Linköping University | Department of Computer Science
Master thesis, 30 ECTS | Datateknik
2017 | LIU-IDA/LITH-EX-A--17/025--SE

Comparing modifiability of React Native and two native codebases

Robin Abrahamsson
David Berntsen

Supervisor : Lena Buffoni
Examiner : Ola Leifler

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

Creating native mobile application on multiple platforms generate a lot of duplicate code. This thesis has evaluated if the code quality attribute modifiability improves when migrating to React Native. One Android and one iOS codebase existed for an application and a third codebase was developed with React Native. The measurements of the codebases were based on the SQMMA-model. The metrics for the model were collected with static analyzers created specifically for this project. The results created consists of graphs that show the modifiability for some specific components over time and graphs that show the stability of the platforms. These graphs show that when measuring code metrics on applications over time it is better to do this on a large codebase that has been developed for some time. When calculating a modifiability value the sum of the metrics and the average value of the metrics between files should be used and it is shown that the React Native platform seems to be more stable than native.

Acknowledgments

We would like to thank our supervisor **Lena Buffoni** and examiner **Ola Leifler**, both from Linköping University, for all the feedback and help throughout our entire thesis. Helping us to stay on the right path and get the best result possible. Thanks to our opponents **Ludvig Helén** and **Petra Öhlin** for reviewing our thesis multiple times and giving valuable feedback to improve the thesis. We would like to thank Valtech AB for providing applications, supervision and a place to do our work.

Contents

List of Figures

List of Tables

1 | Introduction

There are over 2 billion smartphones in the world with different operating systems on them [57]. Android had a significant market share of all the platforms that were sold to end users worldwide in 2014, as can seen in Table 1.1. Even though Android had a vast vast majority of the market share in the world, in the United States Android doesn't have the same kind of dominance. In 2015 Android only had 52.4% of the market share in the United States, as can be seen in Table 1.2. So if a company wants to be available and competetive on the entire market and have applications on all platforms they must either create a native application for each platform, create hybrid applications for the same platforms or a web application that is usable by every platform. Depending on which route one decides to go, it comes with some pros and cons.

Operating system	**2014 Units**	**2014 Market Share (%)**
Android	1,004,675	80.7
iOS	191,426	15.4
Windows	35,133	2.8
BlackBerry	7,911	0.6
Other OS	5,745	0.5

Table 1.1: Worldwide market share by operating system in 2014
[27]

Operating system	**Mar-15 Market Share (%)**	**Jun-15 Market Share (%)**
Android	52.4	51.6
iOS	42.6	44.1
Windows	3.3	2.9
BlackBerry	1.6	1.2
Symbian	0.1	0.1

Table 1.2: United States market share by operating system in 2015
[15]

1.1 Motivation

Creating native mobile applications on multiple platforms generates a lot of duplicate code, especially for the basic functionality and business logic. Due to differences in syntax, language, test suites and packages in the different platforms, reusability and sharing of code is not really possible. This increases the amount of hours needed to release, maintain and test applications for multiple platforms which leads to a higher development and maintenance cost. React Native[1] is a project started by Facebook which addresses the problem in an interesting way. It gives the possibility have one codebase for multiple applications with the possibility to still use platform specific components.

[1] `https://github.com/facebook/react-native`

The work for this thesis has been done at the company Valtech which is an IT-consultant firm located in Stockholm. For many customers Valtech creates web services and often companies also want to have mobile applications for both Android and iOS. To have only one codebase instead of two would require less work and people. If it would be possible to have only one codebase for both applications it would be important to know if it gives any advantages to switch from the native codebases to a cross-platform codebase and how much investment would be necessary.

1.2 Aim

This thesis is exploratory, which means that it investigates a new area. The aim is to evaluate if certain quality attribute can be improved when migrating code from native codebases to a cross platform tool like React Native.

Specifically the modifiability quality attribute, presented in the ISO/IEC 25010 standard [36] is looked at. Modifiability is a measurement of how easily code can be modified without introducing defects or degrading the current product quality.

Some aspects of modifiability is how it has changed over time and also to see if there is some reasonable way to quantify or estimate the impact of changes in the updates of the languages, platforms and frameworks.

1.3 Research Questions

The hypothesis for this thesis is that modifiability will improve when moving from the native codebases to a React Native solution.

Some additional factors that have impact on modifiability of code are changes to the APIs with different releases. There can be releases of new versions of the underlying operating system (like iOS and Android), updates to specific components or new versions of the entire language/framework (like Objetive-c, Java, React Native or JavaScript).

Based on the hypothesis the following research questions were produced:

- Is modifiability a quality attribute that can be calculated to assist in the choice of two implementation solutions for mobile applications?
- How can information about releases of the platforms be quantified to understand the possible impact on modifiability?

1.4 Delimitations

This report will only focus on applications for the two most used mobile operating systems, Android and iOS, which is seen in Table 1.1. Of the different cross platform tools described in Section 2.2 only React Native will be focused on. Only metrics related to modifiability, Section 2.7, were implemented in the static analyzers, Section 3.1.2.

2 | Theory

Part of this chapter contains theory about mobile applications, React Native and similar mobile application platforms. After that comes a part with theory about quality models, metrics, and definitions of maintainability and modifiability. There are also some theory about the method used in the thesis.

2.1 Mobile Applications

There are essentially three different kinds of mobile applications; native, web and hybrid applications.

2.1.1 Native Applications

Native applications are developed to be used on a specific platform or device. They need to be downloaded via a central distribution portal like the App store, Google Play or Windows Store [44]. The application then needs to be installed directly on the device. This enables the ability to use device-specific hardware and software, as the global positioning system (GPS), camera etc. Since the application runs directly on the device it does not need to be connected to the Internet to run, though some tasks in the application might need an Internet connection to be used.

Since native applications are written for a particular device or platform they are being written in platform specific languages. Applications for iOS are written in Objective-C or Swift, applications for Android are written in Java and applications for Windows are written in C#. This prevents code from being reused between the different platforms which increases the time needed to create the applications since the entire codebase has to be rewritten for every platform.

2.1.2 Web Applications

Building web applications is done using technologies such as HTML5, CSS and JavaScript and they can therefore run in a web browser, which make them accessible on most devices [42]. That is also the big advantage of using web applications. Getting one codebase for all platforms and the application will look similar on all platforms.

The main disadvantage with web applications is that they have a hard time coping with heavy graphics and cannot be used when the device is offline [42]. Also they do not have access to low level features and device resources, such as camera, GPS, file system, push notifications or databases. Access to some of those resources has started becoming available by an HTML5 API standard. But the implementations of the standard are still lacking functionality. With this new API a new type of web applications have started to emerge, called progressive web applications.

Progressive Web Applications

A progressive web application uses new technology that make it possible to run the application with limited network connectivity [42]. It also has access to certain native functionality like push notifications. According to Malavolta a web application has to meet four criterias to be classified as a web application:

- A user should be able to install the application so it acts as a native application on the phone.
- It should be served over HTTPS.
- It should include a web manifest, which declare some application metadata.
- It should use service workers, which make it possible to run the application offline.

Progressive web applications have the strengths of a native application in that they can access the resources on the mobile, does not require an app store and they only need one codebase Malavolta. The weaknesses are that the technology is new and not mature. As the web applications progressive web applications also run in a web browser which means that it also has the extra layer of abstractions that comes with running in a web browser.

2.1.3 Hybrid Applications

Hybrid applications are found in the distribution platform for the device, just as one would find a native application [44]. In the same way as a native application it has to be downloaded and installed on the device to be used. But just as the web applications they are built using web technologies such as HTML5, CSS and JavaScript. Hybrid applications are displayed in a native container which is hosting a WebView. Since the hybrid application is still installed on the device it can still access device hardware capabilities.

2.2 Cross-platform Tools

Techopedia defined cross-platform tools as “the practice of developing software products or services for multiple platforms or software environments” [17]. This is conducted with the help of cross-platform development tools, also called cross-platform tools (CPTs). Applications that are created with the help of CPTs will be called *CPT applications* to make reading easier. CPT applications are similar to hybrid and progressive web applications with respect to that it can reuse a lot of code between the platforms. If nothing native specific is needed even the entire codebase could be reused. CPT applications have to be downloaded via an application distribution platform, just as a hybrid application and a native application. What makes CPT applications different from hybrid applications is that they compile the code to native code and therefore have access to native functionality. When CPT applications are compiled they essentially act as native applications.

There are a lot of different CPTs and new ones continues to emerge, in the following chapters the framework of choice, React Native, is presented and some other promising CPTs. Some of these CPTs are similar to React Native while others are vastly different.

2.2.1 React Native

React Native is an open source framework developed by Facebook and it had its initial public release of v0.1.0 on March 26th 2015 [51]. Applications are written in JavaScript and have support for the operating systems Android and iOS [53]. It is built upon on the principles of React.js which is a framework used for building front end web applications.

Virtual DOM

Web pages are built using a Document Object Model (DOM) which is an API for HTML and XML documents. The API allows for things found in an HTML or XML document to be accessed, changed, deleted or added. Often the DOM is represented by a tree structure, especially HTML documents [62]. Rerendering the DOM is a slow operation and manipulating the DOM directly is error prone and hard to maintain [30]. React.js have solved these problems by introducing a virtual DOM that is manipulated directly in-memory and is a replica of the real DOM [26]. When something in the application changes React.js will create a new virtual DOM which includes the changes made and calculates a patch which is the difference between the previous virtual DOM. Then React.js applies only what was changed on to the real DOM. The same approach as with a virtual DOM is also used by React Native where there is a virtual appplication hierarchy to work with instead.

JSX

"JSX is a XML-like syntax extension to ECMAScript without any defined semantics" according to the JSX Specification, where ECMAScript is a scripting-language specification which JavaScript extends. In listing 2.1 it can be seen how JSX code can be used to create HTML with React.js [23]. Both React.js and React Native use the JSX language but instead of HTML building blocks it uses native components [60]. For example p and h1-elements are text-elements in React Native as can be seen in listing 2.2.

```
function test() {
  return (
    <div>
      <h1>title</h1>
      <p>Hello world!</p>
    </div>
    );
}
```

Listing 2.1: JSX code.

```
function test() {
  return (
    <View>
      <Text>title</Text>
      <Text>Hello world!</Text>
    </View>
    );
}
```

Listing 2.2: How the code could look written with React Native instead.

Babel

Babel is a JavaScript compiler that transforms newer versions of JavaScript to older versions [5]. This gives the possibility to use newer versions of JavaScript in browsers that do not yet support it. It can also compile JSX code to pure JavaScript as can be seen in listing 2.1 and 2.3.

```
"use strict";

function test() {
  return React.createElement(
    "div",
    null,
    React.createElement(
      "h1",
      null,
      "title"
    ),
    React.createElement(
      "p",
      null,
      "Hello world!"
    )
  );
}
```

Listing 2.3: JavaScript code compiled with Babel from the JSX code in listing 2.1.

2.2.2 Additional CPTs

There are quite a few additional CPTs that are not covered in this thesis, but could be interesting for future work to look into.

Xamarin

Xamarin is an open source framework developed by Microsoft which enables developers to build native applications for any device in C# and F# [65]. Since Xamarin applications are built using C# it allows the usage of features like Generics, Linq and the Task Parallel Library. It is built on Mono, an open source implementation of the .NET Framework [45]. Mono's .NET implementation is based on the ECMA standards for C# and the Common Language Infrastructure [14]. Xamarin also provides IDE tools such as Xamarin Studio IDE and the Xamarin plugin for Visual Studio to create, build and deploy Xamarin projects.

Adobe AIR

A cross-platform runtime, developed by Adobe Systems [2]. Adobe AIR enables deployment of rich Internet applications (RIAs) that can run on most operating systems, such as OS X, Windows, iOS and Android. AIR applications can be built with ActionScript or HTML/-JavaScript.

The runtime is installed on the device beforehand, and then lets the AIR application use that runtime. This is possible on Windows and Android, but not on iOS. On iOS the application needs a captive AIR runtime in the application, this is also possible on Android. On windows the AIR runtime is already pre-installed, but on Android it has to be installed separately unless the application runs captive AIR runtime.

Adobe AIR lets the developer target the runtime for development, instead of developing for specific operating systems.

NeoMAD

NeoMAD is developed by Neomades and lets the developer write all code in Java, with some of it being reusable cross devices while some of it is implementation specific [47]. The Java code is then transpiled to native binaries and source code which is then compiled using the native development tools of each platform.

NeoMAD has an option that allows the developer to write platform specific code, in the native language, to the NeoMAD project. When writing in the native language, platform specific features that are not available in the NeoMAD Generic API. It also allows reuse native code that has already been written for a specific platform.

Titanium

Titanium is an open source project created by Appcelerator to create native mobile application using JavaScript [58]. It has a JavaScript-based SDK with APIs for iOS, Android, Windows, Blackberry and HTML5. The interface consist of native UI components, which is supposed to result in a real native experience.

NativeScript

NativeScript is an open source framework which lets the develeoper write native iOS and Android applications using Angular, TypeScript or JavaScript [46]. Code can be reused between both platforms and the web without using WebViews. NativeScript gets access to native APIs via JavaScript, and it can reuse packages from npm, CocoaPods and Gradle. CocoaPods is a dependency manager for Swift and Objective-C Cocoa projects and Gradle is a build tool for Android applications.

2.3 Cost of Software Development

Bakota et al. presented a cost model in *"A cost model based on software maintainability"* which adopts the concept of entropy, or disorder, from thermodynamics to measure maintainability [6]. Where high disorder means low maintainability. Their model is based on two assumptions:

1. Changes made to software without the explicit aim to improve the code, will either increase the disorder or leave it unchanged.
2. Performing changes to software with high disorder is more expensive.

From the laws of thermodynamics, Bakota et al. says that a closed system's disorder can only decrease by applying external forces to it. And if left unchanged its disorder can only increase or stay the same.

According to Bakota et al. cost can be measured in any type of effort, e.g. salary, work month, time or similar [6]. They did not have access to historical records regarding the development cost. Therefore they made the assumption that the cost of development was proportional to the elapsed time which they treated as a threat to validity.

2.4 Code Metrics

Also known as software metrics, is a measurement of some property on a system. Depending on which definition of the metric is being used the way the metric is calculated can differ, which can be seen in the chapters below.

