IMAGE COMPARISON BASED MOBILE USER INTERFACE VERIFICATION FRAMEWORK

Maha Kumarage Dinu Sandaru Kumarasiri

168238F

Degree of Master of Science

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March, 2020

IMAGE COMPARISON BASED MOBILE USER INTERFACE VERIFICATION FRAMEWORK

Maha Kumarage Dinu Sandaru Kumarasiri

168238F

Thesis submitted in partial fulfillment of the requirements for the degree Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

University of Moratuwa

Sri Lanka

March, 2020

DECLARATION

I declare that this is my own work and this MSc thesis project report does not incorporate without acknowledgement any material previously submitted for the degree or diploma in any other university or institute of higher learning and to the best of my knowledge and belief it does not contain any material previously published or written by another person except where the acknowledgement is made in the text.

Also, I hereby grant University of Moratuwa the non-exclusive right to reproduce and distribute my dissertation, in whole or in part in print, electronic or other medium. I retain the right to use this content in whole or part in future works (such as articles of books).

Signature: ……………………. Date: ……………………….

Name: M.K.D.S. Kumarasiri

I certify that the declaration above by the candidate is true to the best of my knowledge and that this report is acceptable for evaluation for the CS-6997 MSc Thesis.

Name of the supervisor: Dr. Indika Perera

Signature of the supervisor: ………………. Date: ………………………

ABSTRACT

Due to the highly competitive market, user interface of a mobile application plays a major role in attracting and retaining its user base. In a full stack or web application development, there is usually a user interface (UI)/ user experience (UX) designer or a front-end engineer who implements the front end. On the contrary, in mobile applications, the app developers themselves implement the front end according to the mockups provided by the UI/UX designers. The verification of the user interface of the actual application against the provided mockup happens manually by developers and testers and not by the designers which makes it less accurate and time consuming since their eyes are not trained to identify pixel level visual differences.

Until now various researches have been done on automating the verification step of the event flow and underlying functionalities. But verifying the user interface of mobile applications is still left for the human eye.

The main objective of this research is to develop a mechanism to get a quantifiable score based on how much the user interface of a mobile application matches with its initial mockups. Two accuracy levels are considered for computing this score; layout and overall. Layout score limits the comparison to the layout of the user interface whereas the overall score compares layout along with color, orientation, etc.

For the layout level comparison, three local feature matching algorithms namely, SIFT (Scale-Invariant Feature-Transform), CSIFT (Color SIFT), PCA-SIFT (Principal Component Analysis SIFT) along with a simple bob detection matching algorithm are considered to be experimented with. For the overall level comparison, a pixel level comparison algorithm is used.

In parallel a survey is conducted where UI/UX designers would provide a layout and overall score for a set of selected use cases. These scores were compared with the scores from the image comparison algorithms. Based on this, CSIFT is chosen as the underlying algorithm to compute layout comparison scores as it outputs the closest values mimicking designers. For the overall value the pixel based scores ended up being stricter than values given by the designers.

In conclusion, the objective of the research is successfully achieved by implementing a framework which will output a score based on the comparison between the mockups and the actual user interface of mobile applications in two accuracy levels; overall and layout only. Overall score based on pixel level matching turned out to be too strict and better suited if the requirement is a strict conformity to the provided user interface. Layout score also has limitations with text intensive applications when the data is dynamically loaded. Both these scores can be used to verify the user interface, but the thresholds and which score to use is dependent on the application and how much deviation the company allows against the provided mockup.

ACKNOWLEDGMENT

I owe my deepest gratitude to my supervisor, Dr. Indika Perera, for his invaluable support in providing relevant knowledge, advice and supervision throughout the project. This would not have been possible without his expertise and continuous guidance.

I am deeply grateful for the support and advice given by Dr. Malaka Walpola by providing feedback on the presented work during the continuous progress evaluations.

Further I would like to thank Mr. Dhanika Perera, Chief Executive Officer, Mr. Chamika Weerasinghe, Chief Technology Officer, of Bhasha Lanka (pvt) Ltd for providing the, access to the mockups and case study details and advise in the development process of mobile domain.

Finally, I would like to thank my colleagues at MillenniumIT and Amazon, for covering my work and helping me to balance the workload. Without them, this project would not have been possible.

Special thanks go to all the UI/UX engineers and designers who participated in the survey and provided an invaluable input to my research.

Last but not least, I am grateful for all the people who supported me throughout this research in various means.

TABLE OF CONTENTS

DECLARATION i

ABSTRACT ii

ACKNOWLEDGMENT iii

TABLE OF CONTENTS iv

LIST OF FIGURES vii

LIST OF TABLES ix

LIST OF ABBREVIATIONS x

LIST OF APPENDICES xi

1 INTRODUCTION 1

1.1 Overview 2

1.2 Background 2

1.1.1 Evolution of Mobile 2

1.1.2 Testing User Interface 3

1.1.3 Mobile User Interface Development 3

1.3 Problem Statement 3

1.4 Proposed Solution 4

1.5 Objectives 4

1.1.4 General Objectives 5

1.1.5 Specific Objectives 5

1.6 Overview of the Document 5

2 LITERATURE REVIEW 7

2.1 Overview 8

2.2 Software Testing 8

2.3 GUI Testing 9

2.4 Automatic mockup validation for web applications 12

2.5 Image Comparison 13

2.6 Template Matching 13

2.7 Feature based matching algorithms 13
2.7.1 Global Feature Based Matching 14
2.7.2 Local Feature Based Matching 14
2.7.3 SIFT (Scale Invariant Feature Transform) 15
2.7.4 PCA-SIFT [32] 17
2.7.5 GSIFT 18
2.7.6 CSIFT [34] 18
2.7.7 SURF [36] 18
2.7.8 ASIFT [37] 19
2.7.9 Comparison of SIFT and its variants 19
3 METHODOLOGY 23
3.1 Proposed Solution 24
3.2 Workflow 24
3.3 Shortlisting algorithms 25
3.4 Modifying the algorithms to give a quantifiable score 26
3.4.1 Pixel matching score 26
3.4.2 Layout matching score 26
3.5 Developing the proof of concept application 28
3.6 Conducting the survey 28
3.7 Data Set 29
3.7.1 Use case 1: Hapan 29
3.7.2 Use case 2: E-Channelling 31
3.7.3 Use case 3: Ada dawasa 32
3.8 Selecting the underlying algorithm for layout based score in the framework .. 33
3.8.1 Statistical Analysis 33
3.9 Determining usability of pixel based matching 34
3.10 Creating the framework 34
4 SYSTEM ARCHITECTURE AND IMPLEMENTATION 35
4.1 Overview 36
4.2 High-level Flow 36
4.3 High level architecture of image comparison engine 37
4.3.1 Request Processor 37

4.3.2 Pixel Matcher 38
4.3.3 SIFT Matcher 38
4.3.4 CSIFT Matcher 39
4.3.5 PCA-SIFT Matcher 40
4.3.6 Blob Detection Matcher 40
4.3.7 Output Processor 43
4.4 Proof of Concept Application 43
4.5 Image Comparison Framework 45
5 RESULTS AND EVALUATION 46
5.1 Overview 47
5.2 Survey 47
5.3 Results 47
5.3.1 Layout Matching 47
5.3.2 Overall Matching 52
5.4 Evaluation 55
6 CONCLUSION 57
6.1 Research Contribution 58
6.2 Limitations 59
6.2.1 User interface vs User experience 59
6.2.2 Dynamic Data 59
6.2.3 Agile environment 59
6.2.4 UX design concepts 60
6.2.5 Orientation changes 60
6.3 Future Directions 61
7 REFERENCES 62
8 APPENDIX 67

LIST OF FIGURES

Figure 2.1 Stages of key point selection [31] 16
Figure 2.2 SIFT and its variants 19
Figure 3.1 Workflow 25
Figure 3.2 Design (left) and actual (right) Hapan-1 29
Figure 3.3 Design(left) and actual (right) Hapan-2 29
Figure 3.4 Design (left) and actual (right) Hapan-4 30
Figure 3.5 Design (left) and actual (right) Hapan-3 30
Figure 3.6 Design (left) and actual (right) E-channelling-2 31
Figure 3.7 Design (left) and actual (right) E-channeling - 1 31
Figure 3.8 Design (left) and actual (right) E-channeling-3 32
Figure 3.9 Design (left) and actual (right) Ada dawasa-1 32
Figure 3.10 Design (left) and actual (right) Ada dawasa -2 33
Figure 4.1 High level flow 37
Figure 4.2 High level architecture 37
Figure 4.3 Pixel to pixel matching 38
Figure 4.4 SIFT matching 39
Figure 4.5 Calculate the matching score for SIFT based algorithms 39
Figure 4.6 CSIFT matching 40
Figure 4.7 PCA-SIFT matching 40
Figure 4.8 Blob definition 41
Figure 4.9 Code segment for removing text from an image 42
Figure 4.10 Creating the monochrome version 42
Figure 4.11 Labeling definition for neighboring pixel pattern 42
Figure 4.12 Get Euclidean distance of the attributes 43
Figure 4.13 Pixel to pixel matching screen 44
Figure 4.14 Layout matching screen 44
Figure 4.15 Underlying implementation of the framework API 45
Figure 5.1 Layout matching scores 49
Figure 5.2 CSIFT Feature point matches 50
Figure 5.3 Design (left) and actual (right) E-Channeling-3 50

Figure 5.4 Feature point matching for E-channelling-3 51
Figure 5.5 Blob detection in matching in E-channeling -3 52
Figure 5.6 Pixel score provided by the system vs. overall score provided by the designers 53
Figure 5.7 The design (left) and the actual image (right) 54
Figure 5.8 The design (left) and the actual image modified (right) 54
Figure 5.9 Design (left) and actual image modified coloring the pixel differences (right) 55
Figure 6.1 Mockup (left) and actual (right) with grass background 60
Figure 6.2 Mockup (left) with grass background and actual (right) without grass background 60

LIST OF TABLES

Table 2.1 The cost of fixing the issue vs stage it was found [16] 9
Table 2.2 Comparison of most used GUI automated test tools.................................. 10
Table 2.3 Comparison between SIFT and its variants [30].. 20
Table 2.4 SIFT and variants algorithm Comparison [30] 21
Table 5.1 Layout Matching Survey Scores 48
Table 5.2 Layout matching algorithm scores 48
Table 5.3 Overall matching survey scores 52
Table 5.4 Results given by the framework for overall pixel by pixel score vs. results from designers 53

LIST OF ABBREVIATIONS

Abbreviation	Description
GUI	Graphical User Interface
UI/UX	User Interface/User Experience
MoGUT	Mobile GUI Testing
SUT	System Under Test
SIFT	Scale-Invariant Feature-Transform
CSIFT	Color Scale-Invariant Feature-Transform
PCA-SIFT	Principal Component Analysis Scale-Invariant Feature-Transform

LIST OF APPENDICES

Appendix	Description	Page
Appendix - A	Survey Sent to the designers	68

1 INTRODUCTION

1.1 Overview

This section introduces the problem domain by building up from the background of the mobile applications and how testing in mobile applications is important. This will lead up to the problem statement and the objectives of the research. This chapter finishes with an overview of upcoming chapters and what to expect from them.

1.2 Background

This section discusses the evolution of mobile devices to the current state where it has now become a gadget in day to day life, followed by the importance of testing the user interface of the mobile applications.

1.1.1 Evolution of Mobile

The Motorola Dyna TAC 8000X was the world's first truly portable commercial mobile phone which was introduced in 1983 [1]. From there to the sleek and stylish smart phones which we use now, mobile manufacturing industry has come a long way. Since the inception of smartphones, mobile application development has evolved from hardware-specific software to high-level platform operating systems [2]. With these new developments, mobile phones are no longer a luxury for the young generation as it has been a decade ago [3].

With the increasing usage of mobile phones, the mobile manufacturing industry is evolved into producing cheaper smart phones with higher specifications. Since the usage of mobile and its applications has become a habit for the current generation, mobile app development has also become one of the leading software development variations. Now the growth of the mobile application industry shapes how we live and work every day [4].

In earlier days' mobile applications are simple. User interacted with the mobile screen by searching through the menus by pressing buttons in their key pads. But now smart phones have dominated the mobile phone industry. Instead of typing or pressing buttons in the keyboard, users now interact with the mobile screen by simply touching it or talking to it.

1.1.2 Testing User Interface

Testing has become the most popular verification and validation method in industry [5]. It's the common belief that the sooner the defect or bug is found, the cheaper it is to fix [6]. The main focus on the testing is the functionality of the software. Research fields like portability, usability and visualization are usually neglected, but in turn can prove quite advantageous to the industry [5].

In most of the software applications, the priority in testing has been given to the functional tests. But mobile applications are different. While the functionality is also important, the user experience plays a vital role. Some researchers suggest that the graphical user interface of a mobile application is one of the core elements which decides the success or failure of an application [7].

1.1.3 Mobile User Interface Development

In web and desktop applications, the job of the UI/UX engineer or front-end engineer is to design and implement the front end of the system and then hand over to the back-end developer to implement the logic and functionality. But in mobile application development, the application size is significantly small when compared to desktop and web applications. Hence, they are usually developed by a single mobile application developer.

In the software industry user experience is highly valued. Almost all the companies get help from a UI/UX engineer or a designer to design the user interface of the applications. In mobile applications, they provide a mockup to the mobile developer to refer when building the application.

1.3 Problem Statement

Once the mockup is given to the mobile developer, they will develop the application to conform to the given mockup. Verification of the application's real user interface and its conformity to the mockup is done by the developer at the development stage and the quality assurer at the testing stage. This will not be verified by a user interface designer after the implementation and before being released due to resource constraints. This happens because majority of the small-scale mobile application

companies do not either have a resident UI/UX designer or availability of the UI/UX designer to participate in the testing and verification step. Since these verifications are done manually by quality assurance engineers or developers whose eyes are not trained to catch the visual and pixel level differences, they can be time consuming and prone to human error. This can eventually lead to user interfaces which are different from what was expected, when they are initially designed. Hence, it would be beneficial to all the three parties; designer, developer and the quality assurer to have an automated mechanism to verify that the application's real user interface doesn't deviate from the one given in the mockup.

The aim of this research is to come up with an automated mechanism to quantify how much the user interface of a mobile application matches with its mockup.

1.4 Proposed Solution

The proposed solution is an image comparison based user interface verification framework. When the mockup and a screenshot of the actual application is given to the framework, it will output a comparison score to quantify how much the actual application matches with the provided mockup. The comparison score will be computed using image comparison techniques and be done in two accuracy levels; overall and layout only.

Overall score can be used to identify overall difference between the design and actually implemented screen whereas layout level can be used to identify only the differences in the layout without concerning other aspects like color which get highlighted in overall matching.

1.5 Objectives

The main objective of this research is to develop a mechanism which allows developers and quality assurers to easily get a quantifiable score based on how much the user interface of a mobile application matches with its initial mockups.

1.1.4 General Objectives

- To come up with a framework to quantify the comparison between the actual user interface and designer provided mockups in an application.

1.1.5 Specific Objectives

- To research and analyze the computer vision algorithms which can be used to give a quantifiable score to identify similarities and differences between two images in layout level.
- Implement an overall matching algorithm using pixel based matching.
- Identify modifications which should be done to these algorithms to adapt the context of this research.
- Implement a proof of concept application which outputs the comparison scores given by the analyzed computer vision algorithms and/or their modifications.
- Conduct a survey using a group of designers to validate the scores given by the layout level algorithms and choose which algorithm will mimic an actual designer and be most functional.
- Use the same survey to identify how usable pixel base matching as an overall user interface verification mechanism.
- Implement a framework to provide the pixel based score and the layout based score between two images.

