University of Amsterdam

Faculty of Science

A metrics-based comparison of secondary user quality between iOS and Android

Master Thesis

Software Engineering

Author:
Tobias Ammann
Student number 10460411
Bilderdijkkade 42h
1053VE Amsterdam
Tel. 0639212394
tj.ammann@me.com

August 11, 2014

Abstract

Native mobile applications gain popularity in the commercial market. There is no other economical sector that grows as fast. A lot of economical research is done in this sector, but there is very little research that deals with qualities for mobile application developers. This paper compares the qualities of the iOS and Android platforms, where developers have to deal with. The base of the research form 45 iOS and 35 Android apps that are developed since 2009 in a Dutch mobile service agency. With the help of the factor-criteria-metric model one project metric, three OO-metrics and two method metrics are defined to analyze the apps. Except the project metric, in all test are statistical significant results found, but with a practical view, only OO-metrics show big differences. These results show that the iOS framework acts more like a white-box framework compared to Android, that acts more like a black-box framework. The calculation to the sub factors show that iOS scores better with adaptability and modifiability. In modularity and testability is almost no difference found. Understandability and self-descriptiveness are better on Android. The geometric mean of all sub-qualities results in no difference.

Contents

1 Introduction 9

1.1 Mobile Software Development . . . 10

1.1.1 iOS development . . . 10

1.1.2 Android development . . . 10

2 Theory 12

2.1 Software Quality . . . 12

2.2 Secondary User Quality . . . 12

2.3 Metric description . . . 15

2.3.1 Source Lines of Code(SLOC) . . . 15

2.3.2 Cyclomatic Complexity(CC) . . . 16

2.3.3 Information Flow(IF) . . . 16

2.3.4 Depth of Inheritance Tree (DIT) . . . 17

2.3.5 Coupling Between Object Classes(CBO) . . . 17

2.3.6 Response For a Class(RFC) . . . 18

2.4 Statistical testing . . . 18

2.4.1 t-Statistics . . . 18

2.4.2 Normality testing . . . 19

2.4.3 Data transformation . . . 19

2.4.4 Non-parametric statistics . . . 19

2.4.5 t-Test vs Mann-Whitney test . . . 20

2.4.6 Calculation of qualities . . . 20

3 Metric Tool 21

3.1 Rascal Metaprogramming Language . . . 21

3.2 Clang LibTooling . . . 21

3.2.1 Xcodebuild . . . 22

3.3 Metric Implementation . . . 22

3.3.1 Source Lines of Code(SLOC) . . . 22

3.3.2 Cyclomatic Complexity (CC) . . . 23

3.3.3 Information Flow (IF) . . . 23

3.3.4 Depth of Inheritance Tree (DIT) . . . 25

3.3.5 Coupling Between Objects (CBO) . . . 25

3.3.6 Response for Class (RFC) . . . 26

4 Results 28

4.1 Source Lines of Code . . . 28

4.2 Source Lines of Code in a File . . . 28

4.3 Cyclomatic Complexity . . . 29

4.4 Information Flow . . . 30

4.4.1 Method Length . . . 30

4.4.2 Fan . . . 31
4.4.3 Information Flow . . . 32
4.5 Depth of Inheritance Tree . . . 32
4.6 Coupling Between Objects . . . 34
4.7 Response For Class . . . 34
4.8 Closing the circle . . . 35

5 Discussion, Threats and Future work 37
5.1 Discussion . . . 37
5.2 Threats of validity . . . 38
5.2.1 Construct validity . . . 38
5.2.2 Internal validity . . . 38
5.2.3 External validity . . . 39
5.3 Future Work . . . 39

6 Conclusion 40

7 Bibliography 41

A Summary of Metric data 44

B Metric Descriptives 50

List of Figures

1.1 Abstraction layers of the iOS SDK 10
1.2 Abstraction layers of the Android SDK 10

2.1 ISO 25010 Software Quality model 12
2.2 Cyclomatic Complexity example 16

3.1 Graphical DIT Algorithm for Android Apps 25

4.1 Distribution Source Lines of Code 29
4.2 Distribution Source Lines of Code in Files 29
4.3 Distribution Cyclomatic Complexity 30
4.4 Distribution Method Length 31
4.5 Distribution Fan 32
4.6 Distribution Depth of Inheritance Tree 33
4.7 Depth of inheritance from SDK in Schiphol app 33
4.8 Distribution Coupling between Objects 34
4.9 Distribution Response for Class 35

List of Tables

2.1 Identified sub-criteria's of secondary user quality in literature 14
2.2 Curvilinear Relationship Between Defect Rate and Module Size 15
2.3 Mapping of identified criteria to metrics 18
2.4 Guidelines for data transformation in a statistical dataset 19

A.1 Mean and Median vlaues with all Metrics for all Projects 49

B.1 Calculated statistical values for the Source Lines of Code in a project 51
B.2 Calculated statistical values for the Cyclomatic Complexity 52
B.3 Calculated statistical values for the Method Length for the Information Flow . . 53
B.4 Calculated statistical values for the Fan for the Information Flow 54
B.5 Calculated statistical values for the Information Flow 55
B.6 Calculated statistical values for the Depth of Inheritance Tree 56
B.7 Calculated statistical values for coupling between objects 57
B.8 Calculated statistical values for the Response of a Class 58
B.9 Calculated statistical values for the Source Lines of Code in a file 59

Chapter 1

Introduction

The incredible rise of smart-phones and tablets is perhaps the biggest economic phenomenon today. Everything started in the 70's, when telephony and computing were conceptualized. In 1997, Ericsson used for the first time the name "Smart-phone to describe a telephone. With the come of the smart phone, the mobile operating system was born. It is a domain specific operating system that is tailored for minimalistic hardware resources and power optimization. There were a lot of contenders that tried to establish a mobile operating system in the market. The first company who succeeded was Apple with iOS in 2007. In 2008 also Google was successful with the release of Android 1.0.

According to Moore's law [1], processors are still getting more powerful. With the rise of computing power, also new features and functionality came into the operating system. Today's mobile operating systems are almost able to perform the same tasks as desktop operating systems.

The success of iOS and Android is based on the software development kit(SDK) that enables developers to develop apps with effective use of the mobile phone's hardware.

The demand for apps is still growing extraordinary. Not only the economical side is of interest, also the demand for reliable tools for developers is growing. One example is the fact that on stackoverflow.com[2], questions with tags "Android" or "iOS" can be found at the top of the most popular tags. Until now, there is very little research found that identifies SDK qualities of Android or iOS. This paper tries to identify qualities from the iOS platform compared to those on Android. Therefore, the following research question is defined:

Which platform offers developers more value to develop high quality source code?

The research is performed in corporation with M2Mobi, a Dutch mobile service agency. In total, 80 apps are analyzed, 45 iOS and 35 Android. More than half a million lines of code are analyzed and 41'927 cases are created for statistical analysis. In table A is a summary of all metric data from the apps.

The first part of the thesis contains an introduction to iOS and Android development. The following section covers the theoretic explanation of software quality and metrics. In the third part, the results for every metric is presented. At the end, the results are mapped back to the sub-qualities with a discussion and conclusion.

[1] Moore's law is the observation that, over the history of computing hardware, the number of transistors on integrated circuits doubles approximately every two years.