2.4.1 Chidamber & Kemerer

Chidamber and Kemerer presented a metric suite for object oriented design in 1994 [12]. These metrics were:

- Weighted Methods Per Class (WMC) is the sum of the complexities of every method in that class, Equation 2.1. What kind of complexity is used in the equation can be change to be appropriate to the current language.

$$\begin{gathered} \text{WMC} = \sum_{i=1}^{n} c_i \\ \text{where } c_i \text{ is the complexity value for component} \\ i \text{ and } n \text{ is the number of methods in that class.} \end{gathered} \tag{2.1}$$

- Depth of Inheritance Tree (DIT), the maximum distance from the node to the root of the tree.
- Number of Children (NOC), the number of immediate subclasses to a class in the class hierarchy.
- Coupling between object classes (CBO), the number of classes to which a class i coupled.
- Response For a Class (RFC), the number of methods that can potentially be executed in response to a message received by an object of that class.
- Lack of Cohesion in Methods (LCOM), is a count of the number of method pairs that whose similarity is 0 minus the number of method pairs whose similarity is not zero. Similarity between two methods M_1 & M_2 is calculated according to Equation 2.2.

$$\begin{gathered}\sigma() = \{I_1\} \cap \{I_2\} \\ \text{where } \{I_1\} \text{ and } \{I_2\} \text{are the sets of instance} \\ \text{variables used by methods } M_1 \text{ and } M_2\end{gathered} \tag{2.2}$$

These metrics, with these calculations, are often called CK metrics or C&K metrics after Chidamber and Kemerer.

2.4.2 MOOD

Abreu defined the MOOD (Metrics for Object Oriented Design) metrics [1]. These metrics consists of:

- Method Hiding Factor (MHF), is the sum of the invisibilities of all methods in a class. The invisibility of a method is defined as the percentage of the class from which the method is hidden. MHF for a system or module is calculated in Equation 2.3. A high value of MHF (100%) means that all methods are private which indicates very little functionality, while a low value (0%) indicate all methods are public which leaves the class unprotected.

$$\text{MHF} = \frac{\sum_{i=1}^{TC} M_h(C_i)}{\sum_{i=1}^{TC} M_d(C_i)} \tag{2.3}$$

  Where the variables in the eqation are:

$$\begin{aligned} \text{TC } &= \text{number of classes} \\ M_d(C_i) &= M_h(C_i) + M_v(C_i) \\ M_d &= \text{the number of methods defined in } C_i \\ M_h &= \text{the number of hidden methods defined in } C_i \\ M_v &= \text{the number of visible methods defined in } C_i \end{aligned}$$

- Attribute Hiding Factor (AHF), is the sum of the invisibilities of all attributes in a class. Is calculated in the same way as MHF, but with attributes instead of methods.
- Method Inheritance Factor (MIF), is the ratio between the inherited methods and all the methods in a class. The calculation of MIF for a system or module is shown in Equation 2.4. A low value of MIF (0%) means either that there are no methods in that class or that no inheritance is used.

$$\text{MIF} = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)} \tag{2.4}$$

Where the variables in the eqation are:

$$\begin{aligned} \text{TC} &= \text{number of classes} \\ M_a(C_i) &= M_i(C_i) + M_d(C_i) \\ M_a &= \text{the number of methods defined in } C_i \\ M_i &= \text{the number of inherited methods in } C_i \\ M_d &= \text{the number of declared methods in } C_i \end{aligned}$$

- Attribute Inheritance Factor (AIF), is calculated the same way as MIF but with attributes instead of methods.
- Polymorphism Factor (POF), the number of possible different polymorphic situations. If a class have 0% POF, then no polymorphism is used. If a class has 100% POF, then every method is overridden in all derived classes. The POF value for a system is calculated in Equation 2.5.

$$\text{POF} = \frac{\sum_{i=1}^{TC} M_o(C_i)}{\sum_{i=1}^{TC}(M_n(C_i) * DC(C_i))} \tag{2.5}$$

Where the variables in the eqation are:

$$\begin{aligned} \text{TC} &= \text{number of classes} \\ M_o &= \text{the number of overridden methods in } C_i \\ M_n &= \text{the number of new methods in } C_i \\ DC(C_i) &= \text{the descendants count in } C_i \end{aligned}$$

- Coupling Factor (COF), is the ratio between the actual of coupling and the highest possible coupling in a system. A class A is coupled to class B only if A calls methods or accesses variables of B.

2.4.3 Other Metrics

Apart from the already mentioned metrics, there are other common metrics:

- Size, can often be defined as lines of code (LOC), source lines of code (SLOC) or logical lines of code (LLOC). Boehm et al. created a document which if a line should or should not be counted [10]. It can also be defined to use number of function points (explained later in this list) instead of the number of lines to get a more practical value.
- McCabe's Cyclomatic complexity (CC). The Cyclomatic complexity metric was developed by Thomas J. McCabe in 1976 and is a quantitative measure computed using a programs control flow graph where the complexity is defined as M in Equation 2.6 [43]. In the equation E is the number of edges in the graph, N is the number of nodes in the graph and P is the number of connected components.

$$M = E - N + 2P \tag{2.6}$$

The complexity number M is an upper bound for how many test cases are necessary to achieve total branch coverage for the code.

- Coupling is the manner and degree of interdependence between software modules [38]. Low coupling is usually a sign of a well structured and designed software system. It consists of:
  - Afferent Coupling, also known as inward coupling, the number of classes that depend on the measured class.

  - Efferent Coupling, also known as outward coupling, the number of classes that the measured class depend on.

- Nesting, the maximum number of nested blocks in a method.
- Cohesion, the relatedness of methods and attributes in a class
- Messaging, the number of public members available as services to others.
- DesignSize, the number of classes in the design.
- Halstead complexity measures, measurable properties of software and formulas that describe the relationship between them [29]. This means that with the correct data it should be easy to convert between the different properties of the codebase. The code data are: η_1 - number of distinct operations, η_2 - number of distinct operands, N_1 - total number of operators, N_2 - total number of operands. Using this data it is possible to get the program vocabulary 2.7, program length 2.8 and volume 2.9.

$$\text{Program vocabulary: } \eta = \eta_1 + \eta_2 \tag{2.7}$$

$$\text{Program length: N} = N_1 + N_2 \tag{2.8}$$

$$\text{Volume: V} = N \cdot log_2\eta \tag{2.9}$$

These properties are used to calculate the properties most related to maintainability, such as difficulty to write and understand the program 2.10, effort to code 2.11, time to code 2.12 and number of delivered bugs 2.13.

$$\text{Difficulty: D} = \frac{\eta_1}{2} \cdot \frac{\text{N}}{\eta_2} \tag{2.10}$$

$$\text{Effort: E} = D \cdot V \tag{2.11}$$

$$\text{Time required to program: T} = \frac{\text{E}}{18} \text{ seconds} \tag{2.12}$$

$$\text{Number of delivered bugs: B} = \frac{E^{\frac{2}{3}}}{3000} \tag{2.13}$$

- Maintainability index, introduced by Oman and Hagemeister in 1992 [48] and later refined by Coleman et al. in 1994 [13] to Equation 2.14. It consists of several metrics, and uses the average value for each of them for the module under inspection. McCabe's cyclomatic complexity (CC), Halstead's volume (HV), lines of code (LOC) and percentage of comments (COM).

$$\begin{aligned} \text{Maintainability index} = 171 \\ -5.2 \cdot \ln(\text{HV}) \\ -0.23 \cdot \text{CC} \\ -16.2 \cdot \ln(\text{LOC}) \\ +(50 \cdot \sin(\sqrt{2.46 \cdot \text{COM}})) \end{aligned} \tag{2.14}$$

- Function points, in a software project the way that function points are measured is by quantifying the functionality that is associated with external input, output and files [10]. In COCOMO 2.0, described in section 2.3, each function is then classified by complexity level which gives the Unadjusted Function Points. Usually the function points are also given a degree of influence of fourteen application characteristics on a scale from 0.0 to 0.5. When adding all the ratings to a base level of 0.65 will give a value between 0.65 and 1.35.

2.5 International Standards

International standards are consensus based and transparent [64]. Standardization has been used by companies to implement strategies on product and business issues [49]. “They give world-class specifications for products, services and systems, to ensure quality, safety and efficiency.” [33]. The models for maintainability, Section 2.6, are based on an international standard.

2.5.1 International Standards Organizations

International Organization for Standardization (ISO) was founded in 1947 and is an independent, non-governmental international organization with a membership of 162 national standards bodies[1] [33]. ISO has published 21580 International Standards and related documents, covering almost every industry.

International Electrotechnical Commission (IEC) was founded in 1906 and is a not-for-profit, quasi-governmental organization [32]. IEC is the leading organization for preparation and publication of International Standards for all electrical, electronic and related technologies [32].

International Telecommunication Union (ITU) was founded in 1845 and is the United Nations specialized agency for information and communication technologies [34]. They protect and support everyone’s fundamental right to communicate.

2.5.2 Organization Collaborations

Apart from individual standards organizations, there are also collaborations between the organizations.

The World Standards Cooperation (WSC) is a collaboration between the International Organization for Standardization[2], the International Electrotechnical Commission[3], and the International Telecommunication Union[4] [64]. “Under this banner, the three organizations preserve their common interests in strengthening and advancing the voluntary consensus-based International Standards system.”[64].

ISO/IEC JTC 1 is a joint technical committee created to develop worldwide Information and Communication Technology (ICT) standards for business and consumer applications. There are currently 3004 published ISO/IEC standards developed by committees in JTC 1 [37].

2.6 Maintainability

There are multiple definitions, variants and adaptations to maintainability, below are two different definitions that have been found to be used more often than others.

ISO/IEC 25010 Standard defines maintainability as one of eight product quality characteristics [36]. Along with functional suitability, performance efficiency, compatibility, usability, reliability, security and portability. ISO/IEC 25010 is a standard formed under ISO/IEC JTC 1 [37].

[1]https://www.iso.org/members.html
[2]https://www.iso.org/
[3]http://www.iec.ch/
[4]http://www.itu.int/

Maintainability is a value of how effectively and efficiently the intended maintainers can modify the product or system. ISO/IEC 25010 defines maintainability to consist of five subcharacteristics:

- Modularity: Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- Reusability: Degree to which an asset can be used in more than one system, or in building other assets.
- Analysability: Degree of effectiveness and efficiency with which it is possible to assess the impact on a product or system of an intended change to one or more of its parts, or to diagnose a product for deficiencies or causes of failures, or to identify parts to be modified.
- Modifiability: Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality.
- Testability: Degree of effectiveness and efficiency with which test criteria can be established for a system, product or component and tests can be performed to determine whether those criteria have been met.

IEEE defines maintainability the following way, which is used by Riaz et al. among others:

> "(1) The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment. See also: extendability; flexibility.
>
> (2) The ease with which a hardware system or component can be retained in, or restored to, a state in which it can perform its required functions." ([16])

2.6.1 Maintainability Metrics

The Software Quality Model for Maintainability Analysis (SQMMA) presented by Chawla and Chhabra uses combinations of the following metrics to quantify the five subcharacteristics of maintainability defined by the ISO/IEC 25010 standard [11]:

- Size (SLOC)
- McCabe's Cyclomatic complexity (CC)
- Coupling between object classes (CBO)
- Response For a Class (RFC)
- Weighted Methods per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Lack of Cohesion in Methods (LCOM)
- Method Inheritance Factor (MIF)
- Nesting (NBD)
- Polymorphism Factor (POF)
- Number of Children (NOC)

- Cohesion
- DesignSize

Definitions of the different metrics can be found in Section 2.4 and SQMMA will be further described in Section 2.8.1.

2.7 Modifiability

As seen in Table 2.1, modifiability consists of the stability and changeability quality attributes previously defined in the old ISO/IEC 9126 standard.

The SQMMA method uses Equations 2.15, 2.16 and 2.17 to calculate a value for modifiability of a codebase [11]. It is composed of both stability and changeability. Stability is explained as how much the software is able to minimize unexpected effects when introducing modifications and that low change density is an indication of stable software. Changeability is described as the capability of the software to incorporate a change in the software without making serious mistakes. For example when needing to touch several modules to implement a change it indicate bad changeability.

$$\text{Modifiability} = 0.42 \cdot \text{Stability} + 0.58 \cdot \text{Changeability} \tag{2.15}$$

$$\begin{aligned} \text{Stability} = & -0.19 \cdot \text{Subclasses} \\ & -0.21 \cdot \text{Coupling} \\ & -0.20 \cdot \text{Hierarchies} \\ & -0.18 \cdot \text{EntExt} \\ & -0.21 \cdot \text{Communication} \end{aligned} \tag{2.16}$$

$$\begin{aligned} \text{Changeability} = & -0.13 \cdot \text{avgSize} \\ & -0.16 \cdot \text{Nesting} \\ & -0.16 \cdot \text{Coupling} \\ & -0.13 \cdot \text{LCohesion} \\ & -0.15 \cdot \text{Hierarchies} \\ & -0.13 \cdot \text{Polymorph} \\ & -0.14 \cdot \text{MethodInheri} \end{aligned} \tag{2.17}$$

Of the modifiability metrics it has been shown that response for class (RFC), lack of cohesion (LCOM) and coupling (CBO) are connected to fault-proneness [28, 25]. It has also been shown that keeping the CK metrics, seen in Section 2.4.1, low can improve the maintainability of software systems [24].

Stability, which is one part of modifiability, can be shown by the change density of the software [11]. If the software contains a lot of faults, either the code has changed a lot previously or will have to change in the future. In the model proposed by Dayanandan modifiability metrics that match the ones in SQMMA are hierarchies, coupling, cohesion [20].

2.8 Quality Models

There are a number of models for getting the quality of a codebase, in this section some different models are presented.

