Elevated Testing Depth and Quality for Design Implementations through Screenshot Testing

BACHELOR'S THESIS

Alexandru-Vlad Vamoș

Submitted on 4 November 2024

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Christian Schrödel
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

Erlangen, 4 November 2024

License

Erlangen, 4 November 2024

Abstract

The human factor in manual design verification makes the process time-consuming and error-prone. This thesis develops a strategy for automating design verification in Android applications with the help of screenshot testing. Such an objective requires choosing the right tool, managing the screenshot data and integrating screenshot testing into the CI/CD process. Maximizing the value that screenshot testing adds to the software project requires formulating a test case implementation strategy that defines what parts of the application UI should be verified and when. This research also touches on topics such as image comparison algorithms and Git's problematic handling of binary files. The objective of the strategy developed in this thesis is the reduction of manual testing efforts in regard to design verification and the quick and reliable detection of UI regressions in Android applications.

Contents

List of Figures

List of Tables

Listings

Acronyms

ADB	Android Debug Bridge
ADR	Architecture Decision Record
AI	Artificial Intelligence
API	Application Programming Interface
APK	Android Application Package
CI/CD	Continuous Integration/Continuous Delivery
CV	Computer Vision
DOM	Document Object Model
E2E	End-to-End
HMI	Human-Machine Interface
HTML	Hypertext Markup Language
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
LFS	Large File Storage
ML	Machine Learning
MVVM	Model-View-ViewModel
OEMs	Original Equipment Manufacturers
PbP	Pixel-by-Pixel
SDLC	Software Development Life Cycle
UI	User Interface
VCS	Version Control System

XML Extensible Markup Language

1 Introduction

A considerable amount of the effort that is poured into developing an Automotive Human-Machine Interface (HMI) is aimed at accurately implementing the design specification. As is generally the case for User Interface (UI) implementation, the typical approach is to attempt to recreate the design specification in code as accurately as possible. However, despite the growing amount of value that Original Equipment Manufacturers (OEMs) place on the precise implementation of design specifications and the safety-critical role that UI plays in Automotive HMI, design implementations are mostly only verified through manual tests. This has several significant disadvantages. On the one hand, manual tests depend on human effort, which makes them less time-effective and raises the probability of error (McMillan, 2024). On the other hand, since manual testing requires ever-growing amounts of human effort to be scalable, design implementations are usually not verified in detail, which leads to UI regressions that only get noticed by chance or by the end user.

The objective of this thesis is to develop a concrete strategy for automating design verification in Android applications. This will be achieved through screenshot testing, which Google describes as being an "effective way to verify how your UI looks to users" (‘Compose Preview Screenshot Testing’, 2024). Among other things, such a strategy seeks to reduce the amount of manual testing effort that is invested into design verification and to enable quick and reliable detection of UI regressions.

In order to achieve that, this thesis will tackle challenges such as the comparison of currently available screenshot testing tools, what layers of the application UI they should target in order to maximize efficiency and how screenshot testing can be integrated into Continuous Integration/Continuous Delivery (CI/CD) processes.

Additionally, the thesis is structured in such a way as to serve as a road map for teams that seek to implement their own testing strategy.

2 Literature Review

This chapter sums up relevant literature, gathers insights that are relevant to screenshot testing and explores the documentation strategy that this thesis will use.

2.1 Test Automation

The practice of automated screenshot testing falls under the broader discipline of test automation. Therefore, developing a screenshot testing strategy requires proper knowledge about the goals, benefits, drawbacks and practices of test automation. This section draws on key insights from Baumgartner et al. (2022).

2.1.1 Testware Relevant for Screenshot Testing

Baumgartner et al. (2022) list several relevant testware that the discipline of test automation "[creates] and [deploys]" (p. 7), of which a few are highly relevant for screenshot testing and which this thesis will examine in detail.

The first is the software, which includes the automation tool (Baumgartner et al., 2022). Its "selection ... [is a] complex task" (p. 7), which this thesis aims to complete.

The second testware that is relevant for screenshot testing is the test data, which is "the fuel that drives test execution" (Baumgartner et al., 2022, p. 7). In particular, screenshot testing relies on graphical data. How this data is generated, stored, used and maintained are all matters of concern that this thesis deals with.

A third and last relevant testware that Baumgartner et al. (2022) mention is the test environment. While older Android screenshot testing tools depend on the Android instrumentation and thus require a physical or emulated Android device, some newer tools can run screenshot tests as simple unit tests and require no instrumentation (Soares, 2023a). This thesis will take this novel development into consideration.

2.1.2 Goals of Test Automation

A screenshot testing strategy is also required to adhere to the two central objectives of test automation that Baumgartner et al. (2022) put forth:

- "improve test efficiency and thus reduce the overall cost of testing" (p. 9),
- "[mainain] or [increase] quality" (p. 10).

2.1.3 Drawbacks and Benefits of Test Automation

Developing a screenshot testing strategy further requires awareness of the potential drawbacks of test automation and the effort to minimize their impact. One relevant drawback that Baumgartner et al. (2022) mention is the additional cost and effort required for automation. For example, this encompasses both the effort to implement the testing tool into the software project and the effort to train the team on utilizing that tool. Additionally, the cost of maintenance needs to be kept down, as it has the potential to heavily reduce the added value that automated screenshot testing provides (Baumgartner et al., 2022).

However, being aware of the general benefits of test automation is helpful as well, as a screenshot testing strategy should aim to maximize them. Baumgartner et al. (2022) mention that automation enables building a constantly growing test suite that can be executed on every software release or at any other time for instant feedback. Indeed, manual testing is not as scalable as automated testing, as the former requires a constantly growing amount of time and effort. Another relevant benefit that they mention is that test automation enables the manual tester to shift their focus to tasks that are harder or impossible to automate and therefore yield more value, especially "explorative testing or the targeted use of various dynamic manual testing procedures" (Baumgartner et al., 2022, p. 11).

2.2 Snapshot Testing and Screenshot Testing

Fujita et al. (2023) define snapshot testing as "a type of output comparison testing technique that asserts whether the outputs by the current state of the product remain unchanged" (p. 335). They also make the important observation that the objective of snapshot tests "differs from those of unit and functional tests that define the correct behavior of the products" (p. 335). In other words, instead of verifying correct implementation, snapshot tests verify whether the implementation "matches a certain golden standard" (Cruz et al., 2023, p. 5). The snapshots can come in several forms: JavaScript Object Notation (JSON) objects, Document Object Model (DOM) trees and images (Yerburgh, 2018). Screenshot testing, then, can be described as a category of snapshot testing that is concerned only with image snapshots.

At the time of writing, there is a pressing lack of academic work on the topic of screenshot testing. However, the slightly more common academic papers that explore snapshot testing can provide important insights for the development of a screenshot testing strategy.

Cruz et al. (2023) have performed research on the adoption of snapshot testing "through a grey literature review" (p. 1). For this, they have selected 50 relevant documents and drawn conclusions from them about the snapshot testing practice.

2.2.1 Benefits and Drawbacks of Snapshot Testing

Out of the 50 documents, 54% mention that snapshot testing is "easy to implement" (Cruz et al., 2023, p. 4). The second-most mentioned benefit is that it "prevents regression" (p. 4). However, according to the authors, only 12% of sources describe snapshot testing as an alternative to End-to-End (E2E) UI tests, which, especially in Android, are known to be "fragile" and "flaky" (Coppola et al., 2023, p. 120) and resource intensive, as they require a running Android device.

On the other side, Cruz et al. (2023) show that the most commonly mentioned drawback is the fragility of snapshot tests. What is meant by this is that the snapshot tests easily break, even with the smallest change in the snapshot. The developer then has to decide whether the snapshot requires an update or whether the test has detected a bug.

However, two of the lesser mentioned drawbacks that Cruz et al. (2023) list are especially relevant for screenshot testing in particular. The first one is the large size of snapshots. As mentioned above, screenshot tests use images in favor of text-based snapshots like JSON objects. This could aggravate the size problem even further, as images usually require larger amounts of storage space than text files. The second relevant drawback is the flaky nature of snapshot tests (Cruz et al., 2023). Furthermore, the flakiness of Android test execution environments (Coppola et al., 2023) makes this problem even worse.

There are also insights from Cruz et al. (2023) that can hardly be applied to screenshot testing in particular. For example, as a remedy for snapshot test fragility, they recommend that "snapshot results should be treated as part of the code" (p. 4) and that they should be "commited and changed the same way as the source code" (p. 4–5). While this is suitable in the case of text-based snapshots like DOM trees, it becomes difficult when dealing with screenshots. Not only can an image not be changed in the same way as a text-based file, but, as ‘Git Large File Storage (LFS)’ (n.d.) shows, Git is inefficient when it comes to versioning binary files. This is yet another challenge that will be tackled in this thesis.

2.3 arc42 and Architecture Decison Records

Accurately documenting the research process and research findings is paramount to the objective of this thesis. This section explores how the arc42 and Architecture Decision Record (ADR) templates can be adapted to fit the style and purpose of this academic work.

"arc42 offers a clear, simple and effective structure to document and communicate your software system" ('arc42 documentation', n.d.). The official documentation states that the arc42 template in its entirety answers the following two questions:

- "*What* should yout document/communicate about your architecture?"
- "*How* should you document/communicate?"

In total, arc42 contains 12 comprehensive sections. However, with efficiency being one of its core principles, the template has been created with the intention of being easily adaptable to specific project needs ('Why arc42?', n.d.). Thus, this thesis will adapt and utilize the components of the arc42 approach that are relevant to its objectives, such as:

- *Introduction and Goals*, with the three proposed subsections *Requirements Overview*, *Quality Goals* and *Stakeholder*,
- *Building Block View*,
- *Architectural Decisions.*

Most importantly, the concept of *Architectural Decisions* will be used to communicate the important conclusions of this research and the resulting decisions about the screenshot testing strategy developed in this thesis. These will be structured according to a simplified version of the ADR template presented by Nygard (2011). As such, each important finding or decision will be followed by an ADR that consists of two parts: *Context* and *Decision.*

3 Requirements

Requirements are crucial in any software engineering project. They provide direction for the implementation of the project and a standard by which the final product can be evaluated. Thus, it is important to carefully consider at an early stage of development what the requirements should include. This chapter contains a comprehensive list of requirements for the screenshot testing strategy developed in this thesis, structured across three key perspectives provided by the arc42 template.

3.1 Functional Requirements

The first perspective is the functional one. The following list gathers the central functional requirements of the screenshot testing strategy:

- The engineer (e.g. manual tester or software developer) will be able to perform automated screenshot regression testing on the software project.
- The engineer will have at his disposal a high-level Application Programming Interface (API) with optional low-level control that can be used to implement screenshot tests.
- The engineer will be able to create and store reference screenshots for any screenshot test case.
- The engineer will be able to run the implemented screenshot tests in a local environment (i.e. on the engineer's local machine).
- The automated screenshot testing system can be seamlessly integrated into the CI/CD pipeline and thus into the larger test suite of the software project.
- The screenshot testing system will output individual test results and test reports that will offer both simple insights (e.g. passed or failed) and complex insights (e.g. a graphical difference representation) about the state of the test object.

3.2 Quality Goals

The second perspective is the quality one. The following table includes the required quality goals of the strategy, partly selected from *ISO/IEC 25010* (2023):

Quality Goal	Description
Functional Suitability	Fulfill the "stated and implied needs" (*ISO/IEC 25010*, 2023) and functional requirements (see section 3.1).
Efficiency and Performance	Minimize the amount of required resources (e.g. time, implementation effort and hardware resources) and maximize the added value per invested effort, especially in comparison to manual design verification.
Reliability	Provide accurate and reliable outputs that aid the software development process.
Maintainability	Allow for easy maintenance of the screenshot test implementations, data and architecture.
Scalability	Accommodate a constantly growing test object and test suite.

Table 3.1: Quality goals

3.3 Stakeholders

The third perspective is that of the stakeholders. The following table lists each stakeholder and their expectations for the screenshot testing strategy:

Role	Expectations
Manual Tester	Automated screenshot tests are easy to implement and to run. There is little required training. There is an obvious reduction in time required for UI validation.
Software Developer	Includes the manual tester's expectations and goals. Additionally, the test delivers a precise test result that the developer can make use of in the debugging process.
Team Lead	The effort required to implement automated screenshot tests is minimized. Strain is taken off the manual tester; they can shift their focus to exploratory and usability testing.
Product Owner	The automated screenshot testing results provide an overview of the UI health. There are less inconsistencies between the design specification and the actual design implementation. Visual bugs and regressions are detected sooner.
DevOps	The source code repositories do not get polluted with binary image files. Automated screenshot testing is tightly integrated into the CI/CD pipeline of the software project. The testing architecture is highly scalable.
End User	The application's UI has no visual regressions, which leads to an enhanced user experience and increased user satisfaction.