1.6 Overview of the Document

This document consists of six chapters. The first chapter gives the introduction to the research by presenting the background of mobile application development, testing and testing mobile applications. It presents the research problem which we are trying to solve and the objectives of the research.

The second chapter contains the findings from the related literature. Starting from the importance of software testing along with the GUI testing, it will continue to discuss the possible image comparison algorithms which can be used to build the layout level matching functionality of the framework.

The third chapter describes the identified methodology to solve this problem. This also involves why the certain algorithms are chosen over others and developing the proof of concept application along with the mechanism of validating the framework values.

Fourth chapter contains the information regarding the high-level flow of the framework, system architecture and implementation details of the proof of concept application.

In the fifth chapter we have included the results of the image comparison algorithms along with the results provided by the UX designers and discuss the similarities and differences between the scores. We have finished the fifth chapter with an evaluation of the research and how its objectives have been met.

Last chapter was dedicated to discuss the research contribution from the research along with the limitations of the approach we took and future directions where there is an opportunity to extend the framework.

2 LITERATURE REVIEW

2.1 Overview

This chapter will cover the findings from the related literature. We have started with software testing in general and doubled down on the GUI testing and automatic mockup validation frameworks as they are the most important parts for his research. From there, we have summarized the research we did on image comparison algorithms for layout matching. Based on their qualities we have shortlisted three candidates to use in our proof of concept application.

2.2 Software Testing

Software engineering research is focused on two key objectives: reducing the cost of production and improving the quality of products [9]. This is where software testing comes in. The process of examining a software application in order to identify differences between the required and existing specifications is called software testing [10].

Software testing comes under "Verification and Validation" part of software engineering practices. The process of assessing a software application to determine whether the outcome of a certain development phase match the expectations set at the start of the phase is called software verification [10]. In simpler terms, software verification answers the question, "Are we building the product, right?". The process of assessing whether a software application satisfy its specified requirements for building that application is called software validation [10]. This can be done at the end at the end or during a development phase. In simpler terms, software validation answers the question, "Are we building the right product?".

In today's industry we use various kind of software testing mechanisms to achieve our quality objectives. Few of these mechanisms are black box testing, white box testing, regression testing, performance testing, usability testing, acceptance testing, market comparison testing, etc. [11][12]

These tests will cover the development lifecycle from the requirement engineering stage to the maintenance stage. But testing is expensive. There are studies which shows that as much as fifty percent of the overall software development cost can be

associated with testing and related activities [13]. Better quality can be obtained by increasing the coverage of testing [14]. Better coverage means more test cases which makes manual testing more tiresome and tedious. The solution for this lies in automation. Automating software testing includes implementing and executing test scripts, verifying testing requirements and use automated test tools [15].

The main objective of the software testing is to find the defects as soon as possible. Table 2.1 gives an approximate idea of how the cost of fixing issues depends on the stage it was found. It proves the fact that the sooner the defects were found, the cheaper the cost will be.

Table 2.1 The cost of fixing the issue vs stage it was found [16]

<table>
<tr><th colspan="2" rowspan="2">Cost to fix a defect</th><th colspan="5">Time detected</th></tr>
<tr><th>Requir-ements</th><th>Archit-ecture</th><th>Construc-tion</th><th>System test</th><th>Post-release</th></tr>
<tr><td rowspan="3">Time intro duced</td><td>Requirements</td><td>1×</td><td>3×</td><td>5–10×</td><td>10×</td><td>10–100×</td></tr>
<tr><td>Architecture</td><td>–</td><td>1×</td><td>10×</td><td>15×</td><td>25–100×</td></tr>
<tr><td>Construction</td><td>–</td><td>–</td><td>1×</td><td>10×</td><td>10–25×</td></tr>
</table>

2.3 GUI Testing

Graphical User Interface (GUI) is the visual way of interacting with the software. It is a common part of most of today’s software applications. Since it allows various degrees of freedom to an end-user, it makes it challenging to the test designers to design test cases covering all the input interaction space of the graphical user interface [17].

As graphical user interfaces (GUIs) have become ubiquitous in almost every software system, the demand for GUI-level testing has been increased [18]. However, this brought a unique set of problems which makes the testers to seek a different approach as opposed to traditional software testing [19]. One example for this is the fact that traditional test coverage criteria is being applicable in GUI testing [19].

GUI is being often neglected in lower stages of testing and comes into play in latter stages. Typical GUI testing tools are based on capture and replay technique. However, with the increase of focus in this area, some new techniques were introduced. These include finite-state-machine based test models and event-flow based test models. Due to the complexity of these models and most of the use cases end up being infeasible problems when these models are used individually, the applicability of these modules are limited [17].

There are several GUI automated testing tools available for industry use. Couple of the best ones is as follows [7].

Table 2.2 Comparison of most used GUI automated test tools

Testing Tool	**Feature**	**input**	**Report Function**
abbot	• Measured via a test script GUI state • An interface for controlling the replay • Event-based testing	Java Application	Coverage Report
Guitar	• Provide a test case generator plug-in • Event flow measurement is useful	Java Application	Unsupported
Pounder	• Records test scripts and provides an interface for measuring the results	Java Application	Unsupported
Selenium IDE	• Records the actions of the tester using HTML script	Web UI	Unsupported

These and several other applications use the following methods to test GUIs [20]:

1. **Record/Playback**
   This is the method of conducting the tests by recording the events occurred in GUI. These events are created by interacting with an application using a mouse and/or keyboard input and by replaying these recorded events. Since this technique is a simple pattern many GUI test tools are developed using this.

2. **Capture/Replay**

   The GUI events are captured through the user setting examples and document them.

3. **Particulars Based**

   As the name suggests particulars-based tests executes the system based on graphical user interface particulars of the system. Compared with other test technologies these tests contain several conditions. Hence it is required to describe and summarize design particulars or needs explicitly.

4. **Beta Testing**

   A lot of enterprises use beta testing as a way of testing graphical user interface and it is the most popular test method for GUI testing. Typically, the software's beta version is released first with the promise of new functionality but with the risk of the software not being stable. This mechanism is being currently used with lot of software in the market. However, the issue here is that since this is done by a non-specialty testers composition, it is difficult to get the common users to use and test the app without having the required knowledge.

Visual GUI testing is another approach for testing GUIs. It uses the same mechanism as record and replay. However, instead of using GUI component coordinates or code to detect and interact with GUI bitmap components such as buttons and images, it uses image recognition [21]. The input of a typical visual GUI testing script is given automating mouse and keyboard commands to GUI bitmap components which are recognized through image recognition. The output is also detected using image recognition and the results are compared to expected results [21].

Another GUI testing method is GUI ripping. GUI Ripper dynamically interact with the applications user interface and construct a navigation model of the application and observes the changes to its state [22].

Even though there are many tools and frameworks which provide visual comparison based GUI testing, their scope was to verify that the event stimulus and transition gives the correct UI views. MoGUT is such a framework [10]. It detects the defects

in the event flow, based on the images of the next screen. Similarly, in most frameworks the image comparison is used to check the functionality and the event flow. But none has closely paid attention to verify the user interface. Recently, a new tool has come into play to automatically validate mockups in web applications.

2.4 Automatic mockup validation for web applications

Applittools Eyes is a commercially available mockup validation tool for web development [23]. In a typical web development workflow, the product manager or graphic designer provides the mockups for the website. Using Applittools eyes you can automatically match the web application you developed to the given mockups to ensure that it matches its expected design. It is a commercial software which provides an SDK for your test automation framework to use. Applitools Eyes SDK provides quick and easy integrations with existing test automation infrastructure like Selenium, Appium (Java, .Net, Ruby, Python and JS), Microsoft Coded UI and HP UFT/ QTP/ LeanFT (24).

This tool allows you to validate all the visual aspects of your application by automating them. For example, one simple test can validate all the fields on a given screen. Hence the developer or tester do not have to write separate test cases for each and every UI element on the screen. There are four layers of match levels for the visual comparison. The default match level is set to strict. But the user can override it through code. The match levels are as follows [25].

- Exact - Pixel to pixel comparison, for demonstration purposes and debugging, will fail a test if a pixel is not in place. (not recommended)
- Strict – Strict comparison is said to mimic the human eyes. Hence, only significant visual changes will be identified and changes which are small and not visible to the human eye will be ignored.
- Content – Content comparison identify the changes in content and ignores all the other differences such as style. If the website includes different styles which are not relevant to your test, this match level will be very handy.

- Layout – layout comparison as the name suggests detected only layout changes and ignore the changes in content. This is very useful when the pages include dynamic or localized content.

Even though this tool is available for web application, there are no similar tools we can use for mobile applications.

2.5 Image Comparison

In order to validate whether the user interface conform to the mockup provided by the designer, we plan to use image comparison techniques across two levels of accuracy; overall and layout.

The simplest similarity measure for overall matching consists of directly comparing the pixels between two images. Pixel based matching is rarely useful as it is extremely sensitive to minor transformations, both in geometry (shifts and rotations) and in imaging conditions (lighting or noise) [26].

Layout matching can be done using template matching techniques, neural networks and feature matching [27]. At this stage we eliminated the high complex neural network matching techniques since our requirement is only to match the layout of the image and doesn't involve identifying complex images.

2.6 Template Matching

Template matching is performed on scale normalized windows in pixel by pixel matching basis [27]. This process computes a numerical index indicating how well the template matches the image in a position by moving the template image to all possible positions in a larger source image [28].

2.7 Feature based matching algorithms

Feature based image matching algorithms can be categorized in to two main areas. [29].

1. Global Feature Based Matching Algorithms
2. Local Feature Based Matching Algorithms

Local feature based matching algorithms are more stable in comparison with global feature based matching algorithms [30]. Some of their real-world use cases are as follows:

1. Object/Texture Recognition
2. Object Category Recognition
3. Image Retrieval
4. Mining Video Data
5. Building Panoramas
6. Robot Localization

2.7.1 Global Feature Based Matching

Global features include contour representations, shape descriptors, and texture features. They are able to generalize an entire object with a single vector making their use in standard classification techniques straight forward. This means that global features have very compact representation of images where each image corresponds to a point in a high dimensional feature space. This allows them to use any standard classifier at the same time making them sensitive to clutter and occlusion [29].

2.7.2 Local Feature Based Matching

Local features are more robust to occlusion and clutter since they are computed at multiple points in an image. In case there are variable number of feature vectors per image, a specialized classification algorithm might be required to handle it [29].

Local feature-based matching algorithms includes feature detection and description. The characteristics of good local features are as follows [30]:

1. Feature detection has a high repeatability rate and high speed.
2. Feature description has a low feature dimension, which is easy to achieve quick matching and robustness to rotation, illumination, and viewpoint change.

There are many popular local feature based matching algorithms starting from SIFT (Scale-Invariant Feature Transform) where the original paper was referenced by

more than 5000. We have considered SIFT and some popular variants of SIFT algorithm to find a suitable matching technique to detect the differences and similarities between the mockup and the actual image.

2.7.3 SIFT (Scale Invariant Feature Transform)

This approach transforms an image into scale invariant features relative to local features. This process can be divided into four stages [31].

1. Scale-space extrema detection:

   This stage involves searching over all scales and image locations. To implement this process efficiently in order to identify the potential scale and orientation invariant interest points, a difference-of-Gaussian function is used.
2. Keypoint localization:

   In this stage each of the candidate locations identified from the previous stage are fitted with a detailed model to determine location and scale. The stability of the key points is the basis of selection.
3. Orientation assignment:

   One or more orientations are assigned to each key point location based on local image gradient directions. The rest of the computations will be carried out on the image data which has been transformed relative to this assigned orientation, scale and location for each feature. This provides the invariance to these transformations.
4. Keypoint descriptor:

   The local image gradients are computed at the selected scale in the region around each keypoint. These are transformed into a representation that allows for significant levels of local shape distortion and change in illumination.

These feature extractions are done in a cascading way where the costlier operations are applied only at the locations which passed the initial tests in order to minimize the cost. [31]

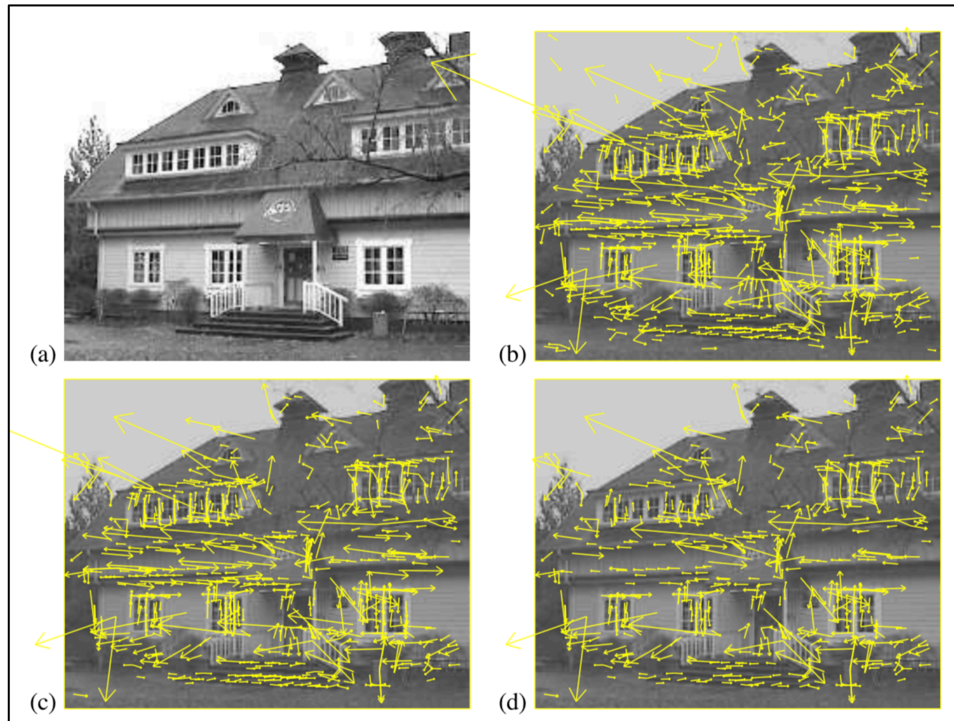


Figure 2.1 Stages of key point selection [31]

Figure 2.1 represents the stages of key point selection. [31]. Keypoints are displayed as vectors in the above image. These vectors indicate scale, orientation and location.

(a) The original image of 233x189 pixel.

(b) The first 832 keypoint locations at maxima and minima of the difference-of-Gaussian function.

(c) Remaining 729 keypoints after minimum contrast threshold being applied to.

(d) Remaining 536 keypoints after applying an additional threshold on ratio of principal curvatures.

The keypoint with minimum Euclidean distance is defined as the nearest neighbor to the invariant descriptor vector. In the database of keypoints the nearest neighbor is considered to be the best candidate to match for each keypoint when matching two images [31].

The key points are invariant to image rotation and scale. They are also robust across a substantial range of affine distortion, change in illumination and addition of noise.

Since the key points are detected over a complete range of scales, the large key points work well for images with noise and blur where small key points work well for small objects and objects with high occlusion. The efficiency of this computation is very high to an extent that thousands of key points can be derived from a typical image in near real time using standard PC hardware [31].

