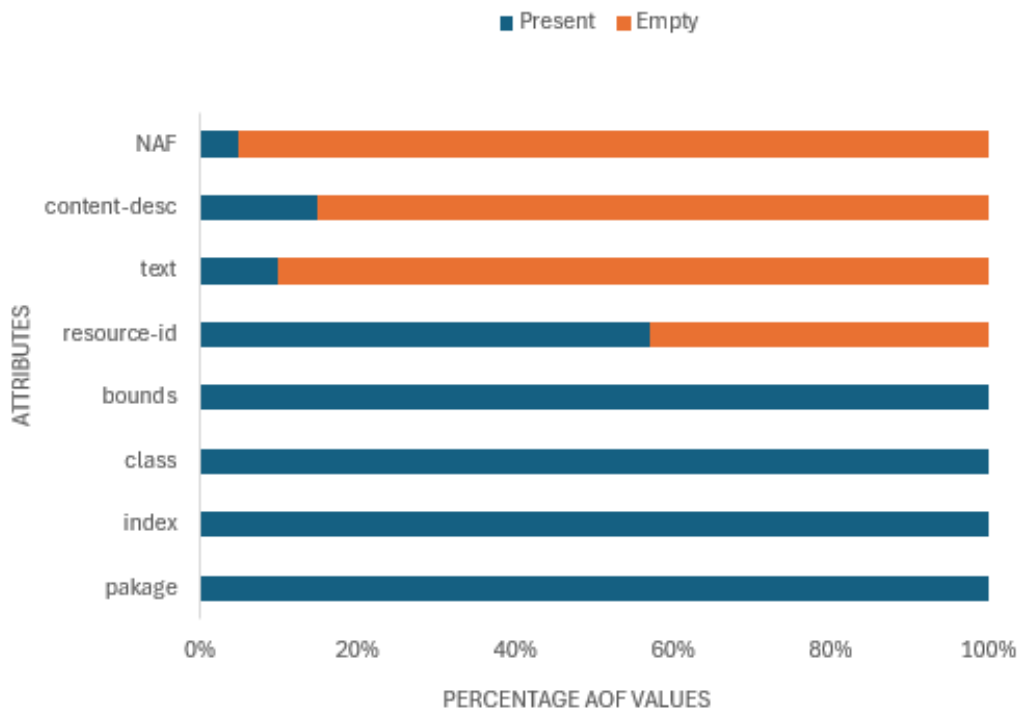

Figura 4.1: distribution of empty and valued attributes in the selected app

dump recovered is shown in Figure 4.2. The distribution of values across the various properties is visible: while most widgets are enabled, about 80 percent are clickable, meaning they may interact. While boolean values aren't efficient for finding widgets, it's important to remember that a weighted combination can help confirm the presence of other, more powerful locators because UI Automator always ensures their presence. It is important to remember that an attribute with a constant value across all widgets will not be helpful as a locator, even if the value is assigned randomly. Indeed, in that scenario, it would be impossible to distinguish between several widgets using the locator's value alone. This means the analysis must be deeper by combining data on the availability of values for a characteristic with the many values assumed by that particular attribute.

The second step for a deeper analysis regarding the nature When utilizing attributes, it is important to consider the different possible values each attribute may contain. Attributes can have values that vary greatly or remain fairly consistent across elements. Considering the range of an attribute The concept of how attributes are assigned is represented in Tabella 4.1. This plots the measure of attribute allocation. different values are distributed for each property: in particular, The value refers to the relationship between the variety of distinct elements and the

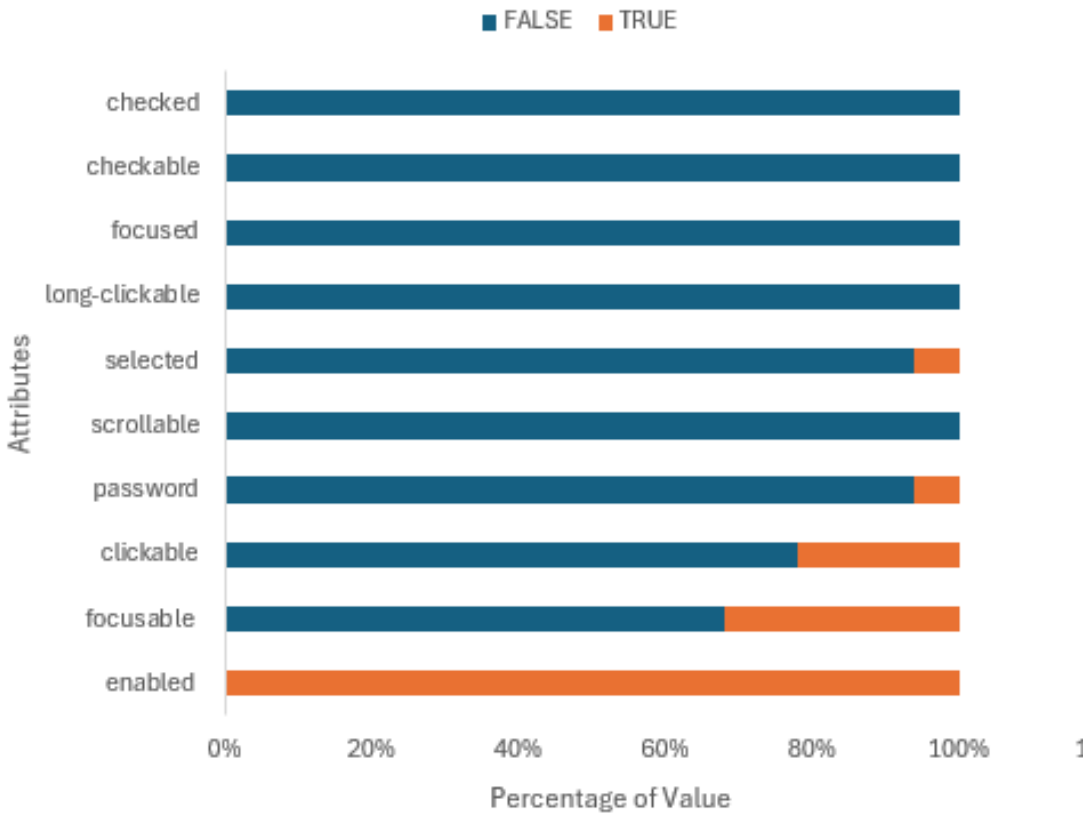

Figura 4.2: distribution of boolean attributes in the selected app

total number of elements. It quantifies how evenly distributed elements are by comparing the number of different types to the total count. A lower value is assumed over the number of valued widget properties. Although this indicator shows that the greatest variability is concentrated in the text, NAF, and content description attributes, from the previous finding, we infer that these are more likely to be left unassigned.

Therefore, we considered the ratio between the number of distinct values for a property and the total number of widgets to merge the information about variability. This specific distribution shows that the resource ID and text properties are the next most variable, with multiple possible values after the boundaries attribute. The boundaries characteristic has the largest variability, but its use as a location method dates back to outdated first-generation technologies with well-known shortcomings. Nonetheless, current location procedures might be strengthened by cross-referencing position-related attributes with other locators.

Notwithstanding the result, it must be taken into account that the value comparison was performed in a boolean way, which implies that for the bound attribute, the widgets must have coincident positions and dimensions to obtain two corresponding values. This constraint appears to be rare, leading us to wonder whether position-related metrics comparisons should be based on more sophisticated treatments, such as the percentage of screen overlapping.