Table 3.2: Stakeholder expectations

4 Architecture

The current chapter explains the architecture of a screenshot test and how screenshot testing relates to the architecture of Android applications, but also how this fits into the larger picture of software development processes.

4.1 Screenshot Testing

When it comes to Android tests, the most wide-spread are those of the functional type. According to ‘Functional Testing: A Detailed Guide’ (2024), functional "tests [ensure] that the system behaves according to the specified functional requirements and meets the intended business needs." Conventional unit, integration and E2E UI tests are fitting examples for this category. For instance, in a contacts application, one unit test might check that the contact sorting function works as intended, while an E2E UI test might verify that the correct contact details are displayed after clicking on a particular contact. On the other hand, snapshot/screenshot testing is generally thought to belong to the opposite, non-functional category of tests (Fujita et al., 2023; Soares, 2023b; ‘What is Non-Functional Testing?’, 2024). This distinction is illustrated in figure 4.1, which places the practice of screenshot testing in the broader context of the Software Development Life Cycle (SDLC).

4.2 Android Application Architecture

Another distinction between more common test types and screenshot testing is the component of the application architecture targeted by them. The Model-View-ViewModel (MVVM) architecture pattern that is common among modern Android applications is depicted in figure 4.2. While conventional unit tests cover any of the three layers and E2E UI tests cover them all at once, for example, screenshot testing is only concerned with the View layer, which holds the UI source code.

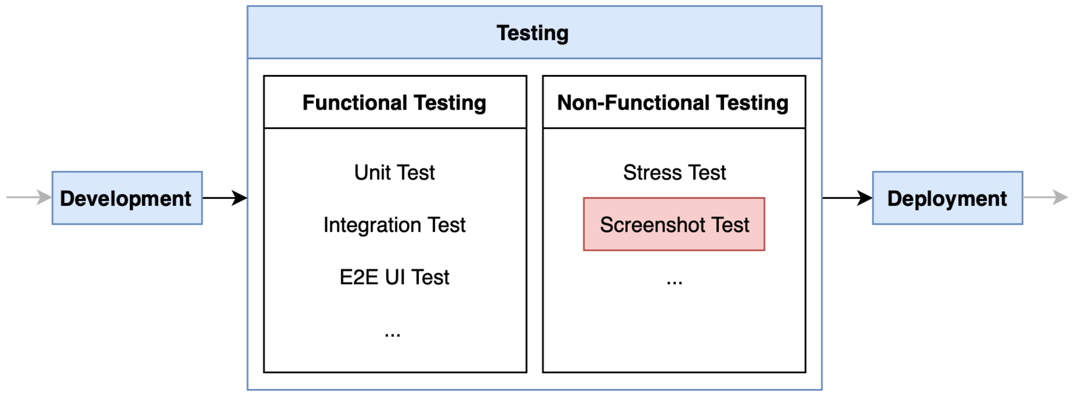


Figure 4.1: Section of the SDLC

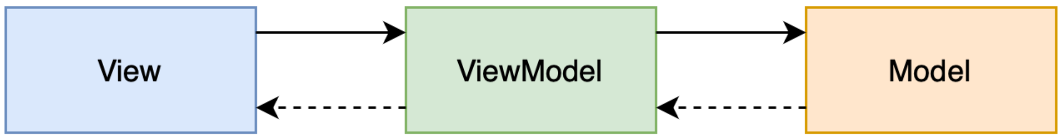


Figure 4.2: Simplified representation of the MVVM pattern

4.3 Screenshot Test Structure

Figure 4.3 shows the basic structure of an Android screenshot test. The inputs are the UI source code and the reference screenshots. The component called *UI Renderer* represents the component of the screenshot testing tool that creates a graphical representation of the UI source code, e.g. the Android graphics pipeline or the Layoutlib library. The screenshot test then uses a comparison algorithm to compare the generated UI representation delivered by the UI Renderer against the reference screenshot and finally delivers the test result.

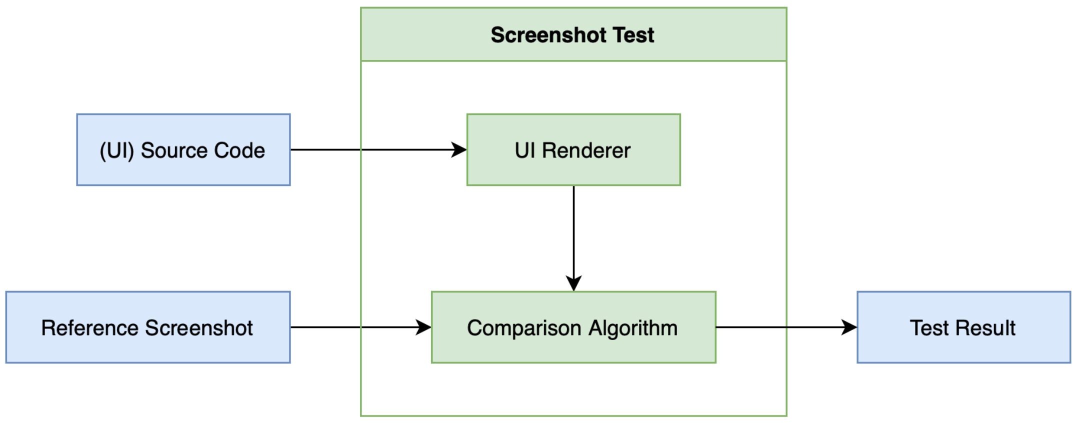


Figure 4.3: Basic representation of the screenshot test structure

5 Design and Implementation

Using the requirements and the architecture considerations presented in chapters 3 and 4 respectively, the objective of this chapter is to develop a concrete screenshot testing strategy. In addition, the topics that the following sections tackle are ordered in such a way as to allow this thesis to serve as a road map for teams looking to develop their own screenshot testing strategy.

5.1 Screenshot Comparison Algorithms

A central aspect of any screenshot testing tool is the comparison algorithm that it uses to decide whether the actual screenshot, which represents the current UI implementation, differs from the reference screenshot. At the time writing, Android screenshot testing tools fall into one of two categories:

- open-source tools that use conventional Pixel-by-Pixel (PbP) comparison algorithms,
- closed-source tools that use their own proprietary algorithms advertised as being augmented by Computer Vision (CV), Machine Learning (ML) and/or Artificial Intelligence (AI) ('Revolutionize UI testing with Visual AI', n.d.; 'Visual testing powered by computer vision', n.d.).

An informed decision between the two options requires a closer look at their algorithms and how they behave.

5.1.1 Open-Source Algorithms

The first category, which makes use of conventional open-source PbP comparison algorithms, contains screenshot testing libraries such as Cashapp's Paparazzi and Takahiro Menju's Roborazzi. The underlying principle, which can be observed in 'PixelPerfect.kt [Source Code]' (2024) and 'ImageComparator.kt [Source Code]' (2024), is simple: the algorithm compares each pixel of the actual screenshot with its coordinate-counterpart from the reference screenshot. Optionally, the comparison is influenced by one or more tolerance parameters, which allows the developer to fine-tune each test to better achieve the desired behavior. For instance, Roborazzi uses Dropbox's open-source "multiplatform image diffing library" ('differ GitHub', 2024) called *differ*, whose API accepts three parameters that alter the library's behaviour:

- One parameter represents the maximum allowed Euclidean distance between the values of two pixels that are being compared. This is used to decide whether the two pixels are considered to be equal or not.
- Two additional parameters increase the tolerance of the algorithm towards small differences caused, for example, by anti-aliasing.

In addition, Roborazzi accepts another parameter that defines the maximum percentage of pixels that are allowed to change before the test fails.

5.1.2 Enhanced Closed-Source Algorithms

The second category of screenshot testing tools, those that are advertised as being augmented by CV, ML and/or AI ('Revolutionize UI testing with Visual AI', n.d.; 'Visual testing powered by computer vision', n.d.), use algorithms that are difficult to analyze closely due to their closed-source nature. And while these tools also offer parameters that alter their behaviour, it often remains unclear what exact influence they have. On top of this, due to the ambiguous language that is used to describe such tools, it is also unclear to what degree they make use of the more advanced software technologies mentioned above.

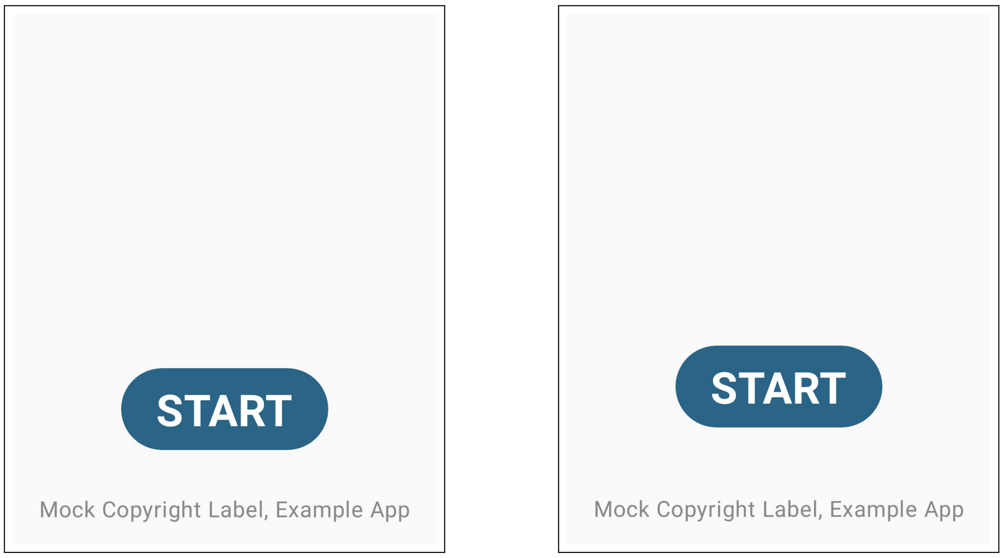
5.1.3 Side-by-Side Comparison

For this reason, this section provides a simple side-by-side comparison of the two tool categories mentioned above. Roborazzi is selected to represent the open-source category, while Aplitools Eyes from Aplitools and App Percy from Browserstack represent the closed-source category. Since a comprehensive comparison would require an amount of effort that exceeds the scope of this thesis, three representative and real-world UI regression scenarios are chosen for this task:

- The first regression scenario targets the common occurrence of a slightly shifted interactive UI element.
- The second regression scenario deals with the accidental change of the color of a UI element.
- The third and last regression scenario targets the slight shift of a text UI element.

These three regression scenarios have been tested on the welcome screen of a mock Android application created solely for the purpose of this research. The complete welcome screen can be seen in appendix C.

In the first scenario, the bottom padding of a UI button has been increased. Figures 5.1a and 5.1b show the reference UI state and the faulty UI state respectively. Figures 5.2a, 5.2b and 5.2c show the difference images provided by each tool after the UI regression has been introduced.

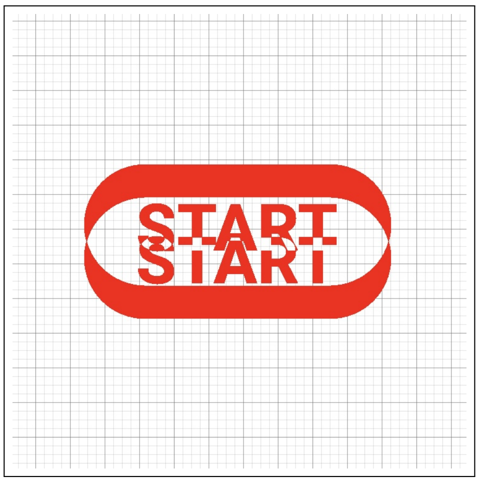

START
Mock Copyright Label, Example App
START
Mock Copyright Label, Example App

(a) Reference screenshot **(b)** Actual screenshot

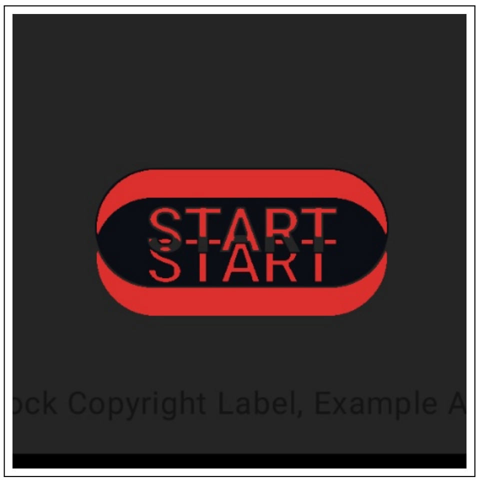
Figure 5.1: Reference and actual screenshots from first regression scenario

With Roborazzi, all pixels that have changed are marked red in the difference image. While Roborazzi's comparison behaviour can be slightly altered, as mentioned above, the general characteristics of the provided difference image are representative for any choice of tolerance parameters. In this scenario, App Percy's difference image strongly resembles that of Roborazzi, as can be seen in figure 5.2b. While its proprietary algorithm has a higher tolerance for differences than the default Roborazzi setup (the area marked with red pixels is smaller than the area taken up by pixels with differing values), figures 5.2a and 5.2b only have minimal differences and it is not evident that App Percy's behaviour is superior to that of Roborazzi. On the other hand, Aplitools Eyes marks the area that contains changed pixels by covering it with one single rectangle shape and thus groups all differences that are in the vicinity of the button together. While this does make for a simpler and cleaner graphical representation, it also makes it more difficult to point out the exact pixels that have been changed.