After SIFT was presented, many subsequent researchers followed and built new algorithms on top of it adjusting the performance of key point detection, descriptor establishing and image feature matching.

Among various other SIFT based improved algorithms, following algorithms are considered for this research.

1. PCA-SIFT
2. GSIFT
3. CSIFT
4. SURF
5. ASIFT

2.7.4 PCA-SIFT [32]

Similar to the standard SIFT descriptor, PCA-SIFT for local descriptors accepts the identical input of sub-pixel location, dominant orientations and the scale of the key point. Then it extracts a 41×41 patch at the given scale, centered over the keypoint, and rotated to align its dominant orientation to a canonical direction.

The process for PCA-SIFT can be divided in to three stages:

1. Express the gradient images of local patches by pre computing an eigenspace.
2. For a given patch, compute the local image gradient.
3. Derive a compact feature vector by projecting the gradient image vector using the eigenspace. Although PCA-SIFT feature vector is considerably small compared to the standard SIFT feature vector, the same matching algorithms can be used with it. The correspondence of the two feature vectors to the same key point in different images is determined using the Euclidean distance between those two vectors.

In comparison to standard SIFT algorithm, PCA-SIFT is more suitable for capturing variation in the gradient image of a localized key point in orientation, scale and space.

2.7.5 GSIFT

GSIFT expands the standard SIFT algorithm by combining local SIFT descriptor with a global context vector which is similar to shape contexts. This global context adds value by differentiating between local features that have similar local appearance. This will give a descriptor which is robust to ambiguities in local appearances and non-rigid transformations [33].

2.7.6 CSIFT [34]

Standard SIFT algorithm uses a gray scale image. CSIFT, also known as COLORED SIFT, extends the SIFT algorithm by embedding the color information in the in descriptors. This makes the features robust to variations in color by using a color variance model [35]. Robustness to geometrical variations are achieved similar to SIFT.

2.7.7 SURF [36]

SURF, which is also known as Speeded-Up Robust Features contains feature descriptors relying on,

1. Gaussian second derivative mask
2. Local Haar wavelet responses

Haar wavelet is a sequence of rescaled "square-shaped" functions which together form a wavelet family or basis.

SURF uses ingenious box filter and integral image tricks to quickly find the features and then describe them robustly using Haar wavelets. This has made SURF faster than its predecessor SIFT.

2.7.8 ASIFT [37]

ASIFT(Affine-SIFT) extends SIFT into a fully affine invariant image comparison algorithm by simulating the camera optical direction and scale, and normalizing the rotation and the translation. The existence of large transition tilts between two images taken from two different viewpoints inspired the exploration of full invariance.

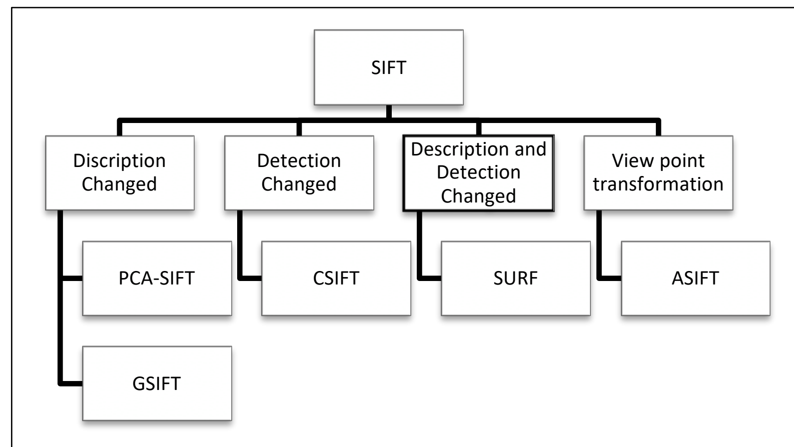


Figure 2.2 SIFT and its variants

2.7.9 Comparison of SIFT and its variants

Figure 2.2 gives a high-level summary of SIFT and its variants followed by Table 2.3 [30] which gives a detailed overview of differences between these local feature based matching algorithms.

Table 2.3 Comparison between SIFT and its variants [30]

	Key point Detection		Key point Description		
	Scale-space	**Selection**	**Main direction**	**Feature Extraction**	**#Dimensions**
SIFT	Different-scale images convoluted with a Gaussian function	Detect extrema in DoG space; do non-maxima suppression	Calculate a gradient amplitude of a square area; regard the direction with the maximum gradient strength as the main direction	Divide a 16×16 region into 4×4 sub-regions; create a gradient histogram for each sub-region	128
PCA-SIFT	Equal to SIFT	Equal to SIFT	Equal to SIFT	Extract a 41×41 patch; form a 3042-dimension vector; use a project matrix to multiply with it	20 or less
GSIFT	Equal to SIFT	Equal to SIFT	Equal to SIFT	For each keypoint, create a vector consisting of SIFT description and a global texture vector	188
CSIFT	Replace grayscale with color invariant; convolute with a Gaussian function	Equal to SIFT	Equal to SIFT	Equal to SIFT	384
SURF	Different-scale box filter convoluting with an original image	Use a Hessian matrix to determine candidate keypoints; do non-maxima suppression	Calculate a Haar wavelet response in x and y directions of each sector in a circular area; regard the direction with maximum norm as the main direction	Divide a 20×20s region into 4×4s sub-regions; calculate a Haar wavelet response	64
ASIFT	After a preprocessing - viewpoint transformation, follow SIFT's steps (i.e., the same as SIFT)				

Table 2.4 has given a qualitative summarization on the algorithms on following areas. [30]

1. Scale and rotation
2. Illumination
3. Blur
4. Affine
5. Time Cost

The scale of evaluation is Fair < Good < Better < Best

Table 2.4 SIFT and variants algorithm Comparison [30]

Algorithm	**Scale & Rotation**	**Illumination**	**Blur**	**Affine**	**Time Cost**
SIFT	Best	Better	Good	Good	Better
PCA-SIFT	Better	Better	Better	Good	Better
GSIFT	Good	Best	Best	Good	Better
CSIFT	Best	Good	Better	Better	Good
SURF	Fair	Fair	Fair	Fair	Best
ASIFT	Good	Fair	Fair	Best	Fair

According to our use case of comparing the mockup with the actual image of a mobile user interface, we only have to consider the scale, affine and time cost. Blur and illumination are not considered because the screenshots of the actual application and mockup do not have blur and illumination since both of them are machine generated and not the actual real-life images. The other factor which is less likely to be change in a mockup of a user interface design and a screenshot of a real-life application is the orientation. The most common UI mistakes in development are on the lines of the position and scale not orientation.

1. Scale
2. Affine
3. Time Cost

Based on the comparison we have three candidate algorithms who has either best or better performance in the qualities that we are interested in.

1. SIFT
2. PCA-SWIFT
3. CSIFT

3 METHODOLOGY

3.1 Proposed Solution

The proposed solution for our research problem is to build a framework which will quantify the comparison between the actually implemented user interface and the designer provided mockups.

3.2 Workflow

When implementing the proposed solution, we decided to go ahead with two accuracy levels; overall and layout only.

For overall comparison between two images, we chose pixel based matching where every pixel of one image is matched with the corresponding pixels in the other image and come up with a score based on the similarity.

Layout level comparison is not as straight forward as the overall pixel based comparison. There were lot of available algorithms which we can use as a base algorithm to compare the similarity between two images. We researched on these available algorithms and chose a set of shortlisted candidates. We added our own very simple blob detection algorithm to the list of shortlisted algorithms as well.

After selecting the algorithms, we built a proof of concept application which will provide a set of scores based on similarities between two images, the two images being the screenshot of the actual application and the mockup provided by the designer.

When building this application, we have to slightly modify the existing algorithms to give a quantifiable score based on the comparison similarities. This score is given as a percentage. The proof of concept application will output scores using all the shortlisted algorithms for layout level and overall pixel based algorithms.

After the proof of concept application is implemented, we applied three industrial use cases and got the comparison scores. We used the same use cases in a survey for a selected group of user interface designers and asked them a comparison score based on layout matching and overall matching between the mockup and actually developed application.

For layout matching, we used the scores provided by the designers as a benchmark to the scores given by the various algorithms and select the best algorithm to be used in our framework.

For overall matching, we compared the scores provided by the designers and the pixel based scores provided by the proof of concept application. The purpose of this comparison is to determine the usability of the pixel based matching as a mechanism to match the overall similarity between the application and the mockup.

Figure 3.1 visualize this workflow.

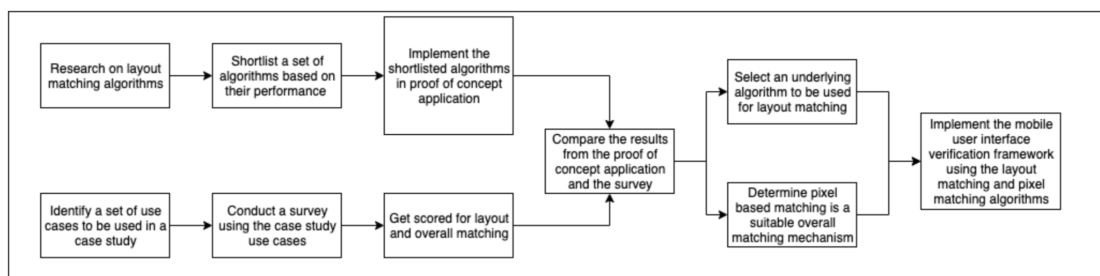


Figure 3.1 Workflow

3.3 Shortlisting algorithms

When analyzing the local feature based matching algorithms in the literature review, we had three candidates who had the best performance of the areas we are interested in, namely scale and rotation, affinity, and time cost.

1. SIFT
2. PCA - SIFT
3. CSIFT

We implemented these algorithms as candidates to be used in our framework for layout matching comparison score.

We also implemented a basic blob detection algorithm. Typically, mobile applications have very simple user interface to allow the user to quickly do their intended task. With the rise of the material design [38], the amount of the widgets available in the user interface has been made very minimal. However, similar to

material design for Android, Apple has its own set of design guidelines [39]. Although fundamentally different, the similarity is that they both suggest a set of basic components to be used in designing and developing. These components can be easily detected by a simple blob detection algorithm and matched based on their size and location in the screen.

3.4 Modifying the algorithms to give a quantifiable score

All of the selected algorithms provide a mechanism to find matches between two images. To make it a quantifiable score we need to use the matching results and formulate an algorithm to get a score based on differences or similarities.

3.4.1 Pixel matching score

This is very easy in the pixel to pixel matching algorithm. Since we have the number of pixels which are different between images, we only have to calculate the percentage of difference and similarity from it.

$$difference = \frac{Number\ of\ unmatched\ pixels}{All\ the\ pixels\ in\ the\ image} \times\ 100\%$$

$$similarity = 100\% - difference\%$$

3.4.2 Layout matching score

Quantifying layout matching is not easy as quantifying the pixel based matching. The initial approach we can take to find the similarities between the two images is to compare their feature point matches to total number of feature points.

$$Layout\ Matching\ Score = \frac{No\ of\ matching\ key\ points}{Max(No\ of\ keypoints\ in\ mockup, No\ of\ keypoints\ in\ actual)} \times 100\%$$

Number of matching keypoints are divided from the maximum value between the keypoints in the mockup and the keypoints in actual rather than taking the number of keypoints in the mockup as the reference. This is to avoid getting a higher score when all the keypoints in the mockup matches with the actual image, but actual image has some other object which gives an extra set of key points and vice versa

where all the keypoints in the actual image matches with the mockup but mockup has a set of extra key points.

The drawback of this approach is that the difference between the matching keypoints are not considered. The percentage of difference between the matching keypoints can be calculated from summing up the Euclidean distance of each keypoint based on position and scale.

$$Distance(f_1, f_2) = \sqrt[2]{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (scale_1 - scale_2)^2}$$

These distances are summed up for all the matching feature points.

$$Total\ Distance = \sum distance(f1, f2)$$

The maximum distance between two feature points is taken as when two feature points are located in two opposite ends of the image. Maximum scale difference is taken by considering the feature point's scale covers whole screen

$$Maximum\ Distance(f_1, f_2) = \sqrt[2]{(width)^2 + (height)^2 + (maxScale)^2}$$

Total distance percentage between the matching keypoints can be calculated as the following.

$$Total\ difference\ percentage = \frac{total\ distance}{maximum\ distance \times number\ of\ matches} \times 100\%$$

Total similarity percentage is calculated using the total difference percentage

$$Total\ similarity\ percentage = 100\% - Total\ difference\ percentage$$

The drawback on calculating only the total similarity percentage as the quantifiable score to match the similarity of two images is that it doesn't take the non-matching feature points into the account.

The actual similarity of the two images is based on both these scores.

$$
\begin{aligned}
&Similarity\ between\ two\ images \\
&\qquad = percentage\ matches\ of\ feature\ points \\
&\qquad \times percentage\ similarity\ of\ matched\ feature\ points
\end{aligned}
$$

A similar calculation is carried out for blob detection as well.

$$
\begin{aligned}
&Similarity\ between\ two\ images \\
&\qquad = percentage\ of\ matching\ blobs \\
&\qquad \times\ percentage\ similarity\ of\ the\ matching\ blobs
\end{aligned}
$$

3.5 Developing the proof of concept application

When developing the proof of concept application, we have used a simple approach where the user can upload two images representing a mockup and an actual screenshot. It uses two different tabs in order to differentiate between pixel to pixel based score and layout scores. Dependent on the tab you are in, you can calculate the pixel to pixel score or layout matching score.

For the layout matching score, you can select what algorithm you want to use among the four shortlisted algorithms.

The purpose of this proof of concept application is to visualize the comparison score when applying the shortlisted algorithms. This can also act as an example implementation of how the framework can be used in a testing framework.

3.6 Conducting the survey

We have conducted a survey using three industrial case studies. Each case study represents a different type of mobile application which is already on the market. The rationale of selecting these applications are as follows.

1. "Hapan": This is an image heavy application designed for kids. Unlike other applications, this has the least amount of text and used bold colors.
2. "E-Channeling": This application is a very text intensive application which has dynamic data loading. Unlike the "Hapan" application, this application has several mobile application widgets like menus, buttons, dialog boxes, etc.

3. “Ada Dawasa”: This application represents the middle ground of the above two applications. It has images and widget elements in the same user interface.

A selected set of local and foreign user interface designers have been asked to fill a survey and provide a layout score and overall score by comparing the mockups and actual images of these mobile applications (Appendix A). The rationale behind including the foreign user interface designers is that they do not have the context represent in the text so that the focus will be only limited to comparing the layout and overall similarity, not the wording on the text as such.

3.7 Data Set

For the survey and get the scores of the framework, we used a data set of ten image pairs. These images are taken from three real applications which is already in the Sri Lankan market.