In figure 4.3, we reported the sum of attribute changes across different releases of an application. Each letter (A, B, C, etc.) corresponds to a specific release, and the numbers represent the total number of attribute changes in each release for various widgets or components. In the application cdisplay, we can see that every time there is a significant release of the application there will be a change in the attributes of the application. This means that if the attribute of the application is changed, there will be a change in the oracles of the application. As we can see from version A, there is a significant attribute change as the application has the first release, and concerning time, some modifications were done; Release A: Various widgets were updated with thirty new attributes. Release B included thirty more attribute changes spread throughout the various parts of the application. Release C: There was a notable surge in attribute modifications, amounting to 67 in the entire program.version D: This version did not include changes to any widget's attributes.Release E: Similar to Release D, there were no attribute changes. Release F: 40 attribute changes were implemented across various application components. Release G: No attribute changes occurred. Release H: No changes were made to attributes. Releases I to V: No attribute changes were recorded in these releases.

Release M: 12 attribute changes were affecting different parts of the application. Release N to V: No attribute changes occurred.

Figure 4.4 shows the distribution of values for the attribute changes in all application versions. We can examine which attribute is most likely to be changed in the longitudinal analysis of the application. The attribute Clickable: This attribute was changed twice across different releases. It indicates whether a widget is clickable or not. The variability in this attribute might suggest alterations in the behavior of clickable elements within the application, possibly affecting user interaction.

Bounds: With 43 changes, the "bounds" attribute experienced significant variability across releases. This attribute defines the boundaries of a widget within the user interface. Changes in this attribute could indicate modifications to the layout or positioning of elements within the application, potentially reflecting redesign efforts or improvements in visual presentation. Content-description: Changed eight times across releases, the "content-description" attribute provides a textual description of a UI element for accessibility purposes. This attribute's variation may suggest updates to enhance accessibility features or improve descriptions for visually impaired users.

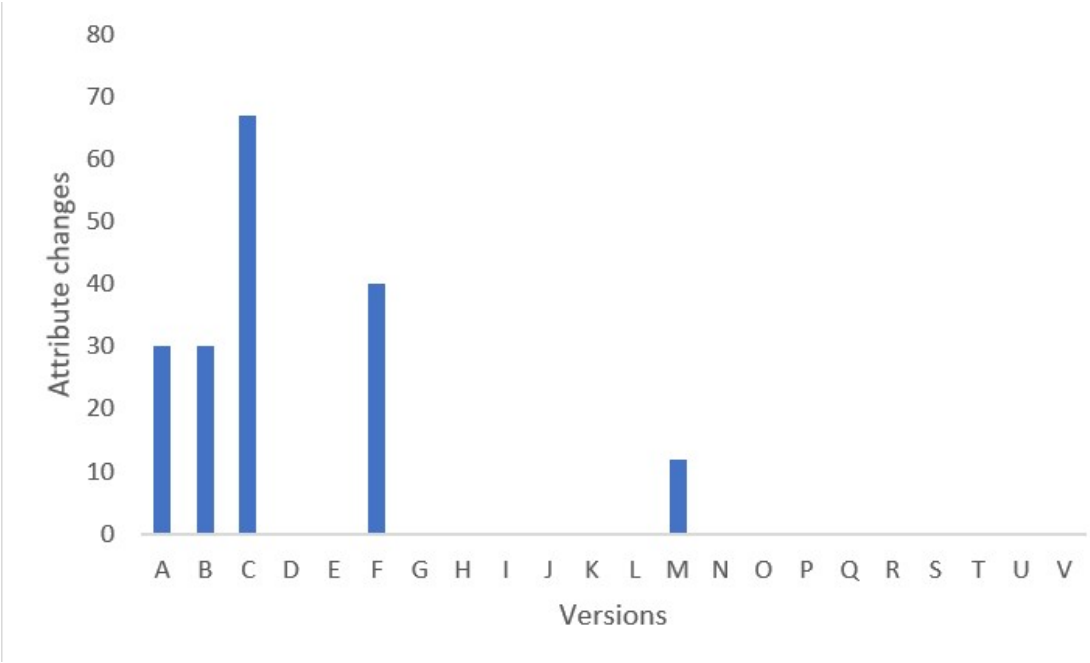

Figura 4.3: cdisplay with all versions and attribute variability

Index: This attribute was altered 19 times across releases. It typically represents the position of a UI element within its parent container. Variability in the "index" attribute could indicate changes in the ordering or arrangement of elements within the application's interface, reflecting layout or navigation flow adjustments.

Focusable: Changed nine times, the "focusable" attribute determines whether a UI element can receive focus. Variability in this attribute might indicate adjustments to the focus behavior of elements, potentially improving navigation or accessibility for users interacting with the application via keyboard or other input devices. Text: With seven changes, the "text" attribute represents the textual content displayed by a UI element. Variability in this attribute could signify updates to the displayed text within the application, such as changes in labels, button text, or other user-facing content.

Class: Altered 24 times, the "class" attribute specifies the class or type of a UI element. Variability in this attribute may indicate changes in the implementation or styling of elements within the application, potentially reflecting updates to the underlying codebase or user interface framework. Resource-ID: Also changed 24 times, the "resource-id" attribute uniquely identifies a UI element within the application's resource hierarchy. Variability in this attribute might indicate updates to the naming or identification of elements, potentially reflecting changes in the application's structure or organization.

Selected: Changed seven times, the "selected" attribute indicates whether a UI element is currently selected. Variability in this attribute may suggest adjustments to the selection behavior of components within the application, potentially improving user interaction or highlighting chosen items.

Long-clickable: This attribute was altered twice across releases. It determines whether a UI element supports long-click interactions. Variability in this attribute might indicate changes in long-press behavior for specific elements, potentially enhancing user interaction options within the application.

Among these attributes, "Bounds," "Class," and "Resource-ID" exhibit high variability with 43, 24, and 24 changes, respectively, suggesting significant adjustments to layout, styling, and identification of UI elements across releases. Additionally, "Clickable," "Index," and "Focusable" also show notable variability, indicating changes in interactive behavior and navigation flow within the application.

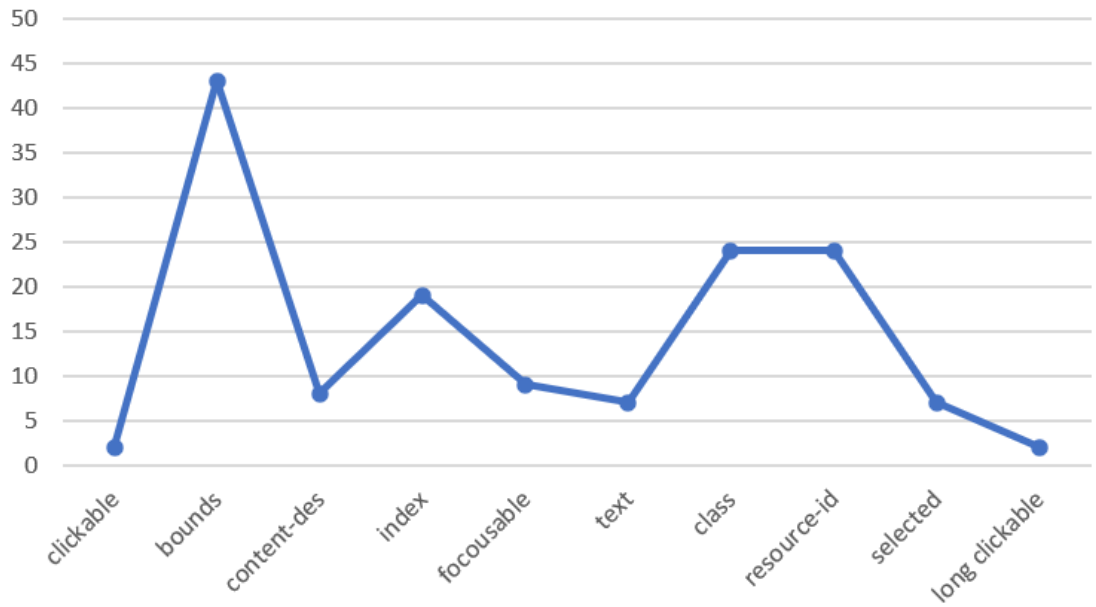

Figura 4.4: cdisplay with all Attributes change

Figure 4.5 shows the variability of widgets in each version of the application, with the number representing the changes or modifications to those widgets. From this data, the versions that saw changes in releases are A, B, C, F, and M. These versions had varying degrees of changes in the number of widgets within the application. Versions D to L and N to V had no changes in the number of widgets, indicating stability or lack of updates in those releases.