[2] Popular website to ask and answer software development related questions

1.1 Mobile Software Development

1.1.1 iOS development

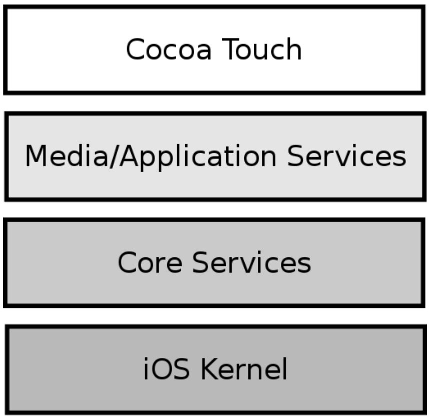


Figure 1.1: Abstraction layers of the iOS SDK

iOS app development is strictly guided by Apple. The iOS framework is based on Objective-C and uses the LLVM compiler. This is an open source project with an *University of Illinois* license and allows Apple to integrate it into their Xcode integrated development environment(IDE) without sharing modifications. Objective-C is an extension to the C programming language and offers users the object oriented(OO) paradigm with Smalltalk-style messaging. Despite Apple is the only main provider of Objective-C, it is the third most popular programming language at the moment [49]. Besides Objective-C, the LLVM compiler enables users to write source code in C, C++ and Objective-C++.
The iOS operating system is based on Apples desktop operating system OS-X, which is based on the Unix operating system. The iOS framework make use of the Model-View-Controller pattern and consists of four layers. On the bottom of the system is the kernel, which is responsible for the file system, hardware drivers and power management. A level higher, the core service layer provides networking, threads and core location. The media and application layer provides all support to run an app and process media. The resulting application is build on the Cocoa touch layer, which acts as interface between the lower layers and the app developer. Development with the iOS SDK is only possible on OS-X and there is no official support to an other IDE than Xcode. Apple included a graphical user interface(UI) builder that allows developers to drag and drop UI elements into a screen without writing any line of code. Before an app goes into the App Store, the app is checked by Apple.

1.1.2 Android development

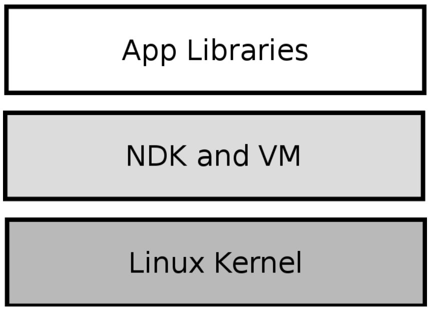


Figure 1.2: Abstraction layers of the Android SDK

The Android operating system is a multi-user Linux system, where each app acts as a different user. The framework consists of three layers as shown in Figure 1.2. On the foundation of the framework acts a tailored Linux Kernel with power savings extensions. The middle layer includes the Native Development Kit(NDK) that is written in C and C++ and the Dalvik Virtual Machine, which is used to translate the Java byte code. Android as well as Dalvik are open source projects. All the standard Android APIs that are used to create apps are defined in terms of Dalvik classes[7]. It is possible to access the NDK from apps, but the advice is to do that only in very specific situations. In the top layer are all application programming interfaces(API) and application support. Apps use the Java syntax and semantics. Android is designed to run on many different types of devices, from phones to tablets to televisions. There are limitations, a device is ”Android compatible” only if it can correctly run apps written for the Android execution environment and each device must pass the Compatibility Test Suite (CTS)[2]. If the app fulfils these requirements, it directly can go into the Play Store [3]. Android development is possible on Linux distributions, Windows and Mac OS-X. The official

[3] Play Store is the official store from Google to purchase and download apps

supported IDE is Eclipse, but it is also possible to use IntelliJ IDEA and NetBeans[4] with the Android development tools plugin. Android has a graphical user interface builder. The created UI is translated into a XML file, that is linked in the virtual machine to the source code.

[4] A new Android development environment called Android Studio, based on IntelliJ IDEA, is now available as an early access preview [2].

Chapter 2

Theory

2.1 Software Quality

A quality is the standard of something as measured against other things of a similar kind; the degree of excellence of something. Unfortunately is this description ambiguous and used through professionals as popular with their own interpretation. This makes quality a commonly misunderstood term[31]. To prevent such misunderstandings, it is important to have a clear definition.

To measure the quality of the source code that is build with a specific SDK, the ISO 25010:2011 system and software quality standard is taken as a starting point. This differentiate between quality in use and product quality. Quality in use can be described as user satisfaction, freedom from risk or context coverage and is only indirect related to source code. Because the focus lies on product quality, the quality in use is neglected.

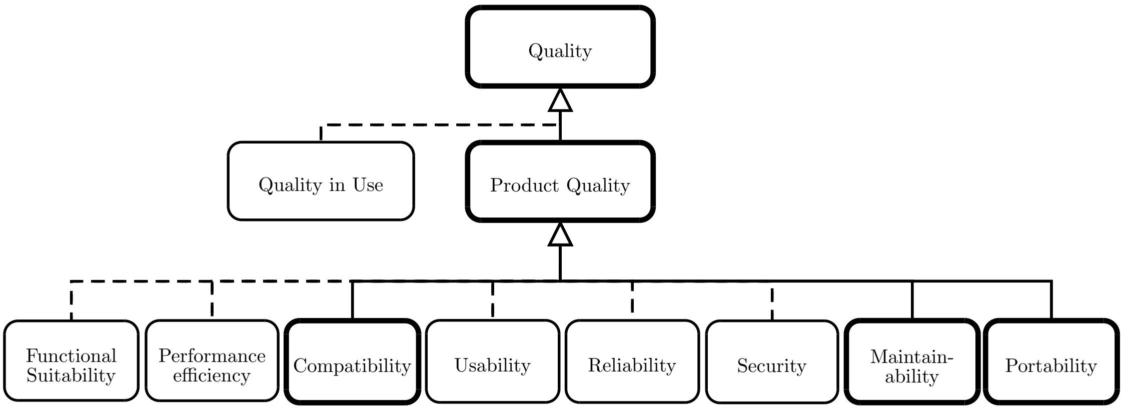


Figure 2.1: ISO/IEC 25010 software product quality model. The thicker bordered cells are critical to secondary product quality. Boxes with stippled lines are essential for quality in use

2.2 Secondary User Quality

Figure 2.1 contains the quality model as defined in the ISO 25010. Compatibility, maintainability and portability have a significant influence on quality in use for secondary users who maintain the system[28], e.g. the user that has to deal with the source code. The three characteristics are described as:

Compatibility

Degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment[28]

Maintainability

Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers[28]

Portability

Degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another[28]

The definition of maintainability, as understood in the context of software systems, is in conformance to the definition provided by the IEEE[43]. Unfortunately are these characteristics too abstract to find a metric that can measure them directly. Research has shown that there isn't any well-known source code metric that is able to predict the subjective maintainability opinions of experts [25]. To get more accurate measurements, the characteristics are divided into sub-characteristics according to the Factor-Criteria-Metric model that was first described by McCall, Richards, and Walters[38]. The establishment of criteria for each factor has several benefits: First, the factor get more specific. Second, if the criteria affects more than one factor, relationships between them get visible. Third, a one to one relationship between metrics and criteria can be established.

The ISO 25010 made a decomposition from factors to criteria. To get a more complete overview, other academic articles are taken and searched for sub-characteristics. The result is presented in Table 2.1. Horizontally are all identified factors and vertically the different articles. The criteria which are identified most, are dark gray shaded. Only researches that identify criteria for compatibility, maintainability or portability are listed. To prevent redundancy, articles that inherit their criteria from other papers, are not included. E.g. the paper "A practical model for measuring maintainability"[26], which gains popularity under researchers as well as practical experts, inherits the sub-characteristics from the ISO 9126[29] and is therefore not included in the table. Beside that, only researches that have a certain reputation are considered. The reputation is measured with the number of times an article is cited[1].

In Table 2.1 are two ISO standards included. The ISO 9126 as well as the ISO 25010 identify software engineering product quality. The ISO 9126 standard is replaced in 2011 by the ISO 25010. There are still a lot of researches that use the ISO 9126 criteria. Table 2.1 shows that four criteria have been changed and seven are still the same between these two. From this seven, three criteria are taken into the further research. These are also the most identified criteria. A look at the other most identified factors show that *self descriptiveness* is identified by papers that are published before 1995 and is neither identified by the ISO 9126 nor by the ISO 25010. The other factor that is not identified by the ISO, is *understandability. Modularity* is identified by McCall, Richards, and Walters in 1977 as important factor. This is one of the factors that is not in the ISO 9126 included, but is added in the ISO 25010. An other interesting fact is that probably every person who ever wrote a piece of code can give an interpretation of the most identified criteria. Criteria that are only identified once, tent to be very complex. E.g. most programmers would hesitate to give a definition of *Ease of impact analysis.*

In the following description, a definition of the most identified criteria in Table 2.1 is presented

Adaptability

Degree to which a product or system can effectively and efficiently be adapted for different or evolving hardware, software or other operational or usage environments[28]. In mobile development, adaptability is the measurement of how easy it is to port existing apps into a newer version OS version or new phones or tablets with different hardware.