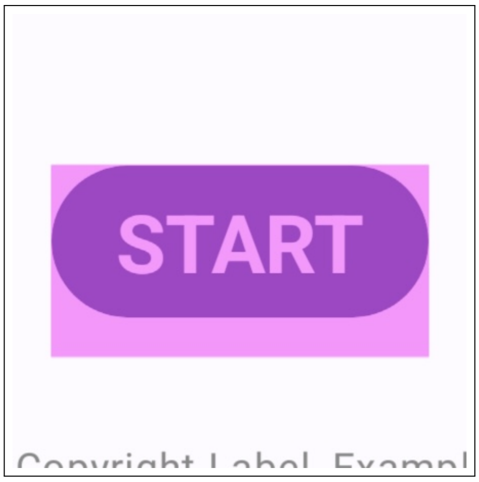
In the second and third scenarios, the button color is changed and the text is shifted, as seen in figures 5.3 and 5.4 respectively. The three tools show the



(a) Roborazzi



(b) App Percy

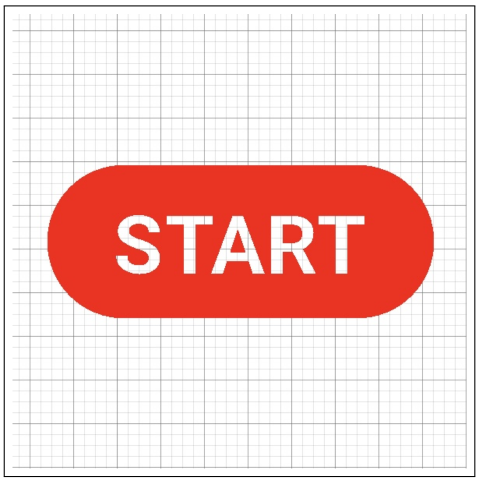


(c) Aplitools Eyes

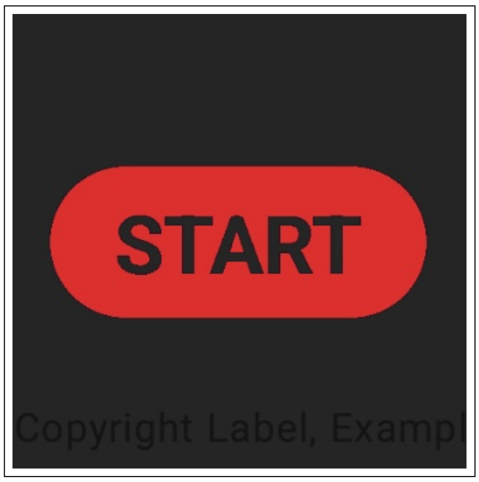
Figure 5.2: Difference images from first scenario (shifted button)

same behaviour as in the first scenario: Roborazzi and App Percy deliver similar difference images that mark all or nearly all different pixels, while Aplitools Eyes highlights the rectangular areas that show pixel differences.

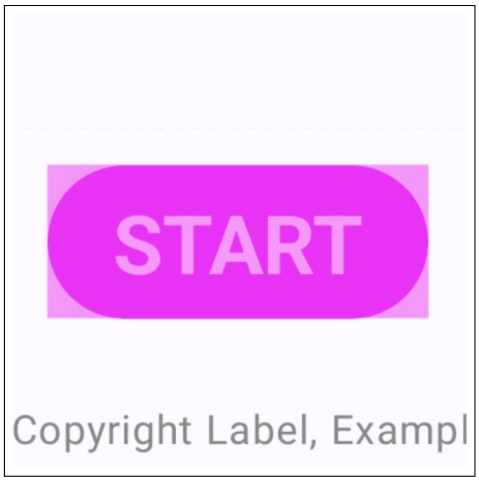
Both categories of screenshot testing tools have further advantages and disadvantages. For example, closed-source tools like App Percy and Aplitools Eyes provide advanced online platforms for analyzing test runs and test results. However, if hosting such tools locally is not supported, they are out of reach for projects where design implementations are confidential. In addition, at the time of writing, the two closed-source tools do not leverage the more advanced technologies to offer contextual descriptions about the UI regressions, neither do they offer possible solutions. On the other hand, open-source tools like Roborazzi



(a) Roborazzi

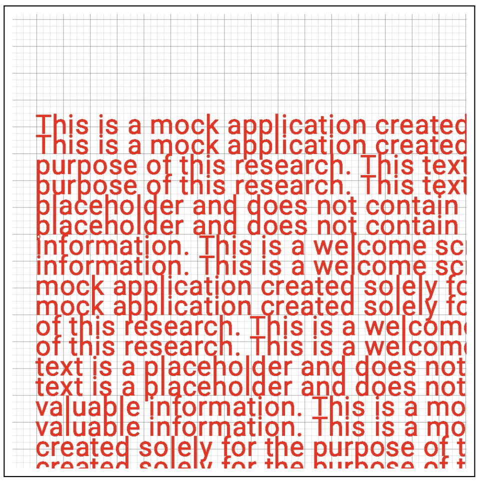


(b) App Percy



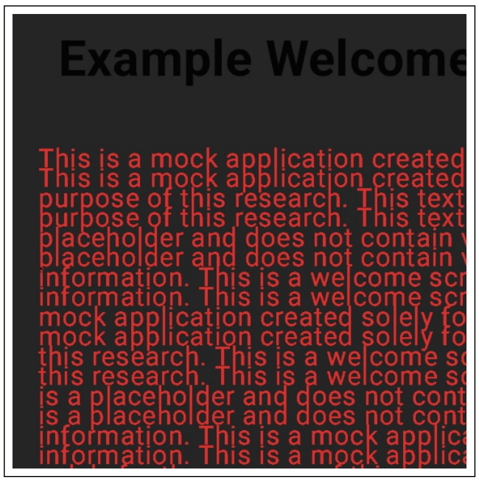
(c) Aplitools Eyes

Figure 5.3: Difference images from second scenario (changed button color)



This is a mock application created
This is a mock application created
purpose of this research. This text
purpose of this research. This text
placeholder and does not contain
placeholder and does not contain
information. This is a welcome sc
information. This is a welcome sc
mock application created solely fo
mock application created solely fo
of this research. This is a welcom
of this research. This is a welcom
text is a placeholder and does not
text is a placeholder and does not
valuable information. This is a mo
valuable information. This is a mo
created solely for the purpose of t

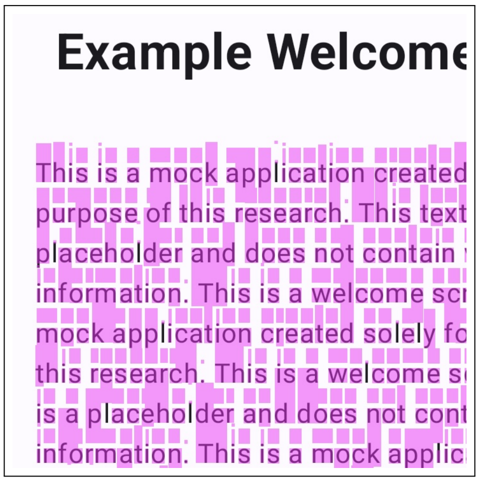
(a) Roborazzi



Example Welcome

This is a mock application created
This is a mock application created
purpose of this research. This text
purpose of this research. This text
placeholder and does not contain
placeholder and does not contain
information. This is a welcome scr
information. This is a welcome scr
mock application created solely fo
mock application created solely fo
this research. This is a welcome sc
this research. This is a welcome sc
is a placeholder and does not cont
is a placeholder and does not cont
information. This is a mock applica
information. This is a mock applica

(b) App Percy



Example Welcome

This is a mock application created
purpose of this research. This text
placeholder and does not contain
information. This is a welcome sc
mock application created solely fo
this research. This is a welcome s
is a placeholder and does not cont
information. This is a mock applic

(c) Aplitools Eyes

Figure 5.4: Difference images from third scenario (shifted text element)

benefit from community engagement and contributions. For example, according to GitHub, Roborazzi has 16 contributors in total ('Roborazzi GitHub', 2024). Additionally, they can be forked and expanded in order to suit the needs of any particular project.

5.1.4 Conclusion

The three examples shown above indicate that there are few significant differences between the visual result representations of the two screenshot testing tool categories. While further and more ample research has to be carried out in order to definitively conclude which one offers superior functionality, in light of the advantages of open-source tools mentioned above, this research currently does not find compelling reasons to recommend closed-source tools that use allegedly AI-enhanced comparison algorithms over open-source screenshot testing tools that use conventional PbP comparison algorithms, such as Roborazzi.

The following ADR sums up the decision of this section:

> **Context**
> At the time of writing, one can choose between open-source tools with conventional PbP comparison and closed-source tools that use proprietary, enhanced ('Revolutionize UI testing with Visual AI', n.d.; 'Visual testing powered by computer vision', n.d.) comparison algorithms.
>
> **Decision**
> The screenshot testing strategy developed in this thesis recommends choosing an open-source screenshot testing tool that uses a conventional PbP comparison algorithm.

5.2 Screenshot Testing Tool

Based on the insights from section 5.1 of this thesis, the current section aims to tackle the challenge of choosing a fitting screenshot testing tool and to offer an overview of its setup and usage.

5.2.1 Tool Selection

Choosing the right screenshot testing tool is a crucial step that requires careful consideration. The first reason for this are the additional costs required for its setup and usage (Baumgartner et al., 2022). Furthermore, making the wrong choice raises the costs even further, as the process has to be repeated.

For this reason, this subsection offers a comparison of three open-source Android screenshot testing tools. The selection of the tools has been made primarily based on the following conditions:

- The tool has been widely-adopted in the Android community. In this research, this means that the respective public repository is required to have at least 500 stars.
- At the time of writing, the tool is being actively developed/maintained, i.e. has had at least one new release in the last 12 months.

The selected screenshot testing tools are:

- Pedro Gómez's Shot (released 2017),
- Cashapp's Paparazzi (released 2021), which disrupted the Android screenshot testing scene by "using an innovative approach that allows screenshot tests to execute as a plain-old JVM unit test" (Soares, 2023a),
- Takahiro Menju's Roborazzi (released 2023), which further "enhances Paparazzi's capabilities" ('Roborazzi documentation', 2024).

Side-by-side Comparison

The comparison that this thesis offers draws on insights from the thorough comparison of Shot, Paparazzi and Dropshots conducted by Soares (2023a).

The first comparison criterion is an aggregate of two important characteristics:

- ease of setup and learning effort, which are concerned with the upfront effort of integrating a tool into the project,
- ease of use, which represents the recurring effort that using the tool requires, e.g. time required to write a screenshot test.