Results shown in figure 4.3, 4.4 and 4.5 that the variability in widgets across releases indicates the extent of changes made to the application's components over

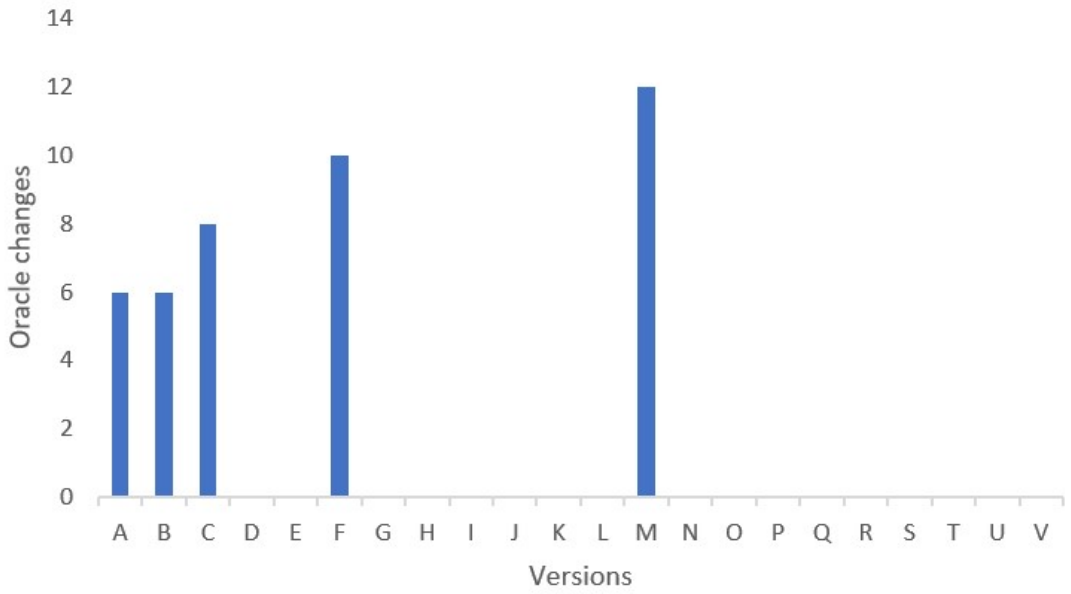

Figura 4.5: cdisplay with all versions and oracles variability

time. Releases with higher variability suggest more significant modifications or updates to the application, while those with lower variability may indicate periods of stability or fewer changes. These releases (A, B, and C) saw incremental changes in the number of widgets, with 6, 6, and 8 widgets changed, respectively. This indicates ongoing development or refinement of features during these early stages. These releases (D to L) experienced no changes in the number of widgets, suggesting potential periods of stability or maintenance where no significant updates were implemented. Release M has 12 widgets changed, indicating a substantial update or overhaul in the application's functionality or user interface. These releases (N to V) also show no changes in the number of widgets, similar to the earlier releases D to L.

. Understanding the variability in widgets across releases helps assess the evolution and progress of the application's development. Higher variability may indicate periods of innovation, feature additions, or bug fixes, while lower variability may signal stability or maintenance phases. Analyzing trends in widget variability can provide insights into the application's development cycle, user feedback incorporation, and overall quality assurance processes.

Results obtained in the figures 4.6, 4.7 and 4.8 we can observe that the bounds attribute in calm application ranked as the most unstable attribute due to its high variability approximately 80 percent of instability in different releases of the application. Despite its frequent presence and variability in assumed values, it's deemed unsuitable as a locator due to instability. Changes in the "bounds"