Modularity

Degree to which a system or computer program is composed of discrete components such

[1] The number of citations is taken from `http://scholar.google.com/`

Secondary user Quality	ISO[28]	Hashim and Key[24]	Peercy [41]	Aggarwal et al. [1]	Coleman et al.[13]	Welker, Oman, and Atkinson[51]	Schneberger[46]	Genero et al.[22]	Broy, Deissenboeck, and Pizka[9]	McCall, Richards, and Walters[38]	Boehm et al.[5]	Sneed and Mérey[47]	Yau and Collofello[53]	Karlsson[32]	Ghezzi, Jazayeri, and Mandrioli[23]	Dromey[17]	Briand et al.[8]	Mari and Eila[35]	Genero et al.[21]	Jung, Kim, and Chung[30]	Rizvi and Khan[44]	ISO/IEC et al.[29]	Total
Adaptability	x			x	x	x									x					x		x	7
Installability	x																					x	2
Replaceability	x																					x	2
Modularity	x	x	x							x				x									5
Reusability	x																	x					2
Analysability	x																			x		x	3
Modifiability	x			x	x	x			x		x					x		x	x		x	x	11
Testability	x								x		x		x	x		x	x	x				x	9
Co-existence	x																					x	2
Interoperability	x																						1
Repairability		x																					1
Evolvability		x																					1
Readability		x										x											2
Programming language		x																					1
Standardisation		x																					1
Complexity		x											x										2
Traceability		x																					1
Stability													x							x		x	3
Consistency			x											x									2
Simplicity			x							x				x									3
Expandability			x																				1
Instrumentation			x																				1
Average number of live variables				x																			1
Average live variable span				x																			1
Comments ratio				x																			1
Understandability							x	x	x		x								x		x		6
Conciseness										x													1
Self descriptiveness			x							x			x	x		x							5
Cohesiveness												x											1
Documentation												x	x										2
Extensibility													x										1
Correctability															x								1
Perfectiveness															x								1
Comprehensibility																	x						1
Ease of impact analysis																	x						1
Flexibility																		x					1
Integrability																		x					1
Changeability																				x			1

Table 2.1: Identified sub-criteria's of secondary user quality in literature

that a change to one component has minimal impact on other components[28].

Modifiability
Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality[28]

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[28].

Understandability
Degree of perceive the intended meaning of words, a language or a construct.

Self descriptiveness
Degree of which a method, module, project or system explain itself.

2.3 Metric description

2.3.1 Source Lines of Code(SLOC)

Size is a project and module metric and one of the most important attributes of a software product[40]. Lines of code(LOC) count executable statements. It has his origin from assembler programs where it is used as a count of instructions. In higher level programming languages, there are different counting methods: Physical counting and logical counting. The physical counting method counts the source lines of code(SLOC) in the source file. The logical counting methods tries to calculate the source code file so that, regardless the coding style, the outcome should always be the same. Nguyen et al. identified that, regardless the counting method, there is always a difference in counting between tools[40].
Basili and Perricone and Kan identified a negative relation between the size and defect density[3][31]. Table 2.2 shows the relation between unit size and defects, identified by Compton and Withrow in two Ada projects[16]. Because of this, SLOC is related to Modularity. Heitlager, Kuipers, and Visser states that a higher volume is more difficult to understand[26]. Berkholz researched the expressiveness of programming languages through inspection of the distribution of lines of code per commit every month for around 20 years, weighted by the number of commits in any given month. The results show that the lines of code in one commit in Java are almost twice as high as in Objective-C. The same applies to the data when it is sorted by consistency[4]. Research by Capers shows that Java needs 53 average source statements per function point, Objective-C needs 27[10]. From this data, it can be assumed that iOS apps should have less SLOC.

Maximum SLOC per module	Average Defect per 1k SLOC
63	1,5
100	1.4
158	0.9
251	0.5
398	1.1
630	1.9
1k	1.3
>1k	1.4

Table 2.2: Curvilinear Relationship Between Defect Rate and Module Size in Ada[16]

Hypothesis 1(H1) *Source code written with the Android framework will contain more SLOC than source code written with the iOS framework*

2.3.2 Cyclomatic Complexity(CC)

Cyclomatic complexity measures the control flow in a program, unit or module. The metric is developed in 1976 by McCabe with the question: How to modularize a software system so the resulting modules are both, testable and maintainable. Many experts in software testing recommend use of the cyclomatic representation to ensure adequate test coverage[31]. The definition of cyclomatic complexity $V(G)$ with a graph G, n vertices, e edges and p connected components is

$$v(G) = e - n + p \tag{2.1}$$

It can also be defined as all the unique paths trough a program. For example, Figure 2.2 has a cyclomatic complexity of $V(G) = 2 - 4 + 4 = 2$. It can be described in unique paths: one path is (A-B-D) and the other is (A-C-D). Further mathematical simplification in "A complexity measure" shows that the cyclomatic complexity of a structured program equals the number of predicates plus one [37]. In Figure 2.2, edge A is a predicate, thus

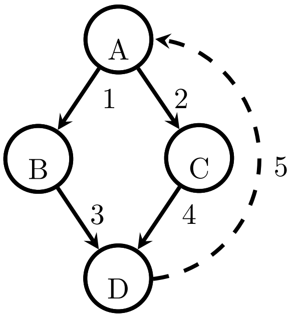


Figure 2.2: Visual program graph with a Cyclomatic Complexity of 2

$$V(G) = \pi + 1 = 1 + 1 = 2 \tag{2.2}$$

Predicate compounds with AND or OR are treated as contributing two to complexity.
In this research, the metric is used as module metric. A module is the smallest testable unit of a program.
A significant influence on cyclomatic complexity could have the handling of adaptability to the different devices and OS versions. There are way more different Android compatible devices than iOS devices. Google is aware of this fact and implemented smart solutions that handle this problem in the background of the framework.
Hypothesis 2(H2) *Source code written with the Android framework has the same cyclomatic complexity as source code written with the iOS framework.*

2.3.3 Information Flow(IF)

The Henry and Kafura metric is a complexity measurement that depends on the information flow into and out of a module.

$$length \cdot (fan_{in} \cdot fan_{out})^2 \tag{2.3}$$

The complexity of a method is defined in Equation 2.3 where the *length* is the amount of lines of code, fan_{in} the local flow into a procedure plus the number of data structures from which information is received and fan_{out} is the number of local flows from a procedure plus the number of data structures which are updated. The measurement shows possible areas where redesign or re implementation is needed and where maintenance (modifiability) of the system might be difficult. A high fan-in and fan-out indicates that this procedure may perform more than one function or there is a missing level of abstraction in the design process (modularity). An implementation difficulty would be indicated by a large procedure i.e. many lines of code(understandability, self descriptiveness). High complexity measures of the metric indicate improper modularization.[27] The information flow is measured with the abstract syntax tree, on Android side parsed with the Rascal m3 engine, on iOS side with the clang/llvm 3.5 engine. Because they address the variables in a different way, the fan-in and fan-out is combined into one variable fan. This variable states the amount of objects that are used (read and write) in a module. The length of a module is measured as is, without any modifications including comments. Finally the information flow is calculated with the formula in Equation 2.4.

