NTNU
Norwegian University of
Science and Technology

A novel approach to GUI layout testing

Kristian Fjeld Hasselknippe

Master of Science in Computer Science
Submission date: June 2017
Supervisor: Jingyue Li, IDI

Norwegian University of Science and Technology
Department of Computer Science

Abstract

Graphical user interfaces has become an important part of most user facing software, and the cornerstone of any mobile app. The GUI is becoming a large part of an applications source code, and is therefore in need of automated testing. The current approaches to GUI testing focus mainly on the functional aspects of an application, trying to assert whether a user interaction has the correct result.

In this thesis we are interested in exploring another side of GUI testing, which we call layout testing. As applications are expected to run in ten to hundreds of different screen sizes, the need for validating that the GUI render correctly is increasing. With layout testing we are interested in being able to validate whether a GUIs elements, controls and widgets are positioned and rendered correctly.

We have focused on identifying common layout errors found in modern GUI applications and categorized them. The next part of this thesis is focused on the implementation of a tool, which incorporates the categorization of layout bugs, and which implements a set of algorithms which identifies these errors in real apps.

This tools was then tested on several real app cases, and compared to other state of the art tools, with focus on the accuracy of the results. We found that our tool was able to discover all common layout error types, and that it had a higher accuracy, for a wider set of test cases, than the compared tool. Since our tool also does not require a custom test script to be developed per app, it is able to lower the cost of testing GUI layouts compared to existing tools.

Sammendrag

Grafiske brukergrensesnitt har blitt en sentral del av det meste av programvare som produseres i dag. Siden brukergrensesnittet er i ferd med å stå for en større og større del av programmers kildekode, er det et økende behov for muligheten til å automatisk teste det. Dagens tilnærming til automatisk testing av grafiske brukergrensesnitt er for det meste opptatt med å teste de funksjonelle aspektene ved programmet, slik at man kan forsikre seg om at de ulike delene av brukergrensesnittet utfører sine oppgaver korrekt.

I denne masteroppgaven er vi interesserte i å utforske en annen side ved testing av grafiske brukergrensesnitt. GUI layout testing, som på norsk kan løst oversettes til testing av grafiske brukergrensesnitts planløsning, er oppgaven å teste hvorvidt alle elementene i et brukergrensesnitt er korrekt plassert på brukerens skjerm, og at elementene blir tegnet korrekt (for eksempel at to elementer ikke havner oppå hverandre).

Vi har fokusert på å identifisere vanlige layout feil i moderne applikasjoners grafiske brukergrensesnitt, og på å finne en kategorisering av disse feilene. Hoveddelen av denne rapporten er fokusert på implementasjonen av et verktøy, som automatisk kan identifisere denne typen feil i applikasjoner og peke brukeren i riktig retning for å fikse feilen i koden sin.

Verktøyet vårt ble testet på flere ekte app caser, og sammenliknet med andre eksisterende verktøy med liknende fokus på layout testing, der vi fokuserte på hvor nøyaktige resultat vårt og deres verktøy klarte å oppnå. Resultatene viste at vårt verktøy hadde høyere nøyaktighet enn tilsvarende verktøy, og er i stand til å teste et bredere sett med test konfigurasjoner. Siden vårt verktøy ikke krever utvikling av spesialtilpassede test scripts, er vi i stand til å senke kostnadene ved GUI layout testing.

Acknowledgment

I would like to thank my supervisor at NTNU associate professor Jingyue Li, for his advice and guidance during the writing of this thesis.

I would also like to thank my friends, family and my girlfriend Nadia Ursin-Holm for their support and encouragement throughout this work.

Contents

Figures

Tables

1. Introduction

GUI applications are now an integral part of most peoples everyday lives in the form of mobile apps. The production of these apps have become a huge industry, with several different operating systems, and app distribution platforms. Currently, the two biggest players in the market of mobile apps, are the iOS and Android platforms, who (as of the writing of this thesis) together consist of roughly 95% of the total market share [15].

Developing GUI applications can be very costly, especially if one is to create native apps for several platforms, which is the norm when creating mobile apps. As a way of making this cheaper and easier, a lot of app developers have ended up creating so called hybrid apps instead. Hybrid apps are web apps, meaning that they are based on web technologies such as HTML, CSS and JavaScript, which run inside a thin wrapper that provides access to native platform features like the camera, accelerometer and GPS.

There are also native solutions that allow one to write an application once, and then compile the app for both native iOS and Android. The solution used during the implementation phase of this project (www.fusetools.com) is one of these solutions.

A GUI usually consists of a large number of GUI elements, which are visual elements that represent either the data of the application, or the actions which can be performed by the user. GUI layout engines are in charge of the process of laying out these elements in such a way so that they fit well on the screen, meaning that they are fully visible. Most GUI applications are required to be run on many different screen sizes, and be able to display lots of different types of data. Making GUI layouts that are able to do this can be an extremely complex task.

1.1. GUI testing

The software development industry has gone to great length to create solutions for automatically testing their application, to make sure they are as free of bugs as possible and that their applications fulfill the required functionality, while simultaneously making it safer to make changes to the code.

GUI testing is a type of testing where the main focus is in validating the correctness of the GUI of an application. There are plenty of out of the box solutions (e.g. [7, 10, 18, 19, 21, 31]) for creating GUI tests, all with their own focus, strengths and weaknesses.

In this thesis, we distinguish between two main types of GUI testing: testing the functionality of a GUI and testing the layout of a GUI.

- **Testing functionality:** When testing for functionality, we are interested in testing that the interaction with the GUI works as we expect. This usually involves having either a tester, or an automated test engine interact with the GUI application, invoking its various actions and asserting that the GUI displays the expected output.

- **Testing layout:** When testing the layout of a GUI, we are interested in validating that the GUI has the correct visual appearance. This can for example mean to verify that all controls, like buttons, sliders and text, are fully visible and positioned correctly. Most GUI applications developed these days are required to run on multiple different screens, which can have both difference size and aspect ratio. They are also required to display dynamic data, which can vary in length and contain unexpected symbols [1].

1.2. The research questions

In this thesis we want to expand on the current state of the art with regards to layout testing and answer the following research questions.

- **RQ1:** What are typical GUI layout errors?
- **RQ2:** How can we automate, and thereby lower the cost of, GUI layout testing to find the typical GUI layout errors defined in RQ1?
- **RQ3:** How can we improve the quality of GUI layout testing by covering as many as possible of the GUI layout errors defined in RQ1?

While there are existing solutions that have been adopted by some members of the industry [10],[18], they either require much manual labor to incorporate, or they provide very limited test coverage. In this thesis, we will therefore explore a new approach to GUI layout testing.

1.3. Research design and implementation

The research questions in this thesis were addressed through the design and implementation of an automated layout testing tool, which we named "The **L**ayout **B**ug **H**unter", or **LBH** for short. The purpose of this tool is to automatically discover layout bugs in GUI applications and report them to the developer so as to aid them in correcting them. The tool was evaluated on

[1] When making an input field for handling user names, one might assume that user names are only of a certain lengths, and then end up creating a GUI which does not support displaying long user names. Making sure this does not happen is an important part of layout testing.

several real world GUI application screens with respect to how many layout bugs it could discover and how accurately it could report them.

1.4. Contributions

When it comes to GUI layout testing, two main approaches are in use today. The first approach is based on comparing screen shots of the app and using diffing algorithms to find the pixels that have change between tests. Then there are the techniques that are based on analyzing the apps layout, meaning that they gather information about the elements of a GUI (like width, height and position on the screen), and uses user defined specifications to perform various assertions, like "Is this Element less than 200 pixels wide?" or "Is Element 1 located to the left of Element 2?

LBH belongs to the layout analysis based type of testing tools, but gets rid of the need for user defined testing specifications. Instead, LBH makes use of the layout error categorization identified as a part of answering RQ1, and implements algorithms that automatically detects these errors in any app from which it is able to gather the correct layout information (like width, height and position) for all elements.

The design of LBH was made with cross platform testing in mind and is structured as a client server architecture, where the actual layout testing is performed on the server. The client communicates through a very simple protocol, which is easy to implement per platform that we want to support. As part of the implementation made in this thesis, the client component was implemented for the Fuse [9] platform (which supports the development of apps for Android and iOS), but we have also made the tool flexible so that it can easily be extended to test both web apps and native Android and iOS apps created using Java and Swift/Objective-C.

1.4.1. A novel approach to automatically discovering GUI layout errors

This thesis presents a novel approach to automatic testing of GUI apps. It provides an alternative to industry standard GUI testing techniques in that it explores the problem of testing a GUIs layout without requiring the developer to write long test scripts. While most standard GUI testing techniques focus on testing a GUI in relation to its functionality, the approach presented in this thesis focuses solely on the aspect of the GUI layout.

The LBH implementation was evaluated and made subject to four separate tests. The first test was done to see that LBH was able to discover the set of layout error types which was discovered in RQ1. We show in sec-

tion 4.2 that LBH is in fact able to correctly identify the set of layout errors we defined.

The second test was structured as a case study, where we implemented a small todo app using LBH to discover bugs along the way. The purpose of this test was to show an example of how LBH can potentially be utilized during the development of an app. In section 4.3 we show that LBH was indeed able to point out layout bugs and aid us in the correction of these bugs.

In the third test we compared LBH to one of the currently leading GUI layout testing tools on the market; AppliTools Eyes [4]. The test was conducted on a slightly complex app screen, where the goal was to identify layout bugs as a result of running the app on a screen which was too small. The test results clearly show how AppliTools is unable to separate between actual layout bugs, and what should just be considered valid layouts, whereas LBH is able to discover layout errors, as well as point the user to where in their code the error originated from.

LBH has two parameters which can be tweaked by the user. These parameters influence how LBH separates between intentional and unintentional layout errors with the goal of increasing the accuracy of its error reports. A final fourth test was conducted to see how changing these parameters affected the results of running LBH. The results of this test showed that tweaking these parameters lowered the chance of reporting false positives for complex cases.

1.4.2. Expanding the possible test coverage of GUI apps

LBH focuses on the problem of validating GUI layouts, making sure they render correctly on all target platforms and screen sizes. This is an aspect of GUI testing that is a lot less explored than the traditional methods, which mainly focus on testing the functionality (business logic in response to user interaction) of the application. With this, we are able to expand the test coverage that it is possible to currently achieve when testing GUI applications.

1.4.3. Automatic GUI testing without having to develop test scripts

Since the LBH performs its validation without the need for developing custom test-scripts, the system can be employed at any point in the development process. By this we mean that LBH can be used from the very beginning of a new app project, be added in the middle of an app being developed, or be employed after the release of an app in order to discover new potential bugs

and be used for regression testing.

1.5. Thesis structure

The rest of the thesis is structured as follows:

- **Chapter 2** takes a look at the current *state of the art* when it comes to GUI testing and layout systems.
- **Chapter 3** goes through the categorization of common layout errors, and the *design and implementation* of LBH.
- **Chapter 4** contains an *evaluation* of LBH with respect to its ability to discover layout errors and compared to other available GUI testing tools.
- **Chapter 5** is a *discussion* about what we have discovered by our implementation in chapter 3 and the evaluation performed in chapter 4.
- **Chapter 6** *concludes* the thesis with an evaluation of the results found in chapter 4, and a suggestion for potential *future work*.

2. State of the art

GUI testing is quickly becoming an important part of many software developers lives and is starting to be taken more and more seriously by the app development industry. GUI testing is significantly different from traditional software testing like unit testing, regression testing, integration testing and so on, in that it requires the tester, be it a person or a testing oracle, to interact with graphical components for the tests be me meaningful. When testing an app for for example the iOS or Android platforms, a testing oracle might simulate actual finger interacting with the touch screen of a phone in order to perform the GUI testing.

2.1. Two main types of GUI testing

In this thesis we make the distinction between functional GUI testing (what is traditionally thought of when discussing GUI testing) and GUI layout testing:

- **Functional testing:** Functional testing can be seen as a systems test of the GUI of an application. The goal of these tests is to make sure that all the functionality of an application is in place, and that it does its jobs correctly. Traditionally, before the process of functional GUI testing was attempted to be automated, this was typically done by a dedicated test employee, who would manually use all the functions of the application and manually record the visual responses it returned. Today, application developers have several options for automated functional GUI testing to choose from.

- **Layout testing:** When doing layout testing we are concerned with verifying that the layout of our GUI is correct on the various devices our application can be expected to run on. When creating an Android application for example, the number of possible screen sizes and screen aspect ratios are enormous. See Figure 1 for an illustration of the huge variation in screen size that an application is expected to support when released on the Google Play Store (the official app marketplace for the Android platform). Modern applications also need to deal with displaying dynamic data. By dynamic data we mean data which is collected from a server or created by the user, meaning the data was not available during the design and development of the application. As a result of this, being certain that a GUI is able to handle these requirements has become increasingly difficult.

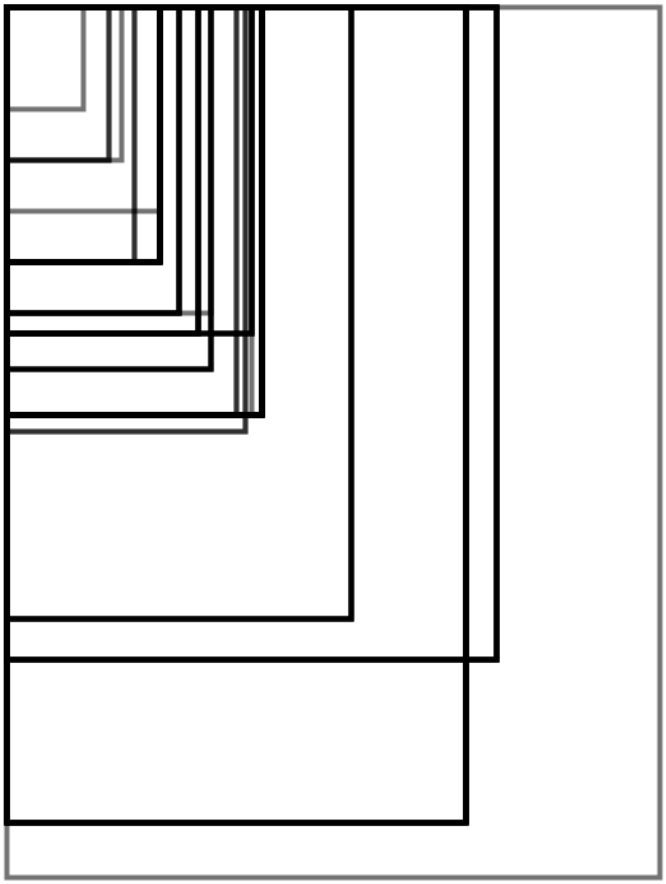


Figure 1. Android screen sizes

Unfortunately, this area of GUI testing is also the least explored, both by the industry and by academia. In their 2013 paper, Banerjee et al. [8] performed a secondary study on the field of GUI testing where they analyzed over 230 publications on the issue. They did not even include the topic of layout testing, but focused solely on the area of functional GUI testing, pointing us to believe that the academic community still sees this as a slightly fringe research topic.

2.2. Functional testing

Functional GUI testing is by far the most well researched and implemented type of GUI testing there is. It is concerned with testing that the GUI elements actually do what they are supposed to, which makes functional GUI testing an extremely useful, and often necessary, part of an applications testing procedures.

While there are several mainstream variants of functional GUI testing, one thing many of them have in common is the need for developing test scripts that can be considered formulas for interacting with a GUI and which assertions should be performed. Test scripts describe actions to be performed on a GUI, which often takes the application through multiple states, each of which have to be verified along the way. As a result of this, these test scripts can easily break as a result of code changes, since they can rely on so many parts of an applications GUI and business logic to work.

Much of the academic work done in this field goes towards reducing the cost associated with developing and maintaining these test scripts. In the next section we will introduce a small set of the techniques proposed in recent years.

- **Record and Replay:** Record and replay (sometimes called capture and replay) is a set of techniques based on simulating user input on the actual running GUI application taking the GUI from one state to another and then performing assertions on that state along the way. The set of inputs that are to be performed are usually represented as a script of actions, and is often created by having a program capture the user interacting with the GUI (hence the name **record** & replay). We will focus mostly on this type of functional GUI testing in this section, as it is the most richly explored type.

- **Model based GUI testing:** With model base GUI one usually creates a model of the application one is to test, which can be considered a partial description of the actual app. One then creates test cases over this abstract model. Each abstract test is then converted to an executable test using a tool which knows the relationship between the model and the actual system.

2.2.1. Dividing functional GUI testing into 3 abstraction levels

Bojko Adzic [1] suggests in his article on GUI test automation, that test scripts can be developed at three different levels of abstraction: the "Business rule / functionality level", the "User interface workflow level" and the

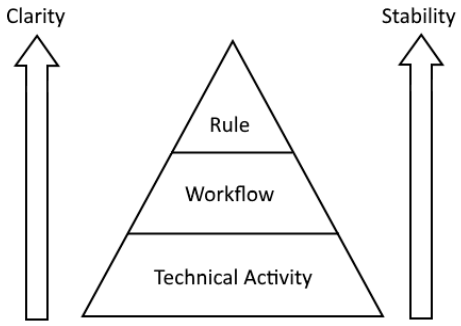


Figure 2. Three levels of abstraction

"Technical activity level". The relationship between these levels are shown in Figure 2.
When describing GUI tests at the business rule level, we express what the test is supposed to demonstrate: "Free delivery is offered to customers who order two or more books".

At the user interface workflow level, we need to be more detailed in our description, as if we were to teach a user how to perform the test: "Put two books in a shopping cart, enter address details, verify that delivery options include free delivery."

Finally, at the technical activity level, we need to describe every action involved in interacting with the GUI: "Open the shop homepage, log in with "testuser" and "testpassword", go to the "/book" page, click on the first image with the "book" CSS class, wait for page to load, click on the "Buy now" link…".

GUI test scripts in the "record, replay and assert" category, are usually described at the technical activity level. This level most easily translates to an actual implementation in code, but it is also most prone to breaking as the project develops.

- **Abstracting over raw user input gestures:** At the lowest level, we can imagine a script that describes the actions a user would perform, where the abstraction is interacting with the same input methods as the user would. In a desktop scenario, that would be the mouse and keyboard, and on a smart phone that would be a set of pointers (where the number depends on the capabilities of the touch screen hardware).

A fictional test script might look something like the following:

```
move cursor to (400,400)
click
wait 10ms
click
assertEqual(someVariable, <some_literal>)
```

In this case, the test script assumes there is something click-able at screen location 400,400. However, as soon as the location of this button is changed, whether it is because it is dynamically placed based on the value of some other controls, or because the designer decides it fits better somewhere else, this script will break and will need to be updated by the developer.

- **Abstracting over user input intent:** At the next level, we can imagine a script where we don't describe cursor movement directly, but instead encode what the users intent is. An example script would look something like the following:

  ```
  double-click elementByName("someButton")
  assertEqual(elementByName("someText").Value,
    <some_literal>)
  ```

As the imagined `elementByName` function illustrates; we can with this system more easily identify interactable objects. Their position on screen is no longer of any concern to the test script developer. They can also more easily describe how that element should be interacted with by using named gestures like `double-click`. They also don't have to care about moving the cursor to the right position, but can directly express that the action should be performed on the GUI element returned by the `elementByName` function.

This abstraction level suffers from many of the same problems as the previous one. Each script is tied directly to the names of each control. If the developer decides to rename a control, the test script breaks.

- **Abstracting over user stories:** At this level, the test scripts aren't expressed in terms of user input at all. Instead, they are expressed in terms of tasks the user is trying to perform.

At the level of abstraction, the developer is tasked with maintaining a set of functions the test developer can use to design and implement the test scripts. The abstraction itself will be considered a part of the code and the developers job to maintain, but should usually allow for a large variety of test scripts to be developed and maintained by someone other than the developer.

In this case, the test scrips themselves are easy to maintain, and can even be expressed in something resembling natural language. The advantage of this is that the task of maintaining them can be offloaded to the designer

or project leader, instead of it being a manual labor task owned by the developer.

The downside is that the developer still has to maintain the abstracted actions the script developer has at their disposal. These abstractions can still break as easily as the previously discussed abstraction levels, although the work involved in maintaining them is potentially reduced.

2.2.2. Record and replay

In this technique, a test developer will in some way interact with a GUI, and by the means of a tool, capture the interaction. The output of such a tool may vary depending on the solution, but commonly a script is generated that will allow an automatic test engine to replay the interaction as if it was performed by the user. A big advantage to this form of testing is that it resembles the industry standard ways of doing normal code unit testing. One performs an operation, and then asserts that certain values (in this case values of the GUI) are as expected.

Record and replay techniques differ from our approach in a major way: they don't test the actual GUI layout. While this is quite a fundamental difference, we've included discussions about record & replay techniques since they are the industry standard for GUI testing. Their inclusion in this discussion is important in order to get an understanding of why there is a missing piece of the puzzle of GUI testing, which we are trying to help fill with this work.

There are many variations of the "record and replay" technique. Most of them have in common that they allow the user to actually interact with the application they are testing in order to generate a script, but some tools combine this technique with other technologies in order to mitigate some of its downsides. We will take a look at some variations of the "record and replay" technique, and also discuss some pros and cons to this approach to GUI testing.

Pure record and replay In its purest form, the record and replay technique allows the test developer to record test scripts, which is then run each time the test engine is started. These tests usually becomes part of a unit test procedure, that is run frequently, with each new code push. GUI tests can be quite a bit slower than normal unit tests because they need to simulate a users input (which sometimes can't be sped up). They are therefore often run less frequently than the code unit test.

There is a reason why this technique has become the industry standard to GUI testing; it is straight forward to both produce scripts, and to run them.

For most of the native mobile app development tools like XCode [30] and Android Studio [2], there exists tools that allows the user to automatically generate GUI testing code by interacting with the app directly.

While some of the testing frameworks tries to generate layout agnostic testing code, by for example capturing input control IDs, the generated scripts are often intimately tied to the current state of the GUI design. This can make it more difficult to make changes to the GUI after the tests have been generated without breaking the tests.

Augmented variations of record and replay

- **GUI test scripts in combination with computer vision techniques:** This approach uses computer vision techniques to capture the appearance of widgets used in the GUI. These visuals are then used to create a dictionary of interactable components and a language of interactions, which the test developer can use. Because this technique uses computer vision to locate the widgets at runtime, it has a higher capacity for script reuse after layout changes are made. It does, however, handle significant changes (like the removal of certain widgets it expected to be present) poorly.