When it comes to the ease of setup, there are no meaningful differences between the three tools, as Shot, Paparazzi and Roborazzi can each be integrated into the project through Gradle plugins and/or library dependencies ('Paparazzi documentation', 2024; 'Roborazzi documentation', 2024; 'Shot README', 2023). For instance, Paparazzi only requires adding one line of code to the build.gradle.kts file, which integrates its Gradle plugin, as can be seen in listing 5.1. According to the source code, the plugin then handles all further dependencies in the background ('PaparazziPlugin.kt [Source Code]', 2024).

```
plugins {
    id("app.cash.paparazzi") version "<version>"
}
```

Listing 5.1: Paparazzi integration (in build.gradle.kts)

There are also no significant differences among these three tools with regard to learning effort and ease of use. For example, compared to popular Android UI testing libraries such as Espresso and UI Automator, which have extensive APIs (see links in appendix B), the Shot, Paparazzi and Roborazzi APIs are all relatively simple (see links to their documentation in appendix A). Listing 5.2 shows a simple Roborazzi screenshot test. Beside taking a screenshot of MainActivity's root view, the test is configured to store the reference screenshot under the outputs directory.

```kotlin
@RunWith(androidx.test.ext.junit.runners.AndroidJUnit4::class)
@GraphicsMode(GraphicsMode.Mode.NATIVE)
class TrivialScreenshotTest {
    @get:Rule
    val composeTestRule = createAndroidComposeRule<MainActivity>()

    @get:Rule
    val roborazziRule = RoborazziRule(
        RoborazziRule.Options(
            outputDirectoryPath = "outputs"
        )
    )

    @Test
    fun capture() {
        composeTestRule.onRoot().captureRoboImage()
    }
}
```

Listing 5.2: Simple Roborazzi screenshot test example

In this example, the Android test rule RoborazziRule receives an object of type Options that specifies the name of the desired output directory (outputs). Taking the screenshot only requires calling the captureRoboImage() method on the object provided by composeTestRule.onRoot(), which is standard Android API.

The second comparison criterion is the required test environment, which Soares (2023a) also mentions. In other words, the deciding question is whether the screenshot tests require an instrumented environment, i.e. a physical or emulated Android device. Similar to conventional E2E UI tests, Shot depends on the Android instrumentation to run its tests. This has several important disadvantages such as the required start-up and shut-down times of physical or emulated Android devices, which can slow down development and the CI/CD pipeline execution. An additional disadvantage is the flakiness of the Android instrumentation, which leads to instrumented tests often failing because of instrumentation failures and not because the test itself has failed, e.g. a failed or improper installation of the application before the test (Pandey et al., 2018). Paparazzi and Roborazzi have none of the above mentioned problems, as they do not require instrumentation; they run the tests on the Java Virtual Machine (JVM), just like any basic unit test, and depend on third-party UI rendering libraries.

The third criterion is the test execution speed (Soares, 2023a), which is tightly related to the second criterion. This goes beyond the already mentioned lack of start-up and shut-down times with tools like Paparazzi and Roborazzi. "There is no need to build test APKs wrapping the test code, no ADB invocations [and] no intensive data transfer via USB or Wifi" (Soares, 2023a). Coppola et al. (2019) state that "the instrumentation of Android emulators is significantly slower than the direct execution on the JVM" (p. 3214). Thus, Paparazzi and Roborazzi screenshot tests offer execution speeds higher than those of Shot screenshot tests.

The fourth criterion covers whether the tool allows the screenshot test to be run against "different device specifications" and "different global states" (Soares, 2023a). For example, a screenshot test for a card view could be performed under a light or dark mode configuration, for small or large font size, for different screen widths and different resolutions. If, for example, the card color for dark mode is accidentally changed, only a screenshot test that covers the dark mode configuration will catch the regression. Thus, it is important that screenshot tests cover the most relevant UI variations. With Shot, this is hard to do, as "changing the device state at the testing time is not trivial when leveraging physical or emulated devices" (Soares, 2023a). However, since screenshot testing tools like Paparazzi and Roborazzi "effectively replace the entire Android graphics pipeline with a fake one" (Soares, 2023a), they offer the possibility to trivially change such "device specifications" and "global states". This significantly reduces the amount of time needed to run the screenshot tests across different application states, because starting two emulators with different screen resolutions is not required, for example.

Finally, Roborazzi offers an important advantage over Paparazzi. It is able to integrate with Robolectric tests (‘Roborazzi documentation’, 2024) and thus enables UI interaction in screenshot tests, e.g. clicking, scrolling, swiping, which further enhances the potential of screenshot testing. Since Shot tests are essentially Android UI tests, interacting with the UI is also possible.

Table 5.1 provides a concise overview of the points described above.

Criterion / Tool	Shot	Paparazzi	Roborazzi
Total Required Effort	minimal	minimal	minimal
Requires Instrumentation	yes	no	no
Execution Speed	slower	fast	fast
Support for UI Variations	hard	easy	easy
Supports UI Interaction	yes	no	yes

Table 5.1: Tool comparison (with insights from Soares (2023a))

Conclusion

The following ADR sums up the decision of this section:

> **Context**
> Shot, Paparazzi and Roborazzi are three widely-adopted and up-to-date Android screenshot testing tools. While they all follow the same general structure of a screenshot test presented in figure 4.3, their implementations of it and thus also their behaviour differ significantly, as the above comparison shows.
>
> **Decision**
> The screenshot testing strategy developed in this thesis adopts Takahiro Menju's Roborazzi as the overall best choice, as it requires minimal effort, does not depend on the Android instrumentation and its screenshot tests are relatively fast. Roborazzi tests also make screenshot testing different UI variations easy and support UI interaction.

5.2.2 Tool Setup and Usage

The remaining part of this subsection provides an introduction into the setup and usage of the Roborazzi screenshot testing library and offers a test implementation example.

Integrating Roborazzi into an Android project is trivial, albeit slightly more complex than Paparazzi because of its Robolectric dependency. Listing 5.3 shows all the steps that are required. First, Roborazzi requires the integration of Robolectric by adding the JUnit framework and the Robolectric framework dependencies. Additionally, the `isIncludeAndroidResources` property is set to true for unit tests. Second, the Roborazzi Gradle plugin, which is responsible for generating the Gradle tasks used to run the screenshot tests, and the three Roborazzi dependencies need to be added to the project. Thus, the complete Roborazzi setup only requires changes to the `build.gradle.kts` source file.

```
plugins {
    ...

    // Roborazzi Gradle Plugin
    id("io.github.takahirom.roborazzi") version "<version>"
}

android {
    testOptions {
        unitTests {
            isIncludeAndroidResources = true
        }
    }
}

dependencies {
    ...

    // Required for Robolectric
    testImplementation("junit:junit:<version>")
    testImplementation("org.robolectric:robolectric:<version>")

    // Required for Roborazzi
    testImplementation("io.github.takahirom.roborazzi:roborazzi:<
        version>")
    testImplementation("io.github.takahirom.roborazzi:roborazzi-
        compose:<version>")
    testImplementation("io.github.takahirom.roborazzi:roborazzi-
        junit-rule:<version>")
}
```

Listing 5.3: Complete Roborazzi integration (in build.gradle.kts)

Furthermore, writing a screenshot test is just as simple as the setup. Listing 5.4 shows a slightly more complex screenshot test used to verify the home screen of a mock Android application. One should note that Roborazzi test classes are located in the test source set of the Android application, similar to conventional unit tests.

As can be seen in the code, the basis of a Roborazzi screenshot test is a simple unit test. A unit test requires a test class, e.g. HomeScreenTest, and a test method annotated with @Test, e.g. testHomeScreen(). Roborazzi further requires two Annotations, @RunWith(androidx.test.ext.junit.runners.AndroidJUnit4::class) and @GraphicsMode (GraphicsMode.Mode.NATIVE), and the central element of the screenshot test, which is the invocation of the screenshot method, e.g. captureRoboImage(). In this example, the screenshot method receives a lambda parameter that defines the UI layout that should be tested. Additionally, the Roborazzi documentation offers several other alternatives for calling the screenshot method, e.g. using Jetpack

```
@RunWith(androidx.test.ext.junit.runners.AndroidJUnit4::class)
@GraphicsMode(GraphicsMode.Mode.NATIVE)
@Config(qualifiers = RobolectricDeviceQualifiers.Pixel7Pro)
class HomeScreenTest {

    private val customImageComparator = SimpleImageComparator(
        maxDistance = 0.01f,
        hShift = 2,
        vShift = 2
    )
    private val customThresholdValidator = ThresholdValidator(0.02F)

    @OptIn(ExperimentalRoborazziApi::class)
    @get:Rule
    val roborazziRule = RoborazziRule(
        RoborazziRule.Options(
            outputDirectoryPath = "baseline-screenshots",
            roborazziOptions = RoborazziOptions(
                compareOptions = RoborazziOptions.CompareOptions(
                    imageComparator = customImageComparator,
                    resultValidator = customThresholdValidator
                )
            )
        )
    )

    @Test
    fun testHomeScreen() {
        captureRoboImage {
            AppTheme {
                HomeScreen()
            }
        }
    }
}
```

Listing 5.4: Complete Roborazzi screenshot test example

Compose test rules.

However, as mentioned in subsection 5.1.1, Roborazzi's behaviour can be altered to better fit the requirements of the project or of individual test cases. In listing 5.4, the RoborazziRule object represents the API through which this can be achieved. First, the reference screenshot directory is changed to baseline −screenshots. Second, the behaviour of the Roborazzi comparison algorithm is altered by changing four parameters: the threshold through ThresholdValidator and the maxDistance, hShift and vShift through SimpleImageComparator. In addition to that, the test class is annotated with @Config(qualifiers = RobolectricDeviceQualifiers. Pixel7Pro), which specifies that the UI should be rendered under *Pixel 7 Pro* device specifications.

XML-based UI Implementations

At the time of writing, the Jetpack Compose UI library is officially recommended by Android for creating application UIs ('Jetpack Compose (Get Started Page)', n.d.), as it "simplifies and accelerates UI development on Android". However, Android projects that have not yet migrated to using Jetpack Compose, or never will, still exist. The Roborazzi API also supports testing Extensible Markup Language (XML)-based, legacy Android UI code.

Listing 5.5 shows how this can be achieved, thanks to Roborazzi's integration with the Espresso UI testing library. Espresso's onView() method delivers an object, which Roborazzi uses as a receiver for its captureRoboImage() method. This takes a screenshot of the UI element with the welcome_text identifier.

```
onView(withId(R.id.welcome_text)).captureRoboImage()
```

Listing 5.5: XML-based screenshot

Gradle Tasks

Screenshot testing tools like Roborazzi are controlled through the Gradle tasks provided by their respective Gradle plugins. Table 5.2 describes the behaviour of the three most important Roborazzi Gradle tasks.

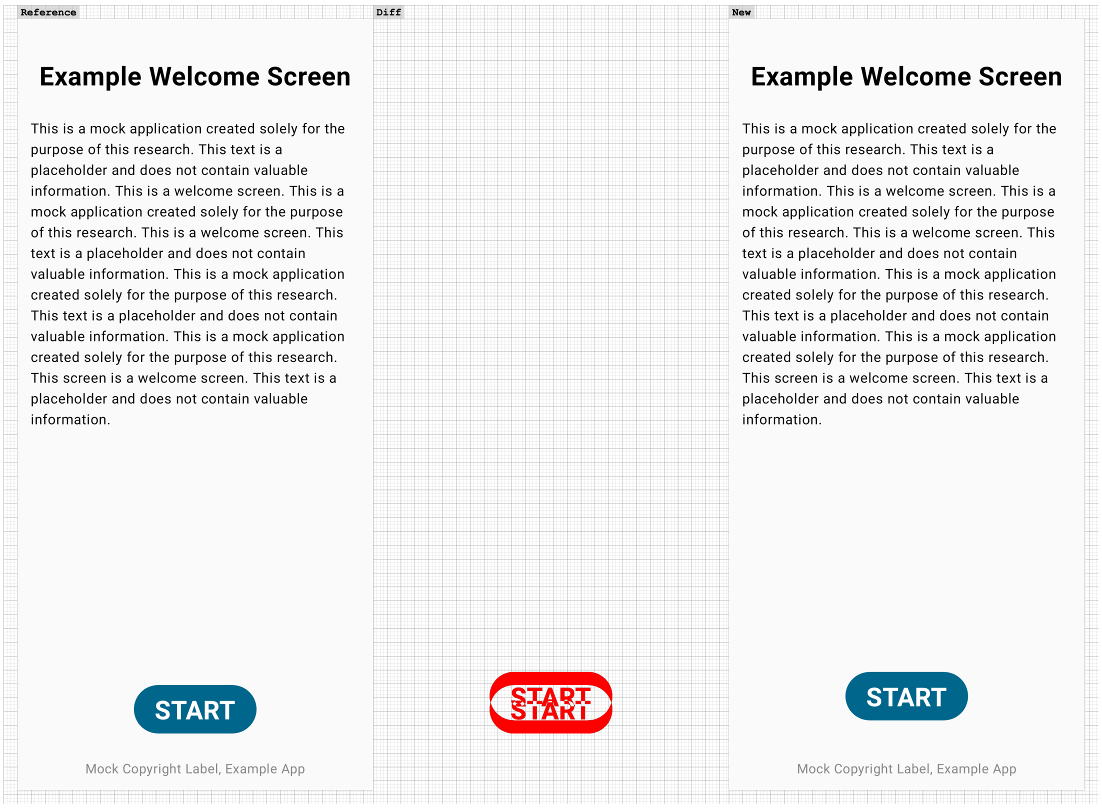
Task	Effect
recordRoborazziDebug	Records new reference screenshots.
compareRoborazziDebug	Runs the screenshot tests and generates reports that contain the test results.
verifyRoborazziDebug	Runs the screenshot tests and fails if one test has failed.