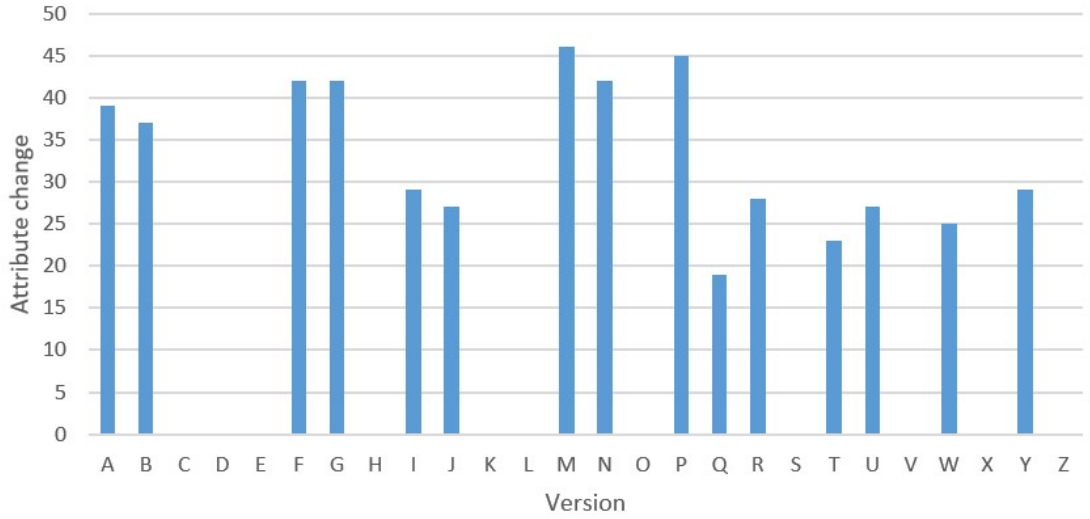

Figura 4.6: calm with all versions and attribute variability

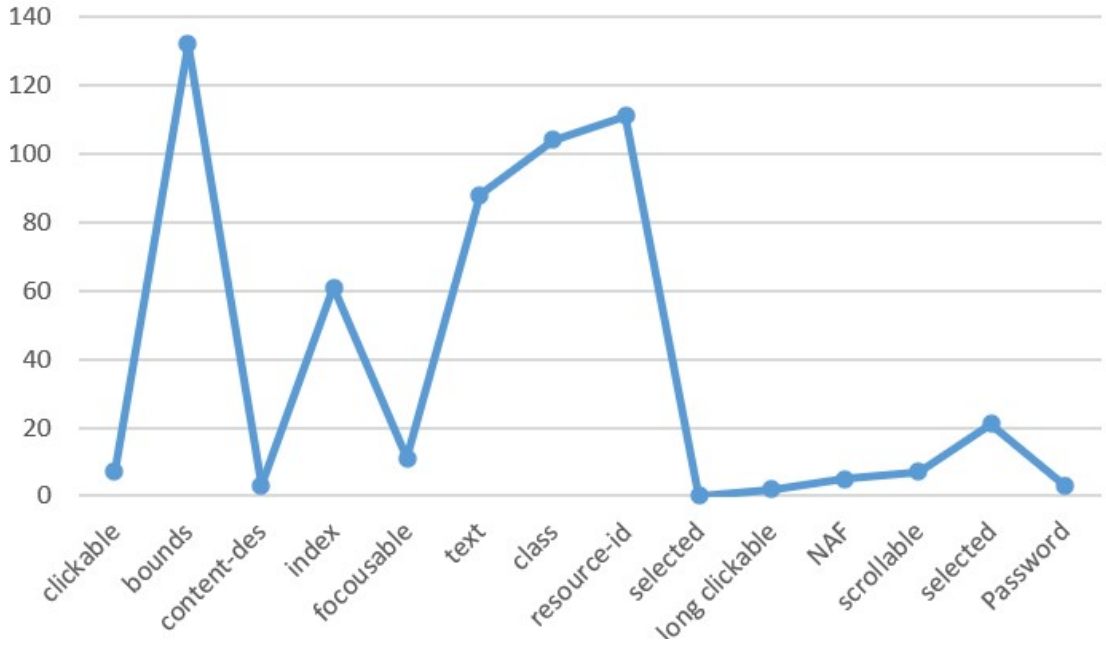

Figura 4.7: calm with all attribute changes in all releases

attribute often coincide with alterations in the "index" attribute, indicating shifts in both visual arrangement and tree structure. The attribute resource-id Ranked second for instability with a variability of 50 percent. Despite Android Developers' recommendation for uniquely identifying resources using this attribute, half of the

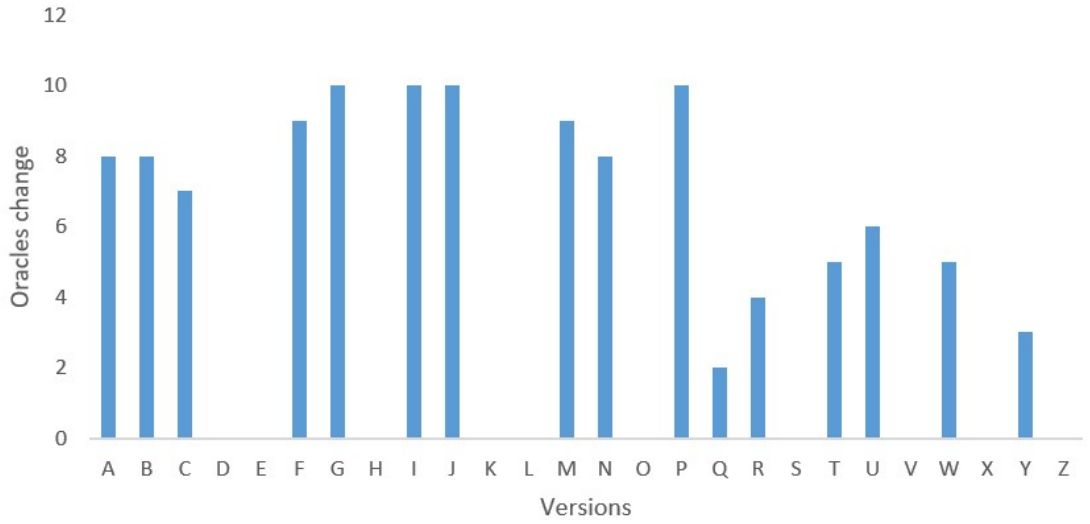

Figura 4.8: calm with all versions and oracle variability

widget changes affect this identifier. Maintaining the "resource-id" updated across the project requires a significant effort, especially during GUI testing outsourcing.

Text and Content Description Attributes (Instability: 23 percent and 17 percent, respectively). These are considered quite unstable attributes with moderate variability. Frequent absence of values adds to their unreliability, contributing to their instability.

For the application youVersion the results shown in figure 4.9 reveal varying degrees of variability across different attributes in different releases. Release such as "A," "B," "C," "F," "G," "I," "J," "M," "N," "P," "Q," "R," "T," "U," "W," and "Y" exhibit substantial changes across versions, indicating significant modifications to these attributes' values.

Figure 4.11 illustrates the impact of attribute changes on UI elements. Attributes like "bounds," "class," and "resource-id" have higher numbers of changes, suggesting their importance in influencing widget alterations across versions.

In Figure 4.10, Certain attributes, such as "bounds," "resource-id," "class," and "content-description," consistently experience significant changes across versions. These attributes play crucial roles in UI element identification, layout, and behavior, making their stability essential for application reliability.

The results obtained from the application Mirror are shown in figure 4.12, which represents the attribute changes in the application during the longitudinal analysis. We can see that Versions A, B, C, D, and E show a relatively lower number of attribute changes, indicating potentially more stable periods in the application's development. Conversely, versions G and J exhibit higher changes, suggesting more