$$InformationFlow = length \cdot fan \cdot fan \tag{2.4}$$

This correspondents not exactly with the original definition in 2.3, but since the goal is to identify differences in platforms and not how the complexities are observed, this way should be sufficient. Null values are excluded, since this code would be otiose.
Java and Objective-C have their origin in the C programming language. The difference lies in the implementation of the OO paradigm. Since the information flow metric is a module metric, there should be no difference.
Hypothesis 3(H3) *There is no difference in information flow in source code written with the Android and iOS framework*

2.3.4 Depth of Inheritance Tree (DIT)

Depth of Inheritance Tree is a object oriented metric that measures the maximum length from the node to the root of the tree for a class. The deeper a class is in a hierarchy, the greater the number of methods it is likely to inherit. This makes it more complex to predict its behavior (understandability, self descriptiveness). On the other side, a particular class in the hierarchy has a greater chance that the methods are reused (adaptability)[12].
The DIT is measured through the complete SDK of the platforms. If the class inherits from no other class, the DIT equals 1. For every super class the file has, the value is incremented by one. The OO paradigm of Objective-C is based on Smalltalk. Chidamber and Kemerer calculated higher DIT in Smalltalk than C++ in their research[12]. The implementation of the OO paradigm of C++ is similar to that of Java. A second experiment conducted by Chidamber in 1998 shows comparable results with an Objective-C and C++ system[11]. Therefore, higher DIT values are expected in the iOS framework.
Hypothesis 4(H4) *The DIT for the Android framework is smaller than that for the iOS framework*

2.3.5 Coupling Between Object Classes(CBO)

Coupling Between Object classes count the number of connections to other classes from a particular class. An object is coupled to an other object if one of them acts on the other i.e. methods of one use methods or instances variables of another. Excessive coupling between object classes is tending to cause harm to modular design and prevents reuse (modularity, adaptability). The higher the coupling, the higher the sensitivity to changes in the class and therefore modifying them gets more difficult (modifiability). The higher the coupling between objects are, the more rigorous the testing needs to be (testability).
The CBO in this research is measured by counting the include/import declarations of a class. Includes from higher level classes, files where the class inherits from, are not added. On iOS, header and class files with the same name are pooled.
Research by Chidamber and Kemerer showed that Smalltalk programs have a higher median CBO value than C++ programs. Since Objective-C offers users Smalltalk-style messaging and Java is closely related to C++, a less obvious but similar outcome is expected [4][10]. Another research by Chidamber, Darcy, and Kemerer where an Objective-C and a C++ system are tested, validate the hypothesis [11]. In this, an almost twice as high mean value and a 3.5 times

higher median value is measured.
Hypothesis 5(H5) *Objects in iOS have higher CBO values than objects in Android*

2.3.6 Response For a Class(RFC)

The response set of a class is a set of methods that can potentially be executed in response to a message received by an object of that class. A consequence of a high value is that testing and debugging of the class becomes more complicated since it requires a greater level of understanding of the tester (understandability, testability)[12].
In case of Objective-C, every method declarations in the interface declaration (header file) count as one response for a class. In the Android source files, every public method count as one. The number of RFC in the super classes are added to the resulting RFC. Thus, the RFC is the number of public methods in a specific class plus all the inherited public methods.
The prediction for RFC is the same as for CBO because Chidamber and Kemerer found that median and maximal values are higher in Smalltalk compared to C++. Chidamber, Darcy, and Kemerer researched an Objective-C and a C++ system for managerial use of metrics. The results shows an almost 3 times higher mean and a 4.5 times higher median value in RFC by the Objective-C system.
Hypothesis 6(H6) *Objects in iOS have higher RFC values than objects in Android*

	SLOC	CC	IF	DIT	CBO	RFC
Adaptability				x	x	
Modularity	x		x		x	
Modifiability			x		x	
Testability		x			x	x
Understandability	x		x	x		x
Self descriptiveness			x	x		

Table 2.3: Mapping of identified criteria to metrics

2.4 Statistical testing

2.4.1 t-Statistics

The general situation in this project is that a metric has a population on iOS with mean μ_1 and variance σ_1^2, while the population on Android has mean μ_2 and variance σ_2^2. Inferences will be based on two random samples of size n_1 with cases $X_{11}, X_{12}, \ldots, X_{1n_1}$ and n_2 with $X_{21}, X_{22}, \ldots, X_{2n_2}$, respectively. These applications arise in the context of simple comparative experiments in which the objective is to study the difference in the parameters of the two populations. To calculate such a difference, the following assumptions must be fulfilled[39]:

1. $X_{11}, X_{12}, \ldots, X_{1n_1}$ is a random sample from population 1
2. $X_{21}, X_{22}, \ldots, X_{2n_2}$ is a random sample from population 2
3. The two populations represented by X_1 and X_2 are independent
4. Both populations are normal

Montgomery and Runger claim that moderate departures from normality do not adversely affect the procedure. If the data set conforms to these assumptions, a t-statistic is used to test the hypothesis because of unknown variances. For some cases, the sample size is huge. In these cases it would be sufficient to test the hypothesis with z-statistics since $\lim_{v \to \infty} t = z$ where v are the degrees of freedom. This is not done because tests are performed in a statistical environment with enough processing power.

2.4.2 Normality testing

Test for normality is done with skewness and kurtosis values. West, Finch, and Curran show in "Structural equation models with nonnormal variables: Problems and remedies." that skewness and kurtosis values are related to sample size [52]. Therefore, critical values for rejecting the non-normality need to be different according to the sample size.

Small samples size (n <50) if absolute z-scores for either skewness or kurtosis are larger than the z-value of the desired α level($z_{0.05} = 1.96$), it can be assumed that the distribution of the sample is non-normal.

Medium sample size(50 <n <300) if absolute z-scores for either skewness or kurtosis are larger than the z-value of the desired α level($z_{0.05} = 3.29$), it can be assumed that the distribution of the sample is non-normal.

Big sample size >300 depend on the histograms and the absolute values of skewness and kurtosis without considering z-values. Absolute Skew values larger than 2 or absolute kurtosis(proper) larger than 7 may be used as reference values for determining substantial non-normality

For sample sizes below 300, the limit is calculated with the z-value

$$kurtosis/skewness \leq z_{value} \cdot std.Error \tag{2.5}$$

2.4.3 Data transformation

A general accepted method for non-normal distributions is to transform the data with mathematical calculations. The goal of modifying data is to fit it more closely to the underlying assumption of the statistical test. Table 2.4 presents the guidelines for transforming data advised by Tabachnick, Fidell, et al.[50]. X is the measured data set, Y is the transformed data set and c is a constant value that is greater than or equal to the smallest value in the measured data set $c \geq X_{min}$

If the distribution has:	Equation
Moderately positive skewness	$Y = \sqrt{X}$
Substantially positive skewness	$Y = \log_{10}(X)$
Substantially positive skewness	$Y = \log_{10}(X + 1)$
Moderately negative skewness	$Y = \sqrt{c - X}$
Substantially negative skewness	$Y = \log_{10}(c - X)$

Table 2.4: Guidelines for data transformation in a statistical data set presented by Tabachnick, Fidell, et al.

2.4.4 Non-parametric statistics

In case of two independent continuous populations X_1 and X_2 with means μ_1 and μ_2 and unwillingness to assume that they are (approximately) normal and transformation didn't contribute to a normal distribution, a Mann-Whitney rank sum test can be performed. This test assumes the following:

1. Data points are independent
2. X_1 and X_2 are continuous

This test is sometimes called Wilcoxon Rank-Sum Test.

2.4.5 t-Test vs Mann-Whitney test

If the normality assumption is correct, the Mann-Whitney rank sum test is approximately 95% as efficient as the t-test in large samples. On the other hand, regardless of the form of the distributions, the Mann-Whitney test will always be at least 86% as efficient[39]. The efficiency of the Mann-Whitney test relative to the t-test is usually high if the underlying distribution has heavier tails than the normal, because the behavior of the t-test is very dependent on the sample mean, which is quite unstable in heavy tailed distributions.