2.8.1 SQMMA

In SQMMA the metrics listed in Section 2.6.1, are weighted for the different quality attributes according to Table 2.1 by Chawla and Chhabra [11]. The weights were extracted by responses of a questionnaire where experts in the domain got to select the importance of the different metrics. This was then aggregated to the weighted quality attributes and finally for maintainability. The final formula for mainatianability based on the quality attributes are presented in equation 2.18. Chawla and Chhabra argues that many frameworks provide good models of quality analysis but lack information about making them handy to use. SQMMA is based on ideas from other models and provides ready to use mathematical formulas to quantify quality attributes. Another thing it provides is formulas for the new quality attributes introduced in ISO 25010. It was validated through trend analysis and bug/change comparison.

$$\begin{aligned} \text{Maintainability} = {} & 0.11 \cdot \text{Analysability} \\ & +0.28 \cdot \text{Modifiability} \\ & +0.11 \cdot \text{Testability} \\ & +0.29 \cdot \text{Modularity} \\ & +0.20 \cdot \text{Reusability} \end{aligned} \tag{2.18}$$

Quality attributes	**Computation formulas**
Analysability	$0.31 \cdot \text{Comments} - 0.39 \cdot \text{CComplexity} - 0.28 \cdot \text{Size} - 0.32 \cdot \text{Complexity} - 0.32 \cdot \text{MethodInheri}$
Modifiability	$\text{Modifiability} = 0.42 \cdot \text{Stability} + 0.58 \cdot \text{Changeability}$
Testability	$-0.23 \cdot \text{CComplexity} - 0.19 \cdot \text{Nesting} - 0.19 \cdot \text{Communication} - 0.18 \cdot \text{Subclasses} - 0.21 \cdot \text{EffCoupling}$
Modularity	$-\text{LCohesion}/\text{Polymorph}$
Reusability	$0.25 \cdot \text{Cohesion} + 0.5 \cdot \text{Messaging} + 0.5 \cdot \text{DesignSize} - 0.25 \cdot \text{Coupling}$

Table 2.1: SQMMA - Computation formulas

2.8.2 A Probabilistic Software Quality Model

Bakota et al. created a probabilistic software quality model based on the ISO/IEC 9126 standard [7]. To calculate the four quality attributes that defines maintainability in ISO/IEC 9126, Bakota et al. used the following metrics. What makes this model different from the rest is that instead of giving one measure of goodness it provides a function over time. This makes it possible to see the evaluation of maintainability for the codebase.

- TTLOC - Total logical lines of code for the system.
- McCage - McCabe cyclomatic complexity.
- CBO - Coupling between objects.
- NII - Number of incoming invocations for the system.
- Impact - Size of the change, computed by the SEA/SEB algorithm.
- Error - Number of serious coding rule violations.
- Warning - Number of suspicious coding rule violations.
- Style - Number of coding style issues.
- CC - Clone coverage. the percentage of duplicate source code parts.

2.8.3 QMOOD

The hierachial Quality Model for Object-Oriented Design (QMOOD) was created by Bansiya and Davis in 2002 [8]. QMOOD aims to compute values for six quality attribues, as can be seen in Table 2.2. The attributes and weights are based on a review of 13 object-oriented development books and publications. From that a table on which metrics affected certain properties positively was produced. All values was normalized and from that the final weights was calculated. What makes QMOOD stand out is that the model can be easily modified to include different relationships and weights which makes it adaptable to different demands. The model have been evaluated using empirical and expert opinion.

Quality attributes	**Computation formulas**
Reusability	$0.25 \cdot \text{Cohesion} + 0.5 \cdot \text{Messaging} + 0.5 \cdot \text{DesignSize} - 0.25 \cdot \text{Coupling}$
Flexability	$0.25 \cdot \text{Encapsulation} + 0.5 \cdot \text{Composition} + 0.5 \cdot \text{Polymorphism} - 0.25 \cdot \text{Coupling}$
Understandability	$0.33 \cdot \text{Encapsulation} + 0.33 \cdot \text{Cohesion} - 0.33 \cdot \text{Abstraction} - 0.33 \cdot \text{Coupling} - 0.33 \cdot \text{Polymorphism} - 0.33 \cdot \text{Complexity} - 0.33 \cdot \text{DesignSize}$
Functionality	$0.12 \cdot \text{Cohesion} + 0.22 \cdot \text{Polymorphism} + 0.22 \cdot \text{Messaging} + 0.22 \cdot \text{DesignSize} + 0.25 \cdot \text{Hierarchies}$
Extendability	$0.5 \cdot \text{Abstraction} + 0.5 \cdot \text{Inheritance} + 0.5 \cdot \text{Polymorphism} - 0.5 \cdot \text{Coupling}$
Effectiveness	$0.2 \cdot \text{Abstraction} + 0.2 \cdot \text{Encapsulation} + 0.2 \cdot \text{Composition} + 0.2 \cdot \text{Inheritance} + 0.2 \cdot \text{Polymorphism}$

Table 2.2: QMOOD - Computation formulas

The metrics in Table 2.2, that aren't already defined in Section 2.4 or uses another definition are:

- Abstraction, Average Number of Ancestors (ANA). The average number of classes from which a class inherits information.
- Encapsulation, Data Access Metric (DAN). the ratio of the numer of private attributes to the total number of attributes declared in the class.
- Hierarchies, number of hierarchies (NOH). The number of class hierarchies in the design.
- Composition, Measure of Aggregation (MOA). The number of data declarations whose types are user defined classes.
- Inheritance, Measure of Functional Abstractions (MFA). The ratio of the number of number of methods inherited by a class to the total number of methods accessible by memer methods of the class.
- Polymorphism, Number of Polymorphic Methods (NOP). The number of methods that can exhibit polymorphic behaviour.
- Messaging, Class Interface Size (CIS). The number of public methods in a class.
- Complexity, Number of Methods (NOM). The total number of methods defined in a class.

2.8.4 SQO-OSS

Software Quality Observatory for Open Source Software, SQO-OSS, is a hierarchial quality model which quantifies product (code) quality and community quality [56]. Product quality consists of maintainability, reliability and security. While community quality consists of mailing list quality, documentation quality and developer base quality. For maintainability, SQO-OSS uses the ISO/IEC 9126 quality model. They argue that OSS has made traditional quality evaluation model non applicable and therefore proposed a new model that takes the community into account.

2.9 Static Analysis

Which can also be called static code analysis or static program analysis, is the analysis of code or a program that is not running, contrary to dynamic analysis. Static analysis can be used to find security vulnerabilities [41, 61, 39] and bugs [9] and is often used together with dynamic analysis.

2.10 Case Study

Runeson and Höst states that "Case study is a suitable research methodology for software engineering research since it studies contemporary phenomena in its natural context." [55]. They list four different purposes for doing a case study.

- Exploratory - find ideas, insights or hypotheses for new research.
- Descriptive - portray a situation or phenomenon.
- Explanatory - find an explanation of a problem or solution.
- Improving - improve a certain aspect.

The purpose of a case study in software engineering is usually to try to improve a certain aspect of a studied phenomenon. While in other fields it is usually used for exploratory purposes.

Depending on the research perspective, a case study can be one of three types. Positivist, critical or interpretive [40]. A positivist case study aims to find evidence for a formal proposition, quantifiable measurements, hypothesis testing or drawing of inferences about a phenomenon. The main task of a critical case study is social criticism. It aims to identify different forms of social, cultural and political domination that may hinder human ability. Interpretive case studies aim to understand phenomena through the participants' interpretation of their context.

2.11 Related Work

Articles related to migrating code from already existing native codebases to a cross-platform framework have not been found. A few master theses have been found about React Native which have shown that in some cases it seem to be a suitable replacement for native applications.

React Native has been investigated in the thesis *"Evaluation Targeting React Native in Comparison to Native Mobile Development"* by Axelsson and Carlström which have looked at developer impression, user experience and system performance [4]. The conclusion was that the users could barely notice any difference on native applications compared to React Native applications. The developer experience was heavily dependent on previous experience and on performance "the CPU and RAM usage was in an acceptable range" according to Axelsson and Carlström.

Also Hansson and Vidhall have compared React Native with native applications in *"Effects on performance and usability for cross-platform application development using React Native"* where they used a performance evaluation, platform code sharing and a user study [30]. The conclusion was that the performance results were promising for React Native and about 75% code were reused. The user study had too few participants to draw any conclusions from but the results points to equal user experience.

The thesis *"React Native application development"* by Danielsson compared a native Android application with a React Native Android application [19]. The results came from a user study and performance tests. The conclusion in this report were that almost all users would have no problems using the React Native app and in performance the native application were better in all cases but the differences in the tests were small.

In the article *"Software quality models: Purposes, usage scenarios and requirements"* Deissenboeck et al. have evaluated software quality models and they proposes a classification of quality models by different purposes [21]. The classification is called DAP - definition, assessment and prediction - where ISO 9126 is an example of a definition model, maintainability index is an example of an assesment model and Reliability growth models (RGM) is an example of a prediction model. According to them an ideal model takes all three classes into account.

In the article *"Size Estimation of Cloud Migration Projects with Cloud Migration Point (CMP)"* by Tran et al. the model for estimating migration is based on function points adapted to Cloud migration [59]. The migration included moving existing code to the cloud and make it run. They looked at the following cost factors:

- Installation and configuration
- Database changes
- Code changes
- Connection Changes

The study shows that their size metric Cloud Migration Point is more suitable for cloud migration project than other existing size metrics.

Some articles about working with cross-platform tools was also found.

Dalmasso et al. presents some desirable requirements of a cross-platform framework and some guidelines for developers that are about to choose a cross-platform framework in *"Survey, comparison and evaluation of cross platform mobile application development tools"* [18]. Some important factors that was mentioned was:

- Native user experience
- Offline and online storage
- Compatibility
- Access to built-in features
- Security

In *"Comparing Cross-Platform Development Approaches for Mobile Applications"* by Heitkötter et al. they present a lot of criterias for evaluating a cross-platform tool for mobile applications [31]. Because a lot of cross-platform tools are based on web technologies - that are standardized, popular, reasonably simple but powerful and well-supported - they are an alternative with low barriers for development. Due to the low barrier they suggest that even if only a single platform is to be developed a cross-platform tool could be the best solution.

Willocx et al. presents a performance analysis of mobile applications with ten different cross-platform tools and native in *"Comparing Performance Parameters of Mobile App Development Strategies"* [63]. They also draw some general conclusion about what platform should be chosen in a specific context based on the performance analysis. Some examples of evaluation criterias are response time, cpu useage, memory useage, disk space and battery useage.

3 | Method

In this chapter the research method is described with problems and important decisions. The metrics, static analysis, the release versions and the implementation of the React Native codebase are covered.

3.1 Description of Research Method

There are five major steps to be walked through when conducting a case study [55], where the three first is presented below, the fourth is the analysis of the collected data which is presented in Chapter 5 and the last step, reporting, is this master thesis.

3.1.1 Case Study Design

The objective of this thesis was to find out if modifiability for one code base developed in React Native is better than the modifiability of two native codebases for Android and iOS. This was done both by evaluating the stability of the platforms and calculating the modifiability of the codebases. The modifiability of the codebases was calculated using the same method as modifiabiltity was calculated in SQMMA, explained in Section 2.8.1.

Case Study Flexibility

According to Runeson and Höst a case study is a flexible design strategy with significant amount of iteration over the five major steps when conducting a case study. However a limitation to the flexibility is mentioned that if the objective changes it is a new case study. The evolution of or work is further discussed in Chapter 5.1.1.

3.1.2 Preparation for Data Collection

Information about what kind of data from the releases of the different platforms had to be found. As well as decisions about what metrcis should be used, how they should be calculated, and on what parts of the code they should extracted from. When making measurements on different components of the code, one step was to decide what code should be included for a specific component. Another step was to find the relevant code to make the measurements on, both of these steps were done according to the authors' opinions.

Version Releases

For the different operating systems there are different release cycles that directly affect how much time and effort is needed to stay up-to-date with the current standards. Some releases just change how a function should be called while others can deprecate or kill certain functionality in the application.

Data for the releases were gathered for iOS[1], Android[2] and React Native[3]. The frequency of the releases differs drastically between the different platforms.

iOS Apple labels its versions X.Y.Z, where X is the new version, Y is a major release and Z is a minor release [35]. Apple has since 2012 released a new version of iOS in early September every year. It is usually in these releases most of the new functionality for the new iOS version is presented, some functionality is also updated in the major releases while the minor releases are generally for bug fixes.

Android Google have not been consistent on when they make releases, but they also label versions X.Y.Z where X is the new version, Y is a major release and Z is a minor patch [3]. Version 1.0 was released on September 23, 2008 and version 7.0 was released on August 22, 2016.

React Native React Native has since v0.40, released in December 2016, entered a monthly release schedule with v0.41 being the January release of 2017, v0.42 being the February release and so on. React Native have not reached a "1.0" release yet and changes rapidly. With each release a list of "breaking changes", added features and bugfixes are presented. The breaking change list shows things that has been changed between the different versions that will break the application if not fixed. Most of these changes can be applied by tools used when upgrading version of React Native. If these changes are not applied with the tool, every breaking change comes with a "how to migrate" text so that migrating between the different releases are going to be as simple and easy as possible. React Native keeps a wikipedia with all the breaking changes[4] that serves two purposes, to get easier upgrades and tracking the breaking changes over time. Every breaking change follows a template that looks as following:

- Who does it affect?
- How to migrate?
- Why make this breaking change?
- Severity (number of people affected · effort)

React Native keeps a roadmap[5] in order to outline some of the upcoming plans. But since React Native is moving in such a fast pace not everything is included in the roadmap. The roadmap is divided into three major parts: adding functionality to the core libraries, improving core performance and stability, and making a better developer experience.

Metrics

In order to be consistent in the measurements of the code, some assumptions had to be made or modifications to the metrics defined in Section 2.6.1. Some of the relevant metrics, the ones related to modifability, were either missing reliable information or it was chosen to make assumptions on about how they should be used or calculated in the work. These metrics are listed below.

- Coupling - The sum of the afferent and efferent coupling metrics. In the measurements both afferent and efferent coupling are calculated by counting the number of imports for the specified files.

[1] `https://developer.apple.com/library/prerelease/content/navigation/`
[2] `https://developer.android.com/about/index.html`
[3] `https://github.com/facebook/react-native/releases`
[4] `https://github.com/facebook/react-native/wiki/Breaking-Changes`
[5] `https://github.com/facebook/react-native/wiki/Roadmap`

- EntExt - As for Chawla and Chhabra, a tool to get good results could not be found for the EntExt value. A neutral value of 1 was chosen.
- MethodInheri - The method inheritance factor was used by taking the number of inherited methods in a class, divided by the total number of methods used in that class. It was chosen to not include methods inherited from the standard libraries of the different systems, but only the ones that were written for the project. React Native classes only extends the standard Component class[6]. Because of that only methods defined in the Component class are inherited which makes this value only dependent on number of methods in the examined class. The Component class is an abstract base class provided by React.
- Hierarchies - For the measurement in Android only the super classes present in the codebase and the object class were counted. In React Native all classes created only had the Component class as super class and therefore the value 1 were set for all classes.
- Subclasses - In the React Native codebase subclassing is not used and that is the reason all subclass values are set to zero. In iOS and Android only direct subclasses were counted, not transitive subclasses.

Table 3.2 is a modified version of Table 1 in [11] containing only the metrics associated with modifiability.