Table 5.2: Roborazzi Gradle tasks ('Roborazzi documentation', 2024)

As is explained in detail in ‘Roborazzi documentation’ (2024), recordRoborazziDebug generates the reference screenshots for each screenshot test, while compareRoborazziDebug and verifyRoborazziDebug can be used to run the screenshot tests, which compare the current UI implementation against the reference screenshots generated by the recordRoborazziDebug.

Outputs

Next to Hypertext Markup Language (HTML) reports and JSON objects that summarize the test runs, Roborazzi also creates comparison images (‘Roborazzi documentation’, 2024), which are also called difference images. Figure 5.5 shows an example of such a difference image, where the test object is the welcome screen of a mock Android application created solely for the purpose of this research. The reference screenshot is on the left side of the figure, while the actual screenshot is on the right side. The differences between the two are depicted in the middle part of the figure.



Example Welcome Screen

This is a mock application created solely for the purpose of this research. This text is a placeholder and does not contain valuable information. This is a welcome screen. This is a mock application created solely for the purpose of this research. This is a welcome screen. This text is a placeholder and does not contain valuable information. This is a mock application created solely for the purpose of this research. This text is a placeholder and does not contain valuable information. This is a mock application created solely for the purpose of this research. This screen is a welcome screen. This text is a placeholder and does not contain valuable information.

Mock Copyright Label, Example App

Example Welcome Screen

This is a mock application created solely for the purpose of this research. This text is a placeholder and does not contain valuable information. This is a welcome screen. This is a mock application created solely for the purpose of this research. This is a welcome screen. This text is a placeholder and does not contain valuable information. This is a mock application created solely for the purpose of this research. This text is a placeholder and does not contain valuable information. This is a mock application created solely for the purpose of this research. This screen is a welcome screen. This text is a placeholder and does not contain valuable information.

Mock Copyright Label, Example App

Figure 5.5: Example of a Roborazzi difference image

5.3 Test Case Implementation Strategy

Section 5.2.2 shows that implementing screenshot tests with Roborazzi is trivial and requires a minimal amount of effort. However, maximizing the added value that screenshot testing provides requires a screenshot testing strategy that answers the following questions:

- What parts of the application UI will be tested?
- When and in what order should the screenshot tests be implemented?

For example, a team that writes the screenshot tests long after each UI feature has been implemented and has already created several UI regressions does not take full advantage of the benefits of screenshot testing. The same is true for a project where historically stable parts of the application UI are tested, leaving the unstable and repression-prone components uncovered. Thus, maximizing the value that screenshot testing adds to a software project depends on finding suitable answers to the two important questions listed above.

5.3.1 What parts of the application UI will be tested?

Before answering the first question, it is helpful to define both extremes of what can be screenshot tested using Roborazzi or any similar screenshot testing tool. On the one hand, testing an individual, low-level UI element, e.g. custom button or custom icon, represents the minimal granularity of a screenshot test. On the other hand, screenshot testing an entire screen, e.g. the home screen or settings screen, corresponds to the maximum granularity. Any answer to the first question lies between these bounds.

Soares (2023b) puts forth two answers to this question and bases the decision between them on the existence of a design system or lack thereof. A design system represents a tool that "explains how a team should create products" (Perez-Cruz, 2019, p. 2) and it includes a UI component library (Fanguy, 2019) that provides common, project-specific UI building blocks and serves to uphold the common design language of the software project, be that one single application or a complete software system such as an automotive infotainment system. Soares (2023b) recommends the following procedure:

- In the absence of a design system, the screenshot tests should be implemented at the screen level.
- In the presence of a design system, implementing screenshot tests at the design system level yields the most value.

Targeting complete screens in the absence of a design system is the most effective strategy, as screen-level tests represent the upper limit of screenshot test granularity and doing so minimizes the amount of screenshot tests that are required. On the other hand, if a design system exists, targeting its individual, low-level UI elements allows for catching UI regressions at their root and thus signals the existence of a regression sooner in the SDLC. Additionally, if the amount of screens outnumbers that of design system elements, this leads to a further reduction in the amount of test cases (Soares, 2023b).

However, this procedure overlooks a problem that arises when application UIs are built using a design system: all the possible points of failure that inevitably appear where the UI elements from the design system are configured, populated with data, arranged and integrated into a complete screen. Thus, the more the application source code has to manipulate the UI elements from the design system in the ways described above, the greater this risk becomes. For instance, a calculator application that receives its button UI elements from a design system still has to label them and properly arrange them, maybe even dynamically arrange them, to fit the typical calculator pattern. Both tasks have the ability to introduce UI regressions into the application.

For those reason, this thesis considers screenshot testing complete screens to be just as valuable towards preventing UI regressions as testing the design system UI elements. Therefore, the recommended procedure is an altered version of the one presented by Soares (2023b):

- In the absence of a design system, the screenshot tests should be implemented at the screen level.
- In the presence of a design system, screen-level screenshot tests should be implemented in conjunction with design system screenshot tests.

The context and decision of the matters explored in this subsection are summed up in the following ADR:

Context
Roborazzi screenshot tests, or those that are implemented using similar testing tools, can cover any type of UI element, be that a complete screen or a low-level, design system UI element. However, as argued above, not all possible test objects are equal in the value that they add.

Decision
The screenshot testing strategy developed in this thesis recommends always implementing screen-level screenshot tests. Additionally, if a design system exists, its individual UI elements should be screenshot tested as well.

5.3.2 When and in what order should the screenshot tests be implemented?

The timing of screenshot test implementation is another crucial concern surrounding the screenshot testing strategy. For instance, postponing the implementation of a screenshot test for a new and unstable UI component means a higher risk of UI regression. Similarly, prioritizing the verification of an old and stable screen over a new and unstable one is relatively ineffective, as the latter has by definition a higher probability of introducing UI regressions into the application.

Considering the already mentioned triviality of implementing a Roborazzi test, this thesis recommends implementing the screenshot test immediately following the implementation of the UI feature, be that a low-level UI component or a complete screen. While this does slow down the pace of feature implementation, such a coupled feature-test-approach maximizes screenshot test coverage at any given moment and prevents the buildup of test debt in the form of a screenshot test backlog. Figure 5.6 illustrates an example of the recommended screenshot testing approach.

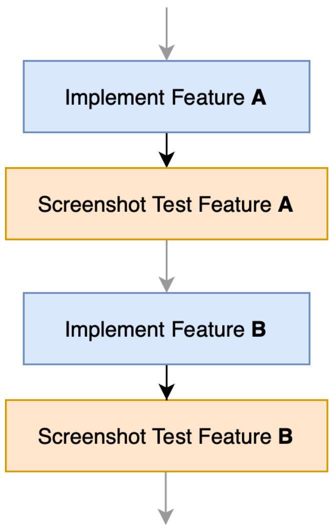


Figure 5.6: Recommended implementation approach

However, in projects where screenshot testing has not been implemented from the beginning, the screenshot test backlog has to be dealt with. As argued above, not every potential screenshot test is equally valuable at any given time. Thus, the objective is to choose the most impactful screenshot test. Due to the large diversity of project requirements and application UIs, instead of prescribing a concrete approach, this thesis provides a list of general and open questions that aid in choosing an appropriate screenshot test case from the backlog. This is illustrated as a Venn diagram in figure 5.7. The list includes questions for test cases that involve both low-level UI components or complete screens. It targets takes both importance – through questions such as "Which screens are the most central to the application?" – and urgency – through questions like "Which screens have caused the most visual regressions?". In addition, the list can be further expanded by any development team to better suit their specific project requirements and challenges.

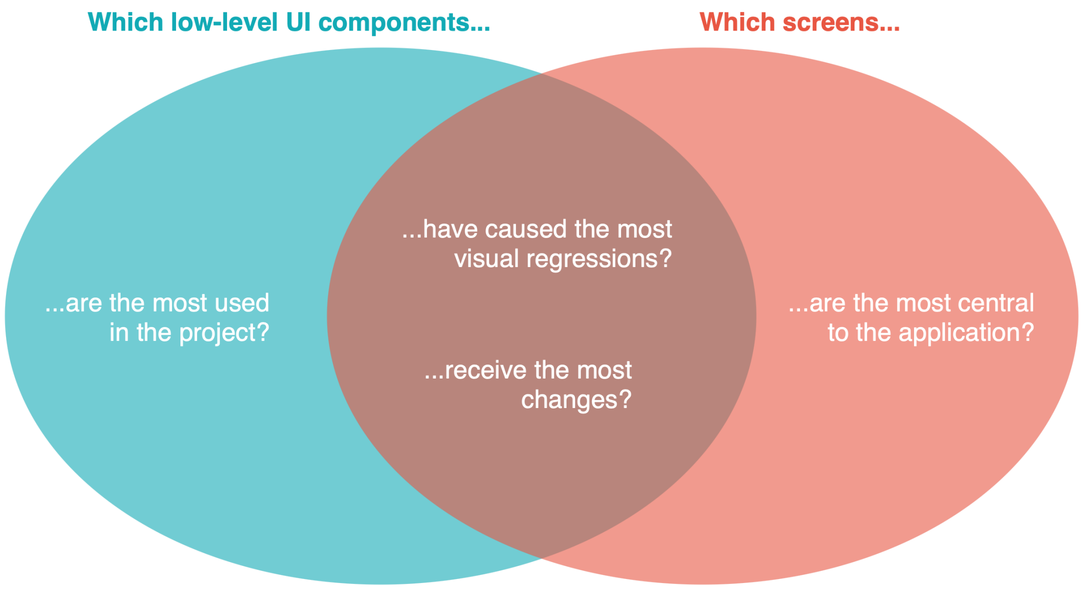

Which low-level UI components...
Which screens...
...are the most used in the project?
...have caused the most visual regressions?
...receive the most changes?
...are the most central to the application?

Figure 5.7: Questions for filtering the screenshot test backlog

The following ADR sums up the decision of this section:

> **Context**
> At any given time, some potential screenshot tests are more valuable than others. Additionally, the list of potential screenshot tests grows constantly as the UI evolves.
>
> **Decision**
> The screenshot testing strategy developed in this thesis recommends implementing the screenshot tests immediately following the implementation of the UI feature. In the case of an existing screenshot test backlog, the most valuable screenshot test will be chosen according to the list of questions presented in figure 5.7.

5.4 CI/CD Pipeline Integration

CI/CD is a widely-used DevOps best-practice (Pittet, n.d.). It is typically achieved through a CI/CD pipeline that is run on an automation server and is made up of several critical steps. For example, a pipeline for a typical Android application includes building the code, running tests and delivering the final binary files (Harsh, 2024). This section of the thesis is concerned with the integration of Roborazzi screenshot testing into a CI/CD pipeline, which is illustrated in figure 5.8.

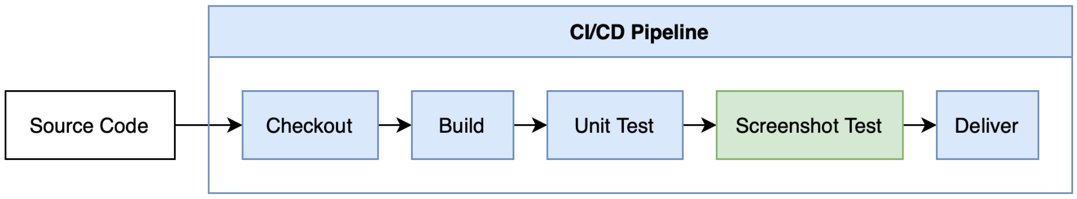


Figure 5.8: CI/CD pipeline with screenshot testing

How Roborazzi can be controlled through Gradle tasks is already explained in subsection 5.2.2. Since Roborazzi tests run on the JVM, i.e. do not require instrumentation, and the reference screenshots are stored directly in the project directory, integrating Roborazzi screenshot testing into a CI/CD pipeline only requires the execution of a suitable Gradle task. For instance, listing 5.6 shows an example of a screenshot test execution in a Jenkins CI/CD pipeline.

```
stage('Run Roborazzi Screenshot Tests') {
    sh('./gradlew verifyRoborazziDebug')
}
```

Listing 5.6: Example screenshot test execution in Jenkins

5.4.1 Platform-Dependent Rendering Differences

Roborazzi screenshot tests exhibit platform-dependent rendering behavior (Hoisie, 2024), which "is due to variations in how graphics libraries render components on different platforms" (‘Roborazzi FAQ’, 2024). Figure 5.9 provides an example of such rendering differences and shows UI renderings of two stacked, pill-shaped UI elements. Figure 5.9a represents the Roborazzi UI render on the macOS operating system, while figure 5.9c represents the Roborazzi UI render on Windows. To the naked eye, these renders look identical. However, as seen in figure 5.9b, the difference image generated by Roborazzi shows that some pixels around the borders of the two UI elements have slightly different color values. Thus, if the reference screenshot is generated on macOS and the screenshot test is run on Windows, for example, the test will detect UI changes and fail.