2.4.6 Calculation of qualities

Every metric has a rank sum mean, μ_1 for iOS and μ_2 for Android. The rank sum (Mann-Whitney) mean is taken because it is less sensitive to extreme values and distributions. To compare these two means, a ratio p is created with the equation presented in Equation 2.6.

$$f(p) = \frac{1}{p} = \frac{\mu_1}{\mu_2} \tag{2.6}$$

All results for $f(p)$ can be graphically modeled as rectangular hyperbola with horizontal and vertical asymptotes in the first quadrant.

Because qualities are influenced by multiple metrics, a mean of all metrics that have influence on one quality is calculated. There is a problem, a calculation of the arithmetic mean with $f(p)$ would lead to faulty results because values above 1 have much more impact on the result than those below. Hence, the geometric mean is calculated. The formula is presented in Equation 2.7.

$$Quality = \sqrt[n]{\prod_{i=1}^{n} Metric_i} \tag{2.7}$$

With the result of this equation, a comparison of the different sub-qualities is made as well as a comparison of the secondary user quality for the two platforms.

Chapter 3

Metric Tool

There are plenty of tools that measure source code metrics. Lincke, Lundberg, and Löwe identified that existing software metric tools interpret and implement the definitions of object oriented metrics differently [34]. Beside that, almost all tools lack support for Objective-C syntax. Because of that, a completely new tool is build. The tool is divided into two parts. On one side, Rascal is used to measure all Android metrics and the SLOC and CBO for iOS. All other metrics are measured with a clang compiler tool. The reason therefore is that these metrics are much easier to compute with an Abstract Syntax Tree (AST) and Rascal is not able to parse Objective-C into a AST. The construct of the clang AST differ in many ways from the format Rascal accepts. That makes it hard to write a tool that import the clang AST into Rascal.

3.1 Rascal Metaprogramming Language

Rascal is a domain-specific language that provides high-level integration of source code analysis and manipulation (SCAM) on the conceptual, syntactic, semantic and technical level. It is not limited to one particular object programming language and generically applicable[33]. Rascal has been designed from a software engineering perspective and not from a formal, mathematical, perspective with a focus on three dimensions of requirements: expressiveness, safety and usability. This makes it ideal to write a metric tool. Rascal can be used in a shell or with a plugin in Eclipse. In this research, the latter is chosen. Unfortunately, there is no AST parser for Objective-C and thus, an other tool is needed.

3.2 Clang LibTooling

Clang is part of the low level virtual machine(LLVM) project. This is a collection of modular and reusable compiler and tool chain technologies. Despite its name, LLVM has little to do with traditional virtual machines, though it does provide helpful libraries that can be used to build them[15]. Clang can be used as a Front-End compiler or as a library. The library provides a infrastructure to write tools that need syntactic and semantic information about a program. There are three ways to do that:

LibClang is a stable high level C interface to clang. When in doubt LibClang is probably the interface to use. Consider the other interfaces only when there is a good reason not to use LibClang[14].

Clang Plugins allow to run additional actions on the AST as part of a compilation. Plugins are dynamic libraries that are loaded at runtime by the compiler, and they're easy to integrate into a build environment [14].

LibTooling is a C++ interface aimed at writing standalone tools, as well as integrating into services that run clang tools. Canonical examples of when to use LibTooling are simple syntax checkers or refactoring tools[14].

A clang plugin cannot be used for a part of a project e.g. one file. With the libClang interface, it is not possible to receive contextual information in the AST. Because of that, a standalone tool with libTooling is created to measure the metrics.

3.2.1 Xcodebuild

To use the standalone tool, the same compiler flags are needed to analyze the file as for a compilation of a file. Because iOS apps use different libraries and different hardware than OS-X, it is a cross-compilation. This require a lot compiler flags, too many to do that by hand for every file. Xcodebuild is the build system from Apple for xcodeprojects (all iOS apps are xcodeprojects). This tool searches all the dependencies for the specific file. Unfortunately, it is not possible to use the output from xcodebuild directly with the tool. The output needs to be transformed into a compile_commands.json file. To do this, the oclint-xcodebuild script is used[1].

3.3 Metric Implementation

3.3.1 Source Lines of Code(SLOC)

In this research the common definition of physical SLOC is used, which are all lines that do not contain blanks or comments. This count can be viewed as language-independent since it does not take in account syntactic and other variations between multitudes of programming languages[40]. Additionally, all brackets are removed. This comes forth from the different programming styles. In Java it is much more common to set the opening bracket of a block directly after the method declaration or statement than it is in Objective-C. There, most developers use new lines for opening brackets. Boehm et al. developed a model in which he identifies the volume of the source code as an important factor in development time [6]. The time needed to develop an app, depends mostly on the time that is needed to write the source code. Because of that, external libraries are excluded by hand from the volume count. The volume is measured for a complete project and for every file in a project. The SLOC metric for iOS as well as Android is measured in Rascal with exactly the same code. The most central piece of code is presented in Listing 3.1. The content of a file is read as a string. In this string, all the comments are removed before all brackets and empty lines are removed. Finally, all “\n” are searched, the index is putted into an array. The size of the array correspondents with the presented SLOC in the file. In Objective-C, the string of header files and source files are combined if they have the same file name.

```
public int countSLOC(str file){
  file = removeComment(file);
  file = removeEmptyLines(file);
  return size(findAll(file, ”\n”));
}
```

Listing 3.1: Essential piece of code for measuring SLOC

[1] OCLint is a static code analysis tool for improving quality and reducing defects by inspecting C, C++ and Objective-C code and looking for potential problems[42]

3.3.2 Cyclomatic Complexity (CC)

Cyclomatic Complexity is used as module metric. A module is the smallest testable unit of a program. Cyclomatic Complexity is measured with an AST, thus for iOS is the clang tool used, for Android the Rascal m3 engine.

In clang, the visitor pattern, for visiting every node in the AST, is implemented in the “RecursiveASTVisitor” class. Every node is implemented as a function. To customize the behavior, a subclass needs to be created and the node function must be overridden. Every function needs to return a Boolean. If a “false” is returned, the visiting of the AST stops and while a true is returned, the visiting of the AST continuous. The implementation of the visitor patter in Rascal is different compared to that in Clang. In Rascal the visitor pattern is implemented like a switch statement. Every node can be accessed through a “case” statement.

```
unsigned CC::calcMethodComplexity(clang::ObjCMethodDecl *decl){
    complexity = 1;
    (void)TraverseDecl(decl);
    return complexity;
}
bool CC::VisitIfStmt(clang::IfStmt *){
    complexity++;
    return true;
}
bool CC::VisitForStmt(clang::ForStmt *stmt){
    complexity++;
    return true;
}
...
```

Listing 3.2: Class for calculating the cyclomatic complexity for iOS with the clang tool

```
public int CCinMethod(Statement ast){
  int complexity = 1;
  visit(ast){
    case \foreach(_,_,_):  complexity += 1;
    case \for(_,_,_,_):    complexity += 1;
    case \if(_,_):         complexity += 1;
    ...
  }
  return complexity;
}
```

Listing 3.3: Method for calculating cyclomatic complexity for Android with Rascal

Albeit the implementation of the visitor pattern differs between clang and Rascal, the result is the same. In both cases, a subclass or function is written in which every predicate is visited. Every time a predicate is visited, a variable that holds the complexity is incremented by one. Method complexities of one, result in a non-valid case. Only values with a complexity higher than one are taken into account. The reason therefore is the iOS SDK. This contains of no source files, only header files. The source files are precompiled. So the method declaration is visible in the AST, but its content is not. The consequence would be that all those method declarations return 1, while this is probably not the case.

3.3.3 Information Flow (IF)

As described in subsection 2.3.3, the information flow contains two parts.