There are other quality models than SQMMA, like QMOOD [8] and SQO-OSS [56]. However, these are either qualitative models or are not quantifying the quality attributes in the ISO/IEC 25010 standard. Another reason for choosing to use SQMMA for measuring modifiability was because both Bakota et al. and Chawla and Chhabra work with the model seemed promising.

The reason for only looking into the modifiability quality attribute of SQMMA was partly that the result would be more focused. Also because it was a quality attribute that was added in the ISO 25010 standard that consisted of two quality attributes Changeability and Stability that was previously used in ISO 9126. That some of the metrics part of modifiability have promising result on showing how good a codebase is was also part of the reason for choosing modifiability.

Static Analysis Tools

There exist some static analysis tools for the different platforms. But some that gave reliable results for both the relevant platforms and techniques and was open source could not be found. The available open source versions were usually out-dated or did not cover the desired measures for this thesis. It was decided to write the analyzers by hand for Android[7], iOS[8] and React Native[9]. All of the analyzers were written in Python. The static analysis was performed mostly with the help of regular expressions, which is a type of pattern matching.

In Table 3.1 the metrics that could be extracted from the different codebases are listed. X means that the metric was possible to extract, - that it was not, and a numerical value if the metric is constant over the entire codebase.

The reason why the metric DIT was not possible to extract from the Objective-C code was that some of the inheritance tree is not part of the user written code and could therefore not be parsed. This metric was easy to extract from XCode and because of this it was chosen not to extract this metric with the static analyzer. Why NSC and DIT are set to numerical values for JavaScript/JSX is explained in Section 3.1.2.

[6] `https://facebook.github.io/react/docs/react-component.html`
[7] `https://github.com/rq-abrahamsson/android-static-analyzer`
[8] `https://github.com/berntsendavid/obj-c-static-analyzer`
[9] `https://github.com/rq-abrahamsson/react-native-static-analyzer`

Language	Objective-C	Java	JavaScript/JSX
Source lines of code (SLOC)	X	X	X
Number of methods (NOM)	X	X	X
Number of overridden methods (NORM)	X	X	X
Number of inherited Methods (#inhMethods)	X	X	X
Response for class (RFC)	X	X	X
Afferent Coupling (affCoupling)	X	X	X
Efferent Coupling (effCoupling)	X	X	X
Nesting level (NBD)	X	X	X
Lack of cohesion (LCOM)	X	X	X
Number of children (NSC)	X	X	0
Depth of inheritance tree (DIT)	-	X	1

Table 3.1: Metrics extracted from the static analyzers

All metrics used to calculate the modifiability value are listed in Table 3.2. The table also contains the source from where the calculations have been collected. Some metrics are compositions of the metrics extracted from the analyzers, the calculations for these can be found in Table 3.3.

Design Metric	Metric Name
Average size of statements (avgSize)	TLOC
Directly called components (Communication)	RFC
Number of entry/exit points (EntExt)	1
Number of nested levels (Nesting)	NBD
Coupling between objects (Coupling)	CBO
Lack of cohesion (LCohesion)	LCOM
Depth of inheritance tree (Hierarchies)	DIT
Polymorphism Factor (Polymorph)	NORM/NOM
Number of children (Subclasses)	NSC (or NOC)
Method inheritance factor (MethodInheri)	MFA (or MIF)

Table 3.2: Metrics, tools and source

avgSize	SLOC/NOM
Coupling	affCoupling + effCoupling
Polymorph	NORM/NOM
MethodInheri	#inhMethods/NOM

Table 3.3: Metrics calculations

```
effCoupling = 0
with open(file_name, 'r') as f:
    s = f.read()
    matches = re.findall('import.*\.\/.*\;',s)
    effCoupling = len(matches)

import_regexp = "import.*" + class_name + "\'*\;"
try:
    affCoupling = len(sh.grep("-r", import_regexp, "src", "index.android.js", "index
        .ios.js").splitlines())
except:
    affCoupling = 0
```

```
coupling = affCoupling + effCoupling
```

Listing 3.1: Example code that gets the coupling value for React Native. Efferent coupling counts the number of imports in the file. Afferent coupling counts the number of times the current class is imported.

```
nrMethods = 0
methods = []
with open(file_path, 'r') as f:
    s = f.read()
    regex_matches = re.findall('[-+] \(.*',s)
    for match in regex_matches:
        methods.append(match[match.find(')')+1:match.find(':')])
    nrMethods = len(methods)
```

Listing 3.2: Example code that counts the number of methods in an iOS file.

```
def get_rfc(file_name):
    number_of_methods = 0
    rfc = 0
    with open(file_name, 'r') as f:
        s = f.read()
        m = re.findall('(?:public|private|protected) \w* \w*\([^(^)^{^}]*\).*{',s)
        methods = map(lambda x: re.findall('\w*\(',x)[0], m)
        numbers = map(lambda x: sh.grep(" " + x,file_name,'-c'), methods)
        count = reduce(lambda x, y: int(x) + int(y), numbers)
        rfc = int(count)-len(m)
    return rfc
```

Listing 3.3: Example code that gets the directly called components of a file in the Android codebase.

In the code Listings 3.1, 3.2 and 3.3 there are some example Python code from the static analyzers that shows how some metrics were extracted which is very similar between the analyzers. What differs the most are the regular expressions used to match for code.

Code Migration

Migrating code can be divided into different phases depending on which codebase should be migrated and to where. Since this have been a merge of two native codebases into one shared codebase it was chosen to divide the migration process into the following phases:

1. Locate code related to component in native code. The native code that should be measured for that specific component.
2. Write shared code for component in React Native.
3. Integrate component into application.
4. Add Android and iOS specific code if needed.

After the migration phases were identified, the application was divided into different functionality that exist in the original applications. After that, the functions was prioritized according to what to develop first. The prioritization was based purely on our opinion as developers on how to proceed when developing an application like this. The functionality is listed in prioritization order below:

1. Navbar
2. Menu
3. List view

The implementation details for every component can be found in Section 3.2.

Since there have been a lot of applications already developed in React Native, like Facebook, Instagram, Airbnb, etc, it was assumed that all parts in the applications could be migrated [52]. In the worst case the parts could be added by creating a native component in the same way React Native componens are made [50]. If something do exist in native code it is always possible to write a wrapper for it.

3.1.3 Collecting Evidence

The version data for iOS were gathered from the apple developer library[10], for Android is was gathered from the Android developer library[11], and for React Native it was gathered from the React Native changelog[12].

Every file related to a component in the native code were identified. Then for every commit that had modified the current file in some way, the metrics to calculate modifiability were extracted. The extraction was performed with the help of the static analyzers described in Section 3.1.2. The results were used to calculate the modifiability over time. The modifiability values for the components were calculated both using the sum and the average modifiability value of the included files in the components. This was done for every day to get a graph of the evolution of the value over time. The same was done with the React Native code after the component had been migrated.

3.2 Components and Files

The two projects were structured quite differently. The iOS application was structured as a classic MVC[13] project while the Android application was structured like the general structure of Android applications where there are Intents[14], Fragments[15] and Activities[16] etc.

Because of the difference of two original projects and the already existing developer experience of React development[17] the migration was not done by trying to mimic the original code. It was done by using the standard way of writing React native code. The code migration process is described in Section 3.1.2

When using the static analyzers to take measurements for the files related to one module it was decided to ignore files that were located in either the standard libraries of the different languages or in the package manager. In iOS for example files located in the folders UIKit or Pods were not added to the tables where the rest of the files were included.

The functionality of the two applications that was migrated was to list, rent and watch movies of different categories. Both of the applications were in a beta phase of development by the start of the project and was released during the project.

Some class and component names are used in this section and they are written with the name that was used in the code.

3.2.1 The applications

The original applications are applications that list movies, series and allows the user to look at the movies and series. The React Native applications replicated part of that functionality with network requests that contained mock data. Due to confidentiality, more information than this cannot be given about the original applications.

[10]`https://developer.apple.com/library/content/navigation/`
[11]`https://developer.android.com/about/versions/marshmallow/android-6.0.html`
[12]`https://facebook.github.io/react-native/versions.html`
[13]`https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`
[14]`https://developer.android.com/reference/android/content/Intent.html`
[15]`https://developer.android.com/guide/components/fragments.html`
[16]`https://developer.android.com/reference/android/app/Activity.html`
[17]One principle of React, "Learn Once, Write Anywhere", see Section 2.2.1

3.2.2 React Native

The reason for using React Native in the analysis was that React Native is based on the large web language JavaScript together with a large web framework React.js where the authors of the report already had experience. According to the home page of React one of the principles for React is "Learn Once, Write Anywhere", which made the authors experience in React really useful. The possibility to create native applications and that many large applications created with React Native already existed was also a main reason. The finished application, after removing sensitive data, can be seen in Figure 3.1.

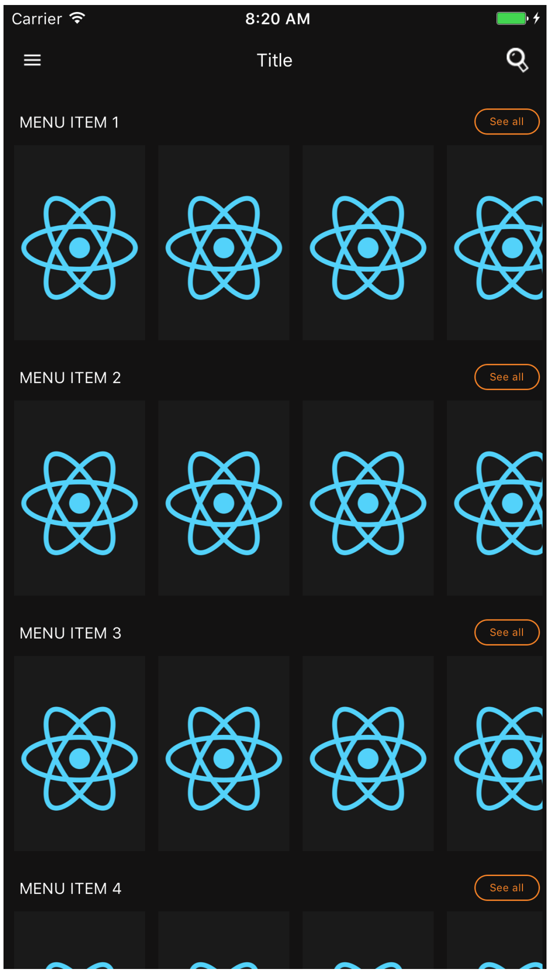


Figure 3.1: The start page of the application, showing navbar and list view.

3.2.3 Navbar

The navbar is the bar in the top of the application. It includes a menu button to the left, a logo in the middle and a search button to the right. It also includes changes of the navbar when pressing the search button, when pressed it gives a text input field and a back button. When pressing the backbutton it goes to the original navbar. The migrated application does not include the Chromecast button which is added to the bar when Chromecast is available. When displaying sub pages only a back button and page name are visible in the bar, this functionality is part of the navbar component. On the video view there is a share button to the right which will not be included. Figure 3.2 is an example of how a navbar looks.

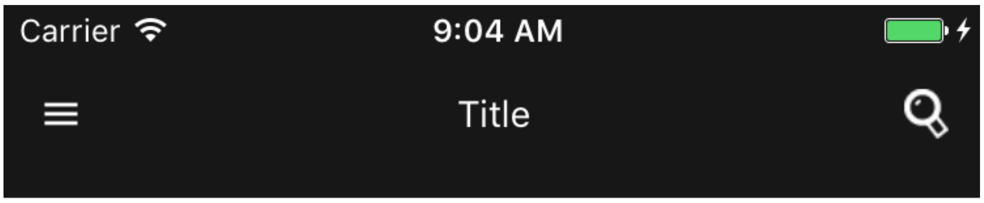


Figure 3.2: Navbar example with hamburger menu, title and search button.[18]

iOS

The navbar for the iOS application consist mainly of the UINavigationController[19] which is part of the UIKit[20]. UIKit is a framework that is crucial for app development in iOS. It presents the view and window architecture to manage the application's UI. It also manages the event handling and the application model needed to drive the main loop and interact with the system. The user written files for the navbar are two controllers.

Android

The navbar in the Android application is built with android.widget.Toolbar[21] and android.app.ActionBar[22] which is standard Android components for handling a navbar. Because of that the code is mostly setting the state of the component for different views. But the initialization code consists of a whole function which is placed in the class that was measured for the navbar.

React Native

The code for the navbar in React Native consists of four files and the main component used for the navbar is the React Native standard component Navigator[23].

3.2.4 Menu

The menu is what appears from the left when either swiping from the left of the screen or when pressing the menu button. In the menu there is a list of options where different views can be accessed. Dummy views or a way to easily add new views to the menu is included in the menu component. Every option should be able to have an image. Figure 3.3 shows an example of a menu in iOS implemented in React Native.

[18]https://en.wikipedia.org/wiki/Hamburger_button
[19]https://developer.apple.com/reference/uikit/uinavigationcontroller
[20]https://developer.apple.com/reference/uikit
[21]https://developer.android.com/reference/android/widget/Toolbar.html
[22]https://developer.android.com/reference/android/app/ActionBar.html
[23]https://facebook.github.io/react-native/docs/navigation.html#navigator

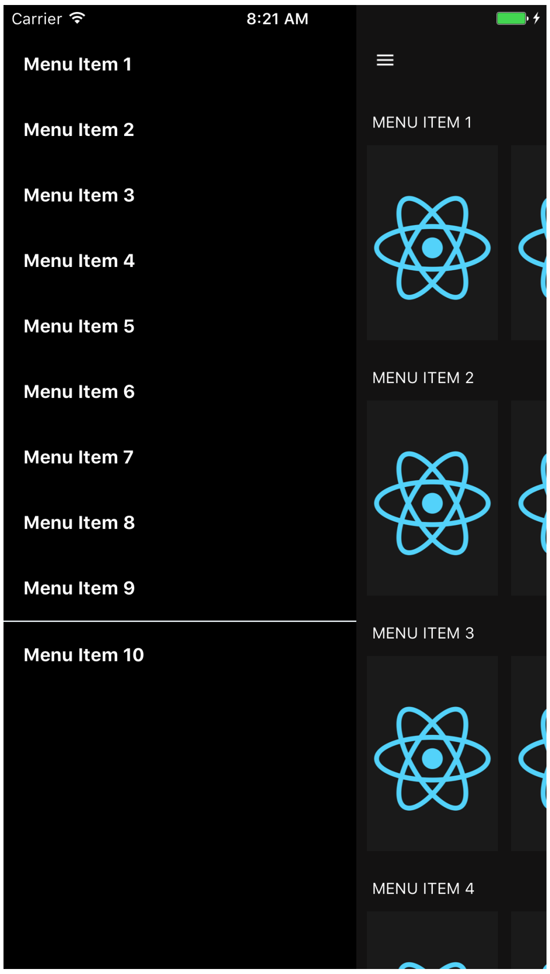


Figure 3.3: Menu example of menu sliding in from the left, with menu items and a separator.

iOS

The menu component in the iOS application is a lot bigger than the navbar and is built up by ten different files. Two models, five views, one controller and two factory classes.