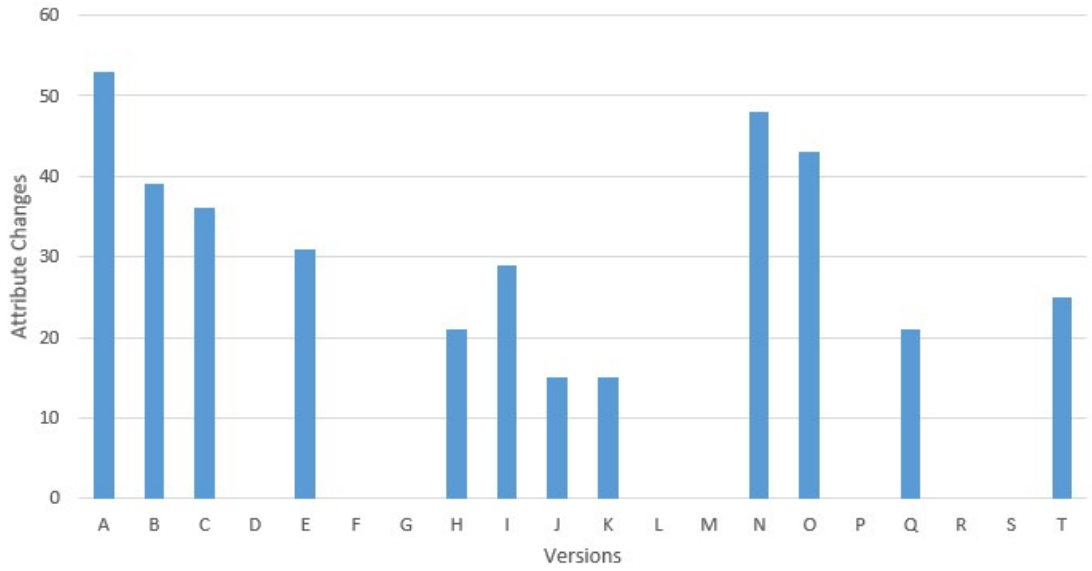

Figura 4.9: youVersion with all versions and attribute variability

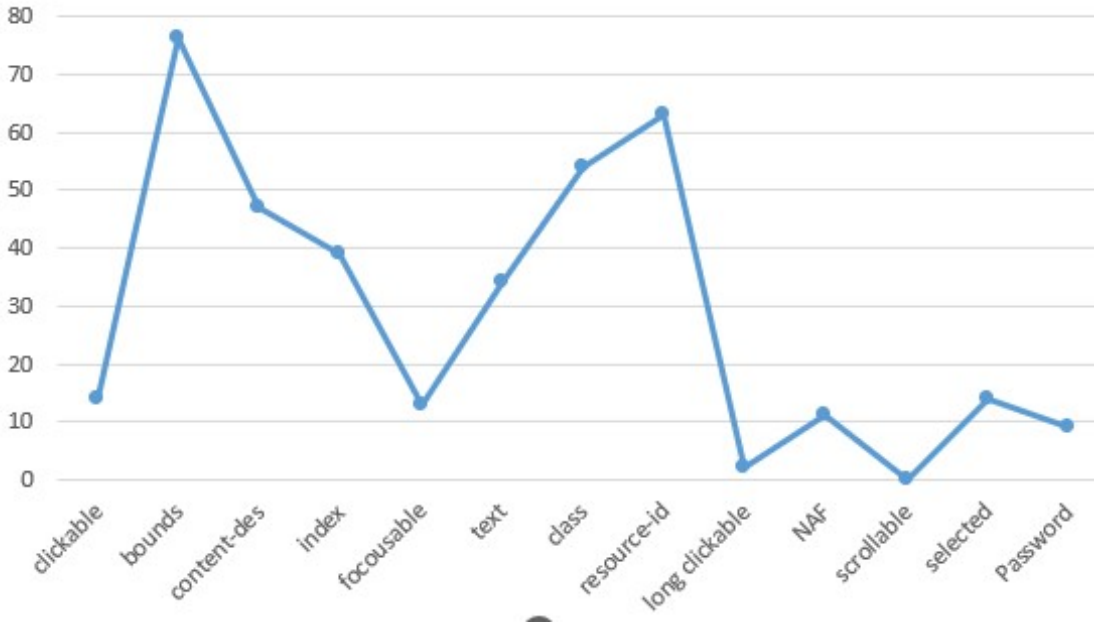

Figura 4.10: youVersion with all attribute changes in all releases

iterative development phases with frequent updates and attribute modifications.

Versions G, J, and K showcase notable spikes in attribute changes, indicating

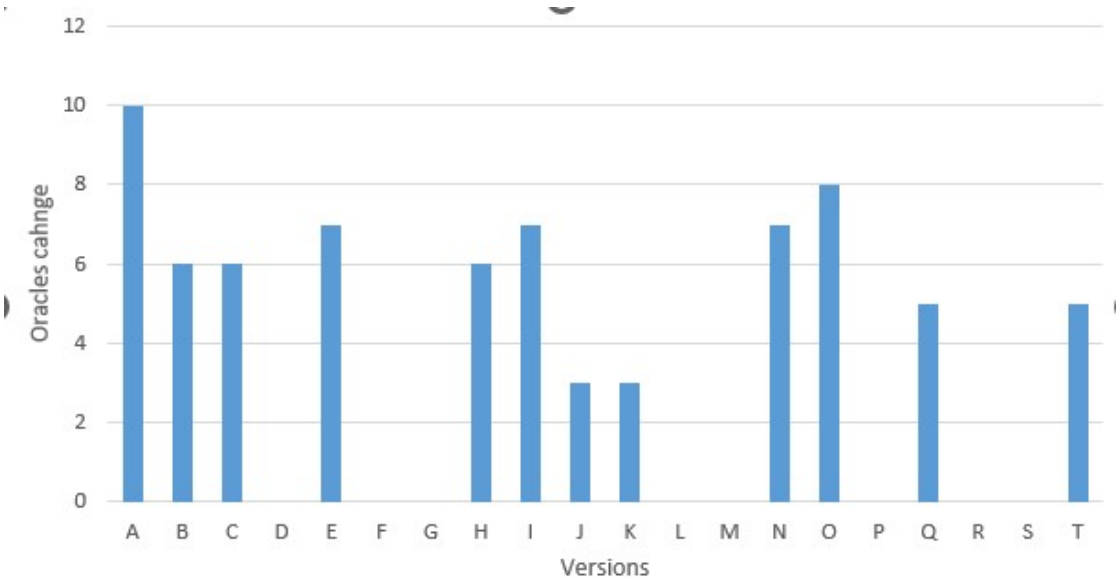

Figura 4.11: youVersion with all versions and oracle variability

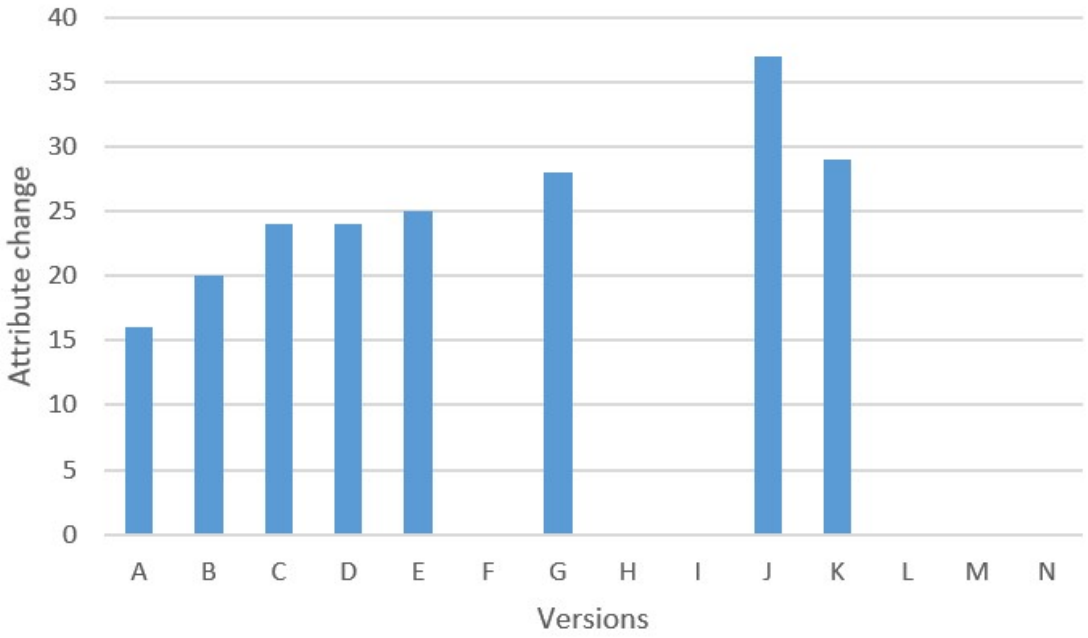

Figura 4.12: Mirror with all versions and attribute variability

potential focus areas for development efforts or areas of significant feature enhancements or refactoring. The substantial increase in attribute changes in version