Method Length

The length of the method is in both platforms calculated with the location of the method in the source code. The location is taken from the AST. In Rascal, the location is stored as annotation to the specific node. A visitor visits every function and extracts the location. The length is calculated with the code in Listing 3.4. The AST in clang doesn't contain location information.

```
public int calcMethodLength(loc l){
  return l.end.line - l.begin.line;
}
```

Listing 3.4: Method for the calculation of the length of a method in Rascal for Android

```
unsigned IF::calcMethodLength(Decl *decl){
    FullSourceLoc end = Context ->
            getFullLoc(decl -> getLocEnd());
    FullSourceLoc start = Context ->
            getFullLoc(decl ->getLocStart());

    return (end.getSpellingLineNumber() - start.getSpellingLineNumber());
}
```

Listing 3.5: Calculation of the method length in the clang tool for iOS apps

This is stored in the context and can be accessed with a pointer from the declaration. This is done in the first part of Listing 3.5. The second part computes the length exactly the same way as this happen Listing 3.4.

Fan

In the clang AST, variables and objects get accessed by a *DeclRefExpr*. This is a node with a reference to the original declaration. Hence, every *DeclRefExpr* in a function is visited and the name of the original variable is stored in an array if it is not already in there. This is done with the code in Listing 3.6.

```
bool IF::VisitDeclRefExpr(DeclRefExpr *decl){
    std::string expr = (decl->getDecl())->getNameAsString();
    if (std::find(fan.begin(), fan.end(), expr) == fan.end()) {
        fan.push_back(expr);
    }
    return true;
}
```

Listing 3.6: Method in the clang tool, which measures the Fan for iOS apps

Unfortunately, Rascal doesn't have a *DeclRefExpr* node as there is in the clang AST. To get the same behavior as in Listing 3.6, the *simpleName* node is taken to work with. This node belongs to the Expression data type. Objects and variables do both have a *simpleName* node declaration. Therefore, it is assumed that programmers used proper naming conventions, where variables start with lower case letters. Listing 3.7 shows the resulting function, where at the end the found variable is putted into a set. In Rascal, a set doesn't contain duplicates.

```
public set[str] getParamNames(Declaration decl){
  set[str] names = {};
  visit(decl){
    case \simpleName(str name): {
      if(/^[a-z]{1}/ := name){
          names += name;
      }
    }
  }
  return names;
}
```

Listing 3.7: Method in Rascal that computes the Fan for Android apps

3.3.4 Depth of Inheritance Tree (DIT)

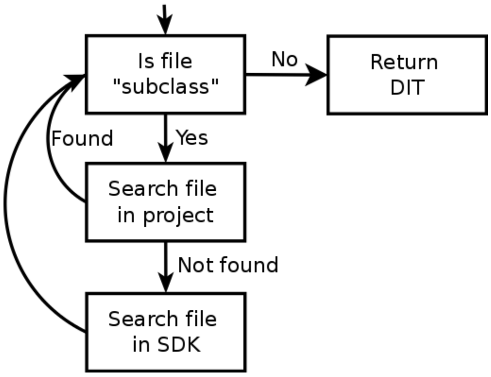


Figure 3.1: Graphical description of the algorithm that is used to count the DIT in Android apps

A graphical abstraction of the DIT algorithm in Rascal is presented in Figure 3.1. The algorithm takes a particular class and looks if it has a super class. If it has one, the specific file is searched in the project directory. When the class is found, the algorithm starts again with the super class. If no file is found, the algorithm searches further in the SDK and returns then the class. Every time a new class is found, the DIT gets incremented by 1. This recursive process continuous until there is no more super class.
As in subsection 3.2.1 described, xcodebuild searches all dependencies for a file in iOS. With these information, the AST consists not only the source file information, but also all dependencies. As in Listing 3.8 shown, clang has a build in function that finds super classes. The only thing left is to recursively iterate through all the classes. If a class has no super class, the actual depth is stored in an array and the DIT value is set back to 1.

```
bool DIT::VisitObjCInterfaceDecl(ObjCInterfaceDecl *decl){
    ObjCInterfaceDecl *superDecl = decl -> getSuperClass();
    if (superDecl == NULL) {
        ditArray.push_back(dit);
        dit = 1;
        return true;
    }
    dit++;
    TraverseDecl(superDecl);
    return true;
}
```

Listing 3.8: Essential piece of code in the clang tool that calculates DIT of iOS apps

3.3.5 Coupling Between Objects (CBO)

Because the clang tool produces an AST with all dependencies, there are no *import* statements in the AST. String operations are used to get the CBO in iOS. The string is first modified as described in subsection 3.3.1. After all comments are filtered, the CBO is measured with the code in Listing 3.9. In Rascal, the visitor pattern is used to visit all import nodes. Here, only the file is parsed into an AST without dependencies. The code is presented in Listing 3.10.

```
public int CBOIos(str file){
  list[int] x = findAll(file, ”#import␣”);
  return size(x);
}
```

Listing 3.9: CBO measurement method with Rascal for iOS apps

```
public int CBOAndroid(Declaration decl){
  int CBO = 0;
  visit(decl){
    case \import(str name): CBO += 1;
  }
  return CBO;
}
```

Listing 3.10: Method to measure CBO for with an AST in Rascal for Android apps

3.3.6 Response for Class (RFC)

The Java syntax uses modifiers to specify how methods can be used in a OO environment. Private methods are not accessible in other classes, protected methods are only accessible in subclasses and public methods are accessible in every instance and subclass. Listing 3.11 show the code how the RFC is measured. A visitor visits every method in the AST and check the modifier. If the modifier equals *public*, RFC is incremented by one. The RFC is measured through all the classes where the base class inherits from. To do this, the Rascal algorithm described in subsection 3.3.4 is used. In iOS, methods don't have modifiers. Methods that are declared in the interface file (header file), are public. Methods in the implementation file are accessible in every subclass and thus comparable with *protected* modifier in Java. In this research, public methods are measured. The reason therefore lies in the iOS SDK. Only interface files are visible, implementation files not.

```
public int calcRFC(Declaration decl){
 int RFC = 0;
 visit(decl){
   case m:\method(_, str name,_,_,_):  RFC += searchModifier(m@modifiers?[]);
   case m:\method(_, str name,_,_):  RFC += searchModifier(m@modifiers?[]);
 }
 return RFC;
}
```

Listing 3.11: Method for measuring the RFC for Android

The RFC in iOS is measured with the clang AST. In every interface the amount of methods are count. With a same sort of algorithm used in Listing 3.8, all interfaces where the base class inherits from, are added.

```
bool RFC::VisitObjCMethodDecl(clang::ObjCMethodDecl *decl){
    interfaces++;
    return true;
}
```

Listing 3.12: Method for measuring the RFC for iOS in the Clang tool

Chapter 4

Results

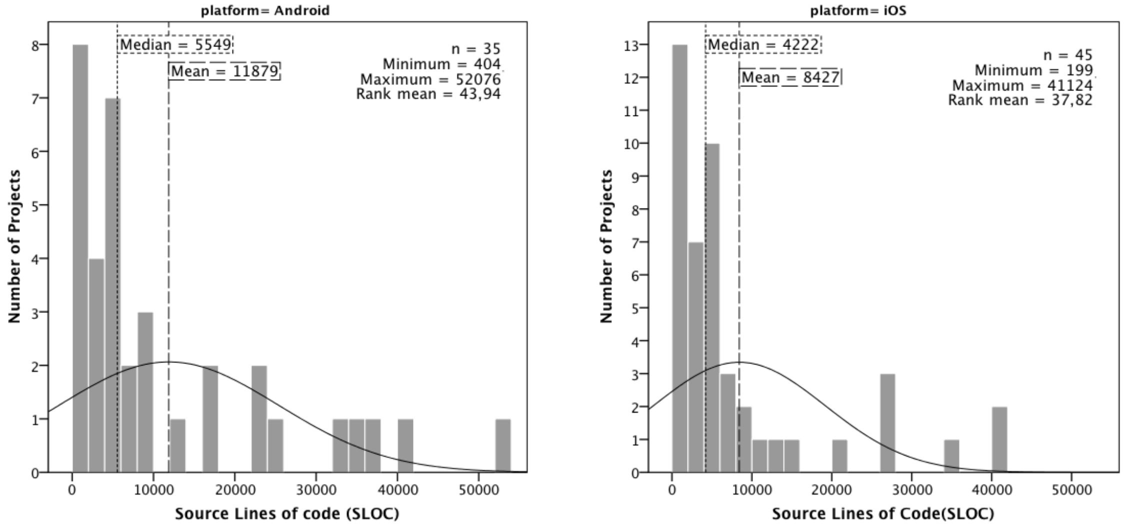
In total are 80 apps available to extract data with the developed tool. 35 of them are Android apps, 45 are iOS apps. There are 27 projects that have an iOS and an Android app, hence have the same specification. A summary of all apps with all metrics is presented in Table A.1. The visual analysis of this data show that beside the volume for a project, there is very small correlation in a project between iOS and Android apps with respect to the metrics. Because of that, statistical testing is always done with the data set generated from all 80 apps.