Android

The Android menu consist mostly of two files where one file contains a base class and the other contains the implementation of the SideMenu.

React Native

The menu is one component in one file. The component is placed in different standard components for the different platforms, DrawerLayoutAndroid for Android and Drawer for iOS.

3.2.5 List View

The list view are a list of movies or movie packages that are related by some topic, for example top list, news, tv series etc. The list view is scrollable to the sides and when pressing on a movie image the application go to a dummy video view for that movie. Included in the list view are also the topic name and a button "show all" that goes to the library view as a sub view for that topic. All video data comes from a rest API. Figure 3.4 shows an example of how a horizontal list view could look, this list view is larger than the width of the phone so in order to show all items in the list view one has to scroll sideways.

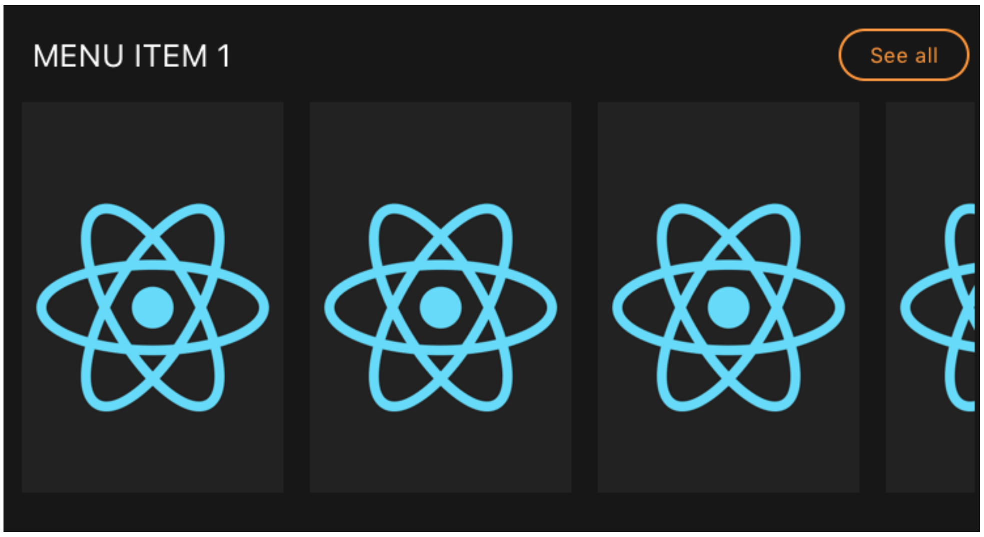


Figure 3.4: Example of a horizontal list view that has to be scrolled to show all content.

iOS

The list view consists of six files. One model, two views and three controllers.

Android

The list view is mainly made up of two files where one is an abstract base class and the other one implement the base class.

React Native

The list view for React Native consists of two components where one is the container which is built of the standard component ScrollView. The other component is the image items that fill the container.

4 | Results

This chapter contains the data and results that have been produced from the static analyzers and release data.

4.1 Releases

This chapter contains data retrieved about the different types of releases for the different platforms and frameworks.

4.1.1 iOS

Figure 4.1a and 4.1b illustrates the addition of new methods, modifications of existing methods and removal of deprecated methods to the iOS API. As can be seen in the figures around version 8 in the end of 2014 there was a release with a lot of changes.

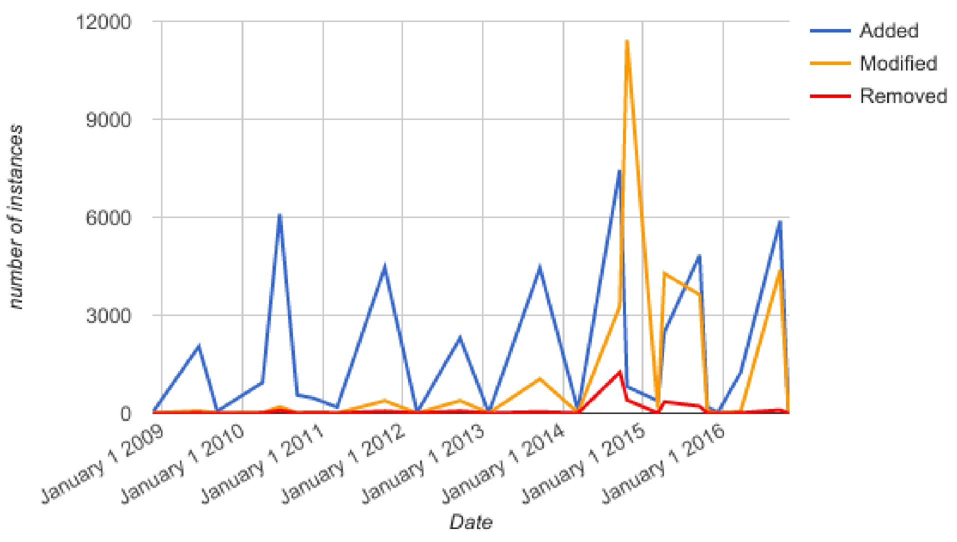
A compiler warning will be shown in XCode when a method have been marked deprecated, but there is no guaranty of how long it will be available after it has been marked. Figure 4.1a displays how the changes have happened over time, while Figure 4.1b shows the changes in respect to the version number of the release. As seen in Figure 4.1b, usually the majority of changes to the API occur in the new version releases. In Table 4.1 the average number of changes can be seen that have happened in the iOS API since the 2.0 release. So in average 16 additions are made to the iOS API every day etc. Table 4.2 shows the average number of changes for every new version of iOS.

	Additions	**Modifications**	**Removals**
average per day	16	10	1
average per release	1877	1226	109

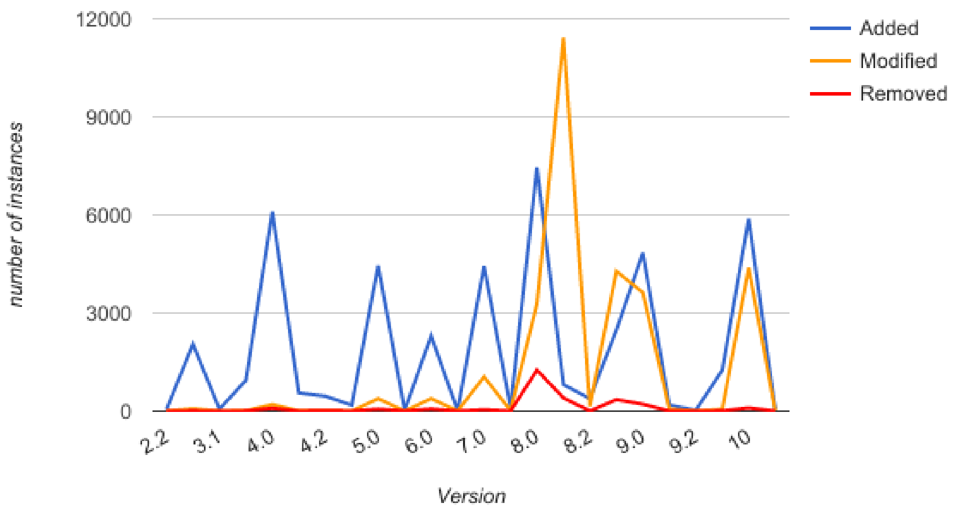
Table 4.1: iOS API changes over time and releases

	Additions	**Modifications**	**Removals**
2.x	47	4	0
3.x	1015	28	5
4.x	1822	56	27
5.x	2236	190	32
6.x	1158	193	33
7.x	2269	531	22
8.x	2779	4785	502
9.x	1574	936	56
10.x	2970	2200	49

Table 4.2: Average iOS API changes per version



(a) iOS releases by date



(b) iOS release by version

Figure 4.1: iOS release data

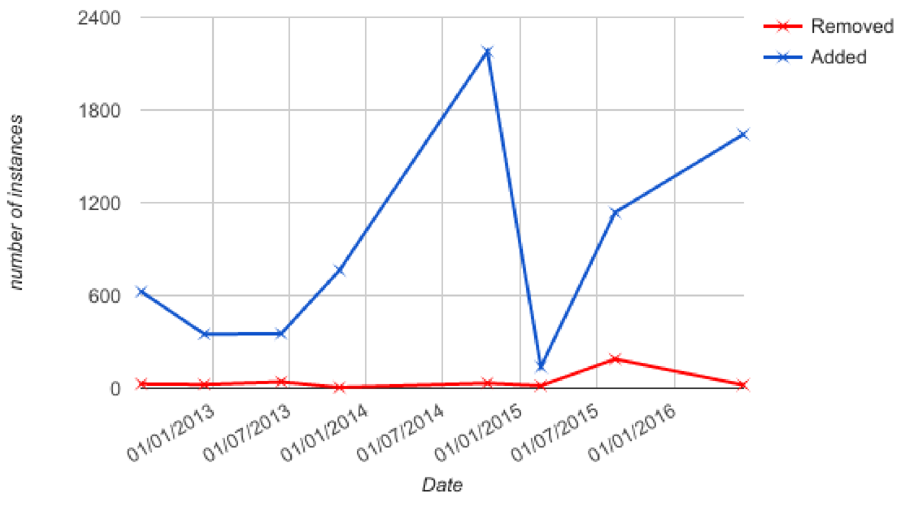
4.1.2 Android

In Figure 4.2b the number of removals and additions of the Android API is shown for API versions 16 to 24. It can be seen that in version 21 and 24 a lot of new features was added and in version 22 the number of added features was less than in any other release. The number of removals is not higher than the number of additions in any release.

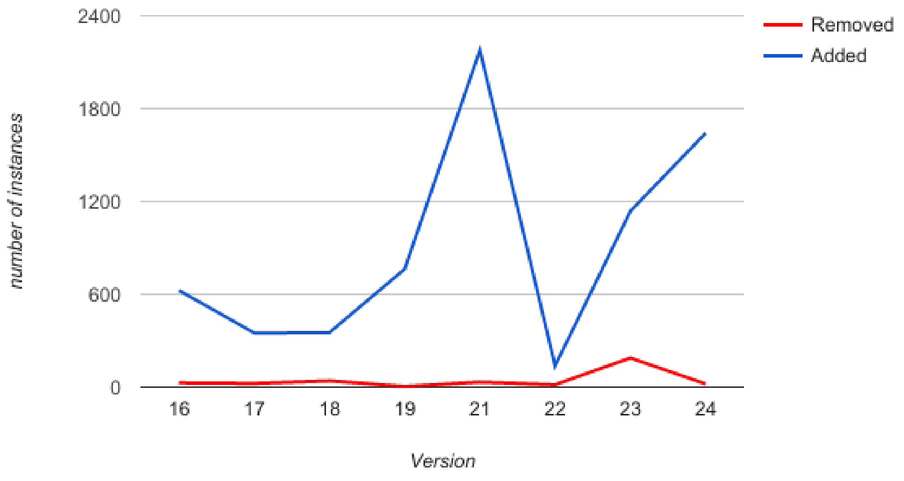
In Android before methods are removed, they will be annotated with being deprecated which discourage usage. Usage of deprecated methods is noticed by the compiler and will generate warnings [22]. Figure 4.2a shows the same data but uses date instead of versions to show when additions and removals occured. In the figures it can be seen that both number of additions and removals for Android are pretty irregular and that the number of removals are fewer than the additions. In Table 4.3 the average additions and removals can be seen which show that the number of additions is 20 times more than removals.

	Additions	**Removals**
day	5	0.25
release	898	45

Table 4.3: Android API changes over time and releases



(a) Android releases by date



(b) Android releases by version

Figure 4.2: Android release data

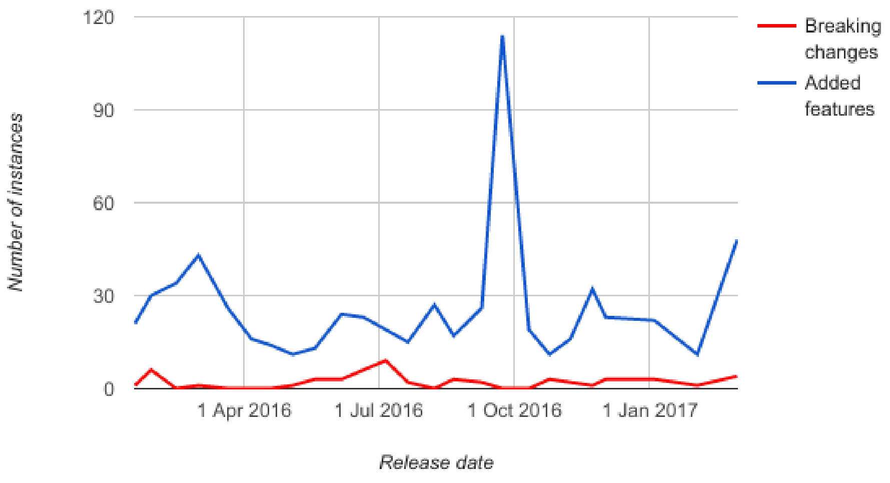
4.1.3 React Native

The data for React Native is not a list of changes to the API although some of them affect the API. Others affect, for example, build tools and integrations with Android and iOS but they will still break the app. In the Figures 4.3a and 4.3b added features and breaking changes are presented. The number of added features is not very different between releases exept for the release of version 0.35 where it was added more than double of any of any other version. It can also be seen that the number of breaking changes has been at most 9 for a release which is less than than the number of added features for the release with the least amount of added features.