Thus, ‘Roborazzi FAQ’ (2024) recommends generating the reference screenshots and running the screenshot tests in the CI/CD pipeline, as it keeps the platform-dependent rendering differences constant.

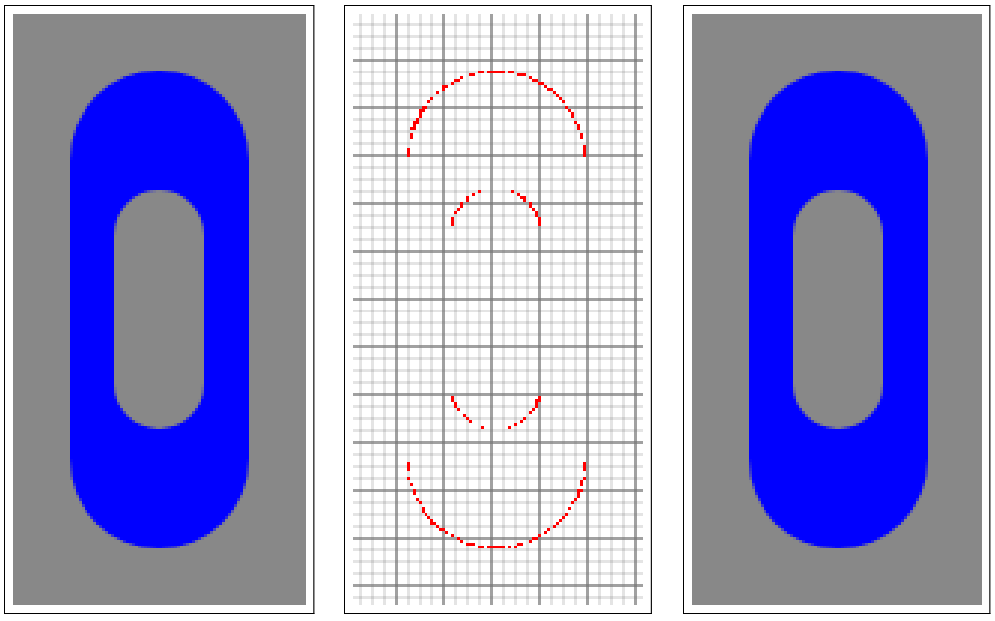

(a) macOS render (b) Difference image (c) Windows render

Figure 5.9: Roborazzi rendering differences between screenshot tests run on the macOS and Windows operating systems

5.5 Data Management

Screenshot tests depend on a reference that the test object is compared against, i.e. reference screenshots. This introduces the challenge of handling and storing constantly growing and potentially large amounts of data. This subsection describes three options for dealing with this challenge.

5.5.1 Storing and Versioning with Git

The first option is to completely rely on the screenshot testing tool and the Version Control System (VCS), e.g. Git. As already mentioned, the directory where reference screenshots are stored can be specified through the RoborazziRule. One would simply have to choose a directory that is being tracked by Git, which is usually not the case for the default build/outputs/roborazzi output directory, and commit the reference screenshots just like any other file. One advantage of this straight-forward approach to screenshot data management is the minimal amount of setup that it requires. The second advantage is the process-related simplicity of using one VCS to version both the reference screenshots and the source code together.

However, the disadvantage is that this can lead to excessively large Git repositories, as Git is inefficient at versioning large binary files such as PNGs. "While text-based files can generate plain text diffs, any change to a binary file requires Git to completely replace the file in the repository" (‘Git Large File Storage (LFS)’, n.d.). Figure 5.10 depicts an experiment that illustrates this problem. The example Git repository has a total size of 12KB and it contains one single PNG file of 10KB, which is a common file size for reference screenshots. Making one minimal change to the PNG file, e.g. chaining one pixel value, and committing the changes results in the repository taking up a total of 22KB, which is 10KB more than before the change. Repeating these steps leads to the repository taking up a total of 32KB. Thus, this experiment confirms that a tiny change in the contents of a PNG image results in an increase in required disk space equal to the size of the file.

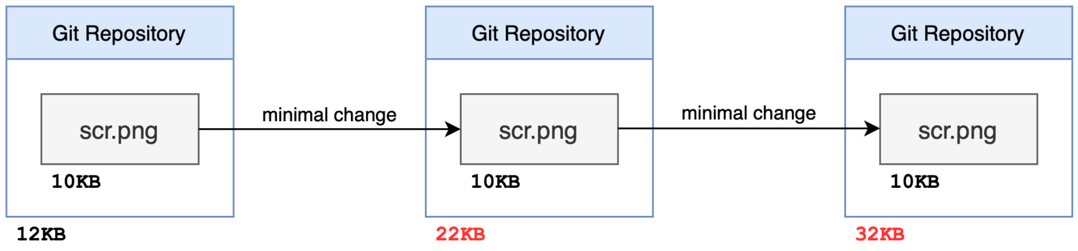


Figure 5.10: Git binary file handling experiment

While adding 10KB to the size of the Git repository is irrelevant, this problem can scale up drastically. The following thought experiment illustrates this risk and uses an imaginary large Android application as an example. The application contains 200 individual screens and 150 individual low-level UI components, each of them being screenshot tested. Each test object is tested under 17 separate circumstances, which represent a selection of the 32 possible variations of the following UI parameters:

- light/dark mode,
- portrait/landscape mode,
- color blind mode on/off,
- 4.5 inch/6.7 inch screen size,
- Left-to-Right/Right-to-Left display mode (e.g. for the Arabic writing system).

Additionally, the UI of the application has been majorly updated twice and each screenshot test has had its reference screenshot changed 5 times on average for each major UI version. Thus, on average, 15 reference screenshot versions exist for each screenshot test. In sum, if each reference screenshot takes up 15KB of disk space on average, the versioning of the reference screenshots alone requires a total of 1.33GB.

Thus, a possibly grave consequence of using the approach presented above is reaching a repository size limit. For comparison, at the time of writing, the repository hosting service called Bitbucket limits the size of cloud repositories at 4GB and, after reaching the limit, one can only "push changes that undo the latest commits" ('Reduce repository size', n.d.). Furthermore, larger repositories mean that Git operations such as cloning take longer, as Git has to handle larger amounts of data.

5.5.2 Storing and Versioning with Git LFS

The second option is to integrate the open-source Git extension called Git Large File Storage (Git LFS). While it does not enhance Git's capabilities of versioning binary files, it does solve the problem of quickly growing repository sizes when working with binary files. Git LFS works by replacing large files in the repository with pointers to an external storage location ('Git LFS Documentation', n.d.). Thus, "large files are downloaded during the checkout process rather than during cloning or fetching" ('Git LFS', n.d.). Well-known source code repository hosting platforms such as GitHub and Bitbucket offer built-in Git LFS support ('About storage and bandwidth usage', n.d.; 'Use Git LFS with Bitbucket', n.d.) and thus handle the storage of the binary files in the background. Additionally, Git LFS is easy to set up and to use. Listing 5.7 shows the steps that are required for setup on local developer machines according to 'Git LFS Documentation' (n.d.).

```
// Install the Git LFS extension.
git lfs install

// Specify what files should be tracked using Git LFS.
git lfs track "*.png"

// Make sure that ".gitattributes" is tracked.
git add .gitattributes
```

Listing 5.7: Git LFS setup

The first command installs the Git LFS extension. Using the second command, the developer can specify which files should be tracked using Git LFS. In this example, all files that have the .png extension are being tracked using Git LFS. The last command makes sure that the .gitattributes file is being tracked by Git, as that is where the record of LFS-tracked files is stored (Gruber, 2019). From this point on, the LFS-tracked files can be updated using conventional Git commands, just like all other files ('Git LFS Documentation', n.d.). Thus, Git LFS enhances Git's functionality at a relatively low cost of setup and use, and requires no extra effort for CI/CD pipeline integration.

5.5.3 Custom Storage and Versioning Solution

The third option is to develop a custom screenshot data management system that stores the reference screenshots on a remote server, e.g. an artifact storage server, and handles the data transfer between the local environment and the remote storage location. One possible implementation is depicted in figure 5.11. When Roborazzi records new reference screenshots in the local environment, it stores them in the temporary screenshot storage location. The custom Gradle task uploadScreenshots then transfers the screenshots further to the remote screenshot storage location. Reversely, before a Roborazzi test is run, the custom Gradle task downloadScreenshots fetches the reference screenshots from the remote storage location and places them back into the temporary storage location, where Roborazzi can use them for comparison.

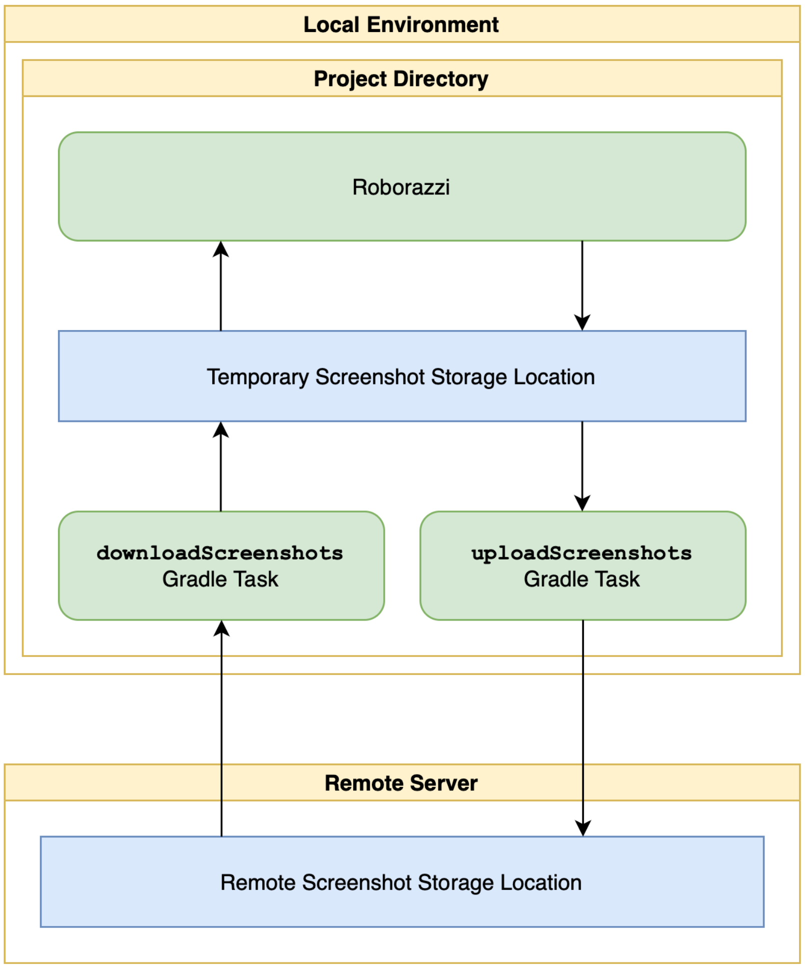

Local Environment
Project Directory
Roborazzi
Temporary Screenshot Storage Location
downloadScreenshots Gradle Task
uploadScreenshots Gradle Task
Remote Server
Remote Screenshot Storage Location

Figure 5.11: Custom screenshot data management solution

One advantage of such a custom screenshot data management system is the decoupling of the screenshot data from the VCS. For example, this allows the development team to only store the latest version of reference screenshots or implement a custom versioning system, as opposed to keeping track of the complete history of the reference screenshots with a VCS such as Git. It also simplifies the screenshot test data by making it possible to define only one set of reference screenshots per project, as opposed to one for each branch.