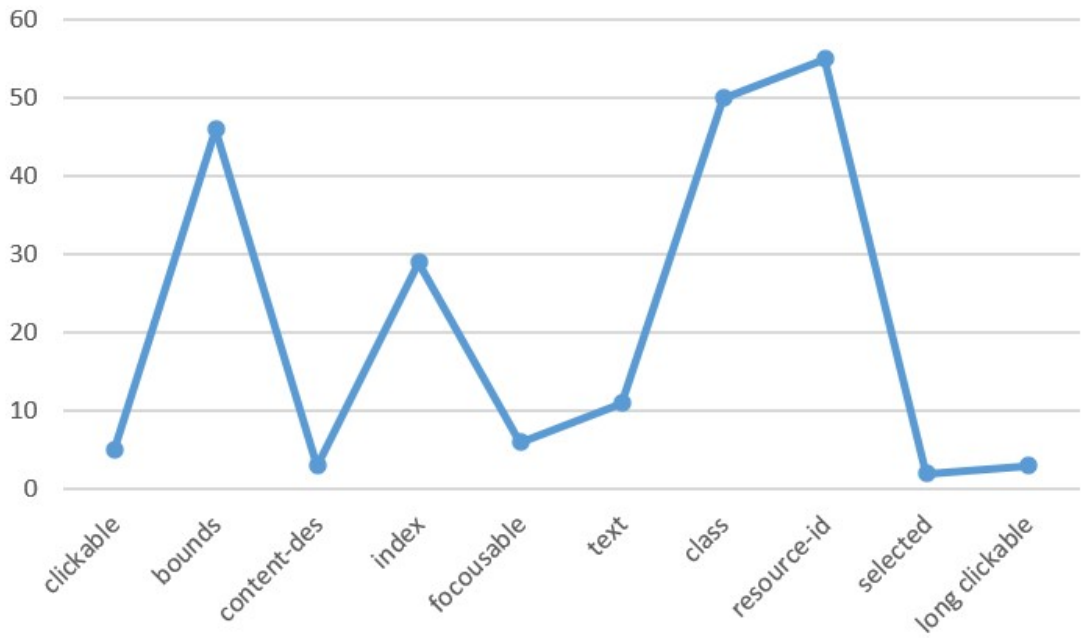

Figura 4.13: Mirror with all attribute changes in all releases

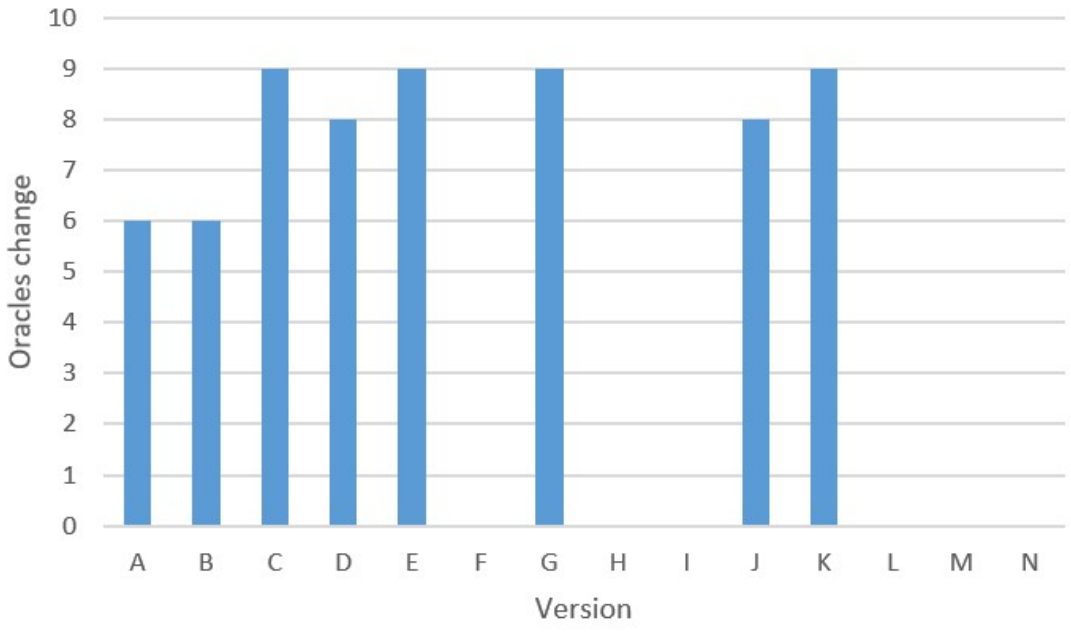

Figura 4.14: Mirror with all versions and oracle variability

J suggests a significant milestone in the application's development, potentially marking the introduction of new features or significant architectural changes.

Figure 4.14 illustrates the impact of attribute changes on UI elements. Attributes like "bounds," "class," and "resource-id" have higher numbers of changes, suggesting their importance in influencing widget alterations across versions.

In Figur 4.13, attributes such as "bounds," "resource-id," "class," and "content-description" regularly undergo substantial modifications between versions. Their stability is critical to the dependability of the application since these properties are vital to the identification, layout, and behavior of UI elements.

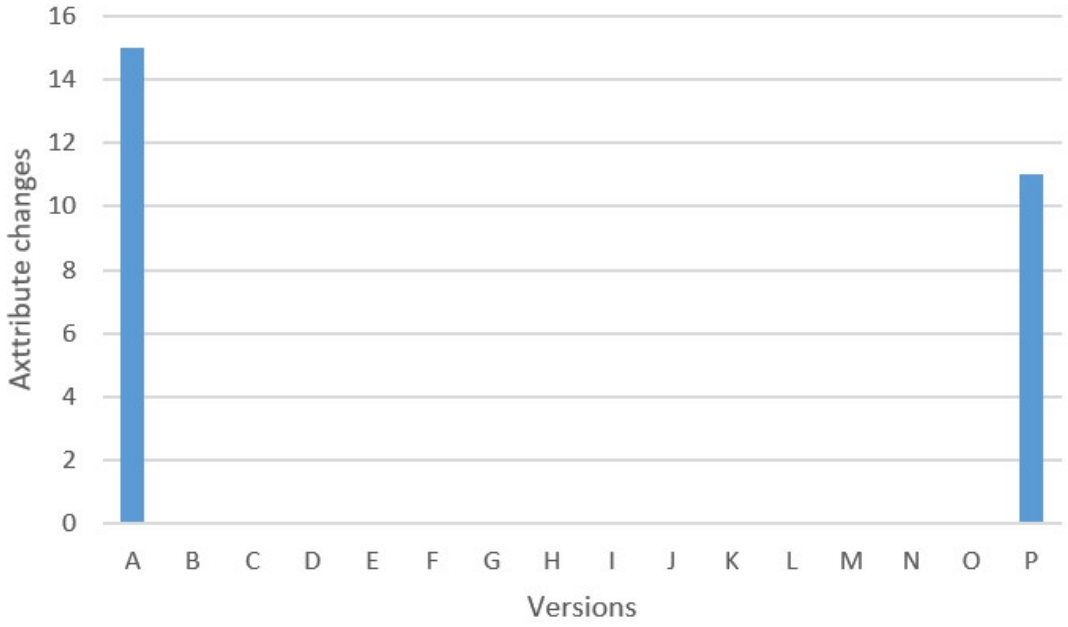

Figura 4.15: Sketchbook with all versions and attribute variability

The results shown in the sketchbook application in figure 4.15 show varying degrees of variability in only two versions of the application, "A" and "P," indicating significant changes to the attributes. Versions B-O show stability during the application development. A total of 25 attributes changed during the consecutive releases of the application, resulting in 15 widget changes. Version "A" and "P" of the application experienced notable attribute changes, primarily focused on the "bounds," "index," and "resource-id" attributes. These changes likely reflect efforts to refine the layout, organization, and identification of UI elements within the application's interface.

Regarding the visual mutations, we found that mutation M11 has the highest frequency of visual change in the GUI, with 67 percent because of the bound