3.7.1 Use case 1: Hapan

Figure 3.2 Design (left) and actual (right) Hapan-1

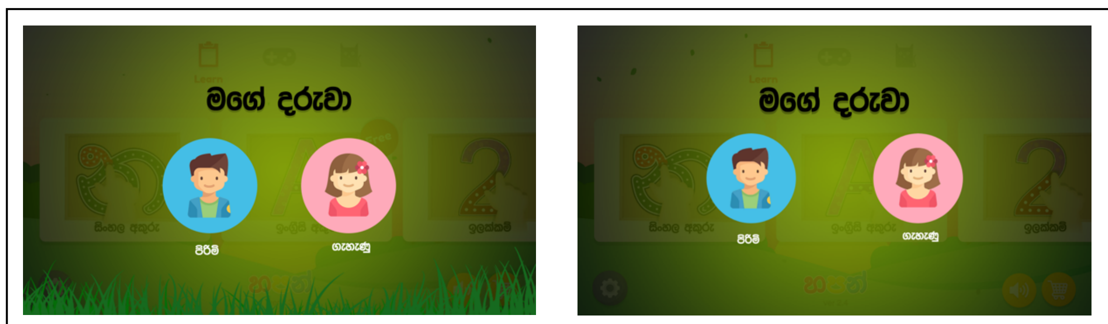


Figure 3.3 Design(left) and actual (right) Hapan-2

Figure 3.5 Design (left) and actual (right) Hapan-3

Figure 3.4 Design (left) and actual (right) Hapan-4

3.7.2 Use case 2: E-Channelling

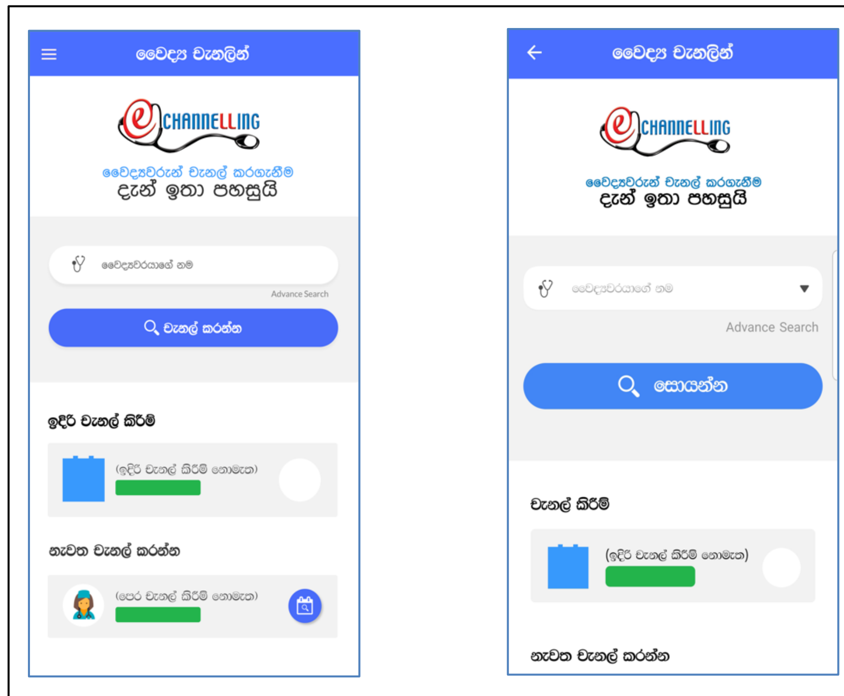


Figure 3.7 Design (left) and actual (right) E-channeling - 1

Figure 3.6 Design (left) and actual (right) E-channelling-2

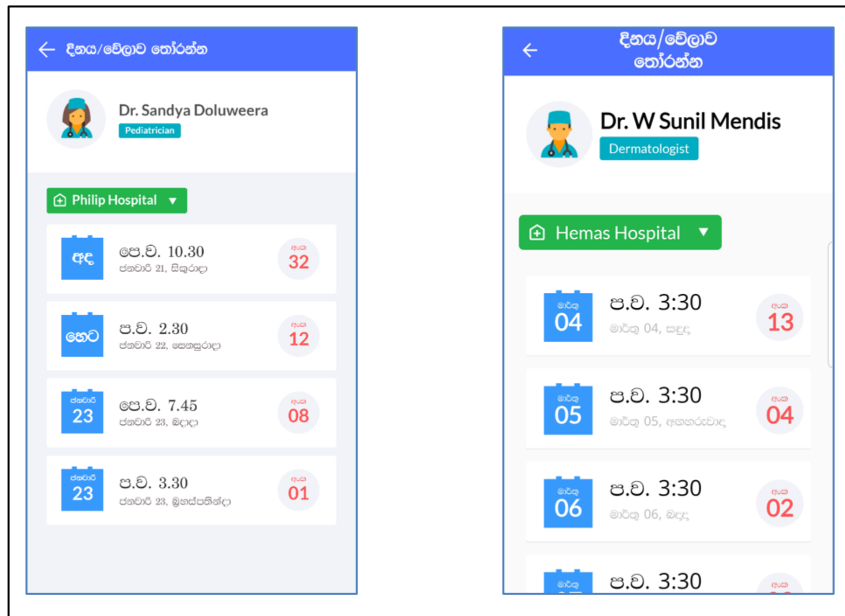


Figure 3.8 Design (left) and actual (right) E-channeling-3

3.7.3 Use case 3: Ada dawasa

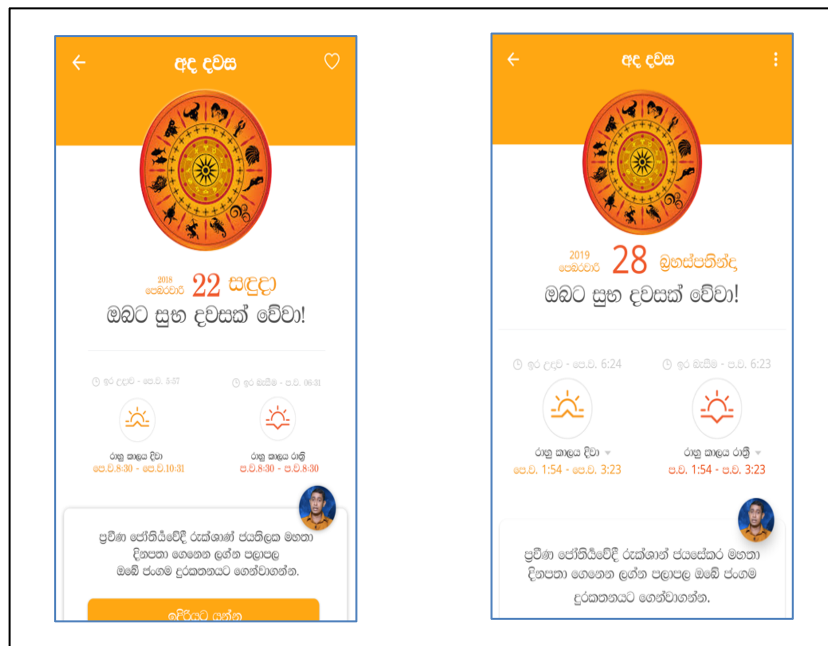


Figure 3.9 Design (left) and actual (right) Ada dawasa-1

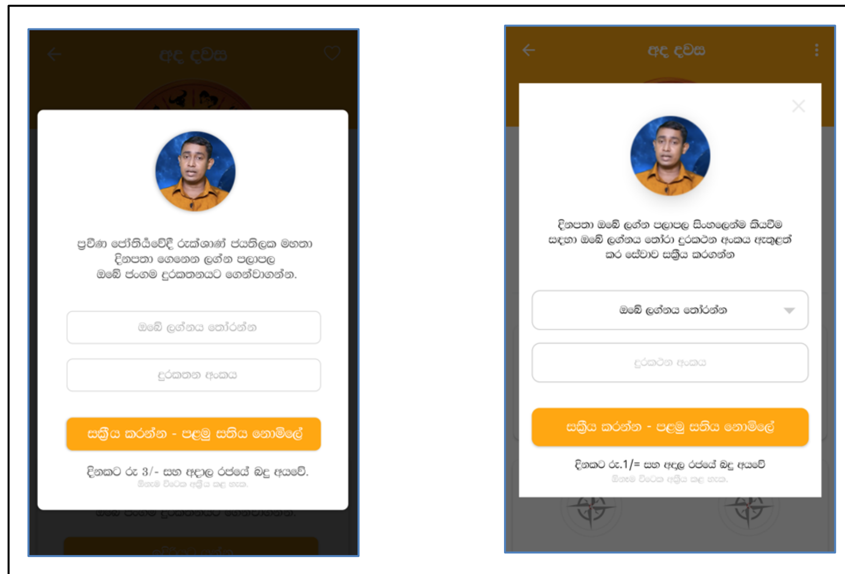


Figure 3.10 Design (left) and actual (right) Ada dawasa -2

3.8 Selecting the underlying algorithm for layout based score in the framework

To select an underlying algorithm for the layout based score in our framework, we have compared the layout comparison scores we received from the survey with the values provided by our algorithms.

The best algorithm whose scores are similar to the scores from the designers is selected as to provide the layout based score for our framework.

3.8.1 Statistical Analysis

For the analysis of the results from the survey we are using mean of the sample and sample standard deviation to measure variation because we want to make a statement about the population standard deviation from which the sample is drawn,

$$s = \sqrt{\frac{\sum(x - \bar{x})}{n - 1}}$$

$\bar{x}$=sample mean

s=sample standard deviation

n=number of scores in sample

We will use a t-distribution to get the upper and lower bounds of the values with 95% confidence level

3.9 Determining usability of pixel based matching

Pixel based matching is very restrictive. We compared the score which we got from pixel based matching to the overall score of comparison by the designers to determine whether it will match the designer's idea of overall similarity. This includes several other aspects like color, orientation, etc. which are not considered in layout matching.

3.10 Creating the framework

The objective of this research is to provide an automated framework to get a quantifiable score to capture similarities between the mockup and actual user interface on the basis of pixel to pixel and layout. This framework should be implemented as a library where the user can easily import and use in their automated testing. It should have a clear API to get the pixel based score and layout score.

To extend the functionality and be flexible for testing frameworks to use the underlying algorithm they prefer instead of the algorithm we suggested, we have also included the following four algorithms as well.

1. SIFT
2. PCA-SIFT
3. CSIFT
4. Blob Detection and matching

4 SYSTEM ARCHITECTURE AND IMPLEMENTATION

4.1 Overview

We followed a slightly different method in architecting the framework and the proof of concept application. This chapter discusses the specifics in how we design the proposed image comparison framework and the proof of concept application.

4.2 High-level Flow

The image comparison framework is designed to be used as a library which provides API methods to get an overall comparison score and a layout comparison score. We also made a design decision to extend this API to output comparison scores based on all of our candidate algorithms in layout comparison. Not only this will provide flexibility to users in selecting an underlying algorithm which is specific to their use cases overriding our general recommendation; this also allows us to use the framework to build the proof of concept application in the first place.

The first iteration of the framework included the implementation of all the candidate algorithm based scores and pixel based overall value score. This made it possible for the proof of concept application to use the framework.

After we got the results of the survey and analyzed the scores given by the designers and chosen a suitable underlying algorithm, we implemented the remaining interface for the layout comparison score by outputting the scores from the selected underlying algorithm.

The input for the API methods will be the mockup and actual image's Buffered Image object. And the output of these API methods will be a single integer score between 0-100 based on the method they choose.

High level of the flow is given in Figure 4.1

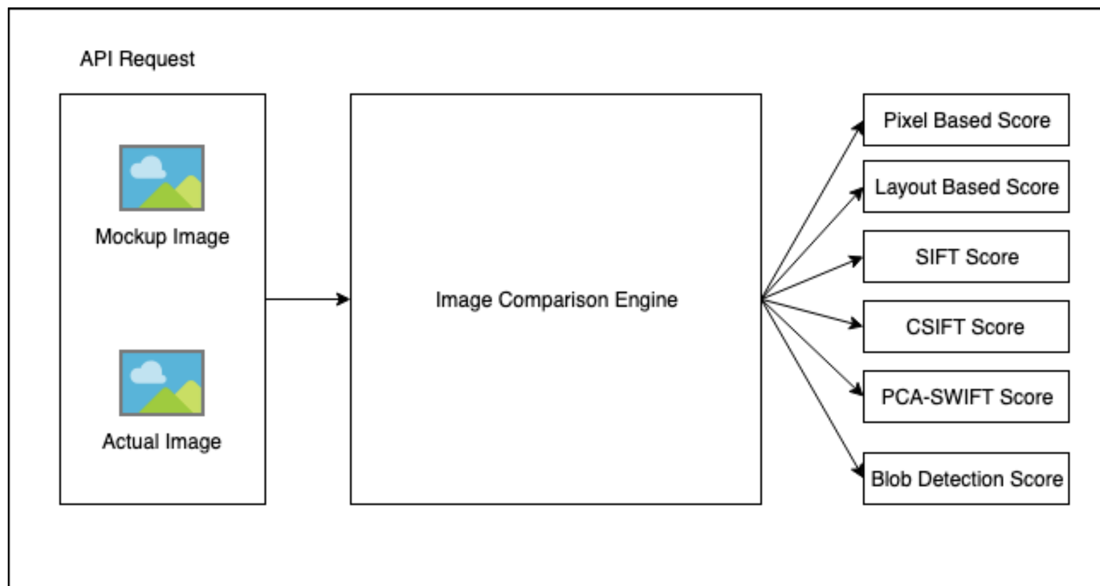


Figure 4.1 High level flow

4.3 High level architecture of image comparison engine

Image comparison engine consist of a request processor, matcher modules and output processor (Figure 4.2)

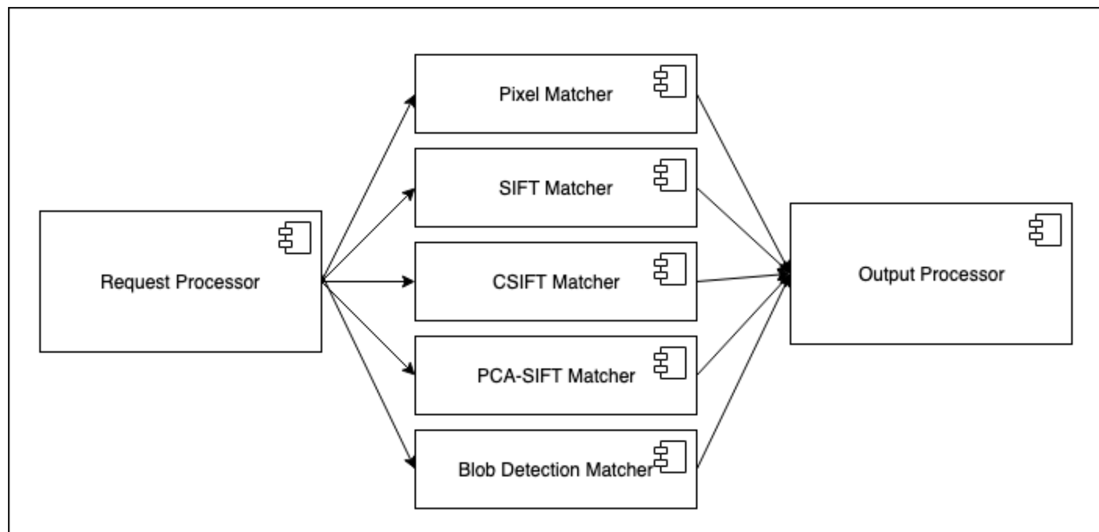


Figure 4.2 High level architecture

4.3.1 Request Processor

The main responsibility of the request processor is to validate the input request and send to the relevant matcher module.