- **Generating tests using accessibility technologies:** This technique leveraged the accessibility technologies commonly found in modern operating systems. These operating systems typically come with built in software called a screen reader, that converts a visual GUI into text that can be communicated to people with visual disabilities using text to speech synthesis. The system used the information gathered by the screen reader both in the creation of tests as well as in the execution of the test engine.

2.3. Layout testing

Layout testing is an area of GUI testing which is concerned with validating the visual aspects of the GUI. We want to be able to verify that each widget of our GUI is laid out and rendered correctly on all target platforms, devices and for all possible data our GUI is supposed to present. Some types of layout testing also attempts to evaluate the level of complexity of a GUI or how easy the text is to read based on its contrast to the background.

It is clear that the role of GUIs has become ubiquitous in the software industry, but yet our methods for testing them visually has not progressed much compared to that of functional GUI testing. In this section we will

give an introduction to some modern GUI frameworks. We will also take a deeper dive in to the Fuse layout engine to get a better understanding of how the layouts of mobile GUIs are calculated.

2.3.1. Background on GUI layout systems

Ever since we started making GUIs, there has a been a need for describing the layout of the various widgets that should be placed in that GUI. By "layout", we mean the task of sizing and placing GUI elements on the screen. The most basic solution, is to just use absolute positioning of elements, meaning the developer specified both and "X" and a "Y" position as well as a "Width" and "Height" for each visual element. This is a viable solution, as long as we know the aspect ratio and size of the window or screen that we are targeting and we know exactly which elements we needs to place. This of course means that absolute positioning won't work with dynamic data, since at that point we don't know how many items we need to place, or how large they are required to be before the application is run.

The need for more sophisticated solution quickly became apparent, and had already been considered in other fields, like print media. They mostly worked with document layout however, which is only partly transferable to GUIs.

There are two main approaches to dynamic GUI layout used by modern layout engines: constraint based layouts, and hierarchical layouts - where both have their strengths and weaknesses. Fuse, the GUI layout engine used in this work belongs the the class of hierarchical layout engines.

Constraint based layouts vs hierarchical layouts The two main schools of dynamic GUI layout techniques boil down to those based on linear programming, and those based on hierarchies of layout containers. In this section we will quickly compare the two approaches, focusing on their general approach and some pros and cons to each technique.

Hierarchical (Fuse[9], WPF[29], HTML[12], Android UI[3]) The Fuse platform [9] (which is discussed in more detail in section 2.3.2), uses a hierarchy of panels which employ various layouting rules to their children. Generally, the hierarchical approach uses a hierarchy of layout containers that subdivide its available space, and is in charge of placing all its children in that space according to some defined rules. A markup language is typically used to describe these hierarchies, and especially the XML family of languages have proved useful here.

The following sample shows how the Fuse XML-based language called UX is used to create a vertical stack of text elements:

```
<App>
    <StackPanel>
        <Text>Text item 1</Text>
        <Text>Text item 2</Text>
        <Text>Text item 3</Text>
    </StackPanel>
</App>
```

The `StackPanel` has three children which is of type `Text`. Since the `StackPanel` is the only panel in the app (denoted by the outermost `App` tags), it is in charge of subdividing the entire screen/window using the stacking rules it defines. In the case of the Fuse `StackPanel`, it will start placing the `Text` elements on the very top of the screen, and make them as small in vertical size as they can, without violating their preferred size (which is determined by the amount of text it contains). The next child is then following the same rule, but being placed in the remaining vertical space, which is the entire screen minus the already placed `Text` element. Fuse' inner workings are discussed in detail in section 2.3.2.

Constraint based (iOS AutoLayout [26]) In constraint based layout solutions, like for example iOS' Auto Layout, layouts are described by a set of linear constraints, instead of by a hierarchical relationship. A hierarchical structure is also used here, but it is used more for organizational purposes, than to dictate the placement and size of elements.

Constraint based layouts give the designer lots of flexibility, but they can be quite cumbersome to write by hand. The need for visual tooling is dramatically increased with constraint based techniques. The textual output of these tools is also less human friendly than their hierarchy based counterparts. Luckily, the design tool that comes with XCode, called "Interface builder", allows the designer to visually edit layouts instead of having to write constraints using a markup language. Following is an example of how the iOS ".storyboard"-format represents a rectangle ("view") which is constrained to be 150 points wide and high, and at all times placed in the center of the available space.

```
<viewController id="BYZ-38-t0r" customClass="ViewController"
    customModule="AutoLayoutTest"
    customModuleProvider="target"
    sceneMemberID="viewController">
```

```
    <layoutGuides>
        <viewControllerLayoutGuide type="top" id="y3c-jy-aDJ"/>
        <viewControllerLayoutGuide type="bottom"
            id="wfy-db-euE"/>
    </layoutGuides>
    <view key="view" contentMode="scaleToFill"
        id="8bC-Xf-vdC">
        <rect key="frame" x="0.0" y="0.0"
              width="414" height="736"/>
        <autoresizingMask key="autoresizingMask"
            widthSizable="YES" heightSizable="YES"/>
        <subviews>
            <view contentMode="scaleToFill"
                translatesAutoresizingMaskIntoConstraints="NO"
                id="4BB-1V-imx">
                <color key="backgroundColor" white="1"
                    alpha="1" colorSpace="calibratedWhite"/>
                <constraints>
                    <constraint firstAttribute="height"
                        constant="150" id="7Ag-oO-djO"/>
                    <constraint firstAttribute="width"
                        constant="150" id="flK-3n-rDl"/>
                </constraints>
            </view>
        </subviews>
        <color key="backgroundColor" red="0.05813049898"
            green="0.055541899059999997" blue="1" alpha="1"
            colorSpace="calibratedRGB"/>
        <constraints>
            <constraint firstItem="4BB-1V-imx"
                firstAttribute="centerY"
                secondItem="8bC-Xf-vdC"
                secondAttribute="centerY" id="LVq-Aw-Wf4"/>
            <constraint firstItem="4BB-1V-imx"
                firstAttribute="centerX"
                secondItem="8bC-Xf-vdC"
                secondAttribute="centerX" id="fDg-WK-ebE"/>
        </constraints>
    </view>
</viewController>
```

```
<Panel>
    <Rectangle Width="150" Height="150"
        Alignment="Center" Color="Blue" />
</Panel>
```

Figure 3. Fuse equivalent of the .storyboard example code.

Note that the .storyboard example code is computer generated, which might be a source of verbosity in this case. It is however easy to argue that this representation is a lot less human readable than the Fuse (hierarchical layout) equivalent, which can be seen in Figure 3.

2.3.2. Fuse

Fuse is a Norwegian company making cross platform mobile app development tools that allows one to create custom layouts and animations that look exactly the same on both iOS, Android and Windows (through .NET [17]). It uses its own custom layout system, which is easy to learn and understand.

Having worked extensively with Fuse, both on its inner workings and as a development tool, it is a fitting platform to use as a base for implementing the ideas proposed in this thesis. It also allows us to illustrate the problems we are attempting to solve with this work.

To get a good understanding of the underlying problem, we'll do an in depth discussion of the layout system used by the Fuse engine. We will discuss how it places and sizes elements, and why it cannot easily guard against certain types of layout errors when we need to deploy the same app to multiple different screen sizes, and that should work with dynamic data (which we will sometimes refer to as test cases in this thesis).

UX Markup Fuse uses a markup language called UX, which is based on the XML family of markup languages. Following is an example snippet that creates a grid with four elements, a red circle, some text, a button and a slider control. We can see the output of this code in Figure 4.

```
<App>
    <Grid RowCount="2" ColumnCount="2">
        <Circle Color="Red"/>
        <Text>Some text</Text>
        <Button Text="Button text"/>
        <Slider />
```

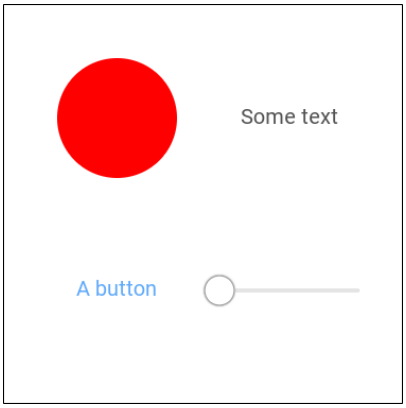


Figure 4. UX markup example

```
    </Grid>
</App>
```

UX Markup (or XML in general for that matter) allows us to easily create a tree structure where each node has a set of attributes associated with them. A node definition is of the form:

```
<NodeType Attribute1="value1" Attribute2="value2">
    ..children...
</NodeType>
```

Notice that we define a matching set of opening and closing tags: `<Type>` vs `</Type>`. In the case where our node doesn't take any more children, we can close it without and end tag by adding a `/` before the end `>`: `<SelfClosing />`.

UX Markup defines a hierarchical structure of panels and elements, where panels can have several other panels or elements as its children. We end up with a tree structure, where the root is an `App` tag. Figure 5 shows an example with more than one branch (the output is shown in Figure 6).
This translates into the following graphical structure which can be seen in Figure 7.
One can also turn a UX subtree into a reusable component by using the `ux:Class` attribute.

The following example component:

```
<Panel ux:Class="MyComponent">
    <Text Value="reusable"/>
</Panel>
```

Can be reused anywhere by using its class name:

```
<App>
    <StackPanel Color="Teal">
        <Panel Width="150">
            <Text Alignment="Left">Hello</Text>
            <Text Alignment="Right">World</Text>
        </Panel>
        <Panel Width="200">
            <Text Alignment="Left">Second</Text>
            <Text Alignment="Right">branch</Text>
        </Panel>
    </StackPanel>
</App>
```

Figure 5. Example showing the hierarchical structure of an app using UX markup.

```
<StackPanel>
    <MyComponent />
    <MyComponent />
    <MyComponent />
</StackPanel>
```

which is equivalent to the expanded version:

```
<StackPanel>
    <Panel>
        <Text Value="reusable"/>
    </Panel>
    <Panel>
        <Text Value="reusable"/>
    </Panel>
    <Panel>
        <Text Value="reusable"/>
    </Panel>
</StackPanel>
```

The layout system used in Fuse Layouts in Fuse are just hierarchies of panels. A `Panel` is an invisible container that can have multiple child panels and elements, and an associated `Layout` type. For convenience, Fuse defines a set of different panel types, like `StackPanel` and `DockPanel`, but in reality they are just wrappers over `Panel` with an associated layout like

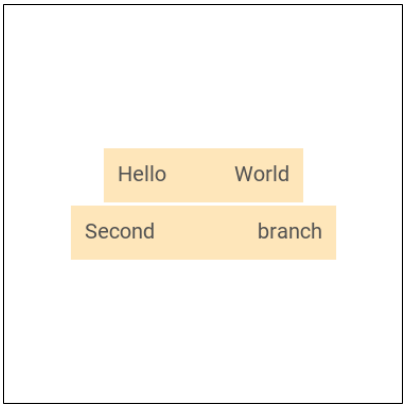


Figure 6. UX markup example.

for example `StackLayout` or `DockLayout`. The layout of a normal `Panel` is called `DefaultLayout`. The most important layouts types will be discussed.

All `Panel` types can contain other panels and also elements which represent visual controls like buttons, sliders, shapes and so on, but the `Panel` is not the most primitive type in the Fuse layout system. The base class (in the object oriented programming (OOP) sense of the word) for all visual elements (elements that have a size and a position) in Fuse is the `Element`. In the following sections we will discuss how the layout system works. While the naming and exact layout of the classes (also in the OOP sense of the word) might differ slightly from the actual implementation of Fuse, they are used here for illustrative purposes.

`Element` contains all basic properties that everything that can be placed by the layout system needs, like `Width`, `Height`, `Margin` and `Padding`. This base class does however not contain a childrens list, meaning it can only act as a leaf node in our layout tree [2].

Following is some pseudo code illustrating the structure of `Element` and `Panel`:

```
class Element
{
    Width : Number
    Height : Number
    Margin : Vector4 of Number
    Padding : Vector4 of Number
    Alignment : Alignment
```

[2]The layout hierarchy is in reality expressed as a tree structure, which is why it is often referred to as a layout tree.

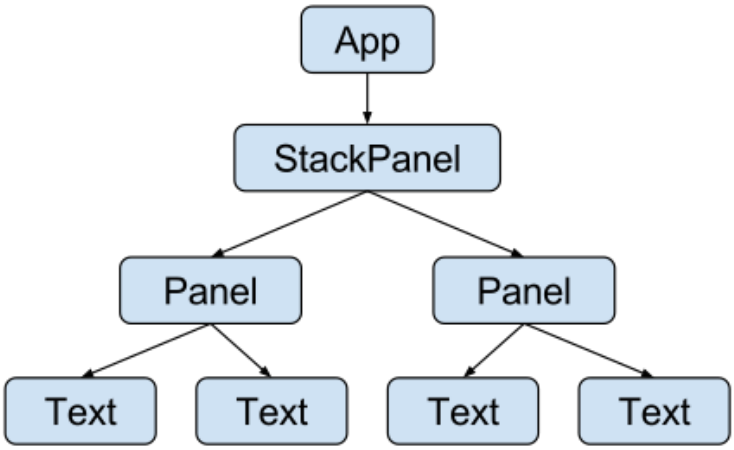


Figure 7. The hierarchical structure of the ux markup shown in Figure 5.

```
    virtual function Vector2
        of Number Measure(availableSize: Vector2 of Number)
    virtual function Arrange(availableSize: Vector2 of Number)
}
```

A simplified structure of `Panel` looks like the following:

```
class Panel inherits from Element
{
    Children : List of Element
    Layout : Layout;
}
```

Note that `Panel` inherits all the properties of `Element`, meaning it also can set its `Width`, `Height`, `Margin` and `Padding`.

The two phases of layout calculation The layout calculation in Fuse has two main phases: the measure phase, and the arrangement phase. During the measure phase, each element is asked to measure itself and return the size it requires. In the arrangement phase, this information is then used to actually assign positions and sizes to all the elements involved in the layout calculation.

Normal elements (which do not have children) will use the measure phase to return its preferred size. This can either be its assigned `Width` and `Height` plus whatever `Margin` and `Padding` it has, but in some cases it is decided

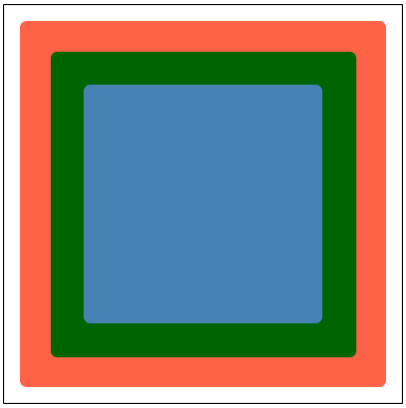


Figure 8. Default layout example

by the value of other of its properties. In the case of the `Text` element for example, the width and height is decided by the text rendering algorithm, which is a much more low level step. It is also in charge of performing proper text wrapping, meaning this information also affects the height of the text element.

When measuring a Panel, it will also call measure on all its children and use the returned information as part of its own measuring calculation. This means that we usually cover the entire layout tree in a depth first fashion, although this is strictly decided by the respective layout types.

After measuring, the placement phase is run. This is where all panels and elements are assigned their actual position and size. After this phase, the layout tree is ready for the rendering phase, where actual drawing is being performed.

DefaultLayout (Panel) The `DefaultLayout` is the most basic layout type, and used by the general `Panel` type. It does not do any explicit arrangement of its children, but it does let us align them based on the set `Alignment` property. It also allows us to layer several elements on top of each other. An example output from the use of DefaultLayout can be seen in Figure 8.

StackLayout (StackPanel) The `StackLayout` lets us stack elements on top of each other, either vertically or horizontally. It will size its children to be as small as possible in the direction of the stack, meaning that elements with no preferred size larger than 0 in that direction will not get any size at all. The StackPanel considers its available space to be infinite in the

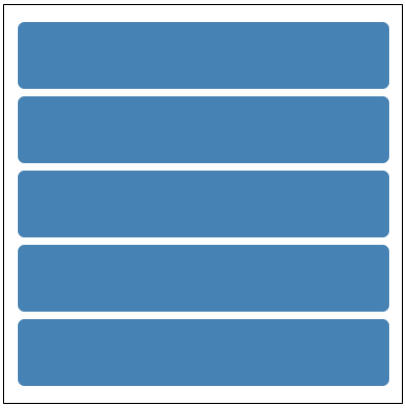


Figure 9. Stack layout example

stacking direction, but prefers to use as little space as possible. This means that, given enough elements of size larger than 0, it risks drawing in a larger area than its parent container has available. This is a potential for erroneous layout when working with lists whose lengths are not predefined and also when the screen size may vary. It is not always the case that an overflowing stack layout is actually erroneous however (for example if the area is user scrollable). For this reason, our system needs to not only consider whether an element is being drawn outside of the visible area, but whether that area is reachable by the user or not. The notion of content reachability will be discussed in more detail. An example of the output of StackLayout can be seen in Figure 9.

Content reachability and the ScrollView In modern apps, the use-case of displaying large lists of data is quite common. These lists are often longer than the screen is tall, and so there is a needs for scrolling views. This is seen in almost every GUI of modern apps. What this means, is that an element can be outside the bounds of its parent `Panel`, but still be "reachable" by interacting with the scroll control. We therefore need to be aware of the fact that an element can be outside of its bounds, but reachable, and thus placed on a valid location. This is discussed further in a section 3.3.1.9.

DockLayout (DockPanel) The `DockLayout` lets us "dock" elements to one of the four edges of the arranging panel (meaning the `Panel` who's children are being arranged by the `DockLayout`): "left", "top", "right" or "bottom". The children are placed one at a time, and for each placement, the available space offered to the next child is reduced by the amount used by

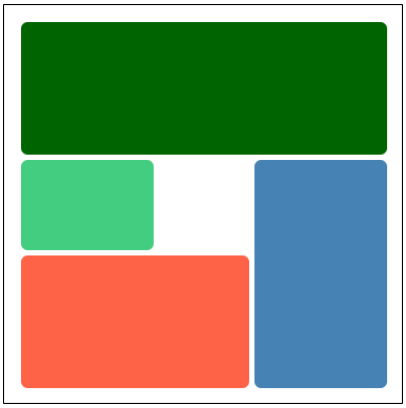


Figure 10. Dock layout example

the previous child. Docking an element to the left, sizes it to its preferred size in the horizontal direction (meaning its `Width`), and placed it near the left edge of the arranging Panel. The layouting works exactly the same for "top" and "bottom", but then `Height` is used instead as the limiting size. Figure 10 shows an example of the use of DockLayout.

GridLayout (GridLayout) The `GridLayout` allows us to place elements in a cellular fashion by defining a set of rows and columns. Elements can then be placed in any (or several) of the cells defined by these rows and columns. Row and column definitions can have either absolute size values, or they can be sized relative to each other. In Fuse, we define rows and columns using a comma separated string of values where we either specify absolute values (e.g. "125"), relative values (e.g. "2*"), or ask it to automatically size itself based on the elements placed in it. Figure 11 shows an example of a GridLayout.

The "*" syntax allows us to create a set of definitions that are relative to each other. The following definition gives us 3 columns, where the first and the last are half as wide as the middle one: `"1*,2*,1*"`. The calculation works as follows:

- Column 1: (1/(1+2+1)) * 100% of Grid Width
- Column 2: (2/(1+2+1)) * 100% of Grid Width
- Column 3: (1/(1+2+1)) * 100% of Grid Width.

Figure 11. Grid layout example

2.3.3. Different layout testing techniques

In this section we will take a look at a handful of the GUI layout testing tools that are available and in use by the software industry today. When we discuss layout testing tools we have decided to place them in one of two categories:

- **Layout analysis based techniques:** These techniques are based on assertions on the layout elements themselves. The input for these methods aren't so much a script as they are declarative documents stating how various layout elements should relate to each other and what values their layout parameters should lie within.
- **Image diffing based techniques:** These techniques rely on using techniques commonly associated with computer vision to discover regressions in GUI layouts. They do this by, among other ways, performing edge detection to discover element borders, calculate the difference between two screen shots to discover changes/regressions, and compare colors of text to its background to discover unreadable text.

Layout analysis based testing techniques The techniques we describe under the category "layout analysis based" are techniques that work directly on information about the layout. What these systems have in common is that they somehow, be it by image analysis or by scraping the GUI, obtain information about the elements (like position, width and height) involved in the GUI. These techniques will use this information and possibly a test script or test description to perform assertions on the GUI. These assertions can for example be of the form "Is element 1 located to the left of element 2?"

or “Does element 1 has a smaller width than element 2?”, and some of these techniques have specialized languages [11] used for describing assertions like these.

Fighting Layout Bugs Michael Tamm presents in his google tech talk [22] a way to use CSS injection along with image diffing techniques to detect which part of a web site is text and whether it overlaps with other elements borders. He achieves this by first changing the color of all text on a web page to black, and then white, while taking screen shots in between. By diffing the two images, he is able to locate the parts of the image that are text because only those pixels will have changed from black to white. He then uses image based techniques to discover the vertical and horizontal lines of the elements in the resulting image. By comparing these lines with the text items discovered in the previous step, he is able to determine cases where text overlaps with element borders. He also uses this technique to determine whether text elements has the appropriate contrast to their background.

This is one of the techniques that are most similar to ours in terms of the goal of not requiring test scripts, and automatically discovering layout bugs. One important difference is that this technique will only work for text, and elements that can easily be identified using image based techniques for object recognition. The scope is slightly smaller since it focuses exclusively on text to element overlap, but the technique is interesting to us since it approaches the same problem from a completely different angle.

Galen Framework Galen framework [10] is a GUI layout testing library for testing Web app layouts. It lets the user create GUI layout descriptions that describe how various elements of a GUI relate to each other and what kind of layout parameters they will consider valid. A test oracle uses this description to verify that the placement and size of elements are valid.

Galen framework achieves this by having a custom test description language that the test developer uses to declare various aspects of an elements position and size relative to other elements or in absolute values. It also allows for specifying ranges of acceptable values. Figure 12 shows an example of what a Galen specification looks like.

Galen is based on a web browser automation tool called Selenium [21], which lets one create automated scripts that simulate user input. By combining these two technologies, the Galen framework allows the user to fully automate the task of regression testing web application layouts.