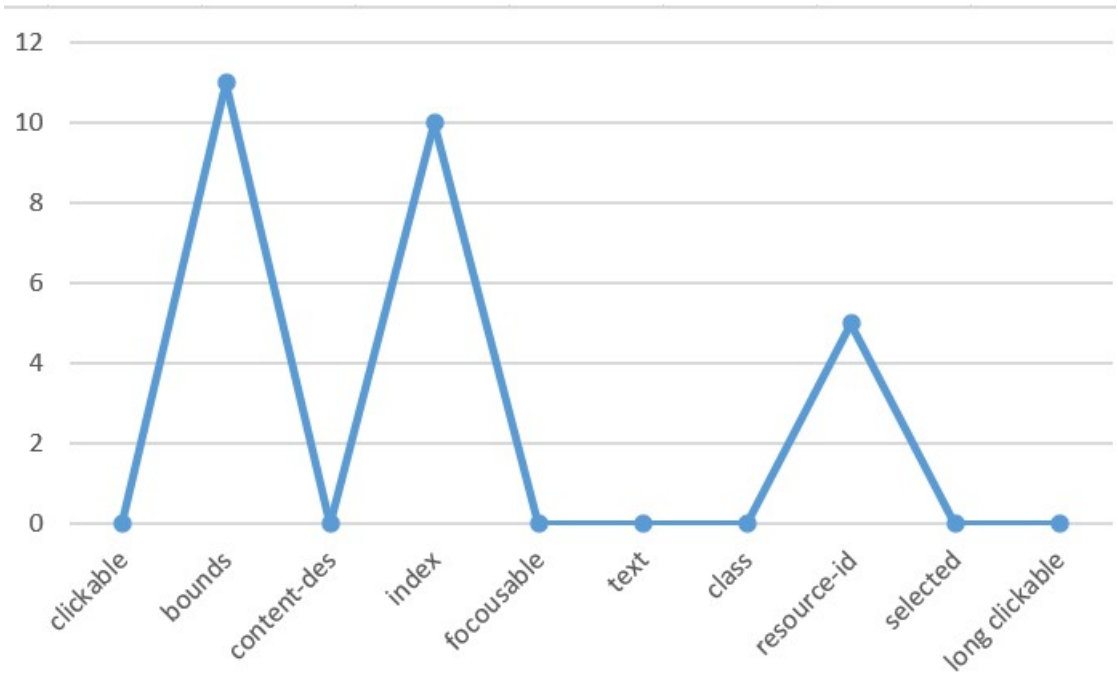

Figura 4.16: Sketchbook with all attribute changes in all releases

attribute. The bound attribute has the highest impact on modifying the widget's location. (eg. The search button location is altered and is placed at the bottom of the GUI). Modifying a widget's graphical appearance (M15, occurring in 50 percent of related widget pairings) is the second visual mutation by frequency. When an application's appearance changed, the widgets were at least slightly rearranged (e.g., the reading and favorite icons in the youVersion app). The absence of visual mutation (M0, occurring in 32 percent of the corresponding widget pairs) was closely followed by the rescaling mutation (M14, happening in 28 percent of corresponding widget pairs), ranked third. Part of this is the need for readjustment after layout changes, but in other situations, a rescale may indicate a shift in the significance of the widget's related function. The mutation about modifications in widget type (M16, which occurred in 17 percent of similar widget pairings) came in last place in frequency. This kind of mutation primarily affects widgets that, as of a particular release, gain new functions that cannot be linked to the previous type. The change from ImageButton to ImageView is an example that occurs repeatedly.

The least common mutation among those considered is a shift in the position to overlap a different widget (M13, occurring in 4 percent of similar widget pairings), indicating that keeping an app's appearance tidy and uncluttered is crucial to its graphical development. One exception is the addition of a Floating Action Button, which makes a specific feature stand out and be easily apparent from the rest of

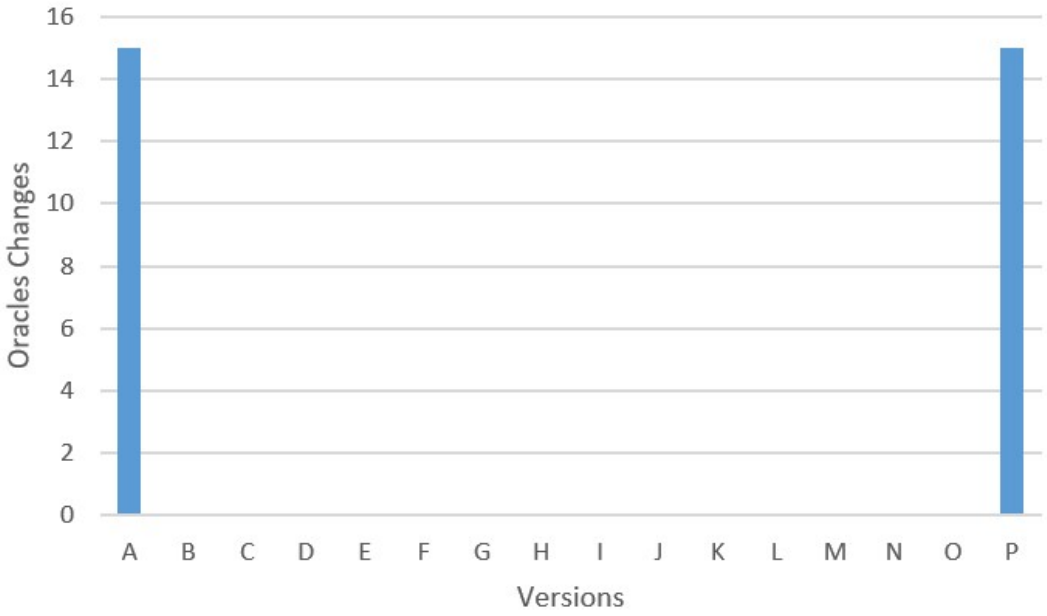

Figura 4.17: Sketchbook with all versions and oracle variability

the GUI.