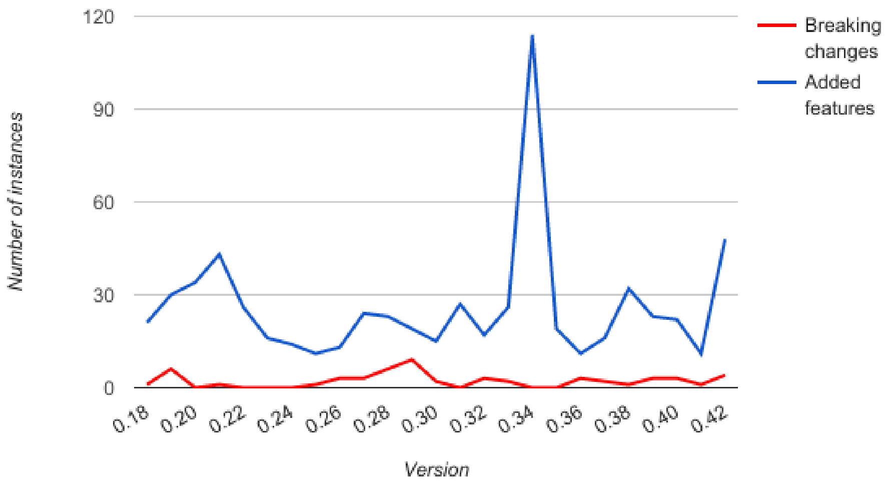
Because React Native allows the writing of native code when Android and iOS makes a new release it is possible to access the new functionality directly. The average number of features added per day is 1.61 and the number of breaking changes per day is 0.13 which can be seen in 4.4. It could also be thougth of as almost every eighth day there will be a breaking change.

	Added features	**Breaking changes**
day	1.61	0.13
release	26.6	2.2

Table 4.4: React Native API changes over time and releases



(a) React Native releases by date



(b) React Native releases by version

Figure 4.3: React Native release data

4.2 Modifiability

This chapter contains the data retrieved from the static analysis tools for the different components and languages. Since native versus cross-platform solutions were being investigated, the modifiability value for the native code was calculated as one codebase. After the modifiability values were calculated, they were normalized to the first functional commit for the component. So an empty file that was generated to create structure of the code wouldn't be included. The normalized value shows how the modifiability have changed relative to itself.

The modifiability value mentioned in the graphs in this section comes from the calculations made for modifiability in Section 2.7.

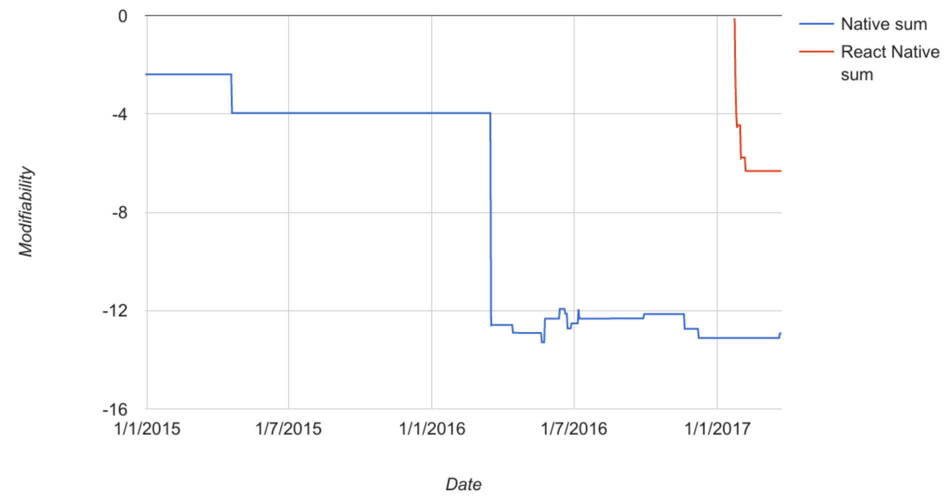
		Native	React Native
Average	Min	-4.5	-1.7
	Max	-0.9	-0.1
Sum	Min	-13.3	-6.3
	Max	-2.4	-0.1

Table 4.5: The maximum and minimum modifiability values for the Navbar component.

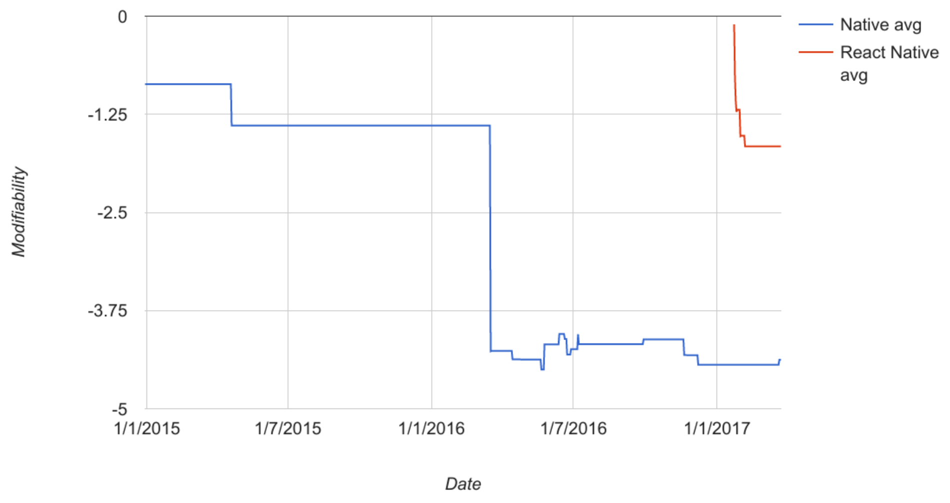
From SQMMA all the quality attributes have a negative impact on maintainability. Therefore a modifiability value closer to 0 indicates that the code is easier to modify. To calculate modifiability for a component the metric value used in Equation 2.15 is the sum of the metric value from all the files regarding that component.

4.2.1 Navbar

The modifiability value for the navbar component, when using the sum of the metric values, is shown in Figure 4.4a . Figure 4.4b shows the modifiability value when using the average metric value for the metric values. The maximum and minimum modifiability values of the codebases can be seen in Table 4.5.



(a) Modifiability when using the sum of the metrics



(b) Modifiability when using the average value of the metrics

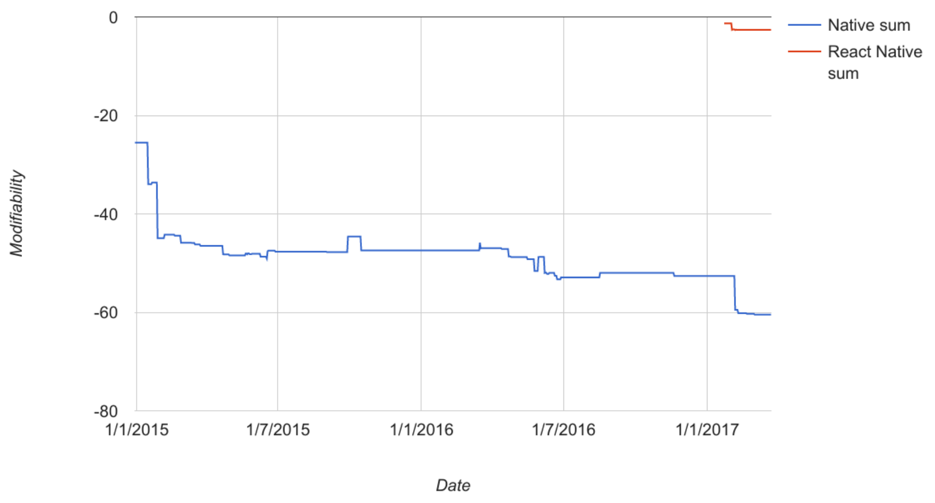
Figure 4.4: Navbar component data from native and React Native applications over time

4.2.2 Menu

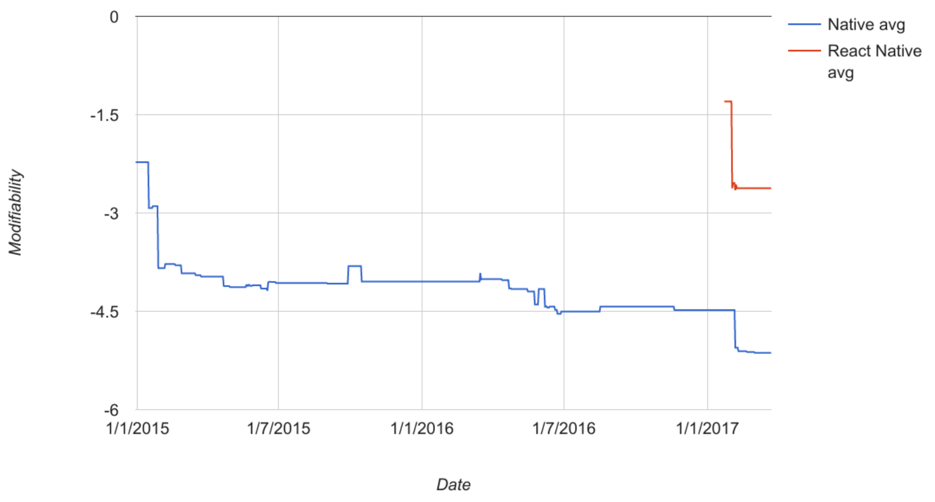
The modifiability value for the menu component, when using the sum of the metric values, is shown in Figure 4.5a. Figure 4.5b shows the modifiability value when using the average metric value for the metric values. The maximum and minimum modifiability values of the codebases can be seen in Table 4.6.

		Native	React Native
Average	Min	-5.1	-2.6
	Max	-2.2	-1.2
Sum	Min	-60	-2.7
	Max	-2.4	-1.3

Table 4.6: The maximum and minimum modifiability values for the Menu component.



(a) Modifiability when using the sum of the metrics



(b) Modifiability when using the average value of the metrics

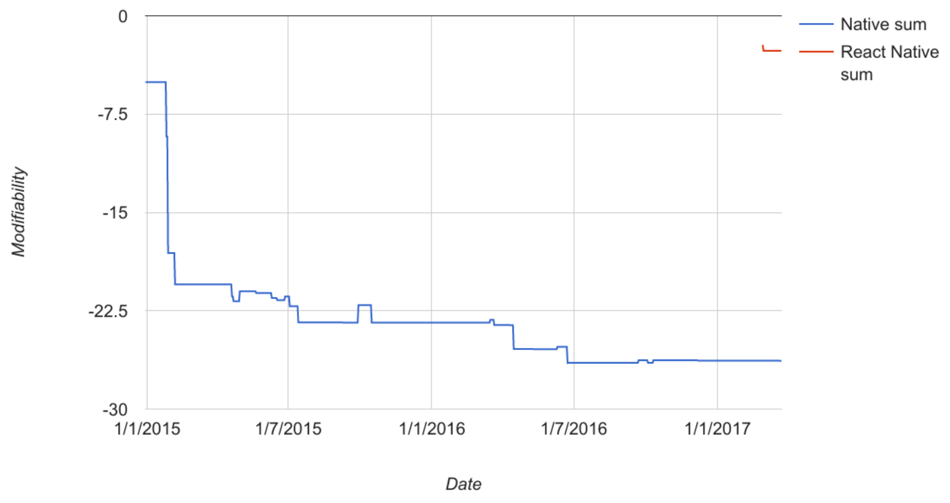
Figure 4.5: Menu component data from native and React Native applications over time

		Native	React Native
Average	Min	-3.4	-1.3
	Max	-0.7	-1.2
Sum	Min	-26.5	-2.7
	Max	-5.1	-2.2

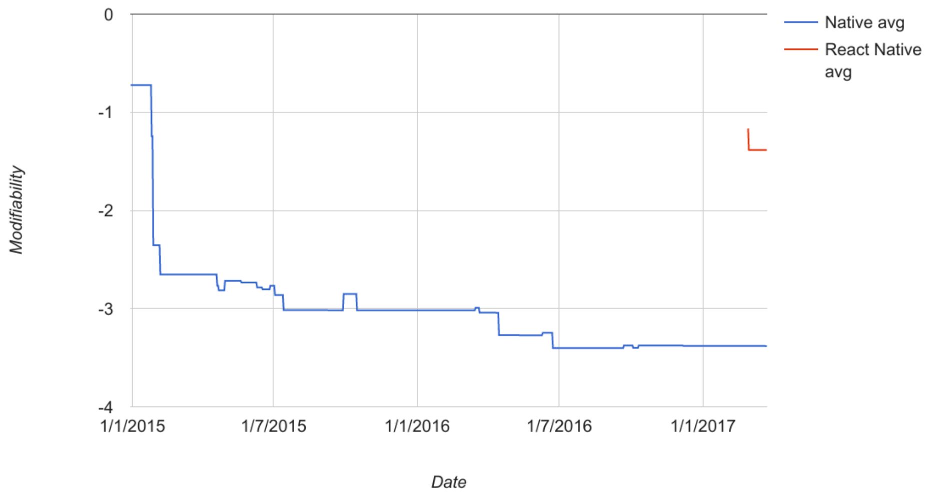
Table 4.7: The maximum and minimum modifiability values for the List view component.

4.2.3 List View

The modifiability value for the list view component, when using the sum of the metric values, is shown in Figure 4.6a. Figure 4.6b shows the modifiability value when using the average metric value for the metric values. The maximum and minimum modifiability values of the codebases can be seen in Table 4.7.



(a) Modifiability when using the sum of the metrics



(b) Modifiability when using the average value of the metrics

Figure 4.6: List view component data from native and React Native applications over time

4.2.4 Applications

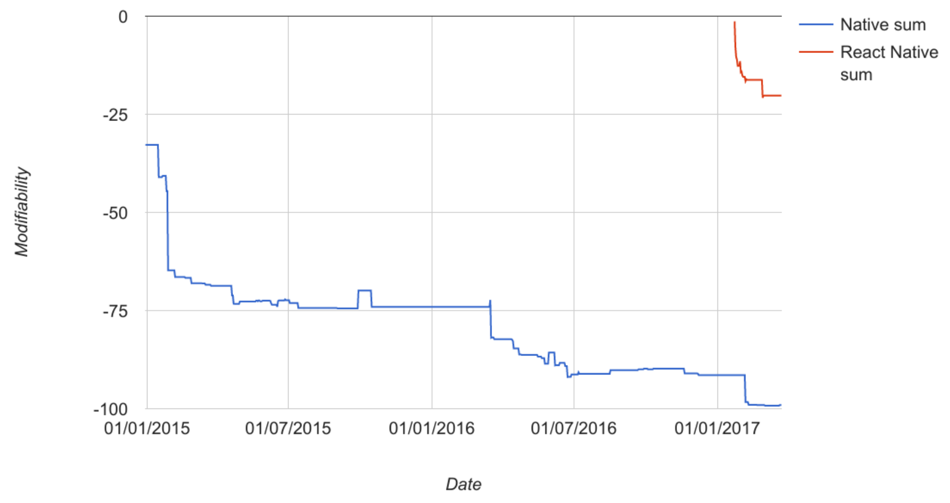
Figure 4.7a shows the aggregated modifiability value for the entire migrated application when using the sum of the metric values. The aggregated value for the native codebases only includes the files related to the migrated components. Figure 4.7b shows the modifiability value of the average metric values for both native and React Native. The maximum and minimum modifiability values of the codebases can be seen in table 4.8.