4.1 Source Lines of Code

From 45 iOS and 35 Android projects the volume is measured and collected in a data set. Figure 4.1a and Figure 4.1b shows that the data is not normally distributed and has a high positive skewness and kurtosis (see Table B.1). According to subsection 2.4.3, a transformation with log_{10} is done. This results in smaller skewness and kurtosis values than the std. error. Thus, the transformed data set fulfills all the criteria described in subsection 2.4.1, and t-statistics are used to test the hypothesis. The independent sample t-test show no statistical significant evidence that there is any difference in volume between iOS and Android, $t(78) = -1.161$, $p = 0.249$. This means that there is no evidence that Android apps are smaller than iOS apps. Hence hypothesis H1, which states that an app written for Android need more SLOC than one for iOS, can not be rejected. Contrary, from the distribution it can be seen that medians and means in Android apps are smaller than in iOS. The outcome of the Mann-Whitney rank sum test show also that Android apps are bigger than iOS apps with a factor 1.162. The distribution of the volume looks comparable between iOS and Android apps. A closer look to the projects that have an Android app and an iOS app, thus the same specifications, show that 17 out of 27 Android apps have more volume than iOS. The Schiphol iOS app, the biggest iOS app, belongs to the apps that are bigger than his equivalent on Android. Specially in this case, an explanation for the bigger size could be that much more developers worked on the iOS app than on the Android app.

4.2 Source Lines of Code in a File

3337 Android and 2025 iOS files are collected over all apps and analyzed for their volume. The distribution and Table B.9 shows a substantially positive skewness and especially kurtosis. Hence, according to section 2.4, a new data set with the log_{10} is created. This satisfies all the assumptions presented in subsection 2.4.1. The independent sample t-test show that volume in an iOS file is statistical significant bigger than the volume in an Android file, t(3975.656) = 7.159, p = 0.000. A closer look on the distribution shows that the spread on iOS is a bit wider than on Android. The consideration of Table B.9 shows that the range on iOS is more than twice as high compared to Android. The Mann-Whitney test show a rank mean factor of 1.115

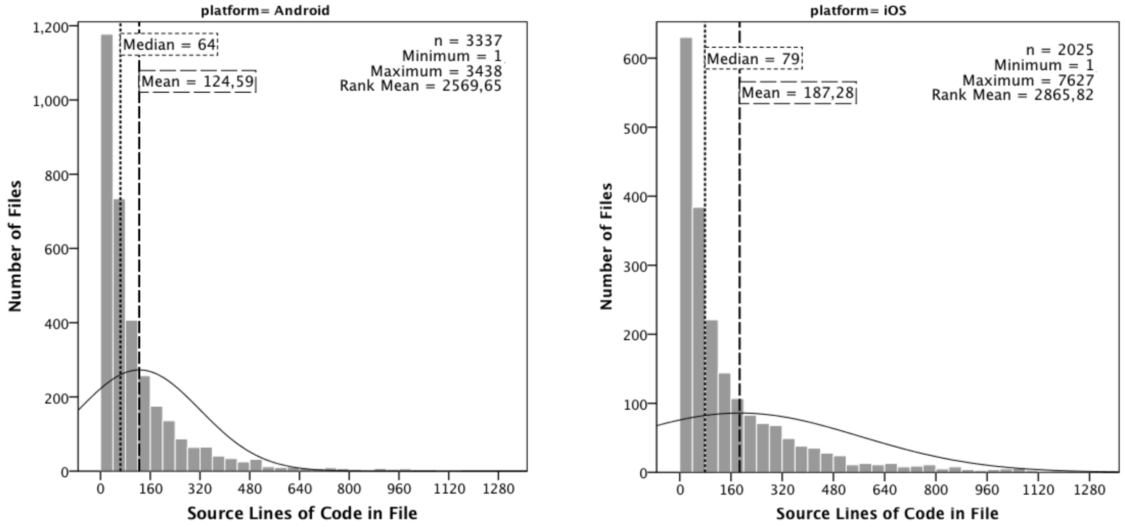


(a) Android

(b) iOS

Figure 4.1: Distribution of the Source Lines of code metric for 45 iOS apps and 35 Android apps.

higher volume on iOS than on Android. The maximal value of a file in iOS is 7626 SLOC, this value is clearly higher than the median for a complete project on iOS. This file contains a XML parser and is written in only one file. The second biggest iOS file contains 4822 SLOC followed by 4694 and 4692. Compared to Android, where the biggest file contains 3439 followed by 3337 and 2457 SLOC, it is much bigger. Table B.9 shows that, beside the mean and median, also the maxima is bigger on iOS.



(a) Android

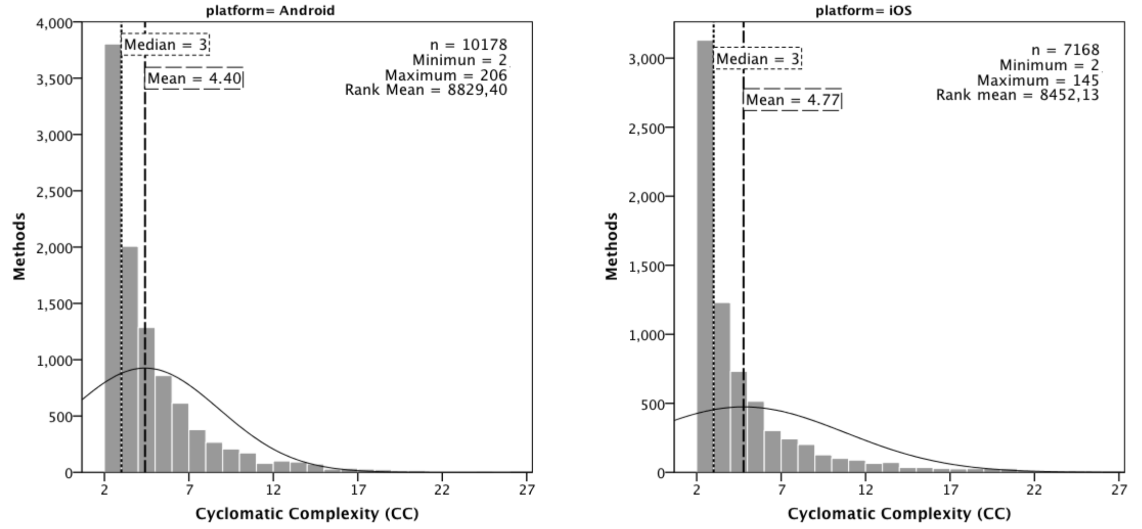
(b) iOS

Figure 4.2: Distribution of the Source Lines of code in files

4.3 Cyclomatic Complexity

7168 iOS and 10178 Android methods with minimal two paths are found in all apps that were available. An examination of the skewness and kurtosis revealed serious departures from normality for the dependent variable, cyclomatic complexity, for Android and iOS. The transformation with the in subsection 2.4.3 proposed formulas still shows serious departures from normality.