4.3.2 Pixel Matcher

This module gets two images and match them pixel by pixel and return a pixel matching score to the output processor. This module is very simple to implement. High-level overview of the method is given in Figure 4.3.

```
private static int getPixelToPixelScore(BufferedImage design, BufferedImage actual) {
    int score = 0;
    int differences = 0;
    for (int i = 0; i < design.getWidth(); i++) {
        for (int j = 0; j < design.getHeight(); j++) {
            if (design.getRGB(i, j) != actual.getRGB(i, j)) {
                differences += 1;
            }
        }
    }
    score = differences * 100 / (design.getWidth()*design.getHeight()) ;
    return score;
}
```

Figure 4.3 Pixel to pixel matching

4.3.3 SIFT Matcher

This module uses the Difference of Gaussian (DOG) SIFT to identify the feature points of the two images. To identify the feature points, the buffered images are first converted to MBF (Multi Band Floating Point) images. A floating-point image is a greyscale image which represents each pixel as a value between 0 and 1. A multi band floating point image has a list of floating-point images to represent each band. For this we have taken the color space as RGB which represent three bands, RED, GREEN, BLUE.

When the feature points are identified, they are sent to a Coordination Distance Matcher. This matcher considers the feature point distances in considering the scale, orientation, x and y coordinates.

```java
public double getSIFTMatchingScore(BufferedImage design, BufferedImage actual) {
    MBFImage query = ImageUtilities.createMBFImage(design, true);
    MBFImage target = ImageUtilities.createMBFImage(actual, true);

    DoGSIFTEngine engine = new DoGSIFTEngine();

    LocalFeatureList<Keypoint> queryKeypoints = engine.findFeatures(query.flatten());
    LocalFeatureList<Keypoint> targetKeypoints = engine.findFeatures(target.flatten());

    LocalFeatureMatcher<Keypoint> matcher = new CoordinationDistanceMatcher<>(8);

    matcher.setModelFeatures(queryKeypoints);
    matcher.findMatches(targetKeypoints);
    List<Pair<Keypoint>> matches = matcher.getMatches();

    displayMatches(query, target, queryKeypoints, matcher.getMatches(), RGBColour.RED, "SIFT");

    return calculateMatchesScore(design, queryKeypoints.size(), matches);
}
```

Figure 4.4 SIFT matching

```java
private double calculateMatchesScore(BufferedImage design, int designPointCount, List<Pair<Keypoint>> matches) {
    if (designPointCount == 0) {
        return 0;
    }
    int matchesCount = matches.size();
    double sum = 0;
    for (Pair<Keypoint> pair : matches) {
        sum += calculateDiff(pair);
    }
    double maxDiff = maxDiff(design.getWidth(), design.getHeight());
    sum += (designPointCount - matchesCount) * maxDiff;

    return 100.0 - ((sum * 100 / designPointCount) / maxDiff);
}
```

Figure 4.5 Calculate the matching score for SIFT based algorithms

4.3.4 CSIFT Matcher

This module matches the two images based on the CSIFT local feature matching algorithm. Figure 4.6 represents the overview of the implementation and 4.5 represents calculating the score from the list of matching key points. Similar to SIFT, this also uses a Coordination Distance Matcher.

```java
public double getCSIFTMatchingScore(BufferedImage design, BufferedImage actual) {
    MBFImage query = ImageUtilities.createMBFImage(design, true);
    MBFImage target = ImageUtilities.createMBFImage(actual, true);
    DoGColourSIFTEngine engine = new DoGColourSIFTEngine();

    LocalFeatureList<Keypoint> queryKeypoints = engine.findFeatures(query);
    LocalFeatureList<Keypoint> targetKeypoints = engine.findFeatures(target);

    LocalFeatureMatcher<Keypoint> matcher = new CoordinationDistanceMatcher<>(8);

    matcher.setModelFeatures(queryKeypoints);
    matcher.findMatches(targetKeypoints);
    List<Pair<Keypoint>> matches = matcher.getMatches();

    displayMatches(query, target, queryKeypoints, matcher.getMatches(), RGBColour.BLUE, "CSIFT");

    return calculateMatchesScore(design, queryKeypoints.size(), matches);
}
```

Figure 4.6 CSIFT matching

4.3.5 PCA-SIFT Matcher

In PCA-SIFT matcher the, the local feature points are identified using the basic SIFT algorithm. The only difference here is instead of using a Coordination Distance Matcher, a PCA algorithm based matcher is used.

```java
public double getPcaSIFTMatchingScore(BufferedImage design, BufferedImage actual) {
    MBFImage query = ImageUtilities.createMBFImage(design, true);
    MBFImage target = ImageUtilities.createMBFImage(actual, true);
    DoGSIFTEngine engine = new DoGSIFTEngine();

    LocalFeatureList<Keypoint> queryKeypoints = engine.findFeatures(query.flatten());
    LocalFeatureList<Keypoint> targetKeypoints = engine.findFeatures(target.flatten());

    LocalFeatureMatcher<Keypoint> matcher = new PcaMatcher<>();

    matcher.setModelFeatures(queryKeypoints);
    matcher.findMatches(targetKeypoints);
    List<Pair<Keypoint>> matches = matcher.getMatches();

    displayMatches(query, target, queryKeypoints, matcher.getMatches(), RGBColour.CYAN, "PCA-SIFT");

    return calculateMatchesScore(design, queryKeypoints.size(), matches);
}
```

Figure 4.7 PCA-SIFT matching

4.3.6 Blob Detection Matcher

A blob for the blob detection matcher is defined with the following parameters. Figure 4.8 shows how this is defined in the code.

1. X coordinate maximum value – X_{max}
2. X coordinate minimum value – X_{min}
3. Y coordinate maximum value – Y_{max}
4. Y coordinate minimum value – Y_{min}

5. Mass (in pixels)

```
public static class Blob {

    public int xMin;
    public int xMax;
    public int yMin;
    public int yMax;
    public int mass;

    public Blob(int xMin, int xMax, int yMin, int yMax, int mass) {
        this.xMin = xMin;
        this.xMax = xMax;
        this.yMin = yMin;
        this.yMax = yMax;
        this.mass = mass;
    }
}
```

Figure 4.8 Blob definition

When blobs are extracted, matched are identified by making a matrix with Euclidean distances between the parameters of the two blobs. A match is defined as the pair which has the lowest score in the matrix.

$$distance(b1, b2) = \sqrt[2]{\begin{array}{c}(x1_{min} - x2_{min})^2 + (x1_{max} - x2_{max})^2 \\ +(y1_{min} - y2_{min})^2 + (y1_{max} - y2_{max})^2 \\ +(m1 - m2)^2\end{array}}$$

When detecting the blobs detecting text differences will make the score lower as the algorithm identify a letter as a blob. In layout matching we are not considering the text since it is subject to change in most use cases. Hence, we have converted regions of text to another blob. We have identified text using SWT detector and marked the set of text as separate blobs [40].

```java
public static BufferedImage removeTexts(BufferedImage bi) {
    final SWTTextDetector detector = new SWTTextDetector();
    detector.getOptions().direction = SWTTextDetector.Direction.Both;
    detector.getOptions().intensityThreshold = 0.12F;

    final MBFImage image = ImageUtilities.createMBFImage(bi, false);
    detector.analyseImage(image.flatten());
    for (final LineCandidate line : detector.getLines()) {
        image.drawShapeFilled(line.getRegularBoundingBox(), RGBColour.BLUE);
    }
    return ImageUtilities.createBufferedImageForDisplay(image);
}
```

Figure 4.9 Code segment for removing text from an image

We first identify the blobs in each image. To do this we first create a monochrome version using basic threshold technique given in Figure 4.10.

```java
byte[] monochromeData = new byte[width * height];
int srcPtr = 0;
int monoPtr = 0;

while (srcPtr < srcData.length) {
    int val = ((srcData[srcPtr] & 0xFF) + (srcData[srcPtr + 1] & 0xFF) + (srcData[srcPtr + 2] & 0xFF)) / 3;
    monochromeData[monoPtr] = (val > 128) ? (byte) 0xFF : 0;

    srcPtr += 3;
    monoPtr += 1;
}
```

Figure 4.10 Creating the monochrome version

The next step is labeling blobs using the neighboring pixel pattern. In our case, this labeling will give us a list of blobs

```java
// This is the neighbouring pixel pattern. For position X, A, B, C & D are checked
// A B C
// D X
srcPtr = 0;
int aPtr = -width - 1;
int bPtr = -width;
int cPtr = -width + 1;
int dPtr = -1;
```

Figure 4.11 Labeling definition for neighboring pixel pattern

Once the pointers are defined, this will iterate through pixels looking for connected regions and assigning labels. A matrix is created with the two list of blobs from the mockup and the actual to identify the matching blobs. The matching score is determined by calculating Euclidean distance between two blobs considering all the attributes including mass itself (Figure 4.12).

```java
public static double getEucledeanDistance(BlobFinder.Blob blob1, BlobFinder.Blob blob2) {
    return Math.sqrt(
                    Math.pow((blob1.xMin - blob2.xMin), 2) +
                    Math.pow((blob1.xMax - blob2.xMax), 2) +
                    Math.pow((blob1.yMin - blob2.yMin), 2) +
                    Math.pow((blob1.yMax - blob2.yMax), 2) +
                    Math.pow((blob1.mass - blob2.mass), 2)
    );
}
```

Figure 4.12 Get Euclidean distance of the attributes

When finalizing matches, we use a threshold to avoid two different blobs to identify within each other because they are not matched with other blobs.

4.3.7 Output Processor

This is just an abstraction layer to receive the result do the result validation and return the output.

4.4 Proof of Concept Application

This is a simple application which will provide a front end to upload the two images and show the results. Figure 4.13 gives you the pixel matching view in the proof of concept application and Figure 4.14 gives you the layout matching view.

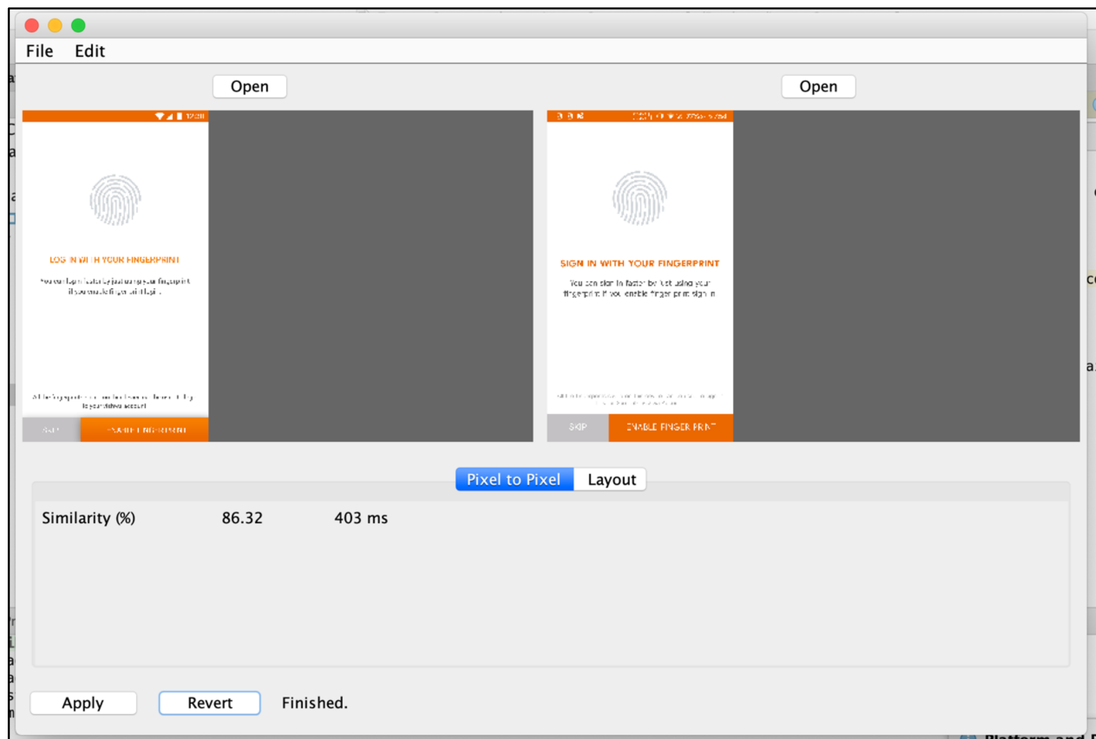


Figure 4.13 Pixel to pixel matching screen

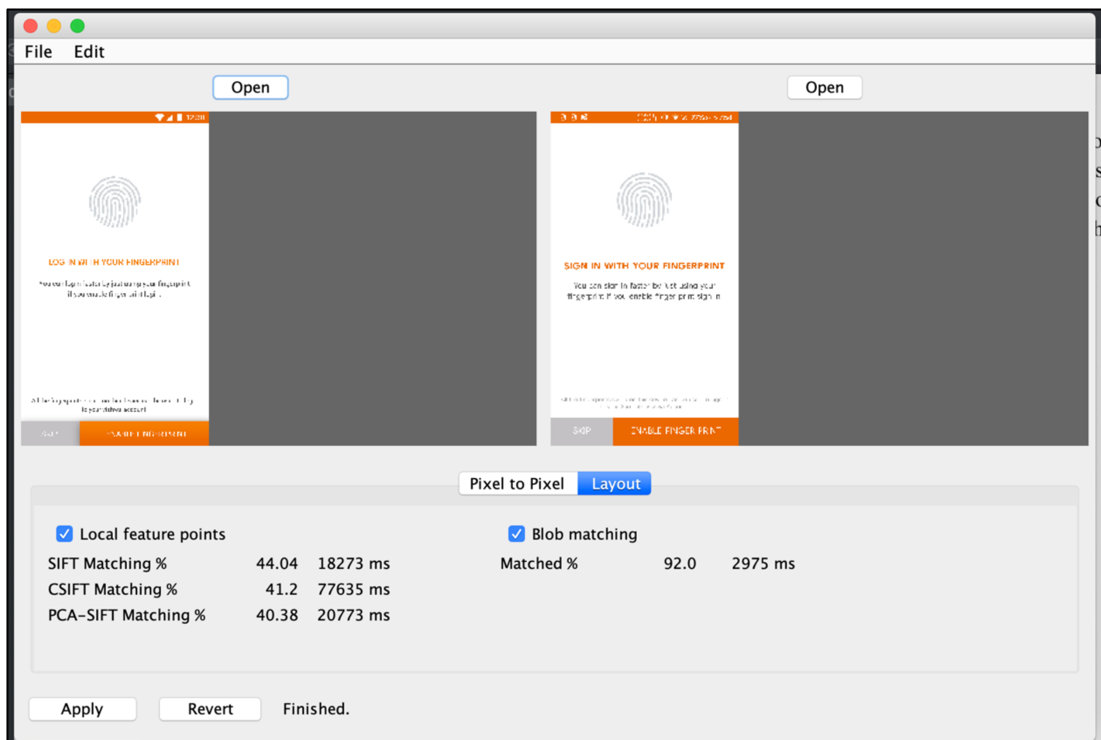


Figure 4.14 Layout matching screen

4.5 Image Comparison Framework

After the survey results are in and when we finalized the underlying algorithm for the layout score, we can go ahead and apply the algorithm to the layout score. Since the recommended algorithm from the results is CSIFT, the underlying structure of the final API looks like the following. How we ended up getting CSIFT as the underlying algorithm can be found in the next chapter.

```
public class ImageComparator {

    public static double getPixelToPixelMatchingScore(BufferedImage mockup, BufferedImage actual) {
        return PixelMatcher.getPixelToPixelMatchingScore(mockup,actual);
    }

    public static double getLayoutMatchingScore(BufferedImage mockup, BufferedImage actual) {
        return CSIFTMatcher.getCSIFTBasedMatchingScore(mockup,actual);
    }

    public static double getSIFTBasedMatchingScore(BufferedImage mockup, BufferedImage actual) {
        return SIFTMatcher.getSIFTBasedMatchingScore(mockup,actual);
    }

    public static double getPCASIFTBasedMatchingScore(BufferedImage mockup, BufferedImage actual) {
        return PCASIFTMatcher.getPCASIFTBasedMatchingScore(mockup,actual);
    }

    public static double getCSIFTBasedMatchingScore(BufferedImage mockup, BufferedImage actual) {
        return CSIFTMatcher.getCSIFTBasedMatchingScore(mockup,actual);
    }

    public static double getBlobDetectionBasedMatchingScore(BufferedImage mockup, BufferedImage actual) {
        return BlobDetectionMatcher.getBlobDetectionMatchingScore(mockup,actual);
    }
}
```

Figure 4.15 Underlying implementation of the framework API

5 RESULTS AND EVALUATION

5.1 Overview

This chapter summarizes the result of the research. It will start from our case study results where we use the same set of use cases in the proof of concept application and the survey to select an underlying algorithm to be used for the layout comparison score. This survey also helps us to determine the usability of the pixel based matching as an overall comparison score.