What parts of the application that was migrated is described in Chapter 3.1.2 under Code migration.

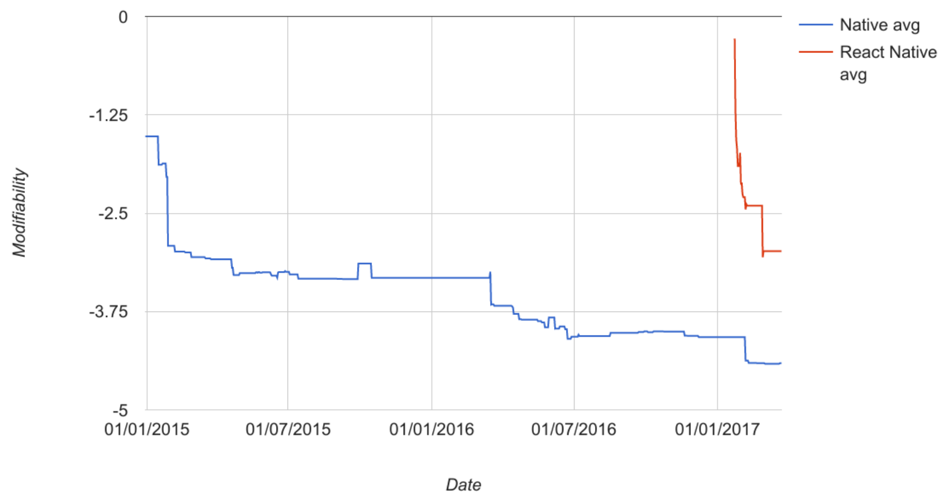
		Native	React Native
Average	Min	-4.4	-3.0
	Max	-1.5	-0.3
Sum	Min	-99	-20.8
	Max	-32.8	-1.3

Table 4.8: The maximum and minimum modifiability values for the all components.

The metric values used in the modifiability calculation were summed up for each date for all the files in the application. For the native application only files that were used in the migrated components were included in the calculations.



(a) Modifiability when using the sum of the metrics



(b) Modifiability when using the average value of the metrics

Figure 4.7: Migrated application data from native and React Native applications over time

5 | Discussion

In this chapter discussion about the used method is presented together with considerations about validity and reliability. The results presented in Chapter 4 is analyzed and discussed and there is also a short section about how the work fits into a wider context.

5.1 Method

This section covers discussion about the method used, how the plan was followed, and validity and reliability of the results.

5.1.1 Method Evolution

The initial plan for this thesis was to compare CPT applications developed in React Native and Xamarin. The applications would be developed to imitate already existing applications for a product. In both CPTs, applications for both Android and iOS were supposed to be developed. These six applications were going to be user tested to examine how applications for the different platforms felt. In total there would be six applications to test, three for each platform.

The initial plan was then discarded because it was found that:

- To achieve better results effort should be focused on one CPT.
- The aim was to get quantitative results because of the increased replicability. User test would have given qualitative results.

Since both of the authors had worked quite a bit with React, and the philosophy of "Learn once, write anywhere" the choice landed on React Native instead of Xamarin. This choice was made to prevent spending a lot of time learning a new framework and language.

With inpsiration from Deissenboeck et al. work on *"Software quality models: Purposes, usage scenarios and requirements"* [21], which is described in Section 2.11. The plan was then to examine if migration cost of the applications could be predicted based on the maintainability of the native codebases. Deissenboeck et al. states that an ideal software quality model should involve a definition model, an assessment model and a predicion model. To validate the prediction of migration cost, time spent on the migration of a component was logged. The idea of linking migration cost to maintainability came from the work of Bakota et al. [6] with their work in developing a cost model for development cost based on software maintainability, this would be the prediction model. They retrieved the value for maintainability from their probabilistic software quality model [7]. A choice was made to change the quality model to SQMMA [11] instead of using the one used by Bakota et al. The choice was made because SQMMA had incorporated the use of the new ISO/IEC 25010 standard instead of the old deprecated ISO/IEC 9126 standard used by Bakota et al. This was an obvious change for the work to make it stay up to date and not get outdated before it was even started. Becuase of this SQMMA was supposed to be the assessment model and ISO/IEC 25010 the definition model.

The cost model by Bakota et al. is based on historical values of the maintainability of the code. It also uses additional factors like an erosion factor and a conversion constant that have to be manually set depending on each individual project. To increase the replicability of the work an important choice was to stick with metrics that could be calculated statically from the code.

The work was then shifted to compare maintainability between the different codebases, and how it changed over time. This shift made the work to only contain a definition model and an assessment model. Maintainability according to ISO 25010, consists of five quality attributes. Each of these quality attributes in turn consist of multiple metrics, and depending on which paper or study is being used, these metrics can differ. Since working with SQMMA there was a metric suite to work with. To get better result it was chosen to narrow the work even more and only consider one of the five quality attribute and evaluate how that specific quality attribute changed over time.

If the metrics only had to be retrieved for the current version of the application, the metrics could have been gathered manually. But since the aim was to get the metric values for the lifetime of the code, doing it manually was out the question. SQMMA had used two different static analyzers in their work, CKJM[1] and Metrics 1.3.6[2] [11]. Both of these static analyzers were outdated for with which system or IDE they could work with and also only worked with Java. No open source static analyzers which worked with all the languages could be found. In order to make sure the metrics were calculated in the same way over all the codebases it was decided to create the static analyzers.

The metric values were extracted for every file that was related to the chosen component. Then for every commit that the file had been modified in, the metric values were extracted. The extracted values were then used to calculate an aggregated value for the native code and for the React Native code, since the aim was to compare native source code to a cross-platform solution.

5.1.2 Source Criticism

Collecting articles and literature have mainly been done through the web library of Linköping University and Google scholar. Wikipedia have been the main source for general topics and facts which is an area where Wikipedia could be seen as a reliable source.

The main source for how to conduct a case study was the article *citetitle[]Runeson2009* from 2009 by Runeson and Höst [55] which is well cited (1807 times). A well cited article indicates that it is a reliable source. Also Per Runeson is cited in total 11091 times alone and Martin Höst is cited 9013 times in total alone which indicate that the authors are reliable sources in the community.

The main sources concerning the modifiability value are:

- *"SQMMA: Software Quality Model for Maintainability Analysis"* from 2015 by Chawla and Chhabra [11] cited by 4 articles.
- *"A Probabilistic Software Quality Model"* from 2011 by Bakota et al. [7] cited by 57 articles.
- *"A cost model based on software maintainability"* from 2012 by Bakota et al. [6] cited by 25 articles.

The articles above are not especially well cited but a reason could be that they are fairly new. One thing which increases the reliability is that they are all based on a ISO/IEC standard and the organization ISO/IEC could be seen as reliable.

[1]https://www.dmst.aueb.gr/dds/sw/ckjm/
[2]http://metrics.sourceforge.net/

5.1.3 Validity and Reliability

Here is a list of threats to validity followed by a text of what have been done to mitigate the threats.

- One threat to validity is that the metrics used does not depend as much on modifiability as it depends on the specific language and framework used. Values on metrics like Number of children (NSC) and Depth of inheritance tree (DIT) have been seen to depend on the language. This might indicate that the language have generally better modifiability but it could also be that the languages are not comparable.
- Modifiability is in the end a metric which depend on the developers that are going to modify the codebase. A codebase that is easy to modify for one developer who knows certain concepts might be hard for another. Skill of the developers might also impact how easy it is to modify the codebase. Experience in the language used could also impact the modifiability and make it differ between developers. The modifiability value of the codebase is therefore probably ambiguous.
- Another threat to validity is the choosing of files that belong to a specific component. Because what files included in a module is not unambiguous. Some files include a lot of things related to other functionality and some files with important code for a module might have been missed. This might have great impact on the result.

According to Runeson and Höst one way to improve validity is to use triangulation, which is a way of taking different angles on the problem and by that getting a broader picture [55]. In this study two view points have been taken from the SQMMA model where one is where the values for a component is summed together and the other is where the average value of the files have been taken. A third view point that have been used is the data for platform stability. Another suggested way to improve the validity is to maintain a case study protocol and the closest to a protocol done in this work is this master thesis. A third suggested way to improve validity is to have peer researchers review protocols, design, collected data, etc. For this master thesis the supervisor and examiner have reviewed the report and collected data during the process. Also two opponents have reviewed the report and given feedback.

The lack of development on the React Native codebase together with the lack of functionality implemented in the React Native codebase is a threat to reliability. Also the lack of modules implemented and measured on is a threat to reliability.

The metrics might not affect the codebase in a linear way as SQMMA suggests. One example is comments in percent which increases the modifiability when it is a high percentage. As an extreme example, if you have a lot of nonsense text with little code the value will be good but will it be easier to modify?

5.2 Results

In this section discussion and analysis of the result is presented.

5.2.1 Releases

When looking at the stability of the different platforms, shown in the figures and tables in Section 4.1, it can be seen that there are fewer number of breaking changes for React Native than for the two other individually. The number of breaking changes are obviously also fewer than when the two native one are combined. This indicates that React Native is a more stable platform to work on in the long run. Though React Native have monthly releases, not a lot is changed in between the versions compared to when the Android or iOS platform have new releases, especially a new version.

A breaking change in React Native is similar to a removal in Android or iOS. In both cases previous functionality cease to work, though a breaking change does not have to remove previously deprecated parts of the code like in Android and iOS. A breaking change can break the behavior of the application from one release to the next without warning. This can make updating inconvenient but all breaking changes are gathered in one place with detailed instructions on how the to fix them. Apart from removals, iOS also uses modifications which changes the call to the API. Those calls also has to be changed in the code.

In the end one can look at how the different platforms/frameworks/APIs changes on a day to day basis, where React Native is the one that have the least amount of changes. React Native seem to be the more stable choice, but one does not necessarily have to be using all of the different parts of the Android or iOS API either. All the changes that happens to one platform might have no impact at all on an application. But looking at the platform/frameworks/APIs as a whole, React Native is the more stable choice according the these measurements.

5.2.2 Modifiability

When looking at the modifiability graphs in Chapter 4.2 it is pretty clear that the native applications modifiability values are worse than the modifiability values for the React Native applications. Though the modifiability is worse for the native code, there are similarities between the graphs. Initially both graphs decline rapidly and when looking at the native graph it stabilizes after a little while. This is similar to the React Native graph, which also stabilizes. It is possible, and most likely, that the React Native graph would continue to decline if development would have proceeded. However the decline would not continue in the same rate as it did initially.

When looking at the modifiability value for the navbar component in Figure 4.4a and 4.4b the modifiability value for the React Native code is closer to the native code than it is for any other component. This is probably because both Android and iOS used already defined components and had very little user written code. This was however not possible for the React Native version, where most of the code was user written.

But at the end of development, the React Native modifiability value was better than the native modifiability value. This is an indication that the React Native code is easier to modify, although there are some other factors which might have made it that way. Some of those factors are described in the subsections below.

The results are statistically inconclusive and only gives indications on how a React Native codebase with the same functionality would compare to the native ones. But it has been shown that modifiability over time for a codebase might be described as a reversed logarithmic function where it decreases rapidly in the beginning and then stabilizes.

JSX and Composition

One thing that might increase the modifiability value for React Native is the style that is used with React Native which is called JSX. It enforces the use of composition in favor of inheritance. A probable reason is that the metrics used does not cover that kind of programming style. The use of inheritance is more common in Java, the language for which the original static analyzers were created. Another probable reason is that it is a better programming style if the goal is to get programs with good modifiability.

Application Size

The React Native codebase was only developed during a short time compared to the native codebases. Because of this the total size of the React Native codebase is much smaller, where size is number of lines of code. It has been shown in previous studies that the size of the codebase is a large factor of maintainability and modifiability. Though the total size of the code is not being used in the modifiability calculations, one can see that modifiability declines more rapidly in the beginning when the size of the codebase increases more rapidly. Because of the limited time spent on developing the React Native applications it is a fact that not all the functionality has been migrated and therefore the React Native applications are not be as big as they would have been. Since the applications were developed under a short period of time, time spent on refactoring that could make the codebase smaller wasn't possible either.

Metric Value

Three different ways the metric values have been used have been mentioned in the report. The sum, the average and the normalized values.

Sum gives the total modifiability value for the entire application. Just like for cyclomatic complexity, this might be a bit misleading. Having a modular application with a lot of files that all have a single purpose and function might yield a higher value than one large complex file.

Average gives the modifiability value for the average file in the application. This might indicate that an application is harder or easier to modify than it really is. It can also indicate if an application is modular or not.

Normalized gives the relative change of the modifiability value according to itself. In order to use normalized values one should not start with a completely new codebase, but this is better to apply on larger codebases. Since a new codebase is usually small, its modifiability value will be fairly close to 0. When more code is added and the modifiability value increase the normalized value will increase a lot. If only comparing two codebases with their normalized values it might indicate that one codebase is easier to modify than another while this would not be the case.

5.3 The Work in a Wider Context

The work produced in this thesis has no direct impact on any people. The result on the other hand might have some impact on other people. If the result shows a great improvement of modifiability for React Native it might increase the usage of React Native and make it go toward being the default way of writing mobile apps. That might force the pure Android or iOS developers to learn new techniques in order to still be attractive on the market. On the other hand if it shows no improvement in modifiability for switching to React Native it might go the other way.

Because this thesis only gives indications that React Native would have better modifiability and that there are many other factors incorporated in the choice of platform the actual results will probably not have a large impact.

The static analyzers produced for this thesis might give value to people that would want to examine similar problems in the future. Comparing cross-platform tools to native with respect to code metrics have not been done much in the past and this thesis might give an opportunity for similar work in the future which makes this thesis mostly give scientific value in that area. For someone that want to study properties of a codebase this thesis might provide some value as a starting point for finding different methods and learn from experience what works and what does not.

6 | Conclusion

The purpose of this thesis has been to investigate if modifiability measurements would be improved with React Native compared to native codebases. For this purpose, we had to answer and understand the impact of our research questions:

- Is modifiability a quality attribute that can be calculated to assist in the choice of two implementation solutions for mobile applications?
- How can information about releases of the platforms be quantified to understand the possible impact on modifiability?