Because of that, a non parametric Mann-Whitney U independent samples test is performed. This revealed a statistically significant difference in cyclomatic complexity(U = 34891174.4, Z = -5.074 p = 0.000) with an iOS rank mean of 8452.13 and an Android rank mean of 8829.40. Although there is a significant difference in mean ranks, there is no difference in median values. Table B.2 show that the 90 and 95 percentiles are even higher on iOS than on Android, while all smaller are the same. A comparable result show the calculation of the mean. There, Android scores lower than iOS. On the other side, range and maximal values are higher on Android. The mean rank factor between Android and iOS is 1.044. Because the contradicting results and the very small mean ranks factor, it is deemed that there is too little evidence to reject hypothesis H2 that states that iOS and Android apps have the same cyclomatic complexity. There is almost no difference in cyclomatic complexity. Because this metric has influence on testing, testability through path coverage should be equal between iOS apps and Android apps. The grouping of the apps with the same specification show that only in 5 project out of 27, Android apps have a higher cyclomatic complexity than iOS. In Table A.1 can be seen that the difference is the most extreme on the KLM Shaker app. On iOS, a cyclomatic mean of 12.5 is measured, 2.53 on the Android app. This is an example of good/bad programming. In the Shaker app for Android is worked a lot with the flow, on the iOS app are many steps that are covered with a superfluous conditional if/switch statement. This is also a reason why this iOS app has a higher volume than the iOS app. Besides that, the mean method length is with 83.4 LOC 4 times higher than the mean on Android.



(a) Android

(b) iOS

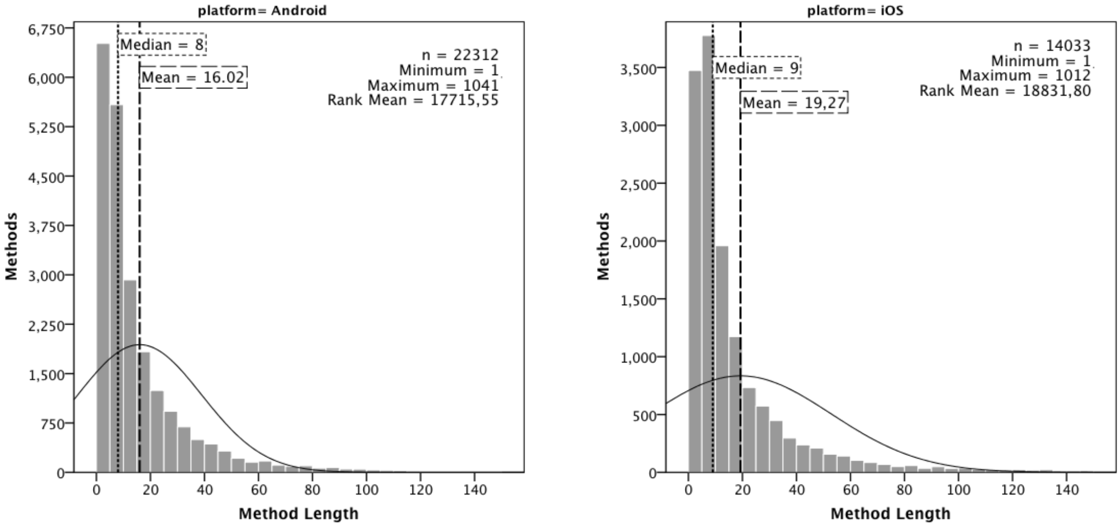
Figure 4.3: Distribution of the Cyclomatic Complexity

4.4 Information Flow

For the information flow, 14033 iOS methods and 19432 Android methods are measured over all available apps. It is measured in two steps, first the length of the method, second the amount of objects that where used in the method.

4.4.1 Method Length

The method length test is performed with a non parametric Mann-Whitney U independent samples test. This is done because of serious departures from normality in skewness and kurtosis and there are a lot of outliers on the positive side. This test revealed a statistical significant difference in method length(Z = -9.896, p = 0.000) where iOS methods have a Mean Rank of 18831.8 and Android methods have 17715.55. The factor between these two is 1.063. Table B.3 show that median method length on iOS is one LOC higher than on Android. The minimum is on both platforms 1 LOC and the maximum is a little bit higher on Android(1041 LOC) than on iOS(1012 LOC). One reason for the higher method length in iOS is the verbosity of the Objective-C language. Often, a developer divides one statement into several lines to improve readability. A test with the Ranger Dierenjournaal iOS app, where a comparison is made between the normal case and the case where all statements use just one line. This test show that methods are approximately 5% longer because of verbosity.



(a) Android

(b) iOS

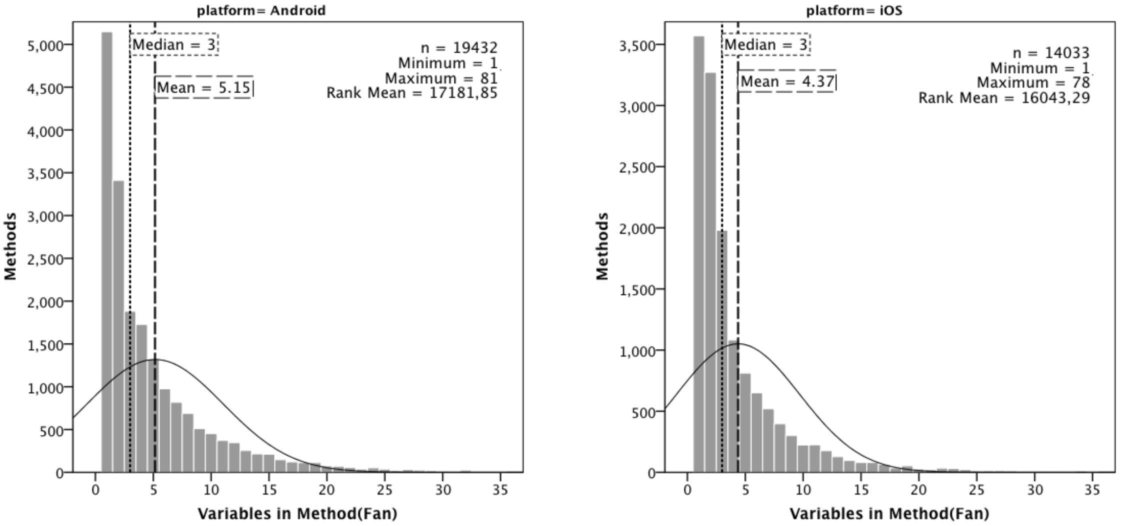
Figure 4.4: Distribution of the measured method length

The results above show that the difference in method length is small. Thus, there is probably not much difference in the two platforms when it comes to the way developers have to implement methods. But it is a good benchmark to see if a developer separates the problems so that every function performs one task. A high ratio between apps with the same specification indicates that methods by the one with the higher number performs more than one task and are more difficult to understand.

4.4.2 Fan

The same amount of methods are measured as for the length. Figure 4.5b and Figure 4.5a show the same median values on iOS and on Android. The mean is a little bit higher on Android. The non parametric Mann-Whitney test, which is performed because of not normal distributed data and not acceptable kurtosis values, confirms with a statistical significant certainty(U = 126553892, Z = -10.797, p = 0.000) that Android methods have a higher Fan than iOS methods. The quotient of the two rank means is 0.934. Not only the mean value is higher on Android, also the inter quartile range, as Table B.4 shows. The maximal value of 78 on iOS is not far away from 81 on Android. One reason for the lower values in iOS could be that an object has more depth than objects in Android. The fact that classes and methods are bigger, confirm this presumption.

An analysis for the highest values shows that there is no match in apps with the same specification. On Android side, the second(81), third(77) and fourth(73) highest values belongs all to the Heineken Eprogram app. A closer look at Table A.1, show that the median value(6) is twice as high as the median for all projects. The app that performs worst is the OCR app for



(a) Android

(b) iOS

Figure 4.5: Distribution of the measured Fan