5.2 Survey

A survey is conducted to identify which layout matching algorithm performs best similar to a designer. This survey includes a set of image pairs which represents the actual implemented application's screenshot and the initial design [Appendix A]. It is given to a set of hand-picked user interface/ user experience designers from local and international companies. They are asked to provide two scores; overall score and a layout score. The overall score should be considering the color, orientation, scale, layout. The layout score only considers the layout of the particular design.

The same image pairs were given as input to the framework using the proof of concept implementation and the scores are recorded. Then the framework scores are matched with the survey scores. The overall score is matched to the pixel by pixel score in the framework and the layout score matched with the four layout scores calculated using SIFT, CSIFT, PCA-SWIFT and blob detection. This will allow us to select an underlying algorithm to calculate layout score for the framework.

5.3 Results

In this section we have compared results of the scores given by our shortlisted algorithms with the scores from the survey conducted.

5.3.1 Layout Matching

Table 5.1 represents the results of the survey data for layout scores along with its confidence interval for 95% confidence

Table 5.1 Layout Matching Survey Scores

Use case	Mean	Sample Standard Deviation	Degrees of Freedom	Margin of Error	Upper Bound	Lower Bound
Hapan - 1	65.5	10.50	19	4.91	70.41	60.59
Hapan - 2	68.75	14.32	19	6.70	75.45	62.05
Hapan - 3	61.25	11.11	19	5.20	66.45	56.05
Hapan - 4	78.25	11.15	19	5.22	83.47	73.03
EC -1	51.75	16.00	19	7.49	59.24	44.26
EC - 2	66.5	14.15	19	6.62	73.12	59.88
EC - 3	71.5	4.62	19	2.16	73.66	69.34
AD -1	72	5.94	19	2.78	74.78	69.22
AD - 2	75.5	14.03	19	6.57	82.07	68.93

Table 5.2 represents the results coming from the algorithms for the same use cases

Table 5.2 Layout matching algorithm scores

Use case	**SIFT**	**CSIFT**	**PCA-SIFT**	**Blob Detection**
Hapan - 1	57.24	57.53	52.84	41.97
Hapan - 2	75.48	61.9	73.99	93.39
Hapan - 3	54.81	55.08	51.82	87.34
Hapan - 4	56.54	51.53	55.14	77.87
EC- 1	34.80	33.99	32.76	61.55
EC - 2	20.78	20.85	17.89	70.54
EC - 3	17.46	12.71	15.17	87.29
AD - 1	49.38	55,73	47.18	94.9
AD - 2	34.66	34.19	32.03	95.06

Figure 5.1 has visualized the algorithm provided layout matching scores with the mean of the designer provided layout matching scores from the survey.

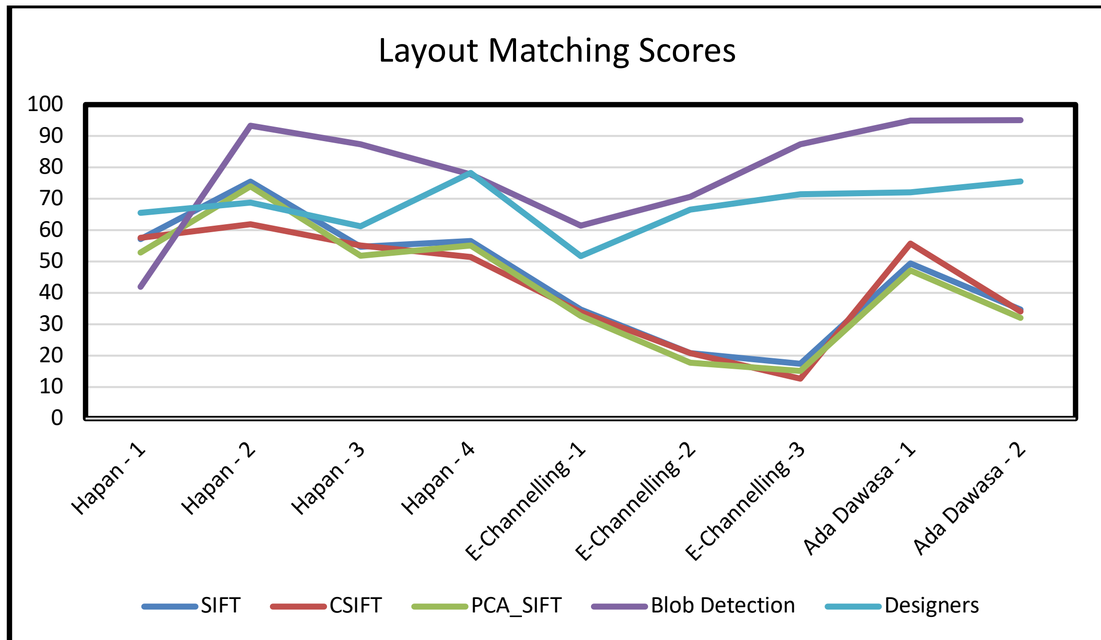


Figure 5.1 Layout matching scores

When we consider the layout matching, none of the algorithms strongly co-relates to the pattern of the scores derived from the survey answers by real designers. However, when taken separately, text heavy user interfaces like E-Channelling and Ada Dawasa use cases, the designer provided scores are slightly follow the pattern of blob detection algorithm scores whereas in picture heavy user interfaces like Hapan, the designer scores follow the local feature matching algorithm score patterns.

When considering Hapan use case, we can see that it is more color intensive and the picture to text ratio is high. And the text for screens we have taken to account is static and not get updated based on the underlying data. In this case, when identifying the feature points, the non-text components like back ground images and buttons plays a major role. Figure 5.2 shows a feature point matches between the design and the actual using CSIFT algorithm which has provided closest result to the designer provided score.

Figure 5.2 CSIFT Feature point matches

But when we consider E-Channeling and Ada Dawasa use cases, both these two apps are more text intensive. So, majority of identified feature points are letters. When the text is actually based on an underlying data source, the values between actual and the design are bound to change. Figure 5.3 shows the design and actual images for one of the E-channeling use cases. According to this figure you can see that the data presented in the screen is dependent on the underlying data source. In the design

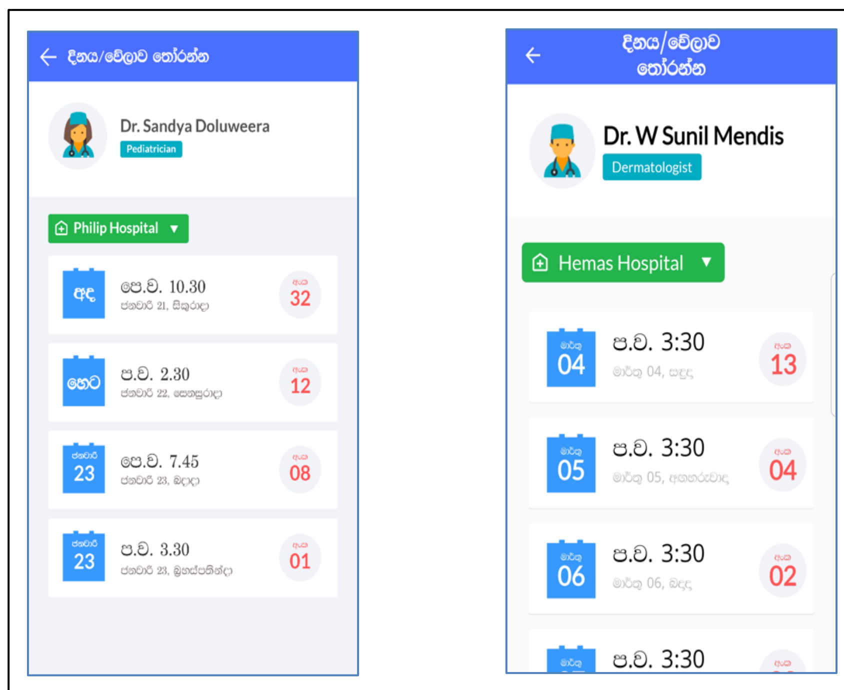


Figure 5.3 Design (left) and actual (right) E-Channeling-3

phase, dummy data is used for these scenarios.

Figure 5.4 shows the feature point matching for the same use case. For example, the design has used a female icon in the top left corner here as in the actual screenshot is

Figure 5.4 Feature point matching for E-channelling-3

an icon of a male based on the data. Similar to the icon, the dates of the design is different from the dates of the actual screenshot.

When the blob detection algorithm is used to this scenario, the text is detected and converted in to blobs as a preprocessing step. Because of this, the location of the text is matched rather than the text itself. The drawback of this approach is that before detecting the blobs, the image in converted into a monochrome image. Because of this it will lose the colors close to the background. For example, in Figure 5.5, the circular objects were not identified as blobs.

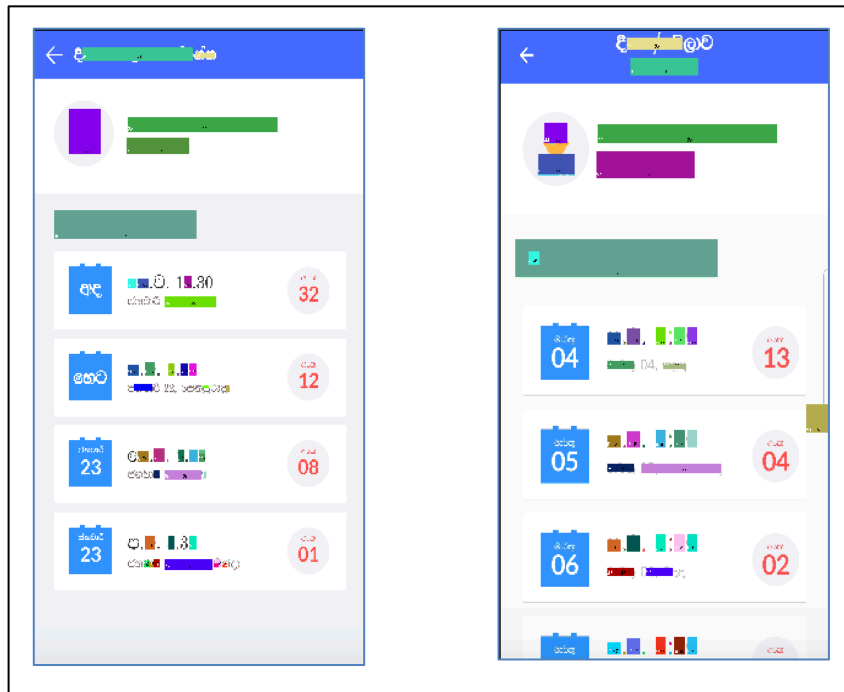


Figure 5.5 Blob detection in matching in E-channeling -3

Because of this blob detection is not suitable to identify the images with similar color palette. However, the underlying data problem can be solved by mocking the same data as in the design or doing the comparison before plug in the actual data source. In this way we can avoid the anomalies for text intensive applications.

5.3.2 Overall Matching

Table 5.3 Overall matching survey scores

Use case	Mean	sample deviation	Margin of Error	Upper Bound	Lower Bound
Hapan - 1	63.75	23.94	11.21	74.96	52.54
Hapan - 2	74.75	20.03	9.37	84.12	65.38
Hapan - 3	66.25	17.91	8.38	74.63	57.87
Hapan - 4	81	8.52	3.99	84.99	77.01
EC - 1	53.75	29.82	13.96	67.71	39.79
EC - 2	67.25	18.39	8.61	75.86	58.64
EC - 3	75.38	6.65	3.12	78.49	72.26
AD - 1	68	17.95	8.40	76.40	59.60
AD - 2	62.25	16.58	7.76	70.01	54.49

Table 5.4 Results given by the framework for overall pixel by pixel score vs. results from designers

Use case	Pixel by pixel matching score	Overall score given by designer
Hapan - 1	6.52	63.75
Hapan - 2	1.35	74.75
Hapan - 3	4.12	66.25
Hapan - 4	11.36	81
EC -1	46.28	53.75
EC - 2	34.62	67.25
EC - 3	40.66	75.38
AD -1	76.56	68
AD - 2	55.96	62.25

Figure 5.6 represents a graph to compare the overall scores given by the designers to pixel to pixel score provided by the framework.

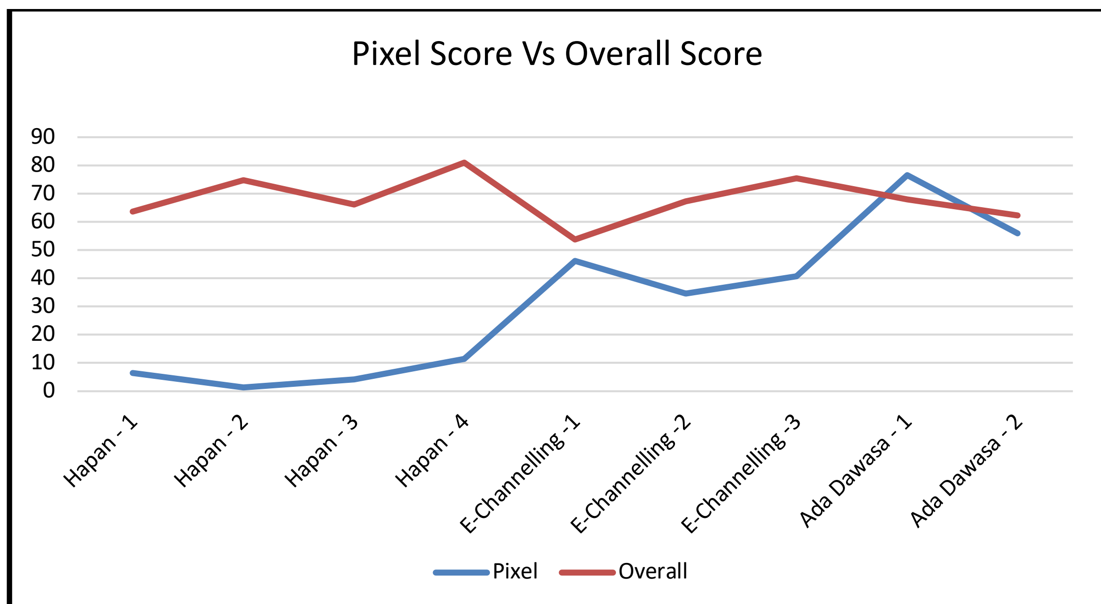


Figure 5.6 Pixel score provided by the system vs. overall score provided by the designers

According to this graph we cannot see any co-relation between the designer's overall score with the framework provided pixel to pixel score.