However, the disadvantage of such an approach is that it requires more effort to implement and set up than using Git LFS. It also requires infrastructure for the remote screenshot storage location. Thus, it is the responsibility of each software development team to decide whether implementing a custom screenshot management solution is worth the extra cost and effort.

5.5.4 Conclusion

The following ADR sums up the decision of this section:

> **Context**
> Screenshot testing introduces the challenge of managing the screenshot data by versioning and storing it appropriately. Subsection 5.5 explores three possibilities for dealing with this challenge.
>
> **Decision**
> The screenshot testing strategy developed in this thesis recommends a screenshot data management solution that takes the drawbacks of Git's binary file handling mentioned in 'Git Large File Storage (LFS)' (n.d.) into account. Therefore, the recommendation is the integration of Git LFS or the implementation of a custom screenshot data management solution, such as the one presented in subsection 5.5.3.

5.6 Pixel Differences and Tolerance Parameters

This section explores the two types of causes behind the pixel differences that one might encounter when using Roborazzi and how each of them can be tolerated. Additionally, it warns against potential pitfalls that can reduce the reliability of the screenshot tests.

On the one hand, pixel differences can be caused by platform-dependent UI rendering behavior. This is also mentioned in subsection 5.4.1. More precisely, according to Roborazzi's main contributor, Takahiro Menju, these differences are caused by how graphics libraries such as Skia and Minikin are implemented for each platform (Menju, 2024). Figure 5.12 provides a simplified example of such rendering differences. The picture on the left shows the original UI render, while the picture on the right shows a render of the same UI element, but with a different UI rendering behavior. As the red markings show, this results in slightly changed pixel values, which can cause the screenshot test to fail.

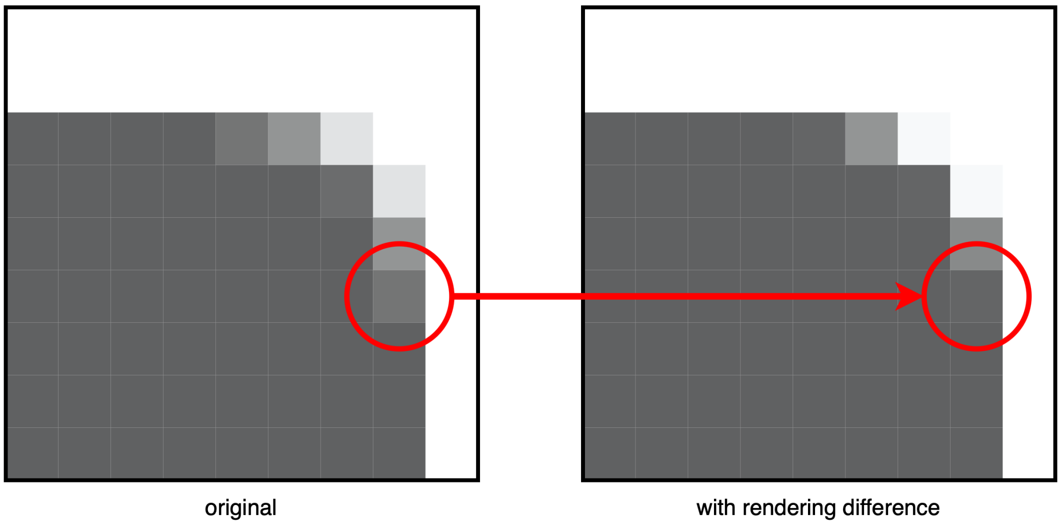


Figure 5.12: Detailed example of rendering difference

With Roborazzi, three parameters can be used in order to tolerate such pixel value differences. The usage of these parameters can be seen in listing 5.4. As noted in subsection 5.1.1, maxDistance defines the maximum Euclidean distance between the color values of two pixels that are considered to be equal. Then, if the maximum distance has been exceeded, hShift and vShift define the area in which the comparison algorithm is allowed to further search for a matching pixel, in case it has been slightly shifted because of small changes in the rendering behavior.

On the other hand, pixel differences can also be caused by the UI implementation itself, which is precisely the pixel differences that screenshot tests aim to detect. An example of this is shown in figure 5.13. The left picture shows the original UI state, while the right picture shows the same UI element that is now shifted left. Such a UI regression could be caused by the accidental change of a padding value, for example.

However, such pixel differences can be tolerated as well. The threshold parameter, whose usage can also be seen in listing 5.4, defines the maximum allowed percentage of pixels with changed values in order for the test to pass.

With regard to these four parameter options that Roborazzi offers, a possible pitfall is adjusting the wrong type of parameter. This is illustrated in the following example. Say that one particular screenshot test is too sensitive to UI rendering differences. According to this section, one should then adjust maxDistance, hShift

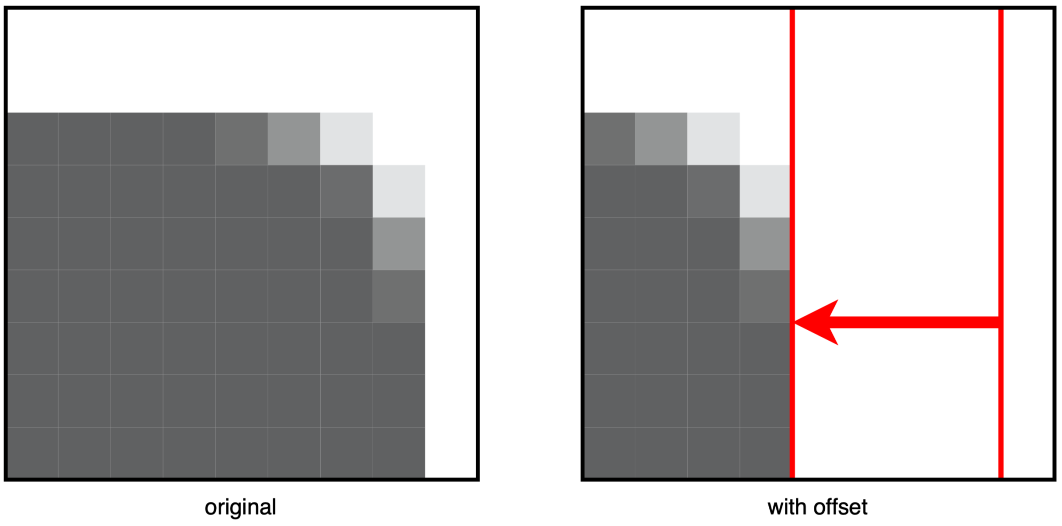


Figure 5.13: Detailed example of UI regression

and/or vShift. Raising threshold instead leads to the following problem. While the test will indeed tolerate more pixels that are slightly different because of the UI rendering differences, it will also tolerate more pixels that are completely different, i.e. UI regressions. This is an unwanted side-effect, which can reduce the reliability of the screenshot test.

6 Evaluation

This chapter documents whether the quality goals listed in section 3.2 have been achieved through the screenshot testing strategy developed in this thesis. The first quality goal, *Functional Suitability*, covers the functional requirements as well. Additionally, it provides direction for future research by pointing out the main shortcoming of screenshot testing as it is implemented in this research.

6.1 Quality Goals

Each quality goal from section 3.2 is graded with either *achieved*, *partially achieved* or *not achieved*.

6.1.1 Functional Suitability

This quality goal requires that the "stated and implied needs" (*ISO/IEC 25010*, 2023) of the project and the functional requirements listed in section 3.1 be met. It has been *achieved*.

The screenshot testing strategy developed in this thesis enables automated screenshot regression testing. Roborazzi, the tool of choice, offers a high-level API that abstracts away the screenshot-taking and screenshot comparison processes, as seen in listing 5.4. It also enables low-level control for defining precise tolerance levels or rendering the UI under particular device specifications. The engineer (e.g. manual tester or software developer) can both create the reference screenshots and run the screenshot tests using the Gradle tasks shown in table 5.2. On top of that, section 5.5 recommends two suitable screenshot data management solutions that allow for seamless data transfer and storage. As seen in section 5.4, the screenshot testing strategy also covers the integration of screenshot testing into the CI/CD processes. Finally, Roborazzi screenshot tests output HTML reports, JSON objects and difference images that describe the test run outcomes in detail, which is presented in subsection 5.2.2.

6.1.2 Efficiency and Performance

This quality goal requires the minimization of invested resources and effort and the maximization of performance. It has been *achieved*.

As already mentioned, Roborazzi requires a minimal amount of effort for its setup and for learning how to use it. The fact that Roborazzi tests run on the JVM and do not require a physical or emulated Android device means that no extra hardware resources are required. Additionally, the test case implementation strategy presented in section 5.3 serves to maximize the amount of value that the screenshot testing strategy adds to the project by recommending what parts of the application UI should be tested and when.

6.1.3 Reliability

This quality goal requires that the screenshot testing strategy delivers accurate and reliable results. It has been *achieved*.

As discussed in subsection 5.4.1, Roborazzi is affected by platform-dependent rendering differences. However, the recommendation to create the screenshots and run the tests in the CI/CD environment ('Roborazzi FAQ', 2024) and the tolerance parameters presented in subsection 5.6 do solve this problem. Also, the warning that subsection 5.6 gives regarding the incorrect use of Roborazzi tolerance parameters further increases the reliability and accuracy of the screenshot testing strategy.

6.1.4 Maintainability

This quality goal requires the easy maintenance of the screenshot tests, screenshot data and testing architecture. It has been *achieved*.

As shown in subsection 5.2.2, the reference screenshots for Roborazzi tests can be generated using only one Gradle task. The decision to use either Git Large File Storage (LFS) or a custom screenshot data management solution, made in section 5.5 of this thesis, also makes for a highly maintainable system. Maintainability is further improved by the fact that screenshot test implementations are self-contained inside of their own test classes, as presented in listing 5.4.

6.1.5 Scalability

This quality goal requires the possibility to accommodate a constantly growing test object and test suite. It has been *achieved*.

The decision surrounding the screenshot data management and the self-contained nature of Roborazzi screenshot tests also play into its high scalability. On the one hand, more screenshot data can be added without polluting the code repository with a large amount of binary files. On the other hand, further screenshot tests can be added to the test suite without affecting or being affected by the existing tests.

6.2 Main Shortcoming and Future Research

The main shortcoming of the screenshot testing strategy developed in this thesis is that it fails to automate the complete design verification process. While it does protect against UI *regressions*, i.e. against visual bugs that break the design implementation that was once valid, it cannot detect visual bugs that the developer introduces into the *first* implementation of the UI. This is illustrated in figure 6.1.

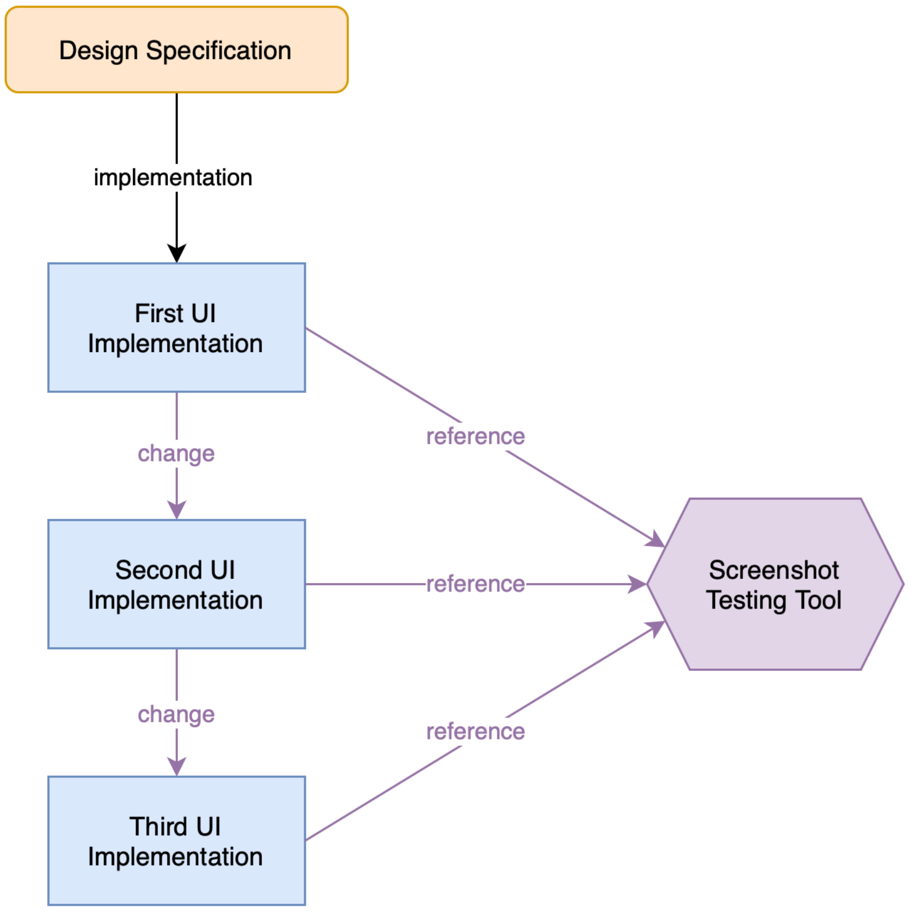


Figure 6.1: What does screenshot testing cover?