One thing to note, is that since the specification file required by the Galen framework needs exact information about what valid values for GUI

```
@objects
    comments            #comments
    article-content     div.article

= Main section =
    @on mobile, tablet
        comments:
            width 300px
            inside screen 10 to 30px top right
            near article-content > 10px right

    @on desktop
        comments:
            width ~ 100% of screen/width
            below article-content > 20px
```

Figure 12. Example of what a galen specification looks like.

elements are, they can be costly to make, and can make it more difficult to make changes to a GUI.

By looking at the Galen specification language [11], we can see that the work required by the test developer is similar to actually specifying the GUI itself. This probably means that the Galen specs have to be updated with about the same frequency as the GUI design itself.

The downside to this is reduced flexibility and increased cost. There is however a strong upside, which is the accuracy to which one can perform regression testing using an approach like this and that one can check if a design works on different screen sizes. By requiring the developer to specify exactly what they consider to be valid layouts, one drastically reduces the chance the system reporting false positives and false negatives.

ITArray - Automotion ITArray's Automotion framework [7] is similar to Galen framework. The difference from many other frameworks is that the assertion library actually lets you assert whether how elements position and alignment relate to each other with calls such as : isElementInside(otherElement) or areElementsAligned(List). The way these tests are specified is even more verbose than the way you specify actual layouts in Fuse, point to its verbosity. An example of an Automotion script is shown in Figure 13.

```
boolean result = uiValidator.init("Scenario name")
    .findElement({rootElement}, "Name of element we validate")
    .sameOffsetLeftAs({element}, "Panel 1")
    .sameOffsetLeftAs({element}, "Button 1")
    .sameOffsetRightAs({element}, "Button 2")
    .sameOffsetRightAs({element}, "Button 3")
    .withCssValue("border", "2px", "solid", "#FBDCDC")
    .withCssValue("border-.radius", "4px")
    .withoutCssValue("color", "#FFFFFF")
    .sameSizeAs({list_elements})
    .insideOf({element}, "Container")
    .notOverlapWith({element}, "Other element")
    .withTopElement({element}, 10, 15)
    .changeMetricsUnitsTo(ResponsiveUIValidator.Units.PERCENT)
    .widthBetween(50, 55)
    .heightBetween(90, 95)
    .drawMap()
    .validate();

uiValidator.generateReport("Report name")
```

Figure 13. Example of what an Automotion test script looks like.

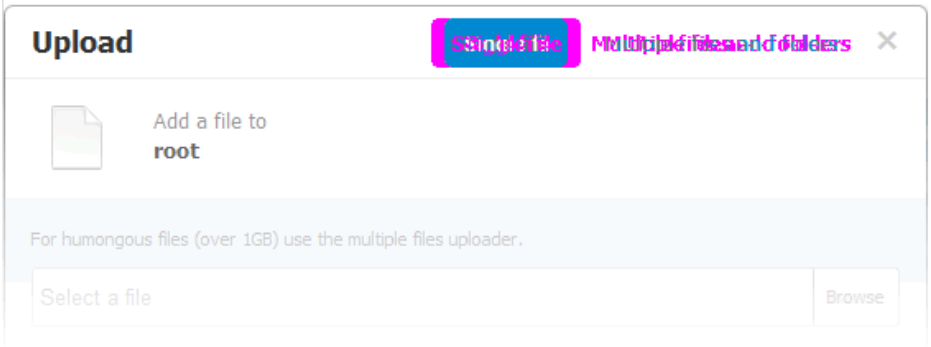


Figure 14. PhantomCSS example

Image based techniques Following is a description of techniques that are based on image comparison. The special thing about these techniques is that they are based on comparing screen shots, pixel by pixel, instead of acting on layout specific information like element sizes and positions. This means that these techniques can employ techniques from computer vision that other techniques can not necessarily use. These tools usually report errors in the form of a screen shot with the errors clearly marked. An example of the output from the PhantomCSS framework can be seen in Figure 14.

PhantomCSS and Webdriver PhantomCSS [18] and Webdriver [28] are two open source GUI testing frameworks that work using many of the same techniques as AppliTools. They use diffing algorithms on two images in order to find changes from one iteration to the next. Another feature that these frameworks offer is the ability to exclude certain parts of the GUI from the testing. This is particularly useful for web pages since they often display data which are not controlled by the developer. This includes elements such as web ads, user data, animated banners, images and text.

This is unfortunately a manual task, meaning the developer has to manually specify which parts of the GUI should not be tested, either by using the name of the element, or by X and Y positions.

PhantomCSS and Webdriver also integrate into AppliTools environment.

AppliTools AppliTools [4] lets the user create test scripts that interacts with their app and upload screen shots of their application to a web server that applies an image diffing algorithm between it and a previous screen shot. By finding the difference between the two images, it is able to detect regressions in a GUI layout. It then presents these changes in a web interface

which allows the developer to easily update the baseline image in the case where the change was intentional.

The web tool also allows for several parallel version of a baseline screen shot, for the cases where people develop on different parallel branches of a GUI.

Since this tool works on image diffing, it is not able to discover layout bugs that are not manifested as a change from one iteration to the next. What we mean by this is that for a tool based on image diffing to work, it needs two images to compare, where one of them is considered the source of truth. Unless we have already been able to establish and capture a source of truth, these techniques won't be able to locate layout errors.

2.4. The problems with the state of the art of GUI testing

There are several problems with the mainstream techniques we have discussed. While developers have more and more options to choose from when considering a GUI testing strategy for their projects, it is still a costly and difficult affair.

2.4.1. GUI testing adds maintenance overhead

A major problem with todays mainstream GUI testing techniques is the maintenance overhead it adds to the project. GUIs are likely to change over the course of both project development and while it is being maintained, and each time it changes, the test scripts have to be updated as well. This means that for each GUI test a project has, the more difficult it becomes to make changes to it. This is of course hard to avoid, and is sometimes the case for regular unit tests as well. However, an important difference between GUI tests and regular unit tests is the fact that a change to the appearance of a GUI that is not a "breaking chance" (meaning a human user would still be able to use it) can still cause its test script to report errors. While breaking changes reported by unit tests usually point to failing business logic, a breaking GUI test can often just mean that the design changed, and that the test script is out of date.

2.4.2. We lack good tools for layout testing

While there has been lots of work done in the field of functional GUI testing, we still lack a good set of options when it comes to layout testing. There is a handful of solutions out there [4, 7, 10, 18], but they are mostly focused on

testing web apps. When it comes to layout testing for mobile apps, the only real options we have for doing layout testing is to use AppliTools Eyes [4], which only supports image diffing based testing, but is able to create tests for mobile platforms.

2.4.3. Problems with layout analysis based techniques

The layout analysis based techniques discussed have the problem of requiring quite verbose test descriptions/scripts, requiring the test developer to describe the rules of their GUI (how large elements are in relation to each other, how they are positioned relative to each other, how they should behave when the window size changes, etc.) in greater detail, to the point where the test script almost becomes as descriptive as the GUI code it tests. This is a problem because it, as all test scripts do, will require updates at almost the same rate as changes are made to the GUI design.

2.4.4. Problems with image diffing based techniques

- **Cannot compare images from different screen sizes:** Since image diffing based techniques uses the difference in pixel color between a baseline image (the source of truth) and a test image (the image to be tested), for them to be considered equal, and therefore valid, they cannot be screen shots from different screen sizes. This means that we can't use the baseline image from one screen size as a means of checking the validity of the layout on a different screen size. This means that we need to establish baselines for all screen sizes if we are to use these techniques to validate the layout of an application that is supposed to run on many different screen sizes.

- **Does not work well with dynamic data:** A major issue with image based techniques is that they only work well with static data. This is because they work on pixel information, instead of content. Most apps will have to display different content each time they are run because this data is often fetched from a web server, or made by the user. This fact significantly lowers the usefulness of image diffing based techniques. We can imagine some ways of mitigating this problem, by for example using the same data each time one is testing, and having separate baselines for each screen size, but this can easily become very costly if the application is expected to run on many screen sizes.

A better approach might be to test only parts of a screen at a time, as stated on the PhantomCSS repository: "If you try to test too much in one screen

shot then you could end up with lots of failing tests every time someone makes a small change.". Increasing the testing granularity might decrease the chance of the system reporting false positives, but it can also decrease the value of the testing, since the system is not tested as a whole.

- **Is prone to reporting false positives:** Image diffing based techniques are prone to reporting false positives, as stated on the PhantomCSS repository website [18]. Pixel values of two consecutive screen shots can easily differ if they are taken on different machines with different setups. These small pixel differences, while not actually layout errors, will sometimes be falsely reported as errors by the testing algorithm. Figure 15 shows an example of this. We can observe small areas (which we have marked with red circles) of purple pixels on what looks like identical images. These are not actually errors, but false negatives reported due to small pixel differences in the two renderings (which could be due to the use of different browsers).

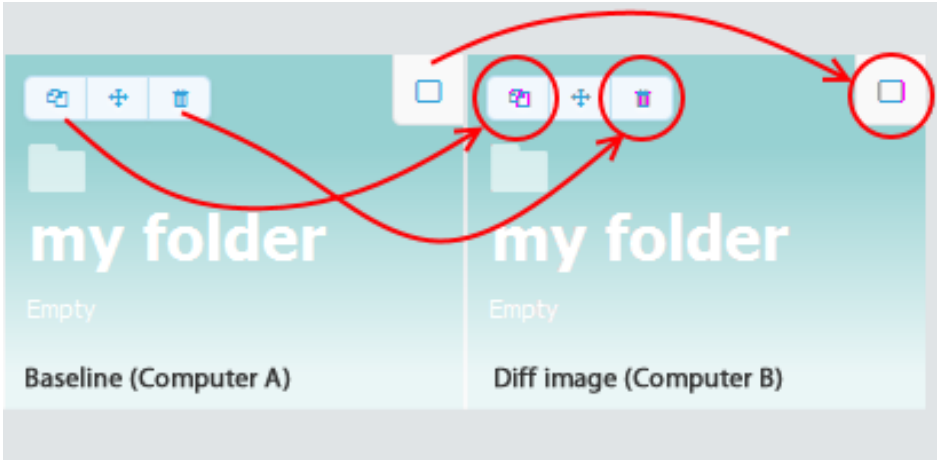


Figure 15. An example of PhantomCSS reporting false negative.

3. Design and implementation

In this section we will present the research questions that guided the work done in this thesis in detail. We then present the main work done in this project and how they answer our research questions.

3.1. Research questions

In this section the research questions will be presented and discussed.

- **RQ1:** What are typical GUI layout errors?

This question regards the problem of defining what a GUI layout error is, so that it can be separated definitively from what would be considered valid GUI layouts. Answering this question becomes the foundation of the prototype implementation of the automatic GUI layout testing system made as a part of this work.

We will try to establish a categorization of GUI layout errors that can be encoded by a set of algorithms, and applied to an actual app to find errors in it.

- **RQ2:** How can we automate, and thereby lower the cost of, GUI layout testing to find the typical GUI layout errors defined in RQ1?

This research question acts as a follow up to the previous one. If we are able to come up with a definition for GUI layout errors, is this definition possible to encode as a set of algorithms, and can these algorithms be used to create a fully automated testing system?

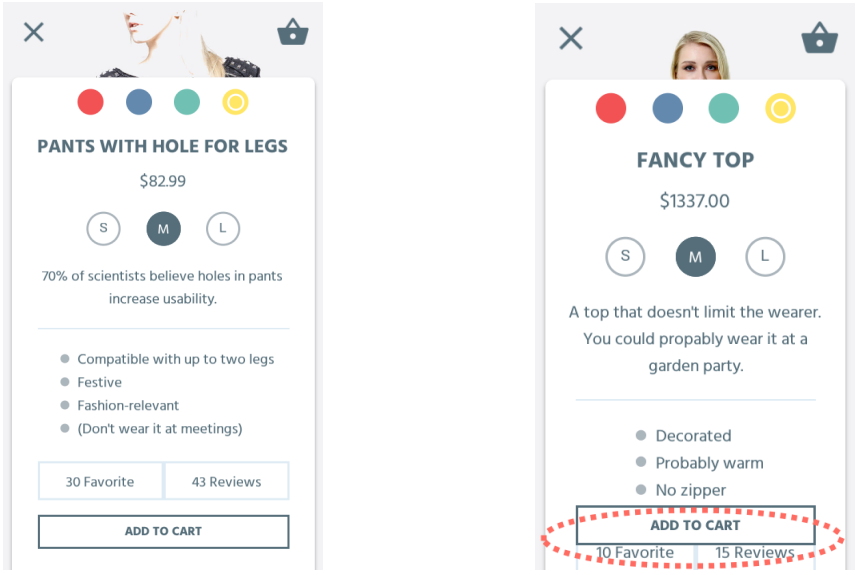
We also want to approach answering this question with the hopes of being able to produce a system that is not only fully automated, but that also required minimal work to set up and maintain. This ties into the final research question.

- **RQ3:** How can we improve the quality of GUI layout testing by covering as many as possible of the GUI layout errors defined in RQ1?

With the final research question, we want to improve the state of the art when it comes to discovering GUI layout errors. Our goal is to be able to increase the quality of testing by increasing the accuracy to which we are able to discover layout bugs.

3.2. Layout errors

When creating GUI layouts for mobile devices, the layouts needs to be valid for a wide variety of different screen sizes with a huge variety of data inputs. It can be a difficult task to validate that a set of layout properties do not lead to overlapping controls when the screen becomes smaller than originally expected (this happens often because the GUI was never designed with all targeted screen sizes in mind) or when a users input is larger than originally assumed. Another common problem is that layouts are often designed with only mock data in mind. By "mock data" we mean the artificial data that is often used when both designing and implementing GUIs. A common theme with mock data is that it is hard to represent all real world cases, and thus, we can easily end up with a design that was never tested on, for example, really large data [3]. These cases can be hard and cumbersome to test for unless one is using an automated process.



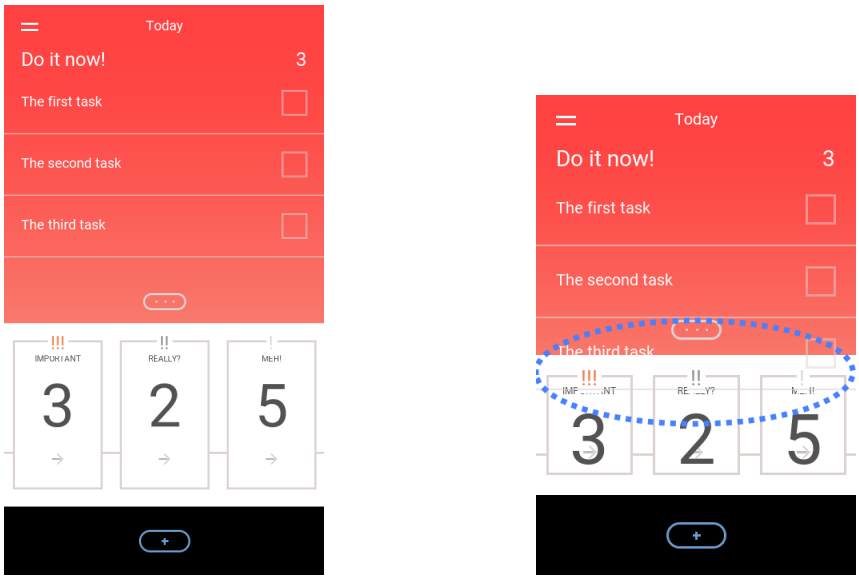
(a) E-commerce app with expected size

(b) E-commerce app on a smaller size screen

Figure 16. E-commerce app example.

Figures 16 and 17 point out examples of layout errors that can easily arise as a result of untested GUI layouts. Some of the errors are very easy to spot by visual inspection, while some of them are more subtle. It can also be difficult and expensive to define tests for all elements to ensure that they are

[3] A common mistake is to wrongly assume the length of names, and to not have text boxes that accommodate really long names.



(a) Overlay-menu app with expected size

(b) Overlay-menu app on a smaller size screen

Figure 17. Overlay-menu app example.

displayed correctly. Another problem with this is the fact that layouts often change quite drastically during development, and even just as a result of it using dynamic data. We therefore propose that much of the potential value of having an automated layout testing system is for it to be fully automated. By this we mean that it should be able to detect common layout mistakes without any custom test script. The focus of this thesis is therefore on identifying these common errors. Classifying them and figuring out a way of automatically detecting them. We also identify some edge cases where our proposed classification might not apply.

3.3. Automatically detecting layout errors

Creating dynamic GUIs that adapt to a variety of different screen sizes and user inputs is hard. There are many different approaches (like [3],[9] and [26]), all with their pros and cons. A problem that has yet to be standardized is the problem of making sure a certain GUI layout is valid for all targeted screen sizes and user input combinations. With todays ever expanding world of mobile GUIs, where an app is expected to be run on tens to hundreds of different screen sizes [27], the issue of validating layout hasn't been properly addressed. The approach discussed in this work combines ideas from constraint based GUI system, with hierarchical GUI systems to automatically

detect errors and flaws in seemingly sound GUI layouts.

3.3.1. Answering RQ1: Defining valid and invalid layouts

Part of the work done in this thesis is to develop a set of properties and algorithms that can be used to discover bugs in a GUI layout. In this work we discovered a categorization of layout errors, which are commonly found in GUI layouts. There are however several exceptions, where these cases should not be considered errors, which are discussed later in this chapter. We do however propose a system for dealing with these cases, which is implemented in LBH and discussed in section 3.4.4.1.

The methodology used for answering this research question was a mix of personal experience with creating app layouts, and especially with the experience of them having layout bugs, and visual inspection of various example apps on different screen sizes to try to identify layout bugs. A lot of the insight into the nature of the various layout bug types was also gained from the implementation of these ideas in LBH, which is presented in section 3.5. The two processes very much affected each other, and so throughout the semester writing this thesis I've been going back and forth between the implementation of ideas, and the tweaking of ideas based on results from the implementation. The implementation of the ideas presented in this thesis is an important part of the work, since the ideas need to be implementable to eventually be of any value to developers.

In the following sections, we will discuss the various kinds of layout errors discovered and the resulting classification.

Horizontal overflow In this case a Text item, sized to be a percentage of the width of its available space, seemingly fits its text within its bounds. We can then observe that this is no longer the case in certain screen sizes, or with different text input if the text is dynamically fetched from a web server.

```
<Panel Width="50%" Alignment="Center">
    <Text Value="This is some realistic text" />
</Panel>
```

Since the `Width` of the `Panel` containing the `Text` item is sized relatively to the screen size, it runs the risk of not being big enough for its text. An example of this can be seen in Figure 18, where the text is longer than its its container and is therefore drawn outside of its bounds.

Vertical overflow This case is almost the same as the above, but acts slightly different with text because of text wrapping. If text wrapping is on,

Figure 18. Horizontal overflow.

it will overflow more and more based on text length, and both the width, and the height of the available size.

```
<Panel Height="20%" Alignment="Center">
    <Text TextWrapping="Wrap" Value=""Lorem ipsum dolor sit amet,
        consectetur adipiscing elit,
        sed do eiusmod tempor incididunt ut
        labore et dolore magna aliqua.
        Ut enim ad minim veniam, quis nostrud
        exercitation ullamco laboris
        nisi ut aliquip ex ea commodo consequat."/>
</Panel>
```

In the above code, we can see that the `Text` item has its `TextWrapping` property set to `Wrap`. What this does is to cause the text to wrap around to the next line if it becomes longer than the width of its container. However, since its containing `Panel` has its `Height` property sized relative to the screen size, the text will potentially wrap too much and become too long in the vertical direction for its container. An example of this can be seen in Figure 19 where the text is too long, and because it is configured to wrap around, it becomes too big in the vertical direction instead.

General overflow Vertical and horizontal overflow are quite similar, and in our implementation (discussed in section 3.5) we don't actually make any real distinction between them. We do discuss them separately here though, because they usually manifest because of different reasons.

It is common to define custom container controls that group and decorate its content. An example of this is the card design commonly associated with

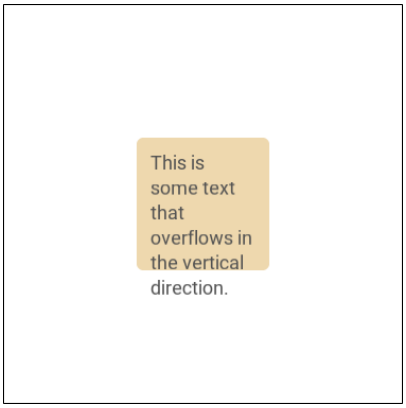


Figure 19. Vertical overflow.

material design guidelines [14]. It is however possible to assign content to these cards that has a minimum size too big for the card at certain sizes, meaning the content will overflow or be clipped by the card bounds. In this case we can end up with just general overflow bugs in either the vertical, horizontal or both directions.

Figure 20 shows an example of a general overflow case where the container ends up being too small for its content. There is both vertical and horizontal overflow in this figure, and the bug has many possible ways of being fixed. Either we would force the container to be big enough for its children. We could add text wrapping to the text item so that it doesn't overflow in the horizontal direction, and we could change the aspect ratio of the image. It is important to keep in mind that there errors points to flaws in a design, which can typically happen if dynamic data and different screen sizes are taken into account.

Overflowing list In many cases we use the StackPanel to stack multiple elements on top of each other. This form of layout is very simple and predictable, but very prone to causing invalid layouts for shrinking screen sizes. The system possibly need more domain knowledge in order to validate based on whether certain parts of the GUI is reachable or not, as discussed earlier. A clipped list should be considered valid if it is inside a `ScrollView` with the same orientation as the direction as that of the overflow. Figure 21 shows an example of a list which has become too long for its container and ends up overflowing in the vertical direction.

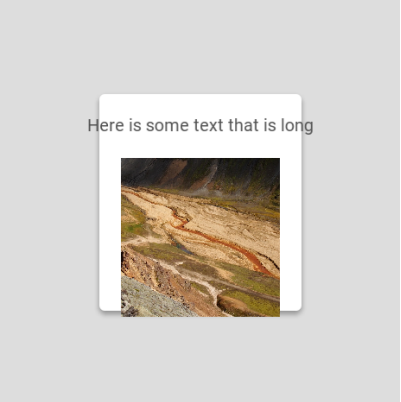


Figure 20. General overflow.