The React Native platform have shown to be more stable than both iOS and Android, when looking at the overall changes to the platform. This means that we can go from native to React Native without having to worry about React Native overhead.

In the graphs, in Section 4.2, the modifiability for React Native is better but it is hard to draw any real conclusions about how that would be in the future because the sample is too small.

When measuring code metrics on applications over time it is better to do this on a large codebase that has been developed for some time. The reason that is when taking the measurements on a small codebase the values have a tendency to be unstable. When just adding or removing one function in a class or an object one can see a distinct change in the metrics, but it does not have to have that large of an impact on the actual modifiability. With a more mature codebase it is easier to see the evolution of the modifiability value.

In order to get a good modifiability value both the modifiability value for the sum of the metrics and the average value of the metrics should be used. Using both aggregations give the best indication of how modifiable the code is since it both gives the overall modifiability of the code and for every file or component.

The consequences of these conclusions are that when making a similar comparison of modifiability. The need to have larger codebases that have been developed over a longer period of time is vital. Having two almost identical projects developed during a long period of time will not be an easy task. Therefore comparing similar applications that are not identical in functionality might be a better way to go.

6.1 Future Work

If the React Native codebase was public a natural future work would be to extend the codebase to include the same functionality as the native ones. Then use the same method to obtain the same metrics. If the thesis work is made at Valtech with access to the code this would be possible and really interesting future work. As mentioned in Section 2.2.2 it would also be interesting to use the same method for some other CPT listed in Section 2.2.2.

The static analyzers produced for this thesis could be used in future work to calculate the metric values for codebases in Java, Objective-C and JavaScript/JSX. Some modifications will be necessary to match some parameters to the current project.

Additional work could be done to evaluate how well certain metrics work with certain languages. A language which relies heavily on principles like polymorphism have appropriate weights assigned to the corresponding metric.

Bibliography

[1] Fernando B Abreu. "Design metrics for OO software system". In: *ECOOP '95, Quantitative Methods Workshop*. 1995.

[2] *Adobe AIR*. URL: `http://www.adobe.com/se/products/air.html` (visited on 11/15/2016).

[3] *Android version history - Wikipedia*. URL: `https://en.wikipedia.org/w/index.php?title=Android_version_history&oldid=776008444` (visited on 03/23/2017).

[4] Oscar Axelsson and Fredrik Carlström. "Evaluation Targeting React Native in Comparison to Native Mobile Development". MA thesis. Division of Certec | Department of Design Sciences Faculty of Engineering LTH | Lund University, 2016.

[5] *Babel transforms your JavaScript*. URL: `https://babeljs.io/` (visited on 11/16/2016).

[6] Tibor Bakota et al. "A cost model based on software maintainability". In: *Software Maintenance (ICSM), 2012 28th IEEE International Conference on* (2012), pp. 316–325. ISSN: 1063-6773. DOI: `10.1109/ICSM.2012.6405288`.

[7] Tibor Bakota et al. "A Probabilistic Software Quality Model". In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on* (2011), pp. 243–252. ISSN: 1063-6773. DOI: `10.1109/ICSM.2011.6080791`.

[8] J. Bansiya and C. G. Davis. "A hierarchical model for object-oriented design quality assessment". In: *IEEE Transactions on Software Engineering* 28.1 (Jan. 2002), pp. 4–17. ISSN: 0098-5589. DOI: `10.1109/32.979986`.

[9] Al Bessey et al. "A few billion lines of code later". In: *Communications of the ACM* 53.2 (2010), pp. 66–75. ISSN: 00010782. DOI: `10.1145/1646353.1646374`.

[10] Barry Boehm et al. "Cost models for future software life cycle processes: COCOMO 2.0". In: *Annals of Software Engineering* 1.1 (1995), pp. 57–94. ISSN: 15737489. DOI: `10.1007/BF02249046`.

[11] Mandeep K. Chawla and Indu Chhabra. "SQMMA: Software Quality Model for Maintainability Analysis". In: *Proceedings of the 8th Annual ACM India Conference*. Compute '15. Ghaziabad, India: ACM, 2015, pp. 9–17. ISBN: 978-1-4503-3650-5. DOI: `10.1145/2835043.2835062`. URL: `http://doi.acm.org.e.bibl.liu.se/10.1145/2835043.2835062`.

[12] Shyam R. Chidamber and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. ISSN: 00985589. DOI: `10.1109/32.295895`. arXiv: `1011.1669`.

[13] Don Coleman et al. "Using metrics to evaluate software system maintainability". In: *Computer* 27.8 (1994), pp. 44–49. ISSN: 0018-9162. DOI: `10.1109/2.303623`. URL: `http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=303623`.

[14] *Common Language Infrastructure*. URL: `http://www.ecma-international.org/publications/standards/Ecma-335.htm` (visited on 11/15/2016).

[15] *comScore Reports June 2015 U.S. Smartphone Subscriber Market Share*. URL: https://www.comscore.com/Insights/Rankings/comScore-Reports-June-2015-US-Smartphone-Subscriber-Market-Share (visited on 11/17/2016).

[16] Standards Coordinating. *IEEE Standard Glossary of Software Engineering Terminology*. Vol. 121990. 1990. ISBN: 155937067X.

[17] *Cross-platform development definition*. URL: https://www.techopedia.com/definition/30026/cross-platform-development (visited on 04/19/2017).

[18] Isabelle Dalmasso et al. "Survey, comparison and evaluation of cross platform mobile application development tools". In: *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)* (2013), pp. 323–328. DOI: 10.1109/IWCMC.2013.6583580. URL: http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=6583580%7B%5C%%7D5Cnhttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6583580.

[19] William Danielsson. "React Native application development". MA thesis. Linköping University, Department of Computer and Information Science, Human-Centered systems., 2016. URL: http://liu.diva-portal.org/smash/get/diva2:998793/FULLTEXT02.pdf.

[20] Udaya Dayanandan. "An Empirical Evaluation model for Software Architecture Maintainability for Object oriented Design". In: *Proceedings of the International Conference on Informatics and Analytics (ICIA-16)* (2016), pp. 1–4. DOI: 10.1145/2980258.2980459.

[21] Florian Deissenboeck et al. "Software quality models: Purposes, usage scenarios and requirements". In: *Proceedings - International Conference on Software Engineering* (2009), pp. 9–14. ISSN: 02705257. DOI: 10.1109/WOSQ.2009.5071551.

[22] *Deprecated - Android developer*. URL: https://developer.android.com/reference/java/lang/Deprecated.html (visited on 03/27/2017).

[23] *Draft: JSX Specification*. URL: https://facebook.github.io/jsx/ (visited on 11/16/2016).

[24] Sanjay Kumar Dubey and Ajay Rana. "Assessment of maintainability metrics for object-oriented software system". In: *ACM SIGSOFT Software Engineering Notes* 36.5 (2011), pp. 1–7. ISSN: 01635948. DOI: 10.1145/2020976.2020983.

[25] Khaled El Emam et al. "The prediction of faulty classes using object-oriented design metrics". In: *Journal of Systems and Software* 56 (2001), pp. 63–75. ISSN: 01641212. DOI: 10.1016/S0164-1212(00)00086-8.

[26] Artemij Fedosejev. *React.js Essentials*. Packt Publishing Ltd., 2015. ISBN: 9781783551620.

[27] *Gartner Says Smartphone Sales Surpassed One Billion Units in 2014*. URL: http://www.gartner.com/newsroom/id/2996817 (visited on 11/17/2016).

[28] Brij Mohan Goel and Pradeep Kumar Bhatia. "Investigating of high and low impact faults in object-oriented projects". In: *ACM SIGSOFT Software Engineering Notes* 38.6 (2013), pp. 1–6. ISSN: 01635948. DOI: 10.1145/2532780.2532807. URL: http://dl.acm.org/citation.cfm?doid=2532780.2532807.

[29] *Halstead complexity measures - Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=Halstead_complexity_measures&oldid=746488955 (visited on 01/12/2017).

[30] Niclas Hansson and Tomas Vidhall. "Effects on performance and usability for cross-platform application development using React Native". MA thesis. Linköping University, Department of Computer and Information Science, Human-Centered systems., 2016. URL: http://liu.diva-portal.org/smash/get/diva2:946127/FULLTEXT01.pdf.

[31] Henning Heitkötter et al. "Comparing Cross-Platform Development Approaches for Mobile Applications". In: *Proc. 8th International Conference on Web Information Systems and Technologies (WEBIST)* 140 (2012), pp. 299–311. ISSN: 18651348. DOI: 10.1007/978-3-642-36608-6.

[32] *International Electrotechnical Commission*. URL: http://www.iec.ch/ (visited on 04/18/2016).

[33] *International Organization for Standardization*. URL: https://www.iso.org/ (visited on 04/18/2016).

[34] *International Telecommunication Union*. URL: http://www.itu.int/ (visited on 04/18/2016).

[35] *iOS version history - Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=IOS_version_history&oldid=775891236 (visited on 03/17/2017).

[36] *ISO/IEC 25010:2011*. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en (visited on 12/21/2016).

[37] *ISO/IEC JTC 1*. URL: https://www.iso.org/isoiec-jtc-1.html (visited on 04/18/2017).

[38] *ISO/IEC/IEEE 24765:2010*. URL: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-1:v1:en (visited on 01/12/2017).

[39] N Jovanovic et al. "Pixy: a static analysis tool for detecting Web application vulnerabilities". In: *Security and Privacy, 2006 IEEE Symposium on* (2006), 6 pp.–263. ISSN: 1081-6011. DOI: 10.1109/SP.2006.29.

[40] Heinz K. Klein and Michael D. Myers. "A Set of Principles for Conducting and Evaluating Interpretive Field Studies in Information Systems". In: *MIS Quarterly* 23.1 (1999), p. 67. DOI: 10.2307/249410. URL: http://dx.doi.org/10.2307/249410.

[41] Benjamin Livshits. "Improving software security with precise static and runtime analysis". PhD thesis. Stanford Univ., 2006, pp. 1–250. ISBN: 9780542984044. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.77.7196%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf.

[42] Ivano Malavolta. "Beyond Native Apps: Web Technologies to the Rescue! (Keynote)". In: *Proceedings of the 1st International Workshop on Mobile Development*. Mobile! 2016. Amsterdam, Netherlands: ACM, 2016, pp. 1–2. ISBN: 978-1-4503-4643-6. DOI: 10.1145/3001854.3001863. URL: http://doi.acm.org.e.bibl.liu.se/10.1145/3001854.3001863.

[43] T J McCabe. "A Complexity Measure". In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233837.

[44] *Mobile: Native Apps, Web Apps, and Hybrid Apps*. URL: https://www.nngroup.com/articles/mobile-native-apps/ (visited on 04/18/2017).

[45] *Mono*. URL: http://www.mono-project.com/docs/about-mono/ (visited on 11/15/2016).

[46] *NativeScript - Wikipedia*. URL: https://en.wikipedia.org/w/index.php?title=NativeScript&oldid=750775394 (visited on 11/23/2016).

[47] *NeoMAD*. URL: http://neomades.com/en/ (visited on 11/16/2016).

[48] Paul Oman and Jack Hagemeister. "Metrics for assessing a software system's maintainability". In: *Proceedings Conference on Software Maintenance 1992* (1992), pp. 337–344. ISSN: 1096908X. DOI: 10.1109/ICSM.1992.242525. URL: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=242525.

[49] Wang Ping. "A brief history of standards and standardization organizations: a Chinese perspective". In: *Eas-West Center Working Papers - Economic Series* 117 (2011), p. 25. URL: http://www.eastwestcenter.org/fileadmin/stored/pdfs/econwp117.pdf%7B%5C%%7D5Cnhttp://hdl.handle.net/10125/21412.

[50] *React Native documentation v0.42*. URL: http://facebook.github.io/react-native/releases/0.42/docs (visited on 03/22/2017).

[51] *React Native Github*. URL: https://github.com/facebook/react-native (visited on 11/15/2016).

[52] *React Native, Showcase*. URL: https://facebook.github.io/react-native/showcase.html (visited on 01/23/2017).

[53] *React Webpage*. URL: https://facebook.github.io/react/ (visited on 01/23/2017).

[54] Mehwish Riaz et al. "A Systematic Review of Software Maintainability Prediction and Metrics". In: (2009), pp. 367–377.

[55] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical Software Engineering* 14.2 (2009), pp. 131–164. ISSN: 13823256. DOI: 10.1007/s10664-008-9102-8.

[56] Ioannis Samoladas and Georgios Gousios. "The SQO-OSS quality model: measurement based open source software evaluation". In: *Open source development ...* 275 (2008), pp. 237–248. DOI: 10.1007/978-0-387-09684-1_19. URL: http://link.springer.com/chapter/10.1007/978-0-387-09684-1%7B%5C_%7D19.

[57] Statista. *Number of smartphone users worldwide*. 2016. URL: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (visited on 11/14/2016).

[58] *Titanium*. URL: http://www.appcelerator.org/ (visited on 11/16/2016).

[59] V T K Tran et al. "Size Estimation of Cloud Migration Projects with Cloud Migration Point (CMP)". In: *2011 International Symposium on Empirical Software Engineering and Measurement* (2011), pp. 265–274. ISSN: 1938-6451. DOI: 10.1109/ESEM.2011.35.

[60] *Tutorial*. URL: http://facebook.github.io/react-native/releases/0.37/docs/tutorial.html (visited on 11/16/2016).

[61] David Wagner and R. Dean. "Intrusion detection via static analysis". In: *Security and Privacy, 2001. S&P 2001. ...* (2001), pp. 156–168. ISSN: 1081-6011. DOI: 10.1109/SECPRI.2001.924296. URL: http://ieeexplore.ieee.org/xpls/abs%7B%5C_%7Dall.jsp?arnumber=924296%7B%5C%%7D5Cnhttp://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=924296.

[62] *What is the Document Object Model?* URL: https://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/introduction.html (visited on 11/16/2016).

[63] Michiel Willocx et al. "Comparing Performance Parameters of Mobile App Development Strategies". In: *Proceedings of the International Workshop on Mobile Software Engineering and Systems* (2016), pp. 38–47. DOI: 10.1145/2897073.2897092. URL: http://doi.acm.org/10.1145/2897073.2897092.

[64] *World Standards Cooperation*. URL: http://www.worldstandardscooperation.org (visited on 04/18/2017).

[65] *Xamarin*. URL: https://www.xamarin.com/ (visited on 11/15/2016).