The first reference screenshot that the screenshot testing tool can create is derived from the first UI implementation. As such, for any subsequent change of the UI implementation, there exists a reference implementation that can be used for comparison. However, this leaves the first UI implementation, which originates directly from the design specification, exposed to visual bugs. If those bugs go unnoticed by the developer, a screenshot testing strategy such as the one developed in this thesis cannot detect them either.

The question whether the complete automation of design verification is possible and how that can be achieved exceeds the scope of this thesis. However, it does represent a possibly fruitful topic for future research.

7 Conclusion

The research presented in this thesis has explored the topic of automating the design validation process for Android applications through screenshot testing. The comprehensive screenshot testing strategy this it has put forth tackles crucial questions such as what different types of tools currently exist, which tools are the best fit for the given requirements and how the tool of choice, in this case Roborazzi, can be integrated into the project and used for implementing screenshot tests. The thesis also offered a test case implementation strategy that helps identify the screenshot test cases that add the most value to the software project by answering the questions *What should be tested?* and *When should it be tested?* Additionally, topics such as the integration of screenshot testing into the CI/CD process and Roborazzi's platform-dependent rendering differences have also been addressed.

Despite the fact that, as shown in section 6.1, the strategy developed in this thesis achieves all the quality goals and meets all the functional requirements that chapter 3 has established, this research has by no means exhausted the topic of automatic design validation and screenshot testing. Further questions such as how the design validation process can be completely automated, minimally explored in section 6.2, are left unanswered and thus hopefully provide motivation for future research.

Thus, through the insights that have been put forth and through the suggestions for future research, this thesis can hopefully be a valuable addition to the body of academic research surrounding design validation and, more specifically, screenshot testing.

Appendices

A Tools and Resources

The following list contains the internet links to tools and resources mentioned in this thesis:

- **arc42**: https://arc42.org/
- **Michael Nygard's ADR template**: https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- **Roborazzi**: https://github.com/takahirom/roborazzi
- **Roborazzi documentation**: https://takahirom.github.io/roborazzi/top.html
- **Paparazzi**: https://github.com/cashapp/paparazzi
- **Paparazzi documentation**: https://cashapp.github.io/paparazzi/
- **Shot**: https://github.com/pedrovgs/Shot
- **Shot documentation (README)**: https://github.com/pedrovgs/Shot/blob/master/README.md
- **Aplitools Eyes**: https://applitools.com/platform/eyes/
- **App Percy**: https://www.browserstack.com/app-percy
- **Git**: https://git-scm.com/
- **Git LFS**: https://git-lfs.com/

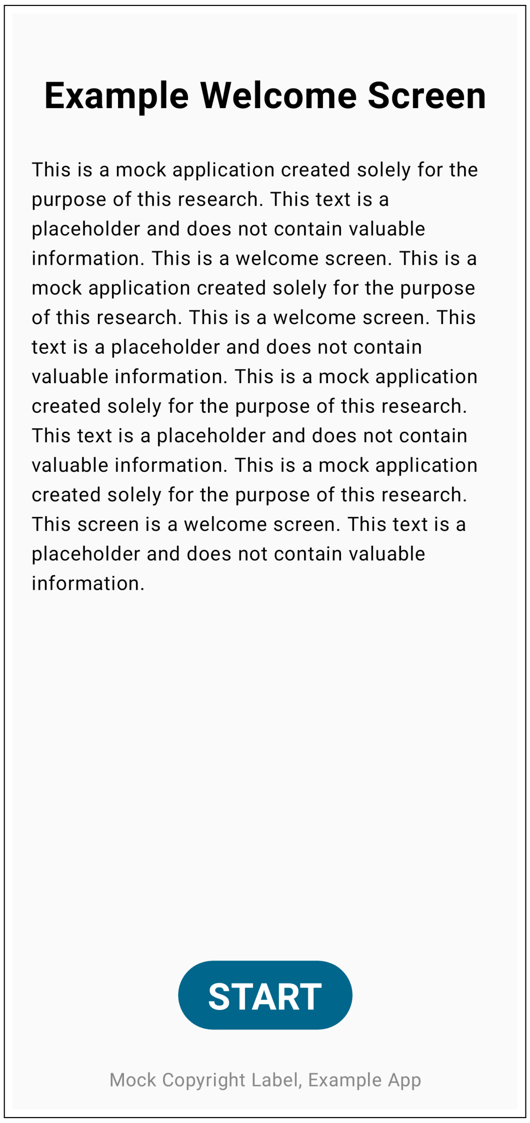
B API References

The following list contains the internet links to references of APIs mentioned in this thesis:

- **Espresso API**: https://developer.android.com/reference/kotlin/androidx/test/espresso/package-summary
- **UI Automator API**: https://developer.android.com/reference/kotlin/androidx/test/uiautomator/package-summary

C Mock Android Application

This mock Android application has been created solely for the purpose of this research. Figure C.1 shows its welcome screen.



Example Welcome Screen

This is a mock application created solely for the purpose of this research. This text is a placeholder and does not contain valuable information. This is a welcome screen. This is a mock application created solely for the purpose of this research. This is a welcome screen. This text is a placeholder and does not contain valuable information. This is a mock application created solely for the purpose of this research. This text is a placeholder and does not contain valuable information. This is a mock application created solely for the purpose of this research. This screen is a welcome screen. This text is a placeholder and does not contain valuable information.

Figure C.1: Welcome screen of the mock Android application

References

About storage and bandwidth usage [Accessed: 28.10.2024]. (n.d.). https://docs.github.com/en/repositories/working-with-files/managing-large-files/about-storage-and-bandwidth-usage

Arc42 documentation [Accessed: 31.10.2024]. (n.d.). https://arc42.org/documentation/

Baumgartner, M., Steirer, T., Wendland, M.-F., Gwihs, S., Hartner, J., & Seidl, R. (2022). *Test automation fundamentals*. Rocky Nook.

Compose preview screenshot testing [Accessed: 31.10.2024]. (2024). https://developer.android.com/studio/preview/compose-screenshot-testing

Coppola, R., Ardito, L., & Torchiano, M. (2023). Multi-device, robust, and integrated android gui testing: A conceptual framework. *Testing Software and Systems*, 115–125.

Coppola, R., Morisio, M., Torchiano, M., & Ardito, L. (2019). Scripted gui testing of android open-source apps: Evolution of test code and fragility causes. *Empirical Software Engineering*, *24*, 3205–3248.

Cruz, V. P. G., Rocha, H., & Valente, M. T. (2023). Snapshot testing in practice: Benefits and drawbacks. *The Journal of Systems and Software*, *204*.

Differ github [Accessed: 27.10.2024]. (2024). https://github.com/dropbox/differ

Fanguy, W. (2019). *A comprehensive guide to design systems*. Retrieved October 30, 2024, from https://www.invisionapp.com/inside-design/guide-to-design-systems/

Fujita, S., Kashiwa, Y., Lin, B., & Iida, H. (2023). An empirical study on the use of snapshot testing. *2023 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 335–340.

Functional testing: A detailed guide. (2024). Retrieved October 29, 2024, from https://www.browserstack.com/guide/functional-testing

Git large file storage (lfs) [Accessed: 28.10.2024]. (n.d.). https://docs.gitlab.com/ee/topics/git/lfs/

Git lfs [Accessed: 28.10.2024]. (n.d.). https://www.atlassian.com/git/tutorials/git-lfs

Git lfs documentation [Accessed: 28.10.2024]. (n.d.). https://git-lfs.com/

Gruber, F. (2019). *Git lfs for dummies* [Translated from German to English]. Retrieved October 29, 2024, from https://codefluegel.com/blog/git-lfs-for-dummies/

Harsh, K. (2024). *How to build a ci/cd pipeline for android projects*. Retrieved October 28, 2024, from https://blog.jetbrains.com/teamcity/2024/07/cicd-for-android/

Hoisie, M. (2024). Comment on issue #1242 in the *Now in Android* repository [Accessed: 30.10.2024]. https://github.com/android/nowinandroid/issues/1242#issuecomment-2032962982

Imagecomparator.kt [source code] [Accessed: 29.10.2024]. (2024). https://github.com/dropbox/differ/blob/main/differ/src/commonMain/kotlin/com/dropbox/differ/ImageComparator.kt

ISO/IEC 25010 (Standard). (2023). International Organization for Standardization/International Electrotechnical Commission.

Jetpack compose (get started page) [Accessed: 29.10.2024]. (n.d.). https://developer.android.com/compose

McMillan, T. (2024). Manual testing vs automated testing: Key differences [Accessed: 31.10.2024]. https://www.testrail.com/blog/manual-vs-automated-testing/

Menju, T. (2024). Comment on issue #351 in the *Roborazzi* repository [Accessed: 31.10.2024]. https://github.com/takahirom/roborazzi/issues/351#issuecomment-2100757751

Nygard, M. (2011). Documenting architecture decisions [Accessed: 31.10.2024]. https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions

Pandey, A., Khan, R., & Srivastava, A. K. (2018). Challenges in automation of test cases for mobile payment apps. *2018 4th International Conference on Computational Intelligence & Communication Technology (CICT)*, 1–4.

Paparazzi documentation [Accessed: 29.10.2024]. (2024). https://cashapp.github.io/paparazzi/

Paparazziplugin.kt [source code] [Accessed: 29.10.2024]. (2024). https://github.com/cashapp/paparazzi/blob/master/paparazzi-gradle-plugin/src/main/java/app/cash/paparazzi/gradle/PaparazziPlugin.kt

Perez-Cruz, Y. (2019). *Expressive design systems*. A Book Apart.

Pittet, S. (n.d.). *Continuous integration vs. delivery vs. deployment*. Retrieved October 28, 2024, from https://www.atlassian.com/continuous-delivery/principles/continuous-integration-vs-delivery-vs-deployment

Pixelperfect.kt [source code] [Accessed: 29.10.2024]. (2024). https://github.com/cashapp/paparazzi/blob/master/paparazzi/src/main/java/app/cash/paparazzi/internal/PixelPerfect.kt

Reduce repository size [Accessed: 28.10.2024]. (n.d.). https://support.atlassian.com/bitbucket-cloud/docs/reduce-repository-size/

Revolutionize ui testing with visual ai [Accessed: 29.10.2024]. (n.d.). https://applitools.com/platform/validate/visual-ai/

Roborazzi documentation [Accessed: 27.10.2024]. (2024). https://takahirom.github.io/roborazzi/top.html

Roborazzi faq [Accessed: 30.10.2024]. (2024). https://takahirom.github.io/roborazzi/faq.html

Roborazzi github [Accessed: 29.10.2024]. (2024). https://github.com/takahirom/roborazzi

Shot readme [Accessed: 29.10.2024]. (2023). https://github.com/pedrovgs/Shot/blob/master/README.md

Soares, U. (2023a). *The landscape of android screenshot testing in 2023*. Retrieved October 27, 2024, from https://ubiratansoares.dev/posts/screenshot-testing-for-android-landscape/

Soares, U. (2023b). *Two strategies to drive screenshot testing in mobile projects*. Retrieved October 27, 2024, from https://ubiratansoares.dev/posts/two-strategies-for-screenshot-testing/

Use git lfs with bitbucket [Accessed: 28.10.2024]. (n.d.). https://support.atlassian.com/bitbucket-cloud/docs/use-git-lfs-with-bitbucket/

Visual testing powered by computer vision [Accessed: 29.10.2024]. (n.d.). https://www.browserstack.com/app-percy/percy-visual-engine?scroll_to=percy-visual-engine

What is non-functional testing? (2024). Retrieved October 29, 2024, from https://www.browserstack.com/guide/what-is-non-functional-testing

Why arc42? [Accessed: 31.10.2024]. (n.d.). https://arc42.org/why

Yerburgh, E. (2018). *Testing vue.js applications*. Manning Publications.