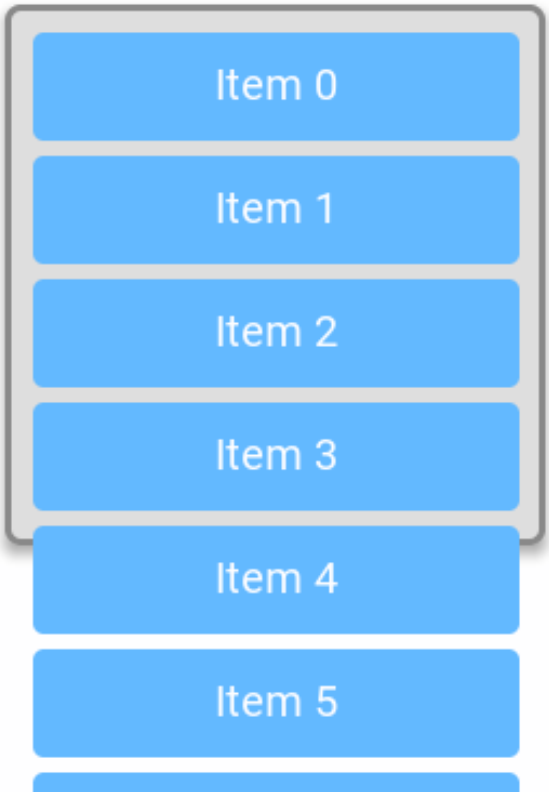


Figure 21. Overflowing list.

Figure 22. Overlapping hit boxes. The gray stroked rectangles are not a part of the app, but there to visualize where the user can tap to hit the button. The figure shows how we can have overlapping hit boxes without the buttons actually overlapping.

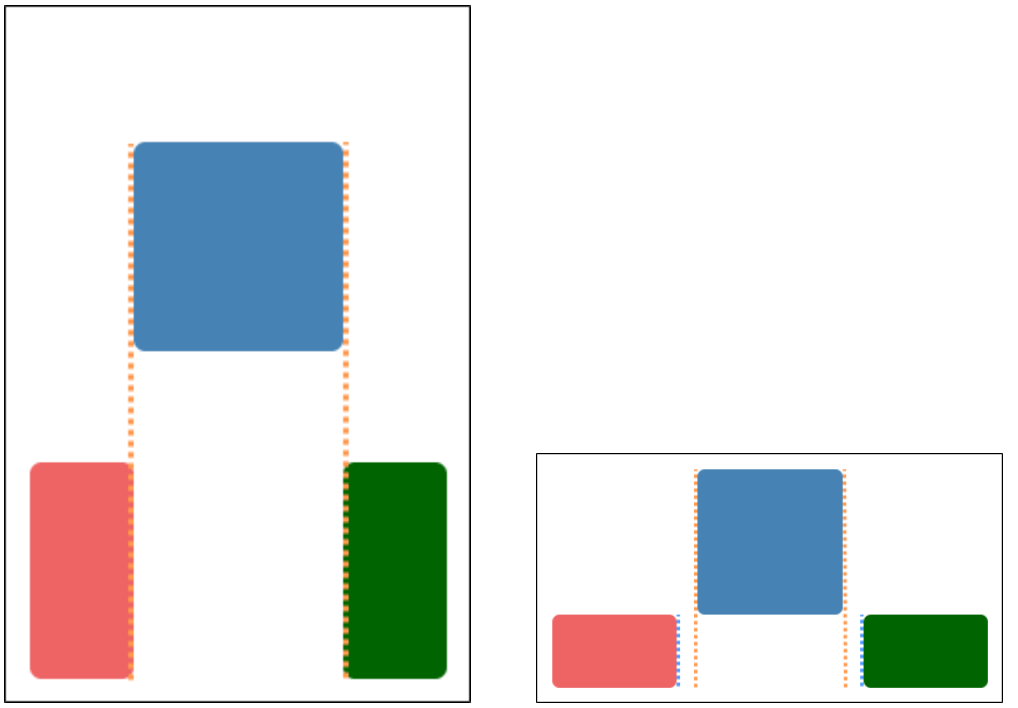
Overlapping hit boxes A hit box is an invisible shape (usually a rectangle) which is used by interactable controls to define the area on screen through which the controls is hittable.

Some controls have an intrinsic minimum size in order to maintain usability. This includes most interactable controls, like buttons, sliders and switches, which should have a minimum size in order to be easy to hit. While a design might be designed with three buttons spread out horizontally, this might be too many for a smaller screen, and we end up with overlap on the hit boxes, even though the button visuals do not overlap. This can severely decrease usability without being easy to spot even by manual visual inspection. Figure 22 shows an example of buttons whose hit boxes are larger than their visual appearance. In this case, the buttons are placed so close to each other that their hit boxes end up overlapping. The gray outlines are just there to indicate the size of the hit boxes.

Alignment changes between screen sizes Elements in a layout often end up aligned to each other in one or more ways. This alignment is usually intentionally added by the developer using layout containers like for example a Grid or StackPanel. While we can expect the children of a Grid to be aligned to each other (since this is the main function of a Grid), we can not as easily verify this for the cases where elements are not direct siblings of each other. There could be that each item of a grid has children of their own, which should be aligned to the children of the other elements in the grid. This is however a common limitation with hierarchical layout systems, like the one used in this thesis.

Because of this, we can easily end up with a situation where our layout

loses alignments due to changes in screen size or data. Figure 23 shows an example of this, where three elements are aligned to each other vertically, and then loses this alignment when the screen size changes.

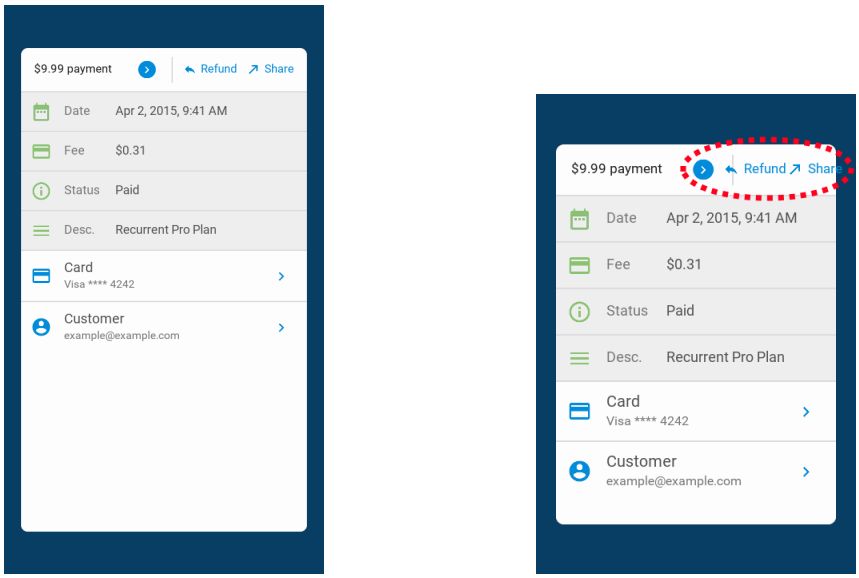


(a) Shows the three elements being aligned.

(b) Alignment has been lost due to changes in screen size.

Figure 23. Shows how elements can lose alignment due to changes in screen size.

Overlap Overlap is when two elements, who are siblings (meaning that they belong to the same container, or parent element as we sometimes call it), are drawn partially over each other. We usually treat these cases are layout errors, especially if the two elements in question are not overlapping on some screen sizes, but then are overlapping in others (especially when run on a smaller screen size than the GUI was designed on). There are of course cases where overlapping elements should not be considered an error, for example when it is an intentional part of the design and put there for aesthetic purposes. There however cases where we can be certain overlap is causing an error, like if the overlapping element contains text, or is an interactable control (meaning that the overlap potentially blocks the user from interacting with it). Figure 24 shows an example of overlapping elements.



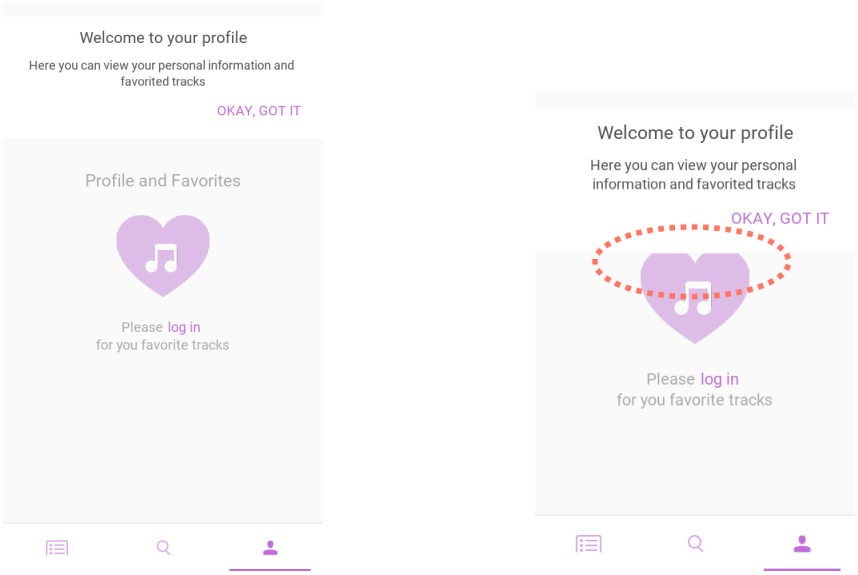
(a) Reveal-actions app with expected size

(b) Reveal-actions app on a smaller size screen

Figure 24. Reveal-actions app example, showing an example of overlapping elements.

Clipping across layers Clipping across GUI layers should in many cases be considered an invalid layout, but some times it is simply a part of the intended design. A sensible default might be to detect whether two panels overlap at some screen sizes, but do not at others. This technique might also be applied to several other proposed default rules. Figure 25 shows and example of this. Notice how the heart shape becomes clipped by the "welcome" message element on the smaller screen size.

Defining reachability Most apps use scrollable areas to display more content than can fit on a single screen. This means that the valid area of a page in an app can be larger than the screen it is drawn on. This is important for the system to pick up, and thus, it needs to be aware of the concept of scrollable, or movable drawing areas. Fuse supports several types of containers that can be moved so that it supports the display of more content than can fit a single screen. The two main types is the ScrollView, which allows the user to move the content up or down (it can also be configured to scroll in any direction), and the PageControl, which allows the user to swipe between several pages. Figure 26 illustrates how an app can display content larger than its screen size, making it reachable by using a `ScrollView`. This means that in the case where we are using one of these scrolling containers,



(a) Welcome screen with expected size

(b) Welcome-screen app on a smaller size screen

Figure 25. Welcome screen example, showing how an element can be clipped by another on a smaller screen size.

we should not report an overflow as an error (since it is a reachable overflow).

3.4. Automatically detecting GUI layout errors

In this section, the proposed solution to the RQ1 will be presented in detail. While this section focuses on the insights gained from the initial research done, a lot of insight was gained during the implementation phase of the project. The technical aspects of the project are thus discussed in more detail in chapter 3.5, since it was such a central part of the research.

3.4.1. Introduction to The Layout Bug Hunter

The system proposed in this thesis focuses on automatically discovering layout related bugs in a GUI. It does so by making certain assumption about what kind of bugs are usually found in GUI layouts. It also makes some assumptions about what the hierarchical relationships between elements in a GUI can tell us about a designer or developers intentions.

The system uses layout data gathered from a running app instance to detect errors in the layout. This is needed in order to get the correct layout values since we need to run the actual layout engine to get these. An im-

Figure 26. Example of how a ScrollView can be used to display content larger than the screen.

portant aspect of the system is that it should not require the developer to manually create test script, but should still be tweakable to some extent.

It would also be important to have an opt out feature to allows for the possibility of turning off testing for certain parts of the layout hierarchy. The implementation of this feature is however outside the scope of this work.

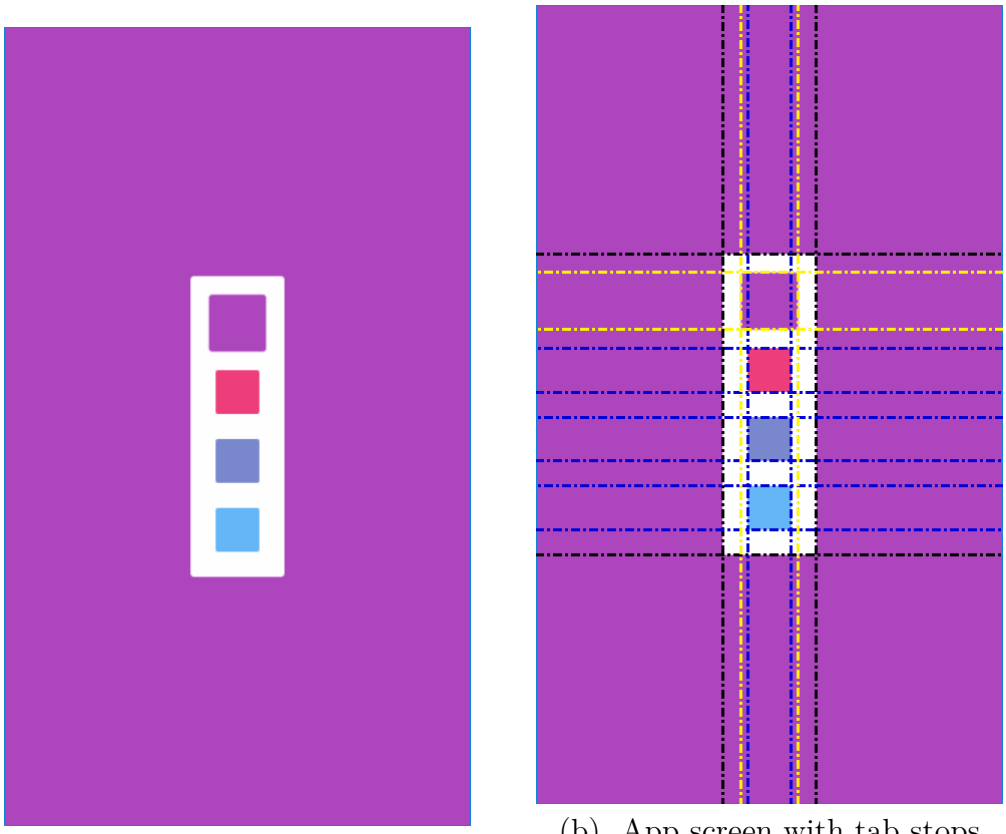
3.4.2. What data do we need to collect?

The system requires certain information about the layout to be gathered on the client running the app that is being tested. While some layout errors (like overlap and overflow) can be discovered by simply gathering the information for one single test case, some layout errors can only be discovered by establishing a baseline over several test cases and using that result to discover outliers. For example when we check for alignment changes, we need to check for changes between more than one test case.

Element positions and sizes The main data gathered on the client running the app are all GUI elements position and their sizes. A realization made from the implementation phase is that many GUI frameworks, and especially the one used in this project, differentiates between element size, and render size. The element size is the area used by the layout system for sizing, while the render size is the size used for actual rendering. The rendering size can in some cases (like for example when we are drawing a drop shadow) be larger than the element size. This can be seen in Figure 28.

Hierarchical relationships between elements An important part of the data gathered is information about the relationship between the various elements visited. The GUI system used in this work uses a hierarchical structure for its elements, and the resulting structure is thus a bidirectional acyclic graph.

Global set of tab stops Another very important set of data required by the layout validation algorithms are a global set of unique tab stops and which elements they are connected to. “Tab stop” is a term borrowed from linear programming based GUI frameworks. A tab stop is a vertical or horizontal line, that divides the screen in two half spaces. This line is used by linear programming techniques to align GUI elements to in order to create all types of layouts, like grids, stacks and docks. With linear programming based GUI frameworks, establishing these tab stops, and assigning elements to them is a part of the developers responsibility. In our case however, we use the



(a) App screen without tab stops

(b) App screen with tab stops marked

Figure 27. Figure illustrating all tab stops for an example app screen.

data gathered about the elements to discover all unique tab stops and which elements they are aligned to. Each element with a Width and Height larger than 0 is aligned to 4 different tab stops. This data becomes an important step to discovering alignment related errors. Figure 27 shows an app screen with all tab stops marked with striped lines.

3.4.3. What types of layout errors can we discover?

A major part of the work done in this thesis was discovering what types of layout bugs can occur during the development of a GUI application. In this section we will discuss a classification of common layout errors, how they manifest, what their characteristics are and how they can be detected. We will also discuss common edge cases where these classifications don't always apply.

The discussion below is based on the assumption that the GUI layout

system used is based on a hierarchical model, as presented earlier in this thesis.

Overflow Overflow is a class of errors where a child element is bigger than its parent container element in one or more directions. Overflow usually occurs when a child elements size is based on absolute values set by itself or by one or more of its children. Some controls, like text and buttons, have an intrinsic size based on their content. If the container of that control has its size based on values that are relative to the screen size, the child element can end up needing more space than the parent container can allocate.

This class of bugs usually manifest by items being clipped, or covered by the overflowing element. In some cases, this can be hard to discover by visual inspection depending on the severity of the overflow.

The overflow category has a few edge cases that was identified during the implementation phase of this thesis:

Edge case: Image based effects Many applications use image based effects like drop shadow and blur. When we validate using the rendering size (the size of the bounds that are actually used for drawing), a drop shadow or blur effect can cause the element to report a size that is larger than the available size given by its parent, while still being a valid element. This happens because shadows often are drawn with an offset with respect to the element.

An example of this can be seen in Figure 28, where the shadow of the rectangle is clearly drawing outside of the bounds of the rectangle. It can also be seen in Figure 29 where a rectangle has had a blur effect applied to it, causing it to draw outside of its render bounds. The render bounds are marked with dark stroked rectangles.

Edge case: Intentional clipping In some cases, a container is used not only as a means of performing layout on its children, but also to intentionally perform clipping, in order to hide certain parts of the element. This happens very commonly when using scroll views. Scroll views are views can contain content that is bigger than itself, allowing users to interact with it to move the content around. They are commonly used to display lists of content that the user can scroll through vertically.

Our system thusly has to account for this fact, and be aware of the fact that scroll views should be considered intentional clipping.

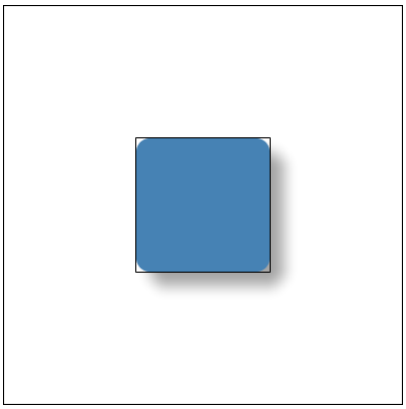


Figure 28. Shows how a drop shadow effect can cause an element to render outside of its bounds.

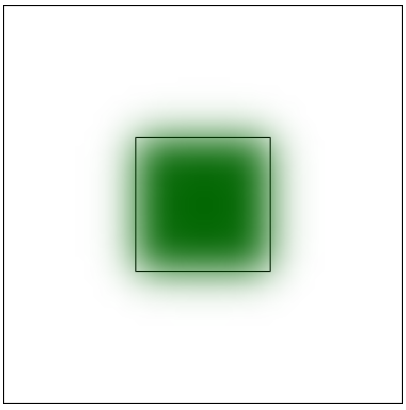


Figure 29. Shows how a blur effect can cause an element to render outside of its bounds

Overlap Overlap is a class of errors similar to overflow, but occurs for a different hierarchical relationship than overflow does. Overlap is reported for cases where sibling elements (elements who have the same parent element) have intersecting render bounds.

There is, however, a criteria that has to apply for this intersection to be considered a layout bug:

Complete containment For the case where the overlap is complete, meaning that one of the elements in question is completely inside the other element, we consider it to be an intentional design choice. The reasoning behind this is because this is a common pattern used to make a background fill for elements.

However, if the intersection rect is smaller than the smallest of the two elements, we know that we don't have a case of complete containment, and the case is reported as a layout error.

Edge case: Overlap for aesthetic purposes Overlapping sibling elements is actually quite commonly used in app designs, and thus a big potential source for false positives. A technique for reducing the chance of reporting false positives is therefore discussed in a later section (see section 3.4.4.1) . Figure 30 shows an example of how overlap can sometimes be used to aesthetic purposes.

Alignment changes Alignment errors is a class of layout bugs that can be quite difficult to discover by visual inspection. An alignment bug is a case where elements who are aligned, either vertically or horizontally, lose that alignment due to changes in data or screen size.

Most cases of alignment are due to the developers choice of layout containers. A grid or a stack for example, are great tools for making both vertically or horizontally aligned elements. In some cases though, the alignment was not explicitly declared by the developers choice of layout containers, but emerged due to the correct choice of margins, paddings and sizes. In this case, the alignment is still deliberate, but can more easily break due to unforeseen data sizes, or changes in screen size or aspect ratio.

Alignment changes are reported as a result of changes in the number of tab stops, or for when tab stops lose or gain aligned elements.

3.4.4. Possible false positives and false negatives

Determining whether a GUI is valid or not is in many ways a subjective decision. We have made an attempt in this thesis to define and classify

Figure 30. An example of how overlapping elements can be used for aesthetic purposes.

layout errors, how they manifest and how they can be identified, but even so, we run the risk of falsely reporting a layout which the designer considers to be valid, as invalid. In this section we propose some techniques which can decrease the chance of reporting false negatives, and false positives [4].

Reducing false positives by establishing a baseline The error classes discussed in the previous section makes many assumptions about what should be considered valid and invalid layouts. While we can easily argue that these error classifications can in many cases be a strong indication of actual layout bugs, the chance of reporting a false positive is quite high.

In this section we will propose and discuss a simple solution that could decrease the chance of reporting a false positive by establishing a baseline for each test run. Errors will only be reported if their error characteristics deviate from the baseline in a significant way.

How can we establish a baseline? As discussed later in the implementation section, the system will gather information for all test cases before it starts on the validation phase. We can then check each error initially reported by LBH and check how many of these errors are preset in each

[4]By false positive we mean that the system reports a valid layout as an error. By false negative we mean that the system reports an error as a valid layout.

test case. We can then create a baseline of errors containing only the errors that are present in a certain amount of the test cases, which indicates that this error, since it is always/often present, was intentionally added by the designer/developer.