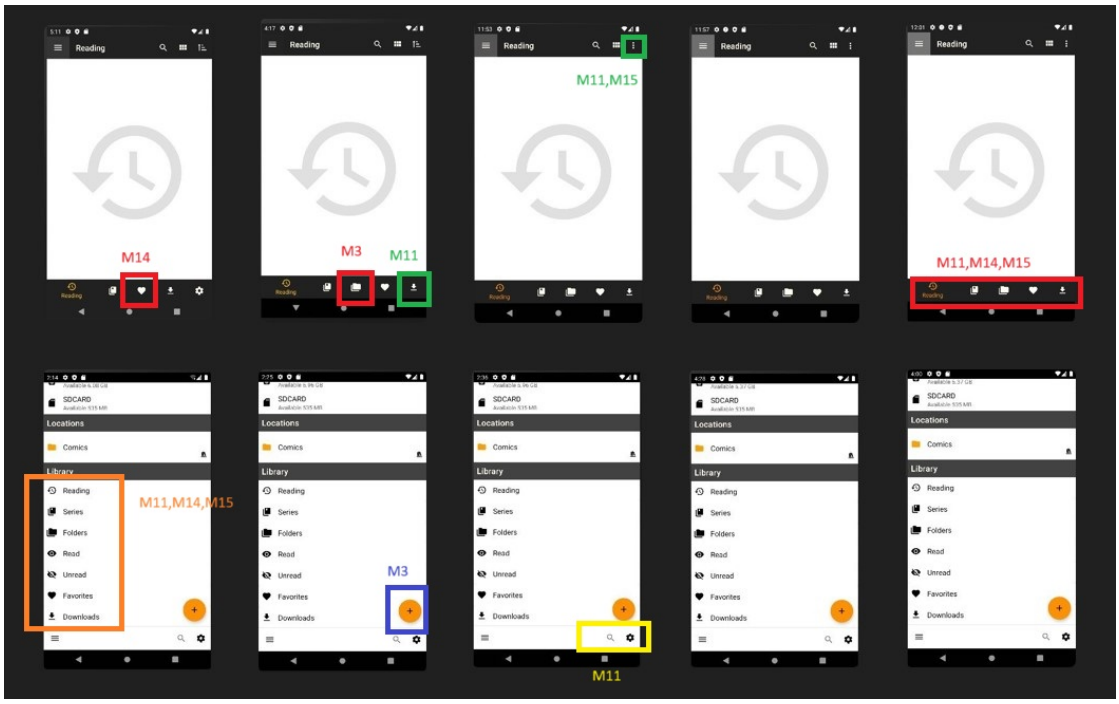

Figura 4.18: Cdisplay all versions mutation

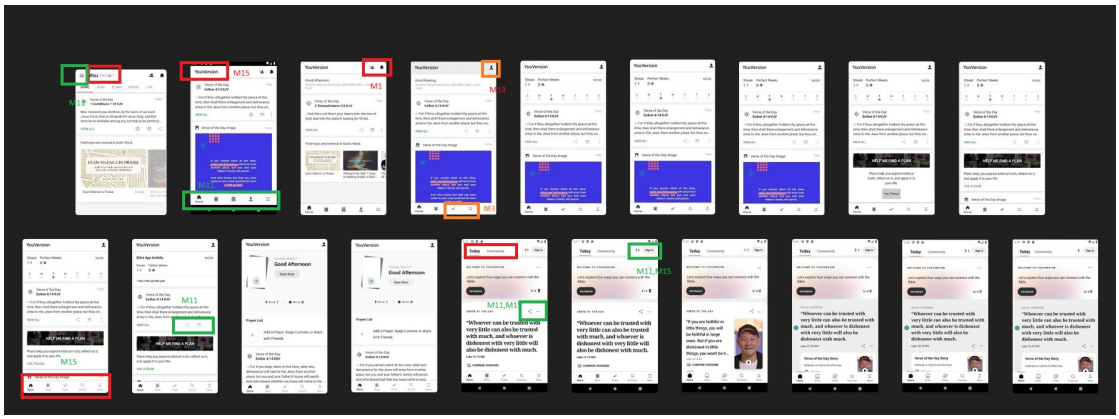

Figura 4.19: youVersion all versions mutation

ATTRIBUTE CHANGES PER VERSION																			
Attribute	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**Total**
Clickable	1	1	2	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	6
Bounds	6	6	9	0	0	10	0	0	0	0	0	0	12	0	0	0	0	0	0	43
Content-dis	1	1	3	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	8
Index	3	3	6	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	19
Focusable	3	3	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	9
Text	2	2	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	7
Class	6	6	6	0	0	6	0	0	0	0	0	0	0	0	0	0	0	0	0	24
Resource-id	6	6	5	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	24
Selected	2	2	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	7
LongClickable	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	2
All Attribute Change	30	30	67	0	0	12	0	0	0	0	0	0	0	0	0	0	0	0	0	179

Tabella 4.1: Attribute Variability

Change in widget Number																				
Widget	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**All versions**
1	0	6	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	10
4	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	7
5	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5
6	0	0	5	0	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	10
7	6	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	7
8	0	0	0	0	0	7	0	0	0	0	0	0	0	0	0	0	0	0	0	7
9	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4
10	4	5	6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15
11	5	4	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	14
12	4	6	3	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	16
13	6	4	4	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	17
14	4	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	7
15	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	2
16	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	4
17	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	4
18	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
19	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
21	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
22	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
24	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
25	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
28	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
31	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
32	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
33	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
34	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
36	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
37	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	1
Total Changes	6	6	8	0	0	10	0	0	0	0	0	0	12	0	0	0	0	0	0	42

Tabella 4.2: Widget Variability

Capitolo 5

Conclusion

In conclusion, this research has provided valuable insights into the significance of understanding attribute variability and its impact on widget changes and the GUI visual appearance in mobile applications. In this research, For a comprehensive analysis, we selected 30 real applications for widget composition. Then, we expanded our experiment to analyze the changes in consecutive application releases for attribute changes and their impact on widget changes and visual changes.

We offered data regarding the attributes' values. Specifically, textual attributes seem to have the highest rate of empty values: only one in four widgets has a non-empty text property, and more than 25 percent lack a resource-id definition. The textual properties and the boundaries attribute show the most flexibility, although their presence is not guaranteed. Starting with a collection of 1277 comparable widgets in all five applications in the longitudinal analysis, we looked at changes that might influence widgets. We found that the five top attributes, Bounds, Resource-id, class, text, and content-description, are the most variable.

Here are the essential findings and contributions of the research.

The study revealed notable variabilities in attributes across consecutive versions, especially in attributes like Bound, resource-id, and content description, which directly influenced the widget changes such as layout, position, and identification of GUI elements.

Attribute variation had a significant visual impact on the GUI, affecting the overall appearance and usability of the application. Variability in attributes such as text and content description influenced the clarity and accessibility of GUI elements.

Using a multilocator proved instrumental in facilitating efficient localization and attribute analysis.

Understanding attribute variations and their impact on widget changes is crucial for improving the user experience of mobile applications. By addressing attribute

inconsistencies and optimizing GUI visual appearance, developers can enhance usability, accessibility, and overall user satisfaction.

Future research should explore advanced multi-locator techniques further to enhance widget localization and attribute analysis in mobile applications. Techniques such as machine learning-based localization algorithms or dynamic attribute tracking mechanisms could offer new insights and efficiencies.

Longitudinal studies tracking attribute evolution over time could provide deeper insights into the dynamics of attribute changes and their long-term impact on GUI visual appearance and user experience.

```
import os os.system("echo 1")
```

Bibliografia

[1] Statista. *Number of smartphone mobile network subscriptions worldwide from 2016 to 2022, with forecasts from 2023 to 2028*. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Accessed on 12-APR-2023 (cit. alle pp. 1, 13).

[2] Shauvik Roy Choudhary, Alessandra Gorla e Alessandro Orso. «Automated Test Input Generation for Android: Are We There Yet? (E)». In: nov. 2015, pp. 429–440. DOI: 10.1109/ASE.2015.89 (cit. a p. 6).

[3] Emil Alégroth. *Visual gui testing: Automating high-level software testing in industrial practice*. Chalmers Tekniska Hogskola (Sweden), 2015 (cit. alle pp. 9–11).

[4] Pingfan Kong, Li Li, Jun Gao, Kui Liu, Tegawendé F. Bissyandé e Jacques Klein. «Automated Testing of Android Apps: A Systematic Literature Review». In: *IEEE Transactions on Reliability* 68.1 (2019), pp. 45–66. DOI: 10.1109/TR.2018.2865733 (cit. alle pp. 11–13).

[5] Tommi Takala, Mika Katara e Julian Harty. «Experiences of System-Level Model-Based GUI Testing of an Android Application». In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*. 2011, pp. 377–386. DOI: 10.1109/ICST.2011.11 (cit. a p. 12).

[6] Chien Hung Liu, Chien Yu Lu, Shan Jen Cheng, Koan Yuh Chang, Yung Chia Hsiao e Weng Ming Chu. «Capture-Replay Testing for Android Applications». In: *2014 International Symposium on Computer, Consumer and Control*. 2014, pp. 1129–1132. DOI: 10.1109/IS3C.2014.293 (cit. a p. 13).

[7] Mario Linares-Vásquez, Kevin Moran e Denys Poshyvanyk. «Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing». In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2017, pp. 399–410. DOI: 10.1109/ICSME.2017.27 (cit. alle pp. 14–16).

[8] Android Fundamentals. *Fundamentals of testing Android apps*. `https://developer.android.com/training/testing/fundamentals`. Accessed on 12-APR-2023 (cit. a p. 16).

[9] Yue Jia e Mark Harman. «An analysis and survey of the development of mutation testing». In: *IEEE transactions on software engineering* 37.5 (2010), pp. 649–678 (cit. a p. 28).

[10] Emil Alégroth, Zebao Gao, Rafael Oliveira e Atif Memon. «Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study». In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–10 (cit. a p. 30).

[11] Tommaso Fulcini, Riccardo Coppola, Marco Torchiano e Luca Ardito. «An analysis of widget layout attributes to support Android GUI-based testing». In: *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE. 2023, pp. 117–125 (cit. a p. 30).