The main reason for this mismatch is the color difference. When comparing the pixels in two images color plays a large role. The framework will treat a small difference of the RGB value as same as a big difference of RGB value. For example, our Hapan-1 use case, the color difference is very small. Figure 5.7 shows the the design and the actual image. In Figure 5.8 design is given with the actual image where the pixels which are different from the design is colored in red. For non-trained eyes, the background color of the two images are almost identical. But for the trained eyes, they can see the difference, but the overall score will be based more on how close the color to the design, rather than whether the color is the same or not.

Figure 5.7 The design (left) and the actual image (right)

Figure 5.8 The design (left) and the actual image modified (right)

Another reason for the score difference is the objects to background ratio. In every use case except one, pixel score has been the lowest when compared to the overall score provided by the designers. But in the Ada Dawasa - 1 use case in figure 5.9, the pixel score (76.56) is greater than the designer provided overall score (68).

Figure 5.9 is the output from the pixel to pixel matching. It is visible that all the objects in the user interface doesn't match, but since both are on white background, the back ground is taken as a similarity.

Figure 5.9 Design (left) and actual image modified coloring the pixel differences (right)

5.4 Evaluation

In this research we have considered five local feature based matching algorithms and analyzed them of their usefulness for this context. This helped us to narrow down and shortlist three algorithms based on scale, affine and execution time.

We implemented the three selected algorithms; SIFT, PCA-SIFT and CSIFT modifying them to match to our context. We also implemented a simple pixel based matching algorithm and a new blob detection algorithm as well.

The implementation of these five algorithms has been used for the proof of concept application. This application will allow a user input two images and find the comparison scores based on the five algorithms.

A survey was done selecting a group of designers and giving them a set of use cases of mockups and actual user interfaces. We have asked them to provide two scores considering the layout matching and overall matching. The aggregated result from these values are used to compare with the scores given by the proof of concept application.

According to the results the pixel based score doesn't have any correlation with the designer provided overall value score. However, this can still be useful for the developers and quality assurance engineers to identify the changes easily between the implementation and the mockup.

Comparing the layout scores, we find that CSIFT algorithm closely follows the pattern of designer scores compared to the other algorithms. Hence, we have used CSIFT as the base algorithm for layout matching when building our framework.

6 CONCLUSION

The world is moving more towards leveraging mobile devices to replace almost everything in the day to life. This allows a lot of appetite for new application domains as well as lot of competition for existing application domains.

To get a competitive advantage, the companies need to provide a good user experience for the users. Unfortunately, unlike web development or other application development, mobile user interfaces are not implemented by the front-end engineers or UI/UX engineers. Although the companies do seek the user interface designer in the designing phase, they do not take steps to allocate the designer to verify once the application is built. This leaves the developer and the quality assurance engineer to evaluate the implemented user interface with the designed one using an untrained eye leaving space for lots of errors and ending up releasing a user interface which is different from the one user interface designer envisioned when the mockup was designed. Hence, we need a proper mechanism to quantify the differences between the mockup and the actually implemented user interface.

6.1 Research Contribution

This research focuses on implementing a solution for the above problem by providing a framework where the developer or the quality assurance engineer can use to verify the user interface by providing the design and the actual implementation.

On the road of achieving this result we have researched for the similar products and researches on this area and found that although similar mockup verification systems exist for web applications, the same has not been explored for mobile user interfaces.

We designed our framework such that the user can use the recommended layout matching score by default. In the same time the framework also provides the ability to use other layout matching algorithms directly to match the layout between two images. The framework also provides the pixel to pixel matching score as well. Although the research suggested that it is not usable and too restrictive, developers and testers can still use it if needed.

Depending on the application type, users can either directly use the CSIFT backed layout score provided by the framework or other layout algorithm.

The framework is built as an open source library so the users can use it and create their own solutions.

We also developed a proof of concept application which users can directly use to test their user interfaces if they do not need any customizations.

6.2 Limitations

6.2.1 User interface vs User experience

This framework only compares the differences in the user interface. In mobile apps user experience also plays a major role. This includes several areas like including animations, response speed, changes to the user interface when the orientation changes, etc.

6.2.2 Dynamic Data

In some mobile applications like the e-channeling use case, when the data like the doctor's name, or available hospitals can be different since they are dynamically loaded. This will affect to the computed scores. If a real user interface designer does the testing, they know these kinds of data is dynamically loaded. To avoid this, we have removed the text in blob detection matcher. But they are still considered in the local feature matchers.

6.2.3 Agile environment

Since software industry is moving towards more agile environment, the business requirements change. When this happens in the middle of the development process, the developer applies the changes without going through the UX designer if there is no resident UX designer present. A good example of this can be removal of a free feature or addition of a free feature. When this happens, the actual implemented screen ends up being different to the mockup. If there is a UX designer in resident they can provide modified mockups with those changes applied or in testing stage they can incorporate that knowledge in testing the user interface. But when we use image comparisons with the initial mockups, these changes will be flagged.

6.2.4 UX design concepts

Designers follow some concepts to ensure accessibility and consistency of the product. But in most cases developers do not have knowledge on these. Hence, they tend to do the easiest development and doesn't worry about the other aspects. For example, in the Hapan mobile app use case, Figure 6.1 and Figure 6.2 the designer has been consistent with the grass background throughout the app whereas the developer has only included them in the app only in the screen of Figure 6.1.

Figure 6.1 Mockup (left) and actual (right) with grass background

Figure 6.2 Mockup (left) with grass background and actual (right) without grass background

6.2.5 Orientation changes

Orientation changes has not been considered in this framework algorithms, specially the layout matching ones, to calculate similarity between the screenshot with the mockup. Unlike position changes and scale, orientation change is very unlikely happen. However, there are use cases where orientation changes can play a major role like animation heavy applications like games.

6.3 Future Directions

To fine tune the score received by the framework by pixel matching to be more correlated to the actual values by designers, we can consider assigning a weighted score based on the color difference or a threshold. More research can be done fine tuning these approaches.

In the blob detection approach the image is converted to monochromic image before detecting blobs. This eliminates the chances of detecting blobs which are close to the back ground color. More research can be conducted on that area to see how we can incorporate the color without converting to a monochromic image.

The framework can be improved and an IDE plugin can be built on top of it where the screenshot of the implemented application screen will be matched with the provided design while in the development phase itself. This will allow the developer to fix the errors before even sending to the quality assurance engineer.

7 REFERENCES

[1] "The evolution of the mobile phone." [Online]. Available: http://www.telegraph.co.uk/technology/mobile-phones/11339603/The-evolution-of-the-mobile-phone-in-pictures.html. [Accessed: 13-Jan-2017].

[2] L. Corral, A. Sillitti, G. Succi, A. Garibbo, and P. Ramella, "Evolution of Mobile Software Development from Platform-Specific to Web-Based Multiplatform Paradigm," Rev. Tecnol. | J. Technol., vol. 12, no. 4, pp. 181–183, 2013.

[3] S. Perez, "For The Young, Smartphones No Longer A Luxury Item | TechCrunch," 2012. [Online]. Available: https://techcrunch.com/2012/02/20/for-the-young-smartphones-no-longer-a-luxury-item/. [Accessed: 13-Jan-2017].

[4] C. Shealy, "A Collection of Mobile Application Development Statistics: Growth, Usage, Revenue and Adoption - Top Mobile Application Development Platforms, Vendors and Developers," 2016. [Online]. Available: http://solutionsreview.com/mobile-application-development/a-collection-of-mobile-application-development-statistics-growth-usage-revenue-and-adoption/. [Accessed: 13-Jan-2017].

[5] C. J. Budnik, R. Subramanyan, and M. Vieira, "Industrial requirements to benefit from test automation tools for GUI testing," Inform. 2007 - Inform. Trifft Logistik, Beitrage der 37. Jahrestagung der Gesellschaft fur Inform. e.V., vol. 2, pp. 410–414, 2007.

[6] P. Yadav, U. K. Yadav, and S. Verma, "Software Testing : Approach to Identify Software Bugs," no. 2, pp. 26–30, 2012.

[7] S. McConnell, *Code complete: [a practical handbook of software construction]*, 2nd ed. Berkeley, CA, United States: Microsoft Press,U.S., 2004.

[8] "List of tablet PC dimensions and case sizes," [Online]. Available: https://en.wikipedia.org/wiki/List_of_tablet_PC_dimensions_and_case_sizes. [Accessed: 13-Jan-2017].

[9] L. J. Osterweil, “Software processes are software too, revisited: an invited talk on the most influential paper of ICSE 9,” in Proceedings of the 19th IEEE International Conference on Software Engineering, pp. 540–548, Boston, Mass, USA, May 1997.

[10] IEEE, "IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology," 1990.

[11] “Types of Software testing and definitions of testing terms,” 2016. [Online]. Available: http://www.softwaretestinghelp.com/types-of-software-testing/. [Accessed: 13-Jan-2017].

[12] L. Williams, “Testing Overview and Black-Box Testing Techniques,” 2006. [Online]. Available: http://agile.csc.ncsu.edu/SEMaterials/BlackBox.pdf. [Accessed: 13-Jan-2017].

[13] M. J. Harrold, “Testing: a roadmap,” in Proceedings of the Conference on The Future of Software Engineering, ser. ICSE ’00. New York, NY, USA: ACM, 2000, pp. 61–72.

[14] E. Dustin, J. Rashka, and J. Paul, Automated software testing: introduction, management, and performance. Boston: Addison-Wesley, 1999.

[15] L. Feng and S. Zhuang, “Action-driven automation test framework for Graphical User Interface (GUI) software testing,” in AUTOTESTCON (Proceedings), 2007, pp. 22–27.

[16] H. Jung, S. Lee, and D. Baik, “An Image Comparing-based GUI Software Testing Automation System,” Elrond.Informatik.Tu-Freiberg.De, 2012.

[17] L. Lu and Y. Huang, “Automated GUI test case generation,” in Proceedings - 2012 International Conference on Computer Science and Service System, CSSS 2012, 2012, pp. 582–585.

[18] G. Liebel, E. Alegroth, and R. Feldt, “State-of-practice in GUI-based system and acceptance testing: An industrial multiple-case study,” in Proceedings - 39th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2013, 2013, pp. 17–24.

[19] A. M. Memon, “GUI testing: Pitfalls and process,” Computer, vol. 35, no. 8, pp. 87–88, 2002.

[20] O. H. Kwon and S. M. Hwang, “Mobile GUI testing tool based on image flow,” Proc. - 7th IEEE/ACIS Int. Conf. Comput. Inf. Sci. IEEE/ACIS ICIS 2008, conjunction with 2nd IEEE/ACIS Int. Work. e-Activity, IEEE/ACIS IWEA 2008, pp. 508–512, 2008.

[21] E. Borjesson and R. Feldt, “Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry,” Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. pp. 350–359, 2012.

[22] A. M. Memon, I. Banerjee, B. Nguyen, and B. Robbins. The First Decade of GUI Ripping: Extensions, Applications, and Broader Impacts. In Proceedings of the 20th Working Conference on Reverse Engineering (WCRE), October 14-17, 2013, Koblenz, Germany.

[23] Applitools, "Visual test automation for web Apps," Applitools -, 2015. [Online]. Available: https://applitools.com/web-app-testing/. Accessed: Aug. 16, 2016.

[24] N. zeldin Administrator, "Overview," 2016. [Online]. Available: https://applitools.atlassian.net/wiki/display/Java/SDK+Guide. Accessed: Aug. 16, 2016.

[25] Applitools - Automated Visual Testing, "How to Configure match level [Advanced visual test automation Techniques]," in *YouTube*, YouTube, 2016. [Online]. Available: https://www.youtube.com/watch?v=DzO-uzOLjUY&list=PLkqF-NUszJY7LZAdiAxf2zk1t1DBwP630&index=5. Accessed: Aug. 16, 2016.

[26] “Chapter 6 Learning Image Patch Similarity.” [Online]. Available: http://ttic.uchicago.edu/~gregory/thesis/thesisChapter6.pdf. [Accessed: 13-Jan-2017].

[27] F. Ren, J. Huang, R. Jiang, and R. Klette, “General traffic sign recognition by feature matching,” 2009 24th International Conference Image and Vision Computing New Zealand, 2009.

[28] R. Brunelli, “Template Matching Techniques in Computer Vision,” 2009.

[29] D. Lisin, M. Mattar, M. Blaschko, E. Learned-Miller, and M. Benfield, “Combining Local and Global Image Features for Object Class

Recognition," *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR05) – Workshop.*

[30] J. Wu, Z. Cui, V. S. Sheng, P. Zhao, D. Su, and S. Gong, "A Comparative Study of SIFT and its Variants," *Measurement Science Review*, vol. 13, no. 3, pp. 122–131, 2013.

[31] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, no. 2, pp. 91–110, 2004.

[32] Y. Ke and R. Sukthankar, "PCA-SIFT: a more distinctive representation for local image descriptors," *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004*.

[33] E. Mortensen, H. Deng, and L. Shapiro, "A SIFT Descriptor with Global Context," *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR05)*.

[34] A. Abdel-Hakim and A. Farag, "CSIFT: A SIFT Descriptor with Color Invariant Characteristics," 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2 (CVPR06).

[35] J.-M. Geusebroek, R. V. D. Boomgaard, A. Smeulders, and H. Geerts, "Color invariance,"/IEEE Transactions on Pattern Analysis and Machine Intelligence/, vol. 23, no. 12, pp. 1338–1350, 2001

[36] J.-J. Seo and K.-R. Yoona, "Modified Speeded Up Robust Features(SURF) for Performance Enhancement of Mobile Visual Search System,"/Journal of Broadcast Engineering/, vol. 17, no. 2, pp. 388–399, 2012.

[37] J.-M. Morel and G. Yu, "ASIFT: A New Framework for Fully Affine Invariant Image Comparison,"/SIAM Journal on Imaging Sciences/, vol. 2, no. 2, pp. 438–469, 2009

[38] "Material Design for Android | Android Developers," *Android Developers*. [Online]. Available: https://developer.android.com/guide/topics/ui/look-and-feel. [Accessed: 27-Mar-2019].

[39] Apple Inc, "Human Interface Guidelines," *Human Interface Guidelines - Design - Apple Developer*. [Online]. Available: https://developer.apple.com/design/human-interface-guidelines/. [Accessed: 27-Mar-2019].

[40] B. Epshtein, E. Ofek, and Y. Wexler, “Detecting text in natural scenes with stroke width transform,” *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010.

8 APPENDIX

Appendix – A: Survey

User Interface Verification and Validation (Mockup vs Actual)

Thank you for agreeing to help with my research on "Image Comparison Based User Interface Verification Framework".

What you have to do?
In the series of questions on this survey, I have given the mockup and the user interface of two case study applications. What you have to do is pick two score between 0 and 100 when comparing these two images. The value should take in to account the layout differences between the mockup and the actual. The first score is considering the layout and the second score is considering the overall differences (eg: color, text, etc)

Scoring Scale
0 - Two images doesn't match entirely
100 - Two images are identical

What is the layout score?
Compare the layout between two images. Is all the components present? Is the components in the correct place? Text changes, color changes are not considered when calculating the layout score

What is the overall score?
When you look at two images, what differences do you see? This includes layout differences, color differences, text differences.

What happens next?
Once I collected the data from you, I will compare these scores with the scores provided by framework I created.

What if you have extra comments?
Please add your comments in the extra comments section at the end.