3.4.5. Tweakability

To further decrease the risk of reporting false positives and false negatives, we allow the user to tweak certain aspects of the system.

There are several possible aspects of the layout validation that could benefit from being tweakable. In this section, we briefly discuss a selection of possibilities for adding tweakability.

Letting the user set a threshold for tab stop merging. For complex GUIs we might end up with a lot of tab stops which are almost on identical, but only separated by a one or a few pixels. By letting the user customize a threshold for merging two tab stops, they can lower the risk of false reporting for cases with a larger concentration of tab stops. By making the threshold larger, more tab stops will be merged, and the chance of an element being reported as having lost alignment will be lowered. Tab stop merging was however deemed to be out of scope for this implementation of LBH.

Letting the user set a threshold for when errors should be added to the baseline. By letting the user tweak parameters of the baseline detection, they can have some control over how represented a certain error has to be in order for it to be considered a part of the baseline. One could for example use a default of including an error in the baseline if it is present in more than 50% of the test cases. In some cases this default might lead to too many false positives, and so letting the user increase this number might be beneficial in some cases.

3.5. Implementation

The system proposed in this thesis was implemented in code and tested on a series of example app screens. In this part, the system design and implementation will be presented. The source code is openly available on GitHub [24].

3.5.1. Implementing the layout bug hunter tool

In this section the implementation of the proposed system is presented. While the theoretical aspects of the solution has been presented in section 3.4, this section is concerned with the technical parts. We discuss the various technologies used, the architecture of the system as well as details about the algorithms developed as part of the layout validation. The various data structures used are also presented in detail.

3.5.2. Tools used

Here follows a very brief explanation of the tools used, and why they proved to be useful choices in the implementation of this system.

Fuse / Uno Fuse [9] is a cross platform app development toolkit which lets one write native apps that run on both Android and iOS. It is the mobile app development platform with which i have the most experience, and thus was the natural choice for this implementation. The work done on this platform consist of two main pieces. A view crawler that gathers information about all visible elements in the app and renders it to a JSON structure. I also put together a small network component that could talk to the layout validation server through TCP. These components were implemented using the Uno programming language, which is considered to be a dialect of C# and powers the Fuse platform. While Fuse allows us to build native iOS and Android apps, it is also able to export to Windows using .NET [17] or MacOS using Mono [16]. This allows us to perform our testing on a desktop machine.

Rust The Rust programming language [20] was chosen for the implementation of the layout validation code itself. Rust is a relatively young systems programming language backed by the Mozilla Foundation. It was chosen because of its extremely useful memory safety guarantees and its features easing the development of concurrent systems, which made this project a lot easier to develop while easily avoiding a huge number of bugs usually encountered in C/C++ projects. It is also cross platform and easily inter-ops with native libraries. It was therefore easy to implement a component that would automatically resize a window (to test multiple sizes of an application) on the Windows 10 operating system using the Win32 API.

3.5.3. Architecture

Since the system clearly defines what information it requires in order to perform its validation, it was implemented as a service to which clients could

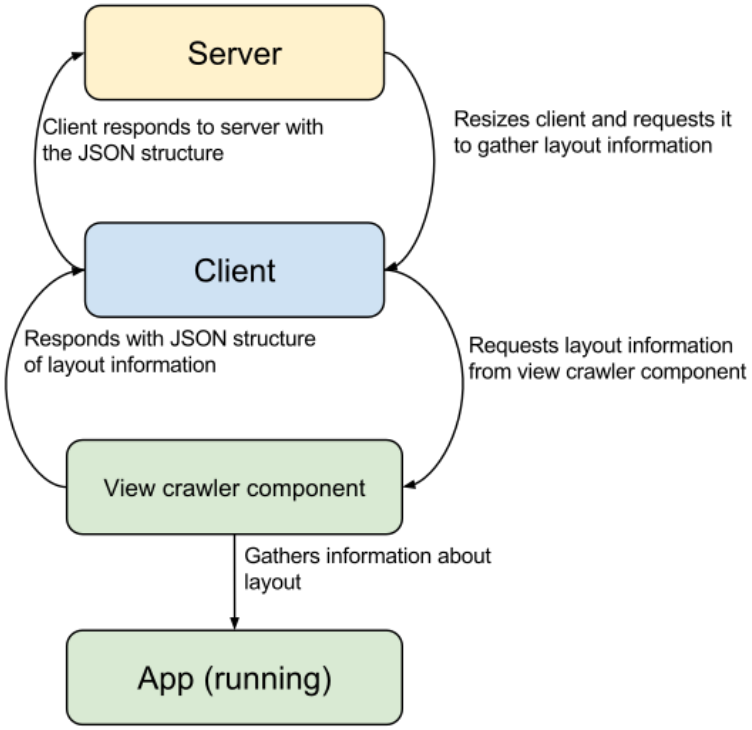


Figure 31. Layout bug hunter architecture.

connect to over TCP. This approach was chosen so as to make the system more easily extensible to other platforms than Fuse, such as for native Android or iOS, or even for web technologies. The system is now more portable because only the data gathering and communication protocol code has to be reimplemented per platform. The actual layout validation code is implemented in the server, and is thus reused for all platforms.

In this implementation, however, we also made the server itself start up the app so as to control a handle to its process. This way we could easily acquire a window handle to the running process so that the system could automatically resize the window and request layout information from the running app. It is important to note that the app has to be running for us to get the correct layout data. This is because the app runs its own layout engine, which generates the actual layout data using the projects UX code as explained in section 2.3.2.

Figure 31 shows the components of LBH and how they interact with each other.

3.5.4. Phases

LBH's work is divided into a set of phases, each performing a set of tasks who's results feed into the next phase. Figure 32 shows a flow chart illustrating the various phases and the steps involved in each phase. The phases can be summed up as follows:

- **Phase 1 - Data collection:** This phase is run in two stages. Data is gathered about the layout as it is presented by the running app. This data is then processed in a second step that discovers more meta data about relationships among the various pieces of data.
- **Phase 2 - Layout validation:** This phase uses the collected and processed data to detect layout errors of various types. Each error type has a specific algorithm associated with it, and also makes certain assumptions about layout in general. The error types as well as their algorithms will be discussed detail. The result of this phase is a set of structures containing information about the violating nodes as well as information about how the error manifested.
- **Phase 3 - Error sorting:** The errors are sorted according to type and severity so as to make the violation rendering phase more structured and meaningful and to aid in the error filtering phase.
- **Phase 4 - Error filtering:** This phase is concerned with filtering out the false positives (cases where the system reports an errors which was actually intentionally added by the developer/designer or which just shouldn't be considered errors) reported by Phase 3. It does this by establishing a baseline of accepted errors. This means that the system makes the assumption that there is a configuration of screen size and data for which a valid (in the sense that it reflects what the developer/designer expected on screen) layout exists. The baseline is automatically discovered using set of heuristics that recognize the same error reported across different test cases. It then makes the general assumption that the more present an error is across test cases, the more likely it is that it should be a part of the baseline, and therefore have its error filtered out the result. This phase is tweakable by the user.
- **Phase 5 - Error rendering:** Detecting layout errors is meaningless unless the results can be reported in a useful way to the user. This phase is concerned with the visual representation of detected errors so that the user can effortlessly notice which part of the app is in violation.

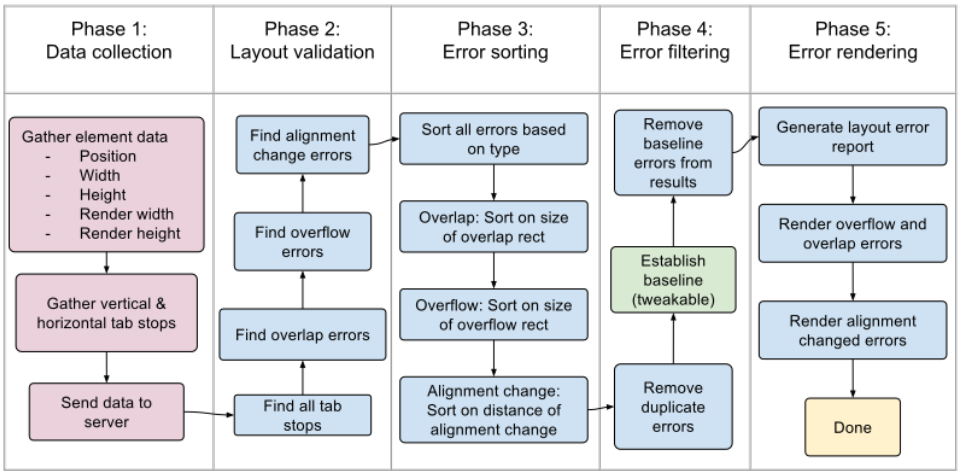


Figure 32. Layout bug hunter phases flow chart. Each column represents a phase in LBHs execution. The steps marked with (tweakable) can be tweaked by the user.

3.5.5. Phase 1: Data collection

The component that actually performs the validation, requires two pieces of information:

- A list of GUI elements and information about their size and position (from now on called "nodes")
  - This list is supplied directly by the connecting client
- A list of vertical and horizontal lines representing the alignment of the various elements in the GUI (from now on called "tab stops")
  - This list is generated as a preprocessing step over the data gathered from the client

`Nodes` structure:

```
Node {
    id: Number
    parent: Optional Node Id

    position: (Number, Number)
    size: (Number, Number)
```

```
    renderSize: (Number, Number)
}

Nodes {
    root_size: (Number, Number)
    nodes: Map from Node Id to Node
}
```

Each node has a position, which marks the position of the nodes top left corner, relative to the screens top left corner. Instead of each node listing all its children, each node has an optional parent id number which can be used to locate its parent.

The `Nodes` structure contains a map from Node Id (which is simply a number) to the corresponding Node. Through a small library developed as a part of this structure, we can easily navigate around the node structure while also easily map over all nodes without having to traverse a complicated tree structure.

This reflects how the system treats each node equally to each other node, but how the parent child and sibling relationships plays a role during the validation phase.

`TabStops` structure:

```
TabStop {
    orientation: Vertical | Horizontal
    position: Number
}

TabStops {
    tab_stops: Set of TabStop
    nodes: Map from TabStop to Set of Node Ids
}
```

A tab stop represents a vertical or horizontal line that splits the screen space into two halves. Each tab stop is represented a position and an enumeration type (as usually found in OOP languages like C# or Java) specifying whether it is a horizontal or vertical tab stop.

A tab stop collection consists of a set of unique tab stops and a map from tab stop to a set of unique node IDs.

This means that we know which nodes are connected to each tab stop. This information is used by the system to figure out alignment changes as a result of resizing of the screen or the elements in the app.

The system requires that we collect data for all test cases that are to be

a part of the validation step before we can go on to the next phase. This is because the validation phase needs to analyze the variation in number of tab stops and their connected nodes in order to figure out how element alignment has changed during test cases. It can also use this global information to figure out what the base line is.

Node Side As a way of uniquely identifying the case of a nodes side being aligned to a tab stop, we have a structure called NodeSide, which group a node id, a side enumeration type, and a tab stop id:

```
Side {
    Left,
    Top,
    Right,
    Bottom
}

NodeSide {
    node_id: Number
    tab_stop_id: Number
    side: Side
}
```

The Client The app to be tested, from now on called the client, implements a crawler component, that is responsible for navigating the hierarchy of nodes, and collecting its data. The data collected by the client is converted to JSON and is represented "in-line" as a hierarchy of nodes where each node has a list of child nodes.

The Server The data sent from the client is collected by the validation system, from now on called the server, which visits all nodes using a tree visiting algorithm, like depth first search, and flattens the list into the format explained in the previous section.

3.5.6. Phase 2: Layout validation

During the layout validation phase, various rules are applied to the data gathered from phase 1, one at the time. The validation phase as it has been implemented in this work, is split into a collection of rules:

- overlap

- overflow
- element alignment (specifically for elements losing alignment)

Each rule is implemented by its own dedicated algorithm, and reported by its own dedicated structure.

Following is a description of each rules implementation. The discussion and reasoning behind each rule is discussed in a previous section

Category: Overflow The overflow category checks whether elements are drawn within the bounds of their parent. The assumption here is that an overflowing element, which can be caused by an absolutely sized control, like a button, being inside a dynamically sized container, can easily lead to visual glitches when tested on screen sizes smaller or of a different aspect than the one used by the designer.

Overflow algorithm The algorithm extracts all pair of nodes that have a parent-child relationship. It then checks whether the child is within the bounds of its parent by performing an intersection test between the rectangles represented by the two elements.

The algorithm then does an intersection test between the overflowing and containing elements, and breaks the overflow rect into several smaller rectangles so that it can accurately report which areas are represented by overflow.

Overflow report structure The report structure is as follows:

```
Overflow {
    parent_node: Node
    child_node: Node
    overflow_rect: {
        left: Optional Rect
        top: Optional Rect
        right: Optional Rect
        bottom: Optional Rect
    }
}
```

The report structure of the Overflow category consists of a collection of up to 4 rectangles that each cover one of the sides for which there could possibly be overflow. The left and right rects are used in the case of horizontal overflow, while the top and bottom rects are used in the case of vertical overflow.

Category: Overlap The overlap rule is concerned with checking whether two elements are *partially* drawn on top of each other. This rule is a bit special, since it contains several identified edge cases where we would consider the layout to be valid even though an overlap has been discovered, an example being that the overlap was intentionally added by the designer for aesthetic purposes.

Overlap algorithm The overlap algorithm is similar to the one used to detect overflow, except that it is only concerned with nodes that have a sibling relationship. The system first extracts all unique pairs of nodes which have the same parent node. It performs a rectangle to rectangle intersection test which returns the intersection rectangle between the two nodes.

There are a few cases which the system considers to be valid even though it finds an intersection:

- If the overlap appears in all test cases, we consider it to be a deliberate overlap (which is sometimes added intentionally by the designer).
  - This is different from the overflow test, where we always consider it to be a layout violation.
- If one rectangle is fully contained within the other, we expect it to be a deliberate background / foreground relationship.

Overlap report structure The report structure is as follows:

```
Overlap {
    node_a: Node
    node_b: Node
    overlap_rect: Rect
}
```

Category: Alignment change This rule works under the assumption that elements that belong to the same tab stop do so deliberately. This is often the case for element arranged in a grid or stack, but this system is also able to detect this for elements who end up in alignment independently from the layout structure it belongs to, which is often the case.

Alignment change algorithm The algorithm works by looking at all the identified node sides and which other node sides they are aligned to. We then compare between the different test sets to see if the groups of aligned node sides differ, and report an error each time this is the case. We then group all

the reports which report on the same pair of nodes, and group their test set ids together so that we only report each node pair once.

Alignment change report structure The report structure for the alignment change rule is as follows:

```
AlignmentChange {
    node: (NodeSide, TestSetId)
    lost alignment to: (NodeSide, TestSetId)
}
```

3.5.7. Phase 3: Error sorting

After the layout errors has been detected they are sorted according to a severity rating decided decided by the system according to the following criteria:

- Error type: the error types themselves are assigned a different degree of severity by default, but can be tweaked by the user.
- Within each violation type, the severity calculations are done individually
  - Overflow:
    - In this case, the total overflowing areas are calculated by adding together the area of each component of the overlap rectangle. The larger the total area, the higher the severity rating it gets.
  - Overlap:
    - The area of the intersection rect is used to determine the severity rating. The larger the area, the higher the rating.
  - Alignment change:
    - The difference in position between each involved tab stop is used to assess the severity. The bigger the difference, the higher the rating. If two violations have the same position difference, but different orientation, vertical orientations are given a higher severity rating by default. The reasoning behind this is that most apps have a natural vertical flow, which makes vertical alignment changes more noticeable than horizontal alignment changes.

3.5.8. Phase 4: Error filtering

In the error filtering phase we are concerned with getting rid of the false positive errors which was reported by phase 2. Each overlap and overflow error reported by LBH is connected to a test case (combination of screen size and data). Alignment lost errors on the other hand are connected to two test cases, since the losing of alignment happens because of a chance from one test case to the other. Because of this, we filter overflow and overlaps separately from alignment lost errors. We start by filtering out what we consider to be duplicate errors. An example of this is that for each overlap, two errors are reported, one from the perspective of the first element, and one from the perspective of the second element. With alignment lost errors we merge all errors that are reported for the same pair of nodes, and maintain a list of each pair of test cases for which this errors occurs, as well as a count of the number of times this error occurs.

After getting rid of duplicate errors, we apply a heuristic to each error type to figure out if it should be a part of the baseline or not. We apply the following heuristics for each error type:

- **Overlaps and overflows:** Overlaps and overflows are filtered by counting the number of times the same error occur across test cases. We then see how many instances there are compared to the number of test cases. If the error occurs the same amount of times as there are test cases, we consider it to be a part of the baseline. The reasoning behind this is that if an error is always present, it was probably put there intentionally. The goal of our tool is to find errors that occur as a result of change between test cases.

This step is tweakable by the user, who can adjust how large a percent of the test cases an error has to be present in for it to be added to the baseline.

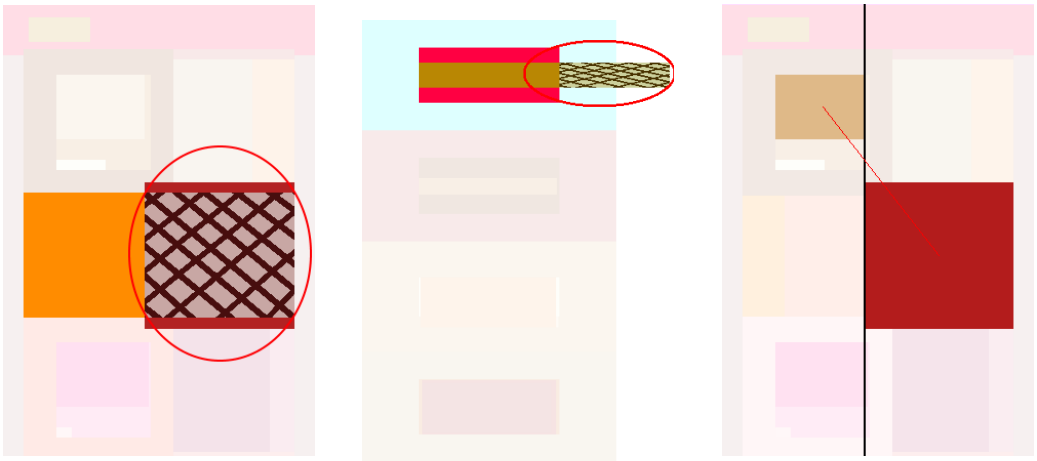
- **Alignment lost:** Alignment lost errors are filtered by a slightly different heuristic than overlaps and overflows. Since an alignment can only change between test sets, we are interested in knowing how for how many test set pairs an error has occurred. Since two node sides are either aligned or not aligned, the maximum number test set pair that can be involved in the same alignment lost error is one less than the number of test sets. In this case, there was an alignment between two node sides one time, and then they were unaligned in all the other cases. The idea behind our heuristic for adding alignment lost errors to the baseline, is to consider how many times the nodes involved in the error were aligned relative to the number of test cases. If we for example have 5 test cases, where two node sides were only aligned once, we

can probably assume that this alignment was unintentional and only occurred because the two node sides were randomly on the same tab stop. If, on the other hand, the two node sides were aligned in four out of the five cases, the one time it was not aligned is probably a real layout error, and is thusly not included in the baseline.

This part is also tweakable by the user, who can adjust how many cases an alignment has to have been a part of in order for it to be included in the baseline. The value supplied by the user is a number between 0.0 and 1.0, which is then multiplied by one less than the number of test cases. We use one less, since if an alignment is present in all test cases, it would never be reported as an error in the first place.

3.5.9. Phase 5: Error rendering

Phase 5 is concerned with presenting the results of the layout validation to the user. It first draws all nodes with different colors so that the user can see an outline of the app. This rendering only displays squares, but could be extended to more accurately display the data gathered from the app. It could possible also use an actual screen shot of the app in this phase.



(a) Example of rendered overlap error. (b) Example of rendered overflow error. (c) Example of rendered alignment lost error.

Figure 33. Examples of the error renderings produced by LBH.

Rendering images Each layout violation is drawn in the order configured in the previous phase. Each render starts with the rendering of all nodes

with a lighter color so that the error nodes stand out more. Overflow and overlap violations are drawn as dark colored rectangles, with the overlap or overflow rectangles drawn with a striped texture. Alignment change violations are highlighted by drawing their common tab stop with a black line, and drawing a line between the two involved nodes. The involved nodes are also highlighted with darker colors. Each violation type is drawn by itself, so that they don't steal attention from each other. A screen shot is then saved to disk as a PNG file and the next violation category is drawn. Figure 33 shows an example how how the three error types are rendered.

- Number of test sets: - 6
- Total errors: - 29
  - Overlaps ———— : 4
  - Overflows ———— : 17
  - Total alignment changes: 8
- Overflow - test-set:4 => 27 @ L-44 - 28 @ L-15 …
- Overlap - test-set:4 => 27 @ L-44 - 18 @ L-44 …
- AlignmentLost => 13:Right @ L:18 - 19:Left @ L:15 - aligned in 2 test sets …

Figure 34. Example of the textual report generated by LBH.

Textual report A textual report is generated of all the errors reported by LBH. First, The report starts with a summary of the number of error of each type and gives shows the number of test sets used. Then comes a detailed list of all errors which were reported. Each error have their own format, which contains information about which nodes were involved, as well as which lines they correspond to in their source code (UX markup in this case). Figure 34 shows an example of what a report by LBH looks like (note that we are only showing one example of each errors type. The ellipses (…) are only there to indicate that we have removed some items from the report for brevity). Each overflow and overlap error is reported with the following information:

- The test set the error occurred in: `test-set:<id>`
- The two nodes involved in the error, where each node is reported with the following structure: `<node_id> @ L-<source_line_number>`

Alignment lost errors have a similar structure in the report bus also includes information about which side of the nodes are involved. Also, since alignment

changes happen between test sets, the report gives us information about how many test sets the two nodes were aligned in.

4. Evaluation

In this chapter we present the evaluation which was conducted to assess the quality of LBH.

4.1. Testing approach

There are several ways this system could be used to enhance the app developer experience and to make it easier to create apps without layout bugs.

- One approach involves regularly applying the validation tool during development, fixing bugs as they are discovered.
- The other approach is to apply the tool to layouts considered production ready in order to discover last minute bugs.

The main testing approach used in the evaluation of this work is the latter of the two. The advantage of this approach is that it is easier to gather a larger volume of data. The downside of this approach is that most of the available examples have already been vigorously tested, meaning that we can't necessarily expect there to be that many bugs for LBH to discover, not because the tools doesn't work, but because the examples might be mostly bug free.

Therefore, the tool was also tested during the development of a simple example app, where the tool was run at regular intervals during the app development. We also tested LBH on a set of constructed example app screen which had cases of all the types of layout errors we have discussed.

4.2. Quantitative results

The LBH implementation was tested on a set of constructed app screens which contained some cases of overflow, overlap and alignment change, and the system was verified to correctly report on the errors, and correctly ignore the non-error cases. During these tests, the system was configured to not establish a baseline, since that would cause some of the constructed errors (which were present on all test cases) to be ignored.

A separate test screen was developed for each of the three error categories (overflow, overlap and alignment change) in order to test LBH's ability to identify them in isolation. All tests were run on two different screen sizes (test cases). The sizes used were that of an iPhone 5 and an iPhone 6 plus who's specifications are shown in table 1.

	Width	Height	Pixels per point
iPhone 5/5s	640	1136	2
iPhone 6 plus	1242	2208	3

Table 1. The two screen sizes used to test LBHs ability to detect overlap, overflow and alignment lost errors.

4.2.1. Overflow results

Figure 35 shows the app screen that was used to test LBH's ability to identify overflow errors. We can see that the figure contains four boxes, where two of them have overflowing text, either in the horizontal direction (row 1), or in the vertical direction (row 3). The app screen contains four parts, that check the systems ability to detect:

1. Horizontal overflow (text is too long for its container)
2. Vertical overflow (text is too long for its container in the vertical direction).

There are also two parts of the test that make sure we do not falsely report a valid case for both normal text and wrapped text.

As the report shows us in Figure 37, LBH reports 3 cases of overflow, and 2 cases of alignment lost. We can ignore the alignment lost cases. They are reported because of random alignments which we do not remove since we have turned off the baseline finder for this test. We can also see from Figure 36 where the overflow errors are located, indicated by the stripe textured boxes.

4.2.2. Overlap result

Figure 38 shows the test app that was used to evaluate LBH's ability to detect overlap errors. The app screen consists of 4 parts, which test the following aspects of our algorithm (from top to bottom in Figure 38):

1. Detecting overlap between two rectangles
2. Ignoring overlap if we have complete containment (see section 3.4.3.2 for details)
3. Correct reporting of overlap error with crossing elements
4. How LBH handles non rectangular elements

Figure 40 shows that LBH detected 3 overlaps, and nothing else. We can be seen in Figure 39, we were able to detect overlap in both the first and third case, which was expected. We see however, that LBH does report an

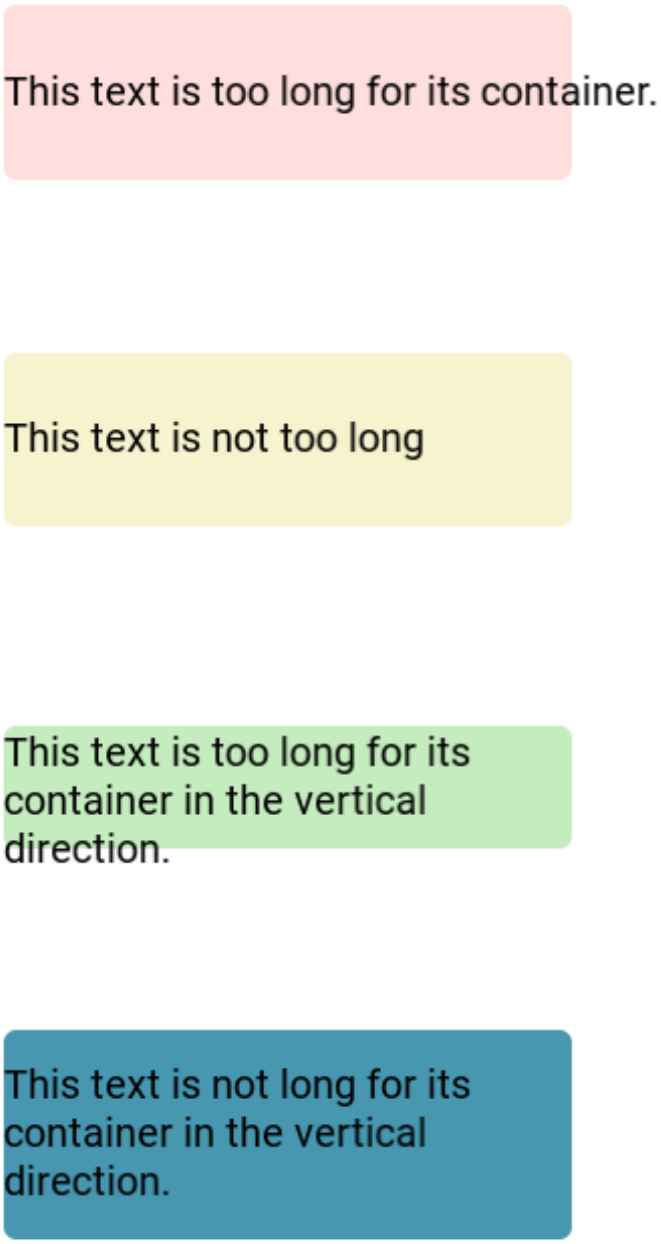
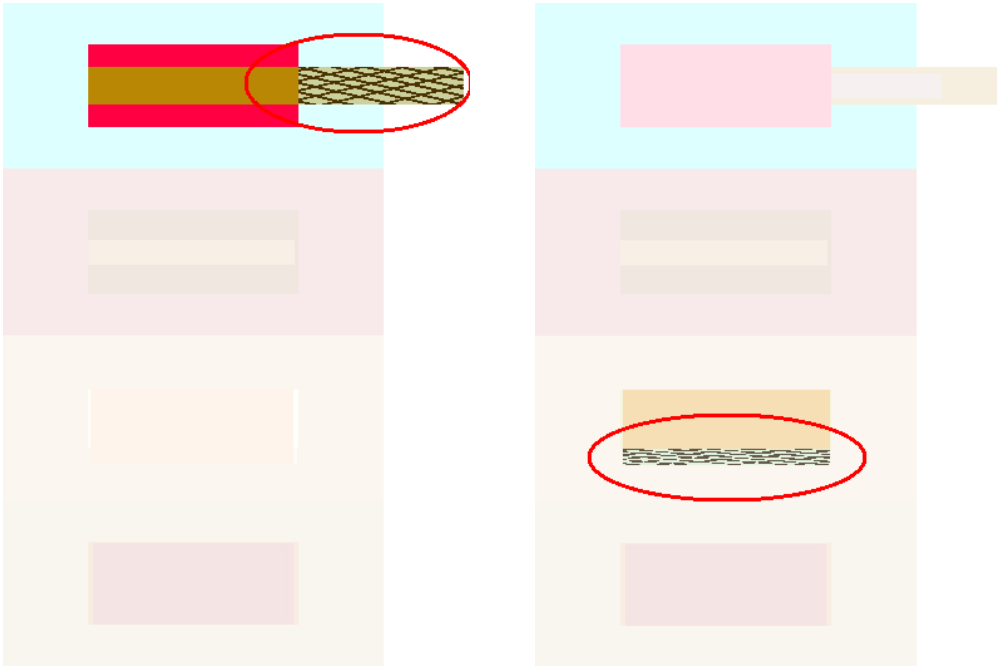


Figure 35. App screen used as input for the overflow test.



(a) Overflow error #1 (b) Overflow error #2

Figure 36. Overflow errors found

- Number of test sets: - 2
- Total errors: - 5
  - Overlaps ———— : 0
  - Overflows ———— : 3
  - Total alignment changes: 2
- Overflow - test-set:1 => 6 @ L-14 - 7 @ L-15
- Overflow - test-set:0 => 6 @ L-14 - 7 @ L-15
- Overflow - test-set:0 => 14 @ L-24 - 15 @ L-25 ...

Figure 37. Results summary for overflow evaluation.

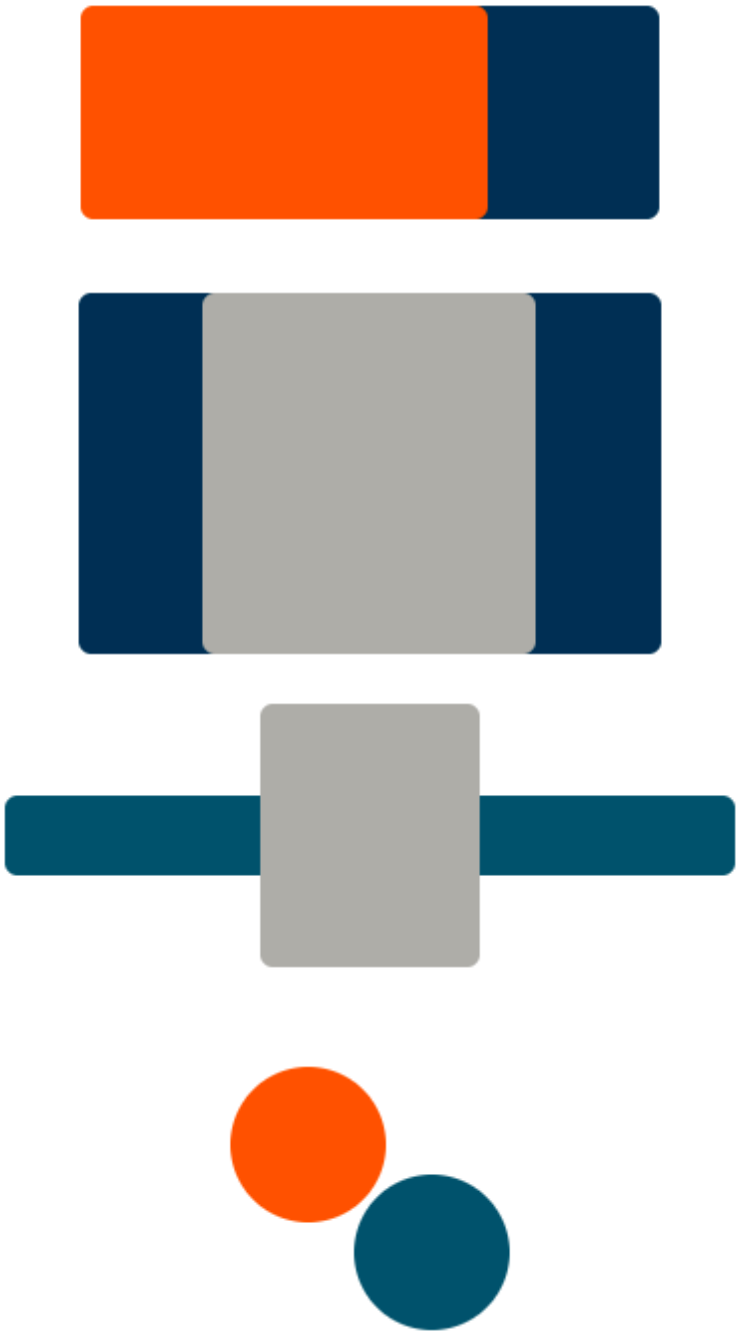
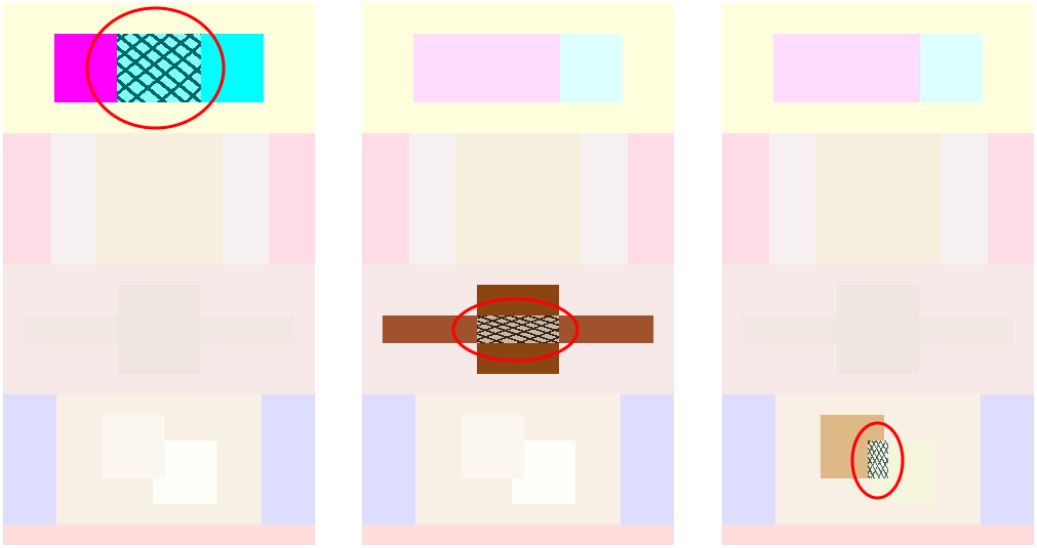


Figure 38. App screen used as input for the overlap test.

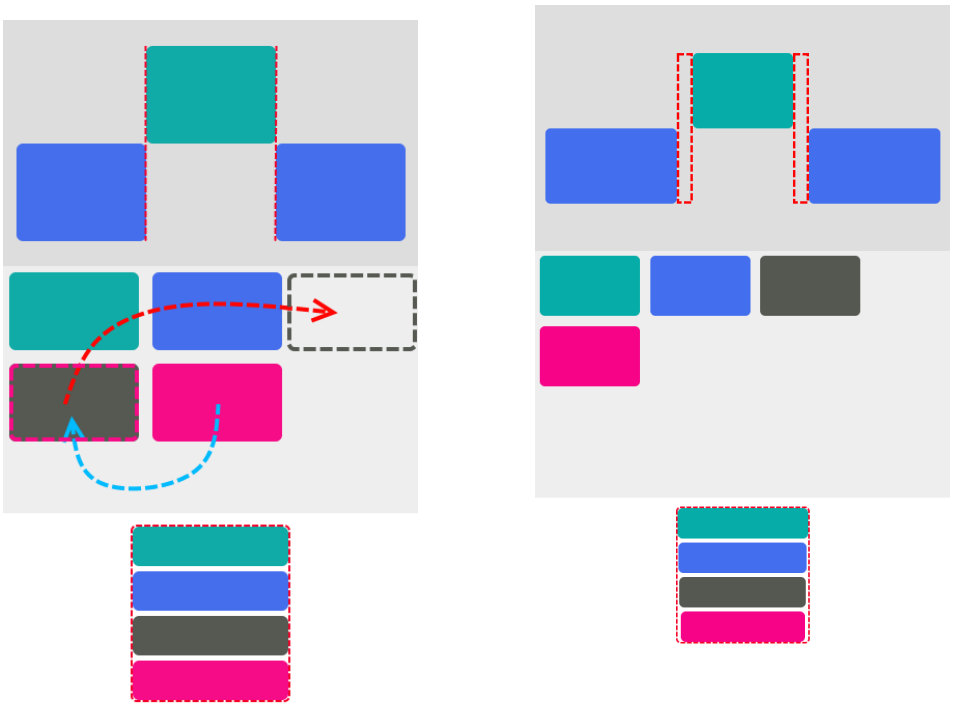


(a) Overlap error #1 (b) Overlap error #2 (c) Overlap error #3

Figure 39. Overlap errors found

- Number of test sets: - 2
- Total errors: - 6
  - Overlaps ———— : 6
  - Overflows ———— : 0
  - Total alignment changes: 0
- Overlap - test-set:0 => 5 @ L-15 - 4 @ L-14
- Overlap - test-set:0 => 11 @ L-23 - 10 @ L-22
- Overlap - test-set:0 => 14 @ L-27 - 13 @ L-26
- Overlap - test-set:1 => 5 @ L-15 - 4 @ L-14
- Overlap - test-set:1 => 10 @ L-22 - 11 @ L-23
- Overlap - test-set:1 => 4 @ L-14 - 5 @ L-15

Figure 40. Results summary for overlap evaluation.



(a) Alignment change test iPhone 5 screen size

(b) Alignment change test iPhone 6 plus screen size

Figure 41. App screens used as inputs for the alignment change test.

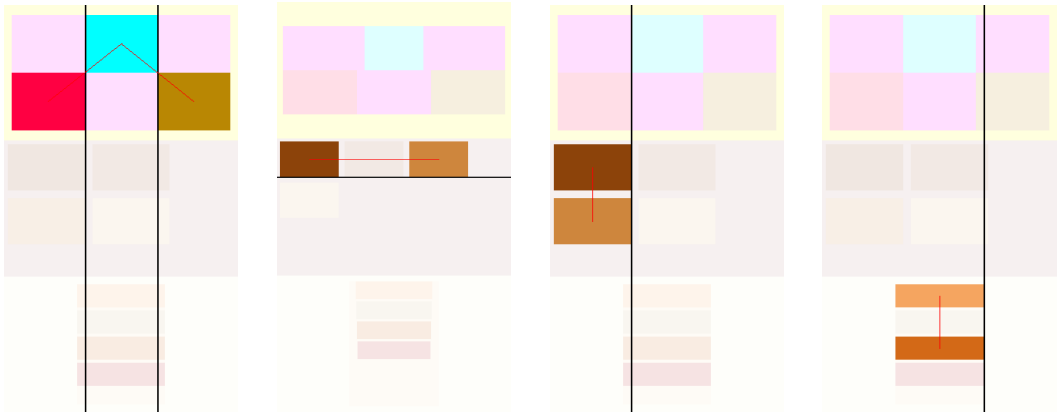
overlap error for the last case (the two circles), which should not happen. This is because LBH doesn't know about circular shapes, and will assume that every element is rectangular. In this case, the rectangles with the same position and size as the circles would be overlapping, which can be seen in Figure 39c. LBH did however not report an error for the second case (complete containment), which is a correct validation.

4.2.3. Alignment change result

Figure 41 shows the test app that was used to evaluate LBH's ability to discover alignment changes between elements of an app rendered on two different screen sizes.

The striped arrows and boxes which can bee seen in Figure 41 were added by hand to indicate the layout changes which happened between the two screen sizes. The test is divided into three parts (which can be seen seen as three rows in the layout).

The first row has a rectangle which is aligned to two rectangles beneath



(a) Alignment change error #1 (b) Alignment change error #2 (c) Alignment change error #3 (d) Alignment change error #4

Figure 42. Overlap errors found

- Number of test sets: - 2
- Total errors: - 40
  - Overlaps ———— : 0
  - Overflows ———— : 0
  - Total alignment changes: 40 ...

Figure 43. Results summary for alignment change evaluation after testing on two screen sizes (iPhone 5 and iPhone 6 plus).

it. As can be seen in Figure 41, the two bottom rectangles lose alignment to the top rectangle on the largest screen size.

In the second row, a `WrapPanel` has been used to display four rectangles. The `WrapPanel` places its children horizontally until it hits the side of the available space. At that point it wraps around and starts placing its children on a new row. In this case we can see that that has happened between the two screen shots of Figure 41.

The last row displays four rectangles which each has a maximum width set (each with a different value) by the `MaxWidth` property in Fuse. This leads to them having the same width (which also makes them aligned to each other) on the small screen size, but causes them to lose alignment on the larger screen size.

Figure 42 shows a selections of the resulting errors which was reported by LBH. We can see that it has detected alignment changes for all the three rows of test cases of the app screen. We can also see, in Figure 43 that LBH has reported 80 cases (where 40 are unique) of alignment change. This high number is because LBH reports on each pair of alignment change.

4.3. Case study - Todo App

In this section we will a conduct a small case study to illustrate how this tool can be used in practice while developing a simple todo app using Fuse

4.3.1. The app

The app will consist of a single screen. Docked to the top will be a text input field for writing a short todo note, with a button to the right which adds that note to the list of todo notes. The rest of the screen consists of a list of todo notes, which can have one of two states, not done, or done, each with their own visual cues. Clicking on the todo note item will toggle its done state.

4.3.2. How we use our tool

Our tool will be invoked at several steps along the production to help us catch layout errors. Tests will be run for 7 different screen sizes which are listed in table 2.

And the errors reported by our tool are fixed as we go. Following is a step by step log of the development process, and the the output we got from our tool.

	Width	Height
iPhone 6	750	1334
iPhone 6 Plus	1242	2208
iPhone 5/5S	640	1136
iPhone 4/4s	640	960
Sony Xperia Z3	1080	1920
Samsung Galaxy S6 Edge	1440	2560
Google Nexus 9	2048	1536

Table 2. Screen sizes used for testing the app.

4.3.3. Development

We start off creating a new project in Fuse using the command in our command prompt:

```
fuse create app TodoApp
```

This creates a new one page Fuse app. We start off with one UX file called MainView.ux, which is where we put all of our UI code.

Adding the client side testing component The current implementation of our tool needs a client side component to both gather information about the GUI tree, and to communicate with the testing server.

To add this component, we add a project reference to the Fuse ViewCrawler project which was implemented as a stand-alone Fuse project. We then add the `GUITestOracle` component to our MainView.ux file:

```
<App>
    <GUITestOracle/>
</App>
```

To make sure our setup is correct, we run our tool, and get the expected results (0 errors) which can be seen in Figure 44.

The input control We then create the note description input control. It is a Grid, with a text input control and a button, which is docked to the top of the app screen.

The UX markup now looks like the following and the visual state of the app can be seen in Figure 45:

```
<App>
    <GUITestOracle/>
```

- Number of test sets: - 7
- Total errors: - 0

```
  * Overlaps ------------ : 0
  * Overflows ----------- : 0
  * Total alignment changes: 0
```

Figure 44. Results after running our tool on an empty project.

Figure 45. The input control

```
    <DockPanel>
        <Grid Columns="2*,1*" Height="60" Dock="Top" Padding="10">
            <Panel>
                <TextInput PlaceholderText="Write a new note" />
            </Panel>
            <Rectangle CornerRadius="5" Color="#0af">
                <Text Value="Add new note" Margin="10,0"
                    TextTruncation="None" Alignment="Center"/>
            </Rectangle>
        </Grid>
    </DockPanel>
</App>
```

By running our tool, we can already now see that we have made some mistakes:

Our tool has reported 4 cases of overflow which all point to the same element. Figure 47 shows the output image generated by LBH which points out the overflow error. Figure 46 shows the textual summary from LBH.
if we look at the detailed reports for the overflow cases, we can see that they all point to line 10 in MainView.ux.