Thank you very much once again for your valuable time.

* Required

Your Info

1. **Name (Optional)**

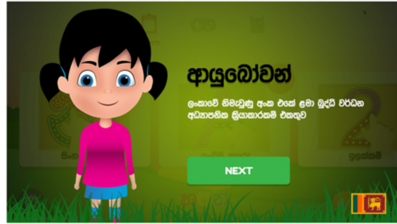
2. **Job Title** *

3. **Company (Optional)**

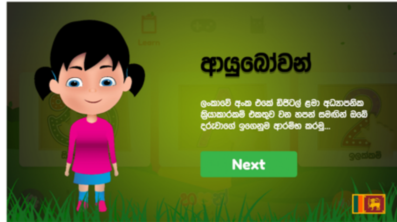
Case Study 1(Hapan) - Subject 1(Welcome Screen)

Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

Mockup



Actual



4. Layout Score - Hapan Welcome Screen *

5. Overall Score - Hapan Welcome Screen *

6. Additional Comments

Case Study 1 (Hapan) - Subject 2 (Gender Choice)

Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

Mockup

Actual

7. Layout Score - Hapan Gender Choice

8. Overall Score - Hapan Gender Choice

9. Additional Comments

Case Study 1 (Hapan) - Subject 3 (Waiting Screen)

Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

Mockup

Actual

10. Layout Score - Hapan Waiting Screen

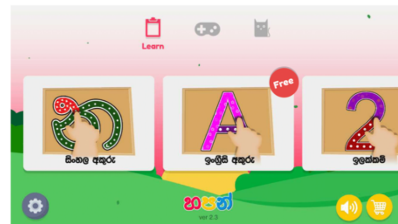
11. Overall Score - Hapan Waiting Screen

12. Additional Comments

Case Study 1 (Hapan) - Subject 4 (Menu)

Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

Mockup



Real

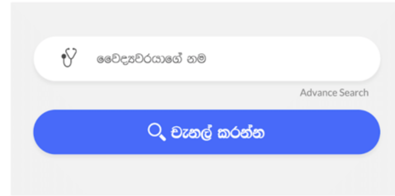
13. Layout Score - Hapan Menu

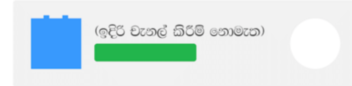
14. Overall Score - Hapan Menu

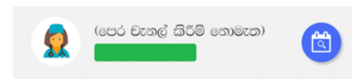
15. Additional Comments

Case Study 2 (Helakuru - E Channel) - Subject 1 (Home Page)
Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

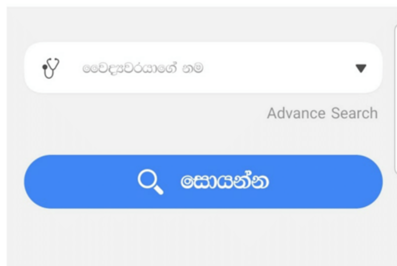
Mockup

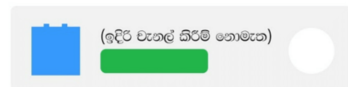






Real





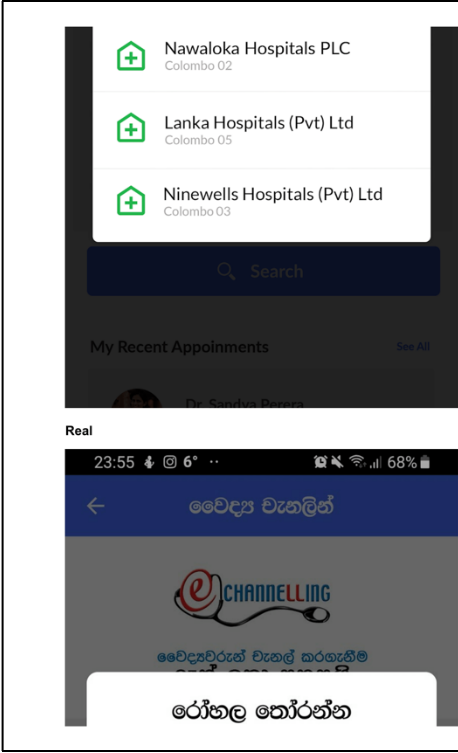
16. Layout Score - Echanneling home page

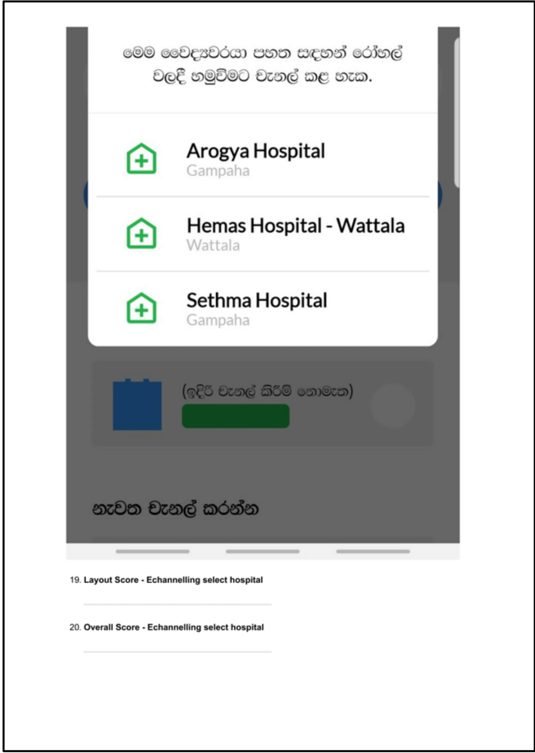
17. Overall Score - Echanneling home page

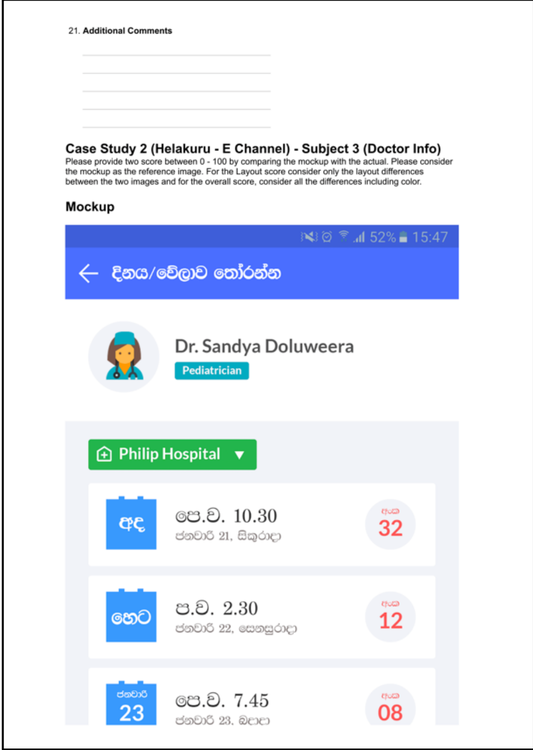
18. Additional Comments

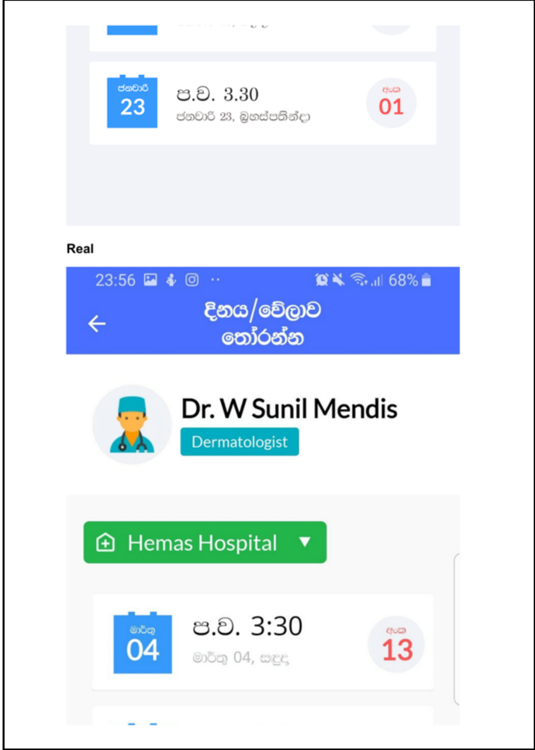
Case Study 2 (Helakuru - E Channel) - Subject 2 (Select Hospital)
Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

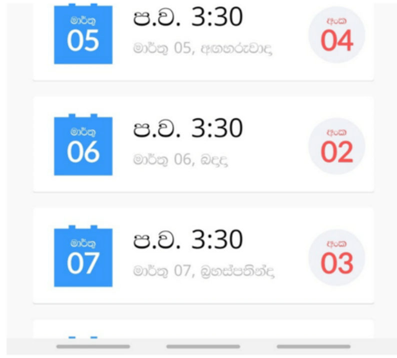
Mockup


Nawaloka Hospitals PLC
Colombo 02
Lanka Hospitals (Pvt) Ltd
Colombo 05
Ninewells Hospitals (Pvt) Ltd
Colombo 03
Search
My Recent Appoinments
Real
23:55
68%
CHANNELLING


Arogya Hospital
Gampaha
Hemas Hospital - Wattala
Wattala
Sethma Hospital
Gampaha
19. Layout Score - Echannelling select hospital
20. Overall Score - Echannelling select hospital


21. Additional Comments
Case Study 2 (Helakuru - E Channel) - Subject 3 (Doctor Info)
Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.
Mockup
52% 15:47
Dr. Sandya Doluweera
Pediatrician
Philip Hospital
10.30
32
2.30
12
23
7.45
08


23
3.30
01
Real
23:56
68%
Dr. W Sunil Mendis
Dermatologist
Hemas Hospital
04
3:30
13



22. Layout Score - Echannelling Doctor Infor

23. Overall Score - Echannelling doctor info

24. Additional Comments

Case Study 3 (Helakuru - Ada Dawasa) - Subject 1 (Home Page)

Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

Mockup

2018 පෙබරවාරි 22 සඳුදා

ඔබට සුභ දවසක් වේවා!

ඉර උදාව - පෙ.ව. 5:57

ඉර බැසීම - ප.ව. 06:31

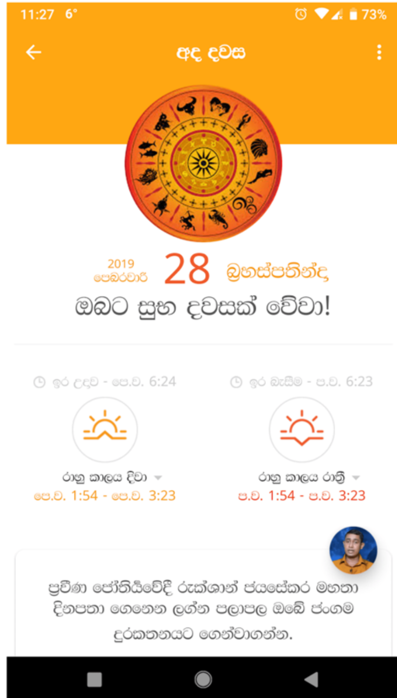
රාහු කාලය දිවා
පෙ.ව.8:30 - පෙ.ව.10:31

රාහු කාලය රාත්‍රී
ප.ව.8:30 - ප.ව.8:30

ප්‍රවීණ ජෝතිෂ්‍යවේදී රුක්ශාන් ජයතිලක මහතා
දිනපතා ගෙනෙන ලග්න පලාපල
ඔබේ ජංගම දුරකතනයට ගෙන්වාගන්න.

ඉදිරියට යන්න

Real



25. Layout Score - Ada dawasa home page

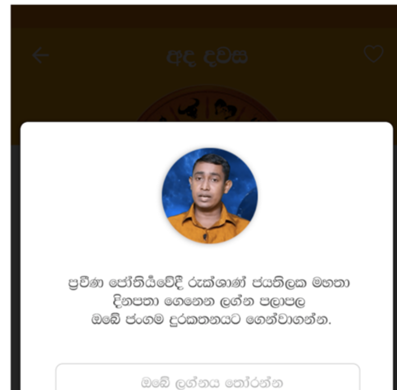
26. Overall Score - Ada dawasa Home page

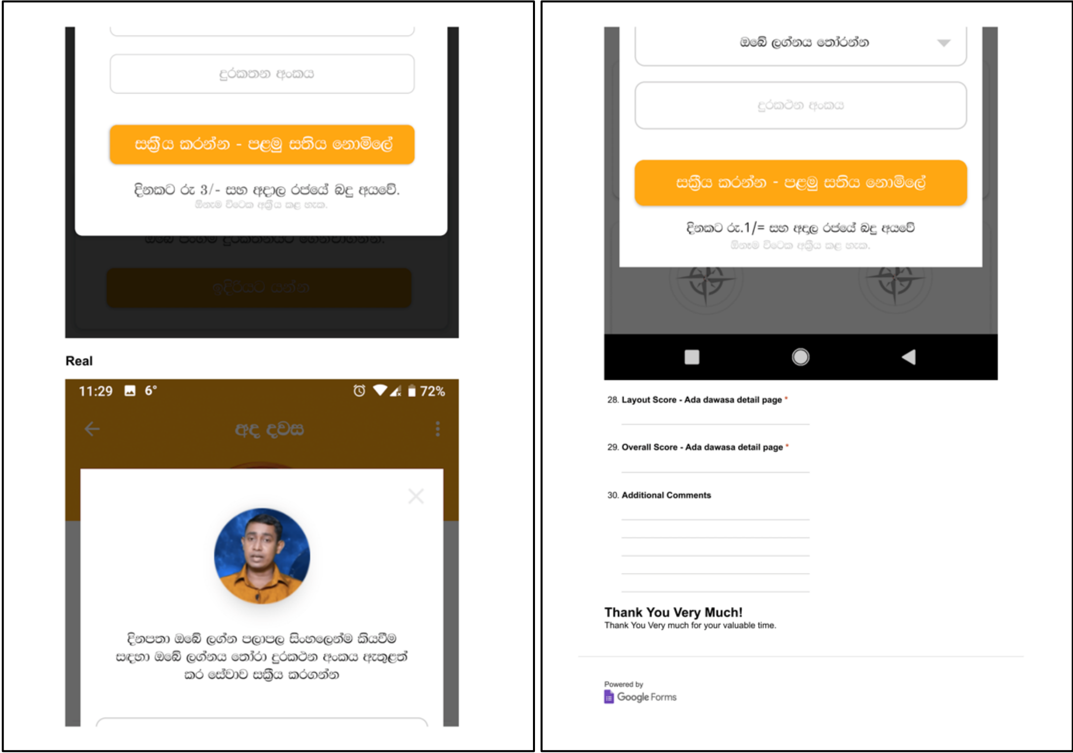
27. Additional Comments

Case Study 3 (Helakuru - Ada Dawasa) - Subject 2 (Detail Page)

Please provide two score between 0 - 100 by comparing the mockup with the actual. Please consider the mockup as the reference image. For the Layout score consider only the layout differences between the two images and for the overall score, consider all the differences including color.

Mockup




දුරකතන අංකය
සක්‍රිය කරන්න - පළමු සතිය නොමිලේ
Real
11:29 6° 72%
අද දවස
ඔබේ ලග්නය තෝරන්න
28. Layout Score - Ada dawasa detail page *
29. Overall Score - Ada dawasa detail page *
30. Additional Comments
Thank You Very Much!
Thank You Very much for your valuable time.
Powered by
Google Forms