At line 10 we find the Text element which is a part of our button. We have a couple of options to fix this problem. We can either make its column

- Number of test sets: - 7
- Total errors: - 5

```
* Overlaps  ------------ : 0
* Overflows ------------ : 4
* Total alignment changes: 1
```

- Overflow - test-set:0 => 8 @ L-10 - 9 @ L-0
- Overflow - test-set:4 => 8 @ L-10 - 9 @ L-0
- Overflow - test-set:2 => 8 @ L-10 - 9 @ L-0
- Overflow - test-set:3 => 8 @ L-10 - 9 @ L-0
- AlignmentLost => 9:Right @ L:0 - 8:Right @ L:10 - aligned in 6 test sets

Figure 46. LBH results summary for input control part.

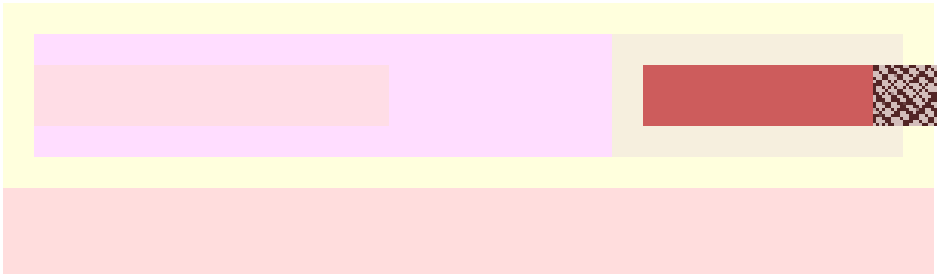


Figure 47. One of the rendered images generated by LBH pointing to the error found during testing of the todo app. The striped area points to the overflow error.

bigger so that it gets enough space on the smaller screens, or we can shorten the button text. In this case we chose the latter; changing it from "Add new note" to just "Add". Running our tool again puts us back to 0 reported errors.

The todo list At this point we are ready to work on the list of todo items. It will have a really simple structure: A stack of DockPanel elements which contains a Text control with the todo note, and a colored Rectangle indicating the "done" state of the item.

We also add a small piece of JavaScript to hook up the "add" button to add notes to the list of notes. We won't go into detail about this JavaScript code however, since it doesn't directly affect the output of our tool. The full source code to the todo list app is included in the appendix.

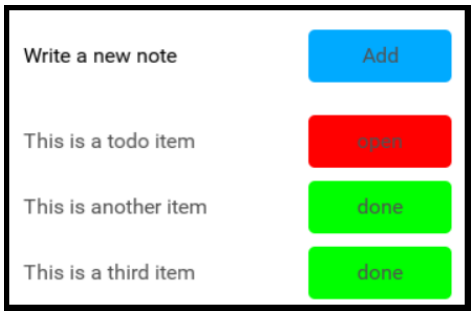
```
<StackPanel Padding="10">
    <Each Items="{todos}">
        <DockPanel Height="50">
            <TextInput Value="{note}" />
            <Rectangle Color="{isDone} ? #0f0 : #f00"
                Width="120" Dock="Right"
                       CornerRadius="5" Margin="0,5">
                <TextInput Value="{isDoneText}" Alignment="Center"/>
            </Rectangle>
        </DockPanel>
    </Each>
</StackPanel>
```

The current state of our app can be seen in Figure 48a.
If we now run our tool, we get the summary output in Figure 49. An example of the visual output can be seen in Figure 48b.

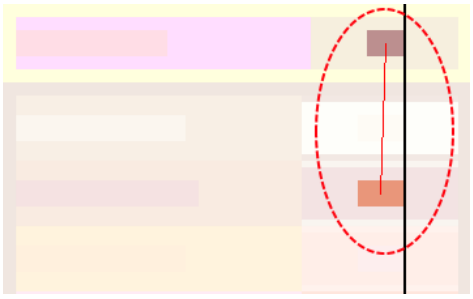
The output tells us that our tools has found 40 cases of alignment lost. Since the app we are testing is quite simple, we've configured LBH to report all alignment lost errors it finds.

Looking at the visual output, we can see that the `isDone` text of our todo items are on some screen sizes aligned to the add button text, while on some they are not. While it is hard to argue that this is an actual bug (since both elements are visible and tidily arranged in all cases), we can make the appearance more consistent by giving the `isDone` indicator the same size as the send button.

```
<Rectangle Color="{isDone} ? #0f0 : #f00"
    Width="width(sendButton)" Dock="Right"
```



(a) The state of the todo app after the todo list has been added before LBH has been run.



(b) Shows one of the alignment errors discovered by LBH when testing the todo app.

Figure 48. Visual state of the app and sample of LBH output.

- Number of test sets: - 7
- Total errors: - 40

```
* Overlaps  ------------ : 0
* Overflows ------------ : 0
* Total alignment changes: 40
```

- AlignmentLost => 40:Right @ L:0 - 8:Right @ L:14 - aligned in 2 test sets
- AlignmentLost => 9:Left @ L:0 - 33:Left @ L:24 - aligned in 2 test sets ... ...

Figure 49. LBH result summary for todo list

```
            CornerRadius="5" Margin="0,5">
    <Text Value="{isDoneText}" Alignment="Center"/>
</Rectangle>
```

After making sure our `isDone` indicator has the same width as the send button (notice the `width(sendButton)` part) and running our tool, we are back to 0 errors reported.

At this point we have a minimal todo list app, and already been able to use our tool to discover several layout bugs.

4.3.4. Case-study Discussion

This case study has been designed to illustrate the potential benefits behind a tool like this and to show that out implementation is indeed able to identify

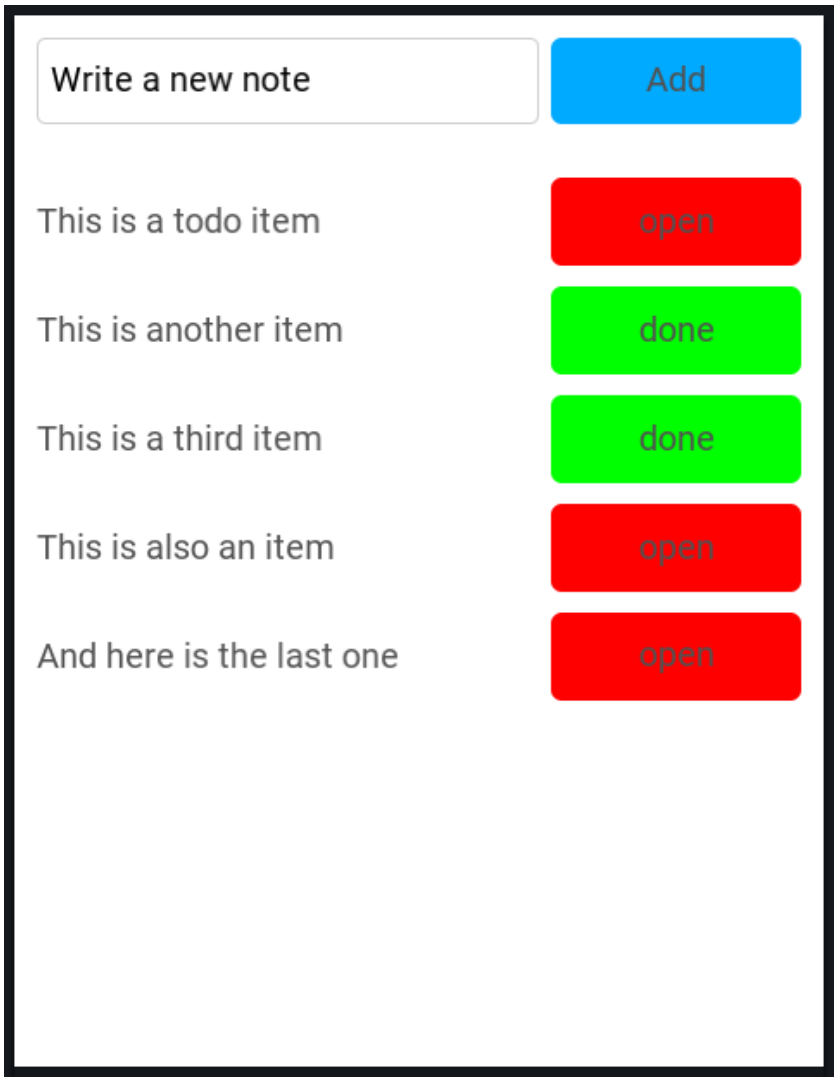


Figure 50. Finished todo list app

real layout errors fully automatically, and to clearly point them out to the user. It is important to be aware of the fact that, while this represents a real world use case, it was created with the intention of illustrating this tool. Had I created it with the intention of making an actual app, I most probably would have been able to avoid these errors in the first place. I would still argue that this case study points towards the usefulness of having a fully automated tool like this as a backup for the production and testing of apps. With the addition of more tweakable parameters, and the option of opting out of, or ignoring testing for certain parts of the app, the goal doesn't even necessarily have to be 0 errors, but just to be made more aware of how to various elements of our app interact as the screen size and data sets changes.

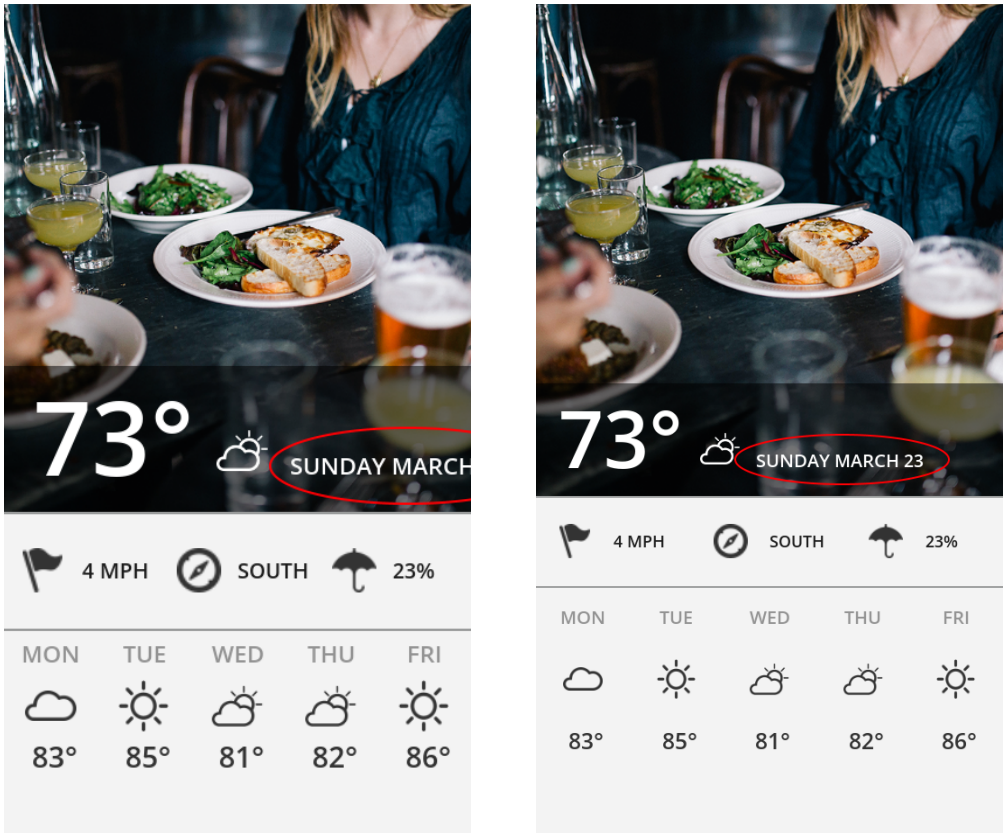
4.4. Comparison with AppliTools Eyes

LBH was compared to AppliTools Eyes [4], which was discussed in section 4, on a small single page app screen which can be seen in Figure 51. AppliTools was chosen as a tool to compare with since it is cross platform, and even has a command line interface. One of the aims of LBH is to be able to discover layout bugs in apps on different screen sizes (iPhone 5 and iPhone 6 plus), and since AppliTools Eyes requires two screen shots to do its image comparison, this was the approach we decided to test for.

The app was implemented in Fuse and contained one case of overflow which we were interested in locating. The element in violation is highlighted with a red circle in Figure 51. When testing AppliTools, screen shots were manually taken from the running app and uploaded to AppliTools server using their command line tool [5].
The way we performed the comparison between AppliTools and LBH was to test the app on two screen sizes, and have our tools performs their validation. Unfortunately, AppliTools requires the compared screen shots to have the same size, which means that we have to scale one of the images either up or down. Because of this, we decided to test for both cases.

In the first test, we use the original screen shot of the app running on an iPhone 5 screen size, and then we downscale the iPhone 6 plus version so that it fits with the iPhone 5 size. The results of this test can be seen in Figure 52a and 52b. In the second test we used the original iPhone 6 plus screen shot and instead upscaled the iPhone 5 screen shot to match. The results of this test can be seen in Figure 52c and 52d.

4.4.1. AppliTools results

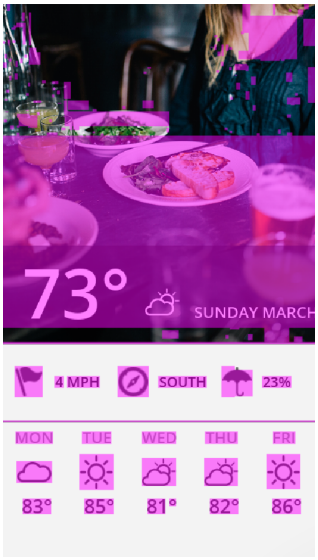


(a) Test app on an iPhone 5 screen size

(b) Test app on an iPhone 6 plus screen size

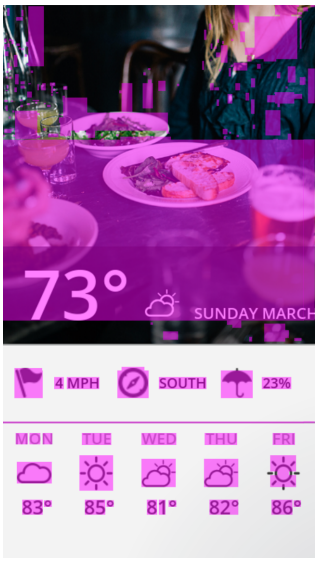
Figure 51. The test app on two screen sizes used as input for the AppliTools test. A red circle shows the element which ends up overflowing on one of the screen sizes.

(a) iphone 5 screen size



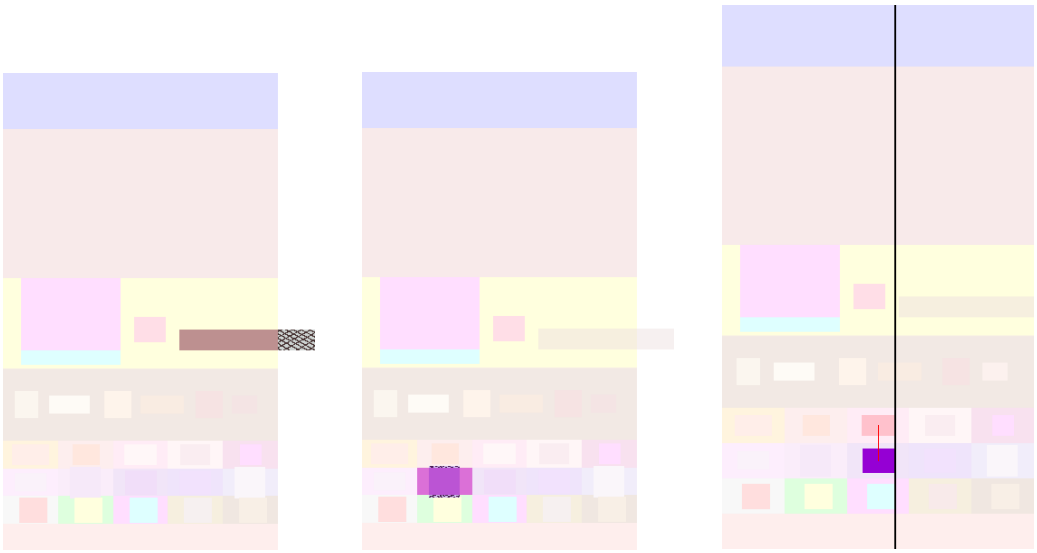
(b) iPhone 6 plus screen size downscaled

(c) iphone 5 screen size upscaled



(d) iPhone 6 plus screen size

Figure 52. Test results from AppliTools of both upscaled and downscaled version. The results are only slightly different, but we can notice that most GUI elements has been highlighted, meaning it is hard to separate the true errors from the false ones.



(a) LBH Overflow error #1 (b) LBH Overflow error #2 (c) LBH Alignment change error #1

Figure 53. LBH test results

AppliTools reports errors by highlighting the areas of the compared screen shot with a purple color. We can see from the two result Figure 52 that AppliTools marks almost all the elements of the app as being part of errors. The difference between the two screen sizes causes the icons and text to become bigger relative to the screen size. Since AppliTools works on image comparison, pixel by pixel, it will then see these changes as errors, since they do not match the original image.

We achieve similar results from downscaling the larger screen size as we do with upscaling the smaller screen size, with most of the screen marked as being in violation.

4.4.2. LBH results

We tested LBH on the same app using the same screen sizes. As can be seen in Figure 54, LBH discovered 0 cases of overlap, 3 cases of overflow and 31 cases of alignment change. A selection of the resulting renders can be seen in Figure 53.

We see from Figure 54 that LBH reports a lot of alignment changes, which we can by visual inspection see are not alignments that should be considered layout errors. This is because LBH is not able to establish a good baseline of alignment errors with only two test cases. If we increase

- Number of test sets: - 2
- Total errors: - 34

```
* Overlaps  ------------ : 0
* Overflows ------------ : 3
* Total alignment changes: 31
```

Figure 54. Results summary for LBH test.

- Number of test sets: - 4
- Total errors: - 6
  - Overlaps ———— : 0
  - Overflows ———— : 4
  - Total alignment changes: 2
- Overflow - test-set:3 => 7 @ L-24 - 8 @ L-0
- Overflow - test-set:0 => 47 @ L-69 - 48 @ L-70
- Overflow - test-set:0 => 7 @ L-24 - 8 @ L-0
- Overflow - test-set:0 => 41 @ L-60 - 42 @ L-61
- AlignmentLost => 31:Right @ L:48 - 44:Right @ L:64 - aligned in 3 test sets
- AlignmentLost => 32:Right @ L:0 - 44:Right @ L:64 - aligned in 3 test sets

Figure 55. Results summary for LBH test.

the number of test cases to 4, we get the results in Figure 55. We can see that LBH now only reports 2 alignment changes, but 4 cases of overflow. This is closer to the what we were able to discover by visual inspection. The two alignment changes which was discovered by LBH was not discovered by visual inspection, and could be caused because of floating point errors during the layout calculation.

LBH was able to point out the overflow error in several of the test cases. Specifically 53a which was a clear visual bug (the text is drawn outside of the screen).

4.5. Comparing results from varying baseline parameter

The baseline discovery step can be tweaked by two parameters which are both numbers between 0.0 and 1.0. The first parameter is used to tweak how the baseline considers overflow and overlap errors, and the second parameter tweaks how the baseline considers alignment lost errors.
We conducted a test to see the variations in how LBH reports layout errors when changing the value of these parameters parameter. The test consisted of running LBH on an app screen which contains some complexity, and some cases of what would be considered false positives if LBH reported them as overlaps. The app used in the test can be seen in Figure 56. We can see that there is an example of overlap with each item, as the blur/purple rectangle overlaps with the background images. It is the job of the baseline phase to make sure we don't report these as errors, but setting the bar too low for adding errors to the baseline, we might end up not reporting the actual errors either.
The results of the tests can be seen in figure 57 and 58.

4.5.1. Tweaking the overflow-overlap baseline parameter

We can see from the graph in Figure 57 that the number of overlaps and overflows reported by LBH goes a bit up and down as we change the value of the parameter from 0.0 to 1.0. Since we in this test has set the alignment lost baseline parameter to 1.0 (meaning all alignment lost errors will be added to the baseline), we don't get any alignment lost errors reported in this test. The overflow-overlap baseline parameter is multiplied by the number of test sets, which is then compared to the number of test sets each overlap and overflow error is present in. If that number is higher than our threshold, we add the error to our baseline. What this means is that for a threshold of 1.0, an overlap or overflow error is only added to our baseline if it is present in all

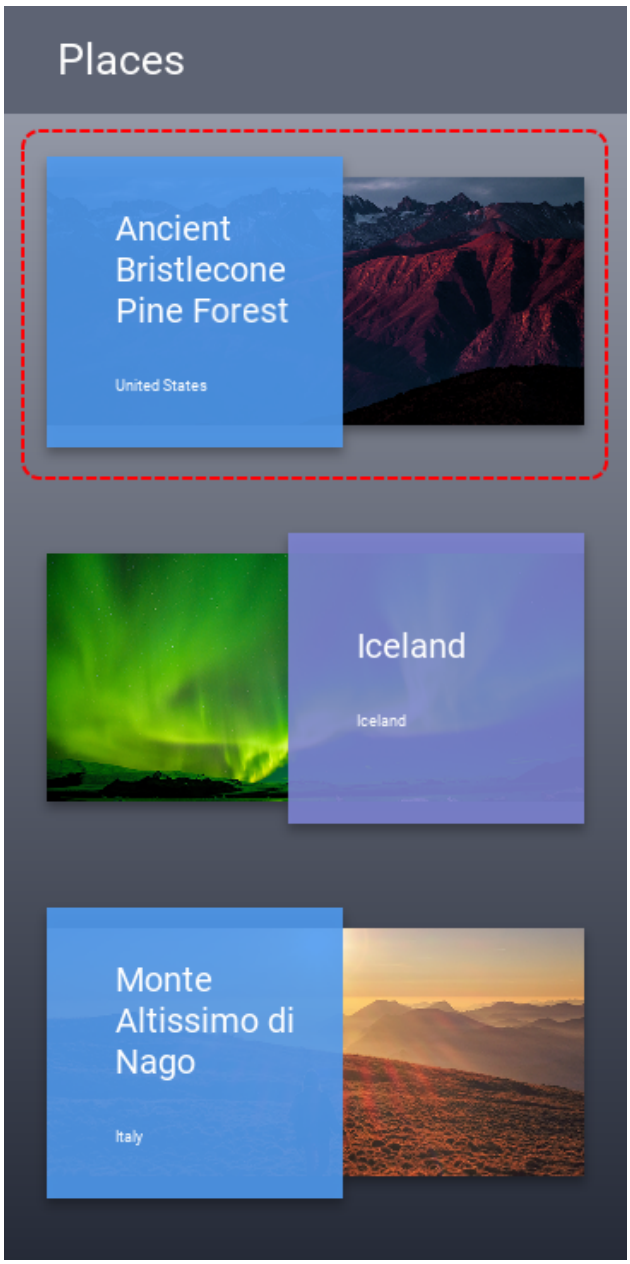


Figure 56. The app used for testing tweaking of the baseline threshold parameters. We can see an example of intentional overlap marked by a striped rectangle.

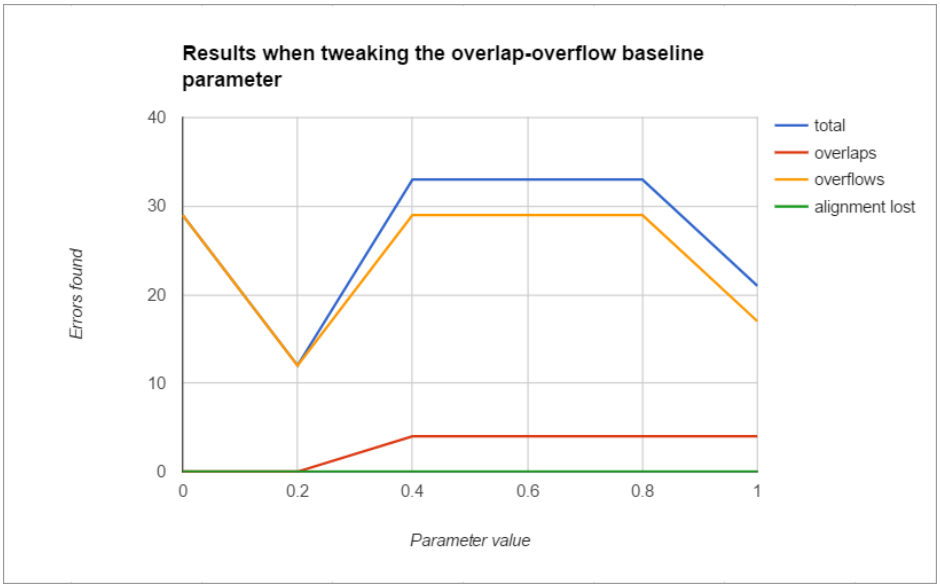


Figure 57. Tweaking overflow overlap baseline threshold parameter.

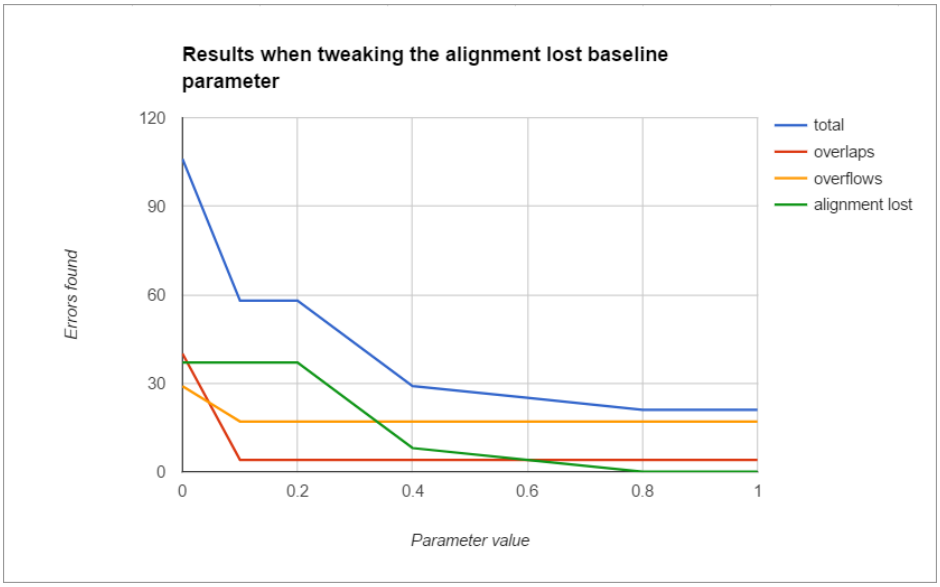


Figure 58. Tweaking alignment lost baseline threshold parameter.

the test cases. On the other hand, the lower our parameter value, the fewer test cases an error has to be a part of for it to be included in the baseline.

We use a default value of 1.0 for this parameter, which is a value that follows our assumption that an overlap or overflow has to be present in all test cases for it to be a part of the baseline. For cases where this is not always true, the developer can lower the value of this parameter to allow for more cases to be added to the baseline.

4.5.2. Tweaking the alignment lost baseline parameter

The graph in Figure 58 shows us the results of the alignment lost baseline threshold parameter test. We can see that as the parameter value goes from 0.0 to 1.0 the number of reported errors decreases. The reason for why it reports a lot more overlap and overflow errors at 0.0 is because at that point the whole baseline system is turned off, and all errors end up being reported.

This parameter is multiplied by one less than the number of test sets. If then an alignment is present in less test cases than that number, the error it corresponds to is added to the baseline. The reasoning behind this heuristic is as follows: Each alignment error has an "aligned node pair" associated with it, which also keeps a count of how many test sets those nodes are aligned in. If the number of test sets a node pair are aligned in low, for example in only 1 test set, we can assume that the reason for why they are aligned is just randomness. However if they are aligned in all test cases except for one, we can assume that the alignment was intentional and that the reason for why they are not aligned in one test case is because of a layout error. The parameter lets us tweak how strict we should be in this regard.

A good default value for this parameters seems to be around 0.8, as long as we have enough test cases (more than 5-6 cases). For situations where we have fewer cases than this, it might be beneficial to lower the value of this parameter.

4.6. Comparison

Table 3 shows a capability comparison of LBH, AppliTools, PhantomCSS and Galen Framework. We can see that among the four tools in the comparison, only LBH has the ability to automatically discover layout bugs. What we mean by this is LBH does not need a user defined script to discover bugs. All of the tools are suitable for regression testing, although what they test for is different between the image based tools, and the layout analysis based ones.

	LBH	AppliTools Eyes	PhantomCSS	Galen Framework
Automatically discover bugs	X			
Supports regression testing	X	X	X	X
Supports design changes	X	X	X	X
Can compare different screen sizes	X			X
Finds overlaps	X			X
Finds overflows	X			X
Finds alignment changes	X			X
Finds pixel differences		X	X	
Works on mobile	X	X		*
Works on the web	*	X	X	X

Table 3. Comparison of the capabilities of LBH and other layout testing tools

Galen Framework is able to discover overlaps, overflows and alignment changes, just like LBH is, but it requires the user to have described exactly how the elements of their GUI relates to each other in order for Galen Framework to be able to discover this. Among the tools, however, only AppliTools and PhantomCSS is able to detect pixel changes between tests. This means that they are for example able to detect if the contents of an image has changed.

AppliTools is the only one of the tools which has implementations for both web and mobile platforms. PhantomCSS and Galen Framework are only implemented for the web, but since the core of Galen Framework is implemented in Java, it is not inconceivable that the framework will eventually be made to work for mobile platforms as well.

While LBH is currently only implemented for one platform, the architecture we've chosen makes it quite easy to extend LBH to work for any other platform as well.

5. Discussion

This chapter will discuss the results from the evaluation, as well as some thoughts on advantages and disadvantages of LBH. We will also discuss how LBH relates to the other GUI layout testing tools we've looked at. We will then discuss some limitations to LBH.

5.1. Evaluation results

After evaluating LBH from several angles, it is clear that a tool like this can certainly be of value, and aid in the discovering of layout bugs during the development of GUI applications. The quantitative results show that LBH is able to correctly detect all the three types of layout errors; overflow, overlap and alignment lost. It was also able to identify cases of overlap which should not be considered errors, like in the case of complete containment which we discussed in section 3.4.3.2.

From the baseline threshold parameter tests, we were able to show how establishing a baseline over several test cases aided us in filtering out the false positives, which would otherwise be reported by LBH as errors even though they were intentionally added by the developer. The addition of the baseline step to LBH, drastically improved the accuracy of the reported errors.

With the case study, we aimed at giving a practical example of how LBH could be used in practice while developing a, while small and simple, real app, and help us discover layout bugs as we developed the app. We showed that the reports generated by LBH was able to show us where in our code the errors originated from, so that we could easily fix them.

With these results, we are certain that a tool like LBH has the potential to aid developers in creating responsive and dynamic GUI layouts. By potentially integrating LBH with existing IDEs and/or testing environments, this will further increase the value of employing a tool like LBH.

5.2. Compared with existing tools

LBH was compared with AppliTools Eyes, with results showing how LBH was more accurately (resulted in fewer false positives) able to discover layout bugs. In this section we discuss how LBH compares to some the other tools mentioned (PhantomCSS, Galen Framework) and how it compares to functional GUI testing techniques.

Functional GUI testing focus on verifying that interactions with the GUI results in the correct business logic being run and that it has the expected result on the GUI. This means that it is difficult or even impossible to use

these techniques to verify that what the GUI is displaying actually looks correct. This was probably never the intention of most of the record and replay based techniques discovered during the initial research for this thesis, but it is also what lead me to realize how unsaturated the field of layout testing is.

Comparing LBH with a tool like Galen Framework, we see that their goals are mostly the same; discovering bugs in how elements relate to each other and their expected sizes and positions. However, the fact that LBH does not require any user defined test specification puts it at a significant advantage, and adds to its potential. In fact, LBH can be thought of as a mix between a static analysis tool and a testing tool. The reason why we feel like this is an appropriate description of LBH is because of the fact that it does not require user defined scripts, but does require the app to be running in order to get its required data. We could conceivably envision a situation where the layout system of whatever platform was to be tested was separated from the execution of the app, and thusly opening up the possibility for making a tools like LBH truly a static analysis tool.

Comparing LBH with image diffing based tools like AppliTools Eyes and PhantomCSS, we see quickly the value analysis based tools can offer over that of image diffing based tools. Image based tools only have the information stored in the pixels of the screen shot images to work with. This means that even though they are able to correctly identify layout errors, they won't have any information about which elements the error corresponds to, and therefore no knowledge about where in the code the errors originated from.

Image based tools are also at a disadvantage when it comes to comparing tests between different screen sizes. This is these tools usually require the compared images to have the same size and aspect ratio. We got around this limitation in our tests by either upscaling or downscaling one of the screen shots that were to be compared. This is of course not a perfect solution since any scaling of images can easily lead to artifacts in the images. The reason why we chose to compare with AppliTools (one of the image diffing based tools), was because it supported the platform we were testing out of the box, and because of the fact that image based tools work without a user defined test script/specification. This makes it an appropriate tool to compare with LBH.

5.3. Limitations

In this section some shortcomings of the system will be discussed.

5.3.1. Cross platform

LBH can currently only test apps made with Fuse. The reason for this is that the client component which is in charge of gathering information about the apps layout is, as of now, only implemented for the Fuse platform. Because of the architecture of LBH however, it is almost trivial to extend LBH to also be able to test other platforms. We only need to re-implement the client component for each platform, whose only task is to gather layout information, and send it to the LBH server.

5.3.2. Multi page apps

The current implementation of LBH does not know how to navigate through the various pages in a complex app, and this is not able to automatically test an entire app if it contains more than one page. This was deemed out of scope of this work, since the implementation of such a component can be considered a medium sized project in itself. The implementation of such a component is proposed as the potential for future work in section 6.1.

5.3.3. Animations

Most modern apps contain a moderate to significant amount of animations. While this system is able to detect errors in a static GUI layout, most animation systems, including the one used in Fuse, does not report animations as separate layouts. This means that the current implementation of LBH has no way of performing its validation on animations. It also means that some errors might be wrongly reported since they don't take into account the fact that an element might be in the middle of an animation.

- Note: By animation in this context we technically mean translational offsets from a rest state. The rest state represents the actual layout that will be tested, although what the user sees might include this animation offset, leading to the possibility of false positives and negatives.

6. Conclusion and future work

It is clear that GUI testing will become an increasingly important part of the GUI application development industry. With the creation of dynamic layouts which is supposed to support to run on a large number of screen sizes and on different screen aspect ratios, the need for good layout testing tools is also increasing rapidly. In this section we will summarize how we answered our research questions.

- **RQ1:** What are typical GUI layout errors?

With the work presented in chapter 3, we presented a set of layout errors that typically arise during the development of GUI layouts. We categorized these errors into three categories: "overflows", "overlaps" and "alignment lost", which together covered all the layout errors we had discussed.

- **RQ2:** How can we automate, and thereby lower the cost of, GUI layout testing to find the typical GUI layout errors defined in RQ1?

We chose to answer RQ2 with the implementation of an automated GUI layout testing tool which we called "The **L**ayout **B**ug **H**unter", or **LBH**. With LBH, we were able to implement algorithms that encapsulated the categorization found in RQ1 and to automatically discover these errors in GUI applications.

- **RQ3:** How can we improve the quality of GUI layout testing by covering as many as possible of the GUI layout errors defined in RQ1?

In chapter 4, we evaluated LBH from several different angles, and compared it to the performance of one of the existing GUI layout testing tools (AppliTools Eyes [4]). With the results of this comparison, and the results of the rest of the tests performed in chapter 4, we showed how we answered RQ3 and improved the quality of GUI layout testing.

Based on the evaluation results of LBH, we believe LBH outperforms current GUI layout tools in terms of test cost-effectiveness and test coverage. To disseminate the design and implementation of LBH, we will make a scientific paper from this thesis and submit it to a software engineering conference, e.g. APSEC 2017 (APSEC 2017 24th Asia-Pacific Software Engineering Conference http://www.apsec2017.org/).

6.1. Future work

The aim of this thesis was to find a new approach to GUI testing, which was fully automated. With this work, we feel confident that we've been able

to achieve that with the implementation of the LBH prototype. LBH is of course only that; a prototype. We believe the need for future work in several directions are necessary in order to make LBH reach its potential and be a good solution and usable by any developer of GUI based application:

- **Further research into layout error categorization:** The layout error categorization (overlap, overflow and alignment lost) must be further tested and evaluated. We have shown how these three categories of layout error can point out obvious flaws in several test apps, but we require even more testing on more complex applications to definitively prove that this categorization is complete and does not break on certain types of GUI applications.

- **Implement the client side of LBH for more platforms:** For LBH to be useful for a wider audience, the LBH GUI crawler (which gathers and sends the layout information) client needs to be implemented for more platforms. Since Fuse apps run on both Android and iOS, the most interesting next platform to support would be the web. Proving that LBH works for both web and native targets, is a great step in the right direction of making this a proper cross platform experience.

- **Make LBH compatible with existing CI systems:** The optimal way of using automated tests in a project is to incorporate them in a CI (continuous integration) system. There are several popular CI systems in use by the industry today, good examples of which include services like TeamCity [23], Travis CI [25], AppVeyor [6] and Jenkins [13]. Making sure LBH is compatible and easy to set up with as many as possible if these services is paramount to making a tool like LBH a viable testing option for the industry.

- **Improving the quality of the error reports:** While the error reports generated by LBH point us in the right direction of figuring out what caused the errors, there is plenty of room for improvement. The textual report is currently verbose and not very reader friendly, meaning it can take a long time to find what we are looking for if the report contains a lot of errors. The PNG rendering are also not optimal. We would have liked to have LBH show a more accurate rendering of the app, so that the user can more easily see which part of their app is in violation. Another possibility is to add the ability to interactively inspect the reports, being able to select an item from the textual report and have the render for that error show up in a window.

- **Automatically navigate through the app:** Most non-trivial apps contain more than just one page. Often the user will have to interact with the application in order to visit all the various pages it contains. In order for LBH to work with these apps, it needs a way of navigating the app without the user having to be present. This requires a component be implemented for each platform that is to support this feature of LBH. Several platforms already have libraries that lets one achieve this, and so the only work needed by LBH is a common wrapper through which LBH can perform its operations.
- **Add support for injecting test data to controls:** The current implementation of LBH does not support the injection of test data into an applications controls. The goal of this feature is to be able to test whether input controls like text inputs do not cause layout errors if their content becomes too large. We chose not to attempt an implementation of such as system since it can potentially be a large project by itself. Creating a good heuristic for what kind of data is sensible to test with and how to test for as many cases as possible without letting the number of test cases become too large seems to us like more than a trivial task. However, integrating such a system with LBH should increase its test coverage potential by a large amount.

References

[1] Gojko Adzic. "How to Implement UI Testing without Shooting Yourself in the Foot." 2010. https://gojko.net/2010/04/13/how-to-implement-ui-testing-without-shooting-yourself-in-the-foot-2/.

[2] "Android Studio Espresso Test Recorder." https://developer.android.com/studio/test/espresso-test-recorder.html.

[3] "Android User Interface Overview." https://developer.android.com/guide/topics/ui/overview.html.

[4] "AppliTools." AppliTools. 2017. https://applitools.com/.

[5] "AppliTools CLI Documentation." AppliTools. 2017. https://applitools.com/resources/tutorial/screenshots/cli#step-2.

[6] "AppVeyor: Continuous Delivery Service for Windows." https://www.appveyor.com/.

[7] "Automotion." ITArray. 2017. https://www.itarray.net/automotion/.

[8] Ishan Banerjee, Bao Nguyen, Vahid Garousi, and Atif Memon. "Graphical User Interface (GUI) Testing: Systematic Mapping and Repository." *Inf. Softw. Technol.* 55 (10). Butterworth-Heinemann, Newton, MA, USA: 1679–1694. Oct. 2013. doi:10.1016/j.infsof.2013.03.004.

[9] "Fuse - Cross Platform App Development Kit." https://www.fusetools.com.

[10] "Galen Framework." Galen. 2017. http://galenframework.com/.

[11] "Galen Language Spec." Galen. 2017. http://galenframework.com/docs/reference-galen-spec-language-guide/.

[12] "Hyper Text Markup Language." https://www.w3.org/html/.

[13] "Jenkins Open Source Automation Server." https://jenkins.io/.

[14] "Material Design Guidelines." Google. https://material.io/guidelines/.

[15] "Mobile OS Marketshare." NetMarketShare. 2016. https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=9&qpcustomb=1.

[16] "Mono Framework." http://www.mono-project.com/.

[17] ".NET." Microsoft. https://www.microsoft.com/net.

[18] "PhantomCSS." Huddle. 2017. https://github.com/Huddle/PhantomCSS.

[19] "Ranorex Android Test Automation." https://www.ranorex.com/mobile-automation-testing/android-test-automation.html.

[20] "Rust Programming Language." https://www.rust-lang.org.

[21] "Selenium." SeleniumHQ. 2017. http://www.seleniumhq.org/.

[22] Michael Tamm. "Fighting Layout Bugs." Google. 2009. https://www.youtube.com/watch?v=WY3C6FHqSqQ.

[23] "TeamCity Powerful Continuous Integration." https://www.jetbrains.com/teamcity/.

[24] "The Layout Bug Hunter Source Code." https://github.com/kristianhasselknippe/layout-bug-hunter.

[25] "Travis CI: Test and Deploy with Confidence." https://travis-ci.org/.

[26] "Understanding iOS AutoLayout." https://developer.apple.com/library/content/documentation/UserExperience/Conceptual/AutolayoutPG/.

[27] "Variations in Screen Sizes." Google. https://material.io/devices/.

[28] "Webdriver." Webdriver. 2017. http://webdriver.io/.

[29] "Windows Presentation Foundation." https://msdn.microsoft.com/en-us/library/ms754130%28v=vs.110%29.aspx.

[30] "XCode Documentation for Recording UI Test Cases." https://developer.apple.com/library/content/documentation/DeveloperTools/Conceptual/testing_with_xcode/chapters/09-ui_testing.html.

[31] "XCTest Framework for Xcode." https://developer.apple.com/reference/xctest.

7. Appendix

7.1. Todo App Full Code

Following is the full UX and JavaScript code for the final todo list app used as a case study during chapter 4.3.

MainView.ux:

```
<App>

    <TextInput ux:Class="Text" IsEnabled="false"
        HitTestMode="None" IsReadOnly="true"/>

    <GUITestOracle ux:Name="testOracle"/>

    <JavaScript File="MainView.js" />
    <DockPanel>
        <Grid Columns="2*,1*" Height="60"
            Dock="Top" Padding="10">
            <Panel>
                <TextInput PlaceholderText="Write a new note" />
            </Panel>
            <Rectangle ux:Name="sendButton"
                CornerRadius="5" Color="#0af" Clicked="{addTodo}">
                <Text Value="Add" Margin="10,0"
                    TextTruncation="None" Alignment="Center"/>
            </Rectangle>
        </Grid>
        <StackPanel Padding="10">
            <Each Items="{todos}">
                <DockPanel Height="50">
                    <TextInput Value="{note}" />
                    <Rectangle
                        Color="{isDone} ?
                            #0f0 : #f00" Width="width(sendButton)"
                            Dock="Right" CornerRadius="5" Margin="0,5">
                        <Text Value="{isDoneText}" Alignment="Center"/>
                    </Rectangle>
                </DockPanel>
            </Each>
        </StackPanel>
```

```
    </DockPanel>
</App>
```

MainView.js:

```
const Observable = require("FuseJS/Observable");

const input = Observable("");

function Todo(note, isDone) {
    var ret = {
        note: note,
        isDone: Observable(isDone)
    };
    ret.isDoneText = ret.isDone.map(x => {
        if (x) {
            return "done";
        } else {
            return "open";
        }
    });
    return ret;
}

const todos = Observable(
    new Todo("This is a todo item", false),
    new Todo("This is another item", true),
    new Todo("This is a third item", true),
    new Todo("This is also an item", false),
    new Todo("And here is the last one", false)
);

function addTodo() {
    let note = input.value;
    todos.add(new Todo(note), false);
}

module.exports = {
    input,
    todos,
    addTodo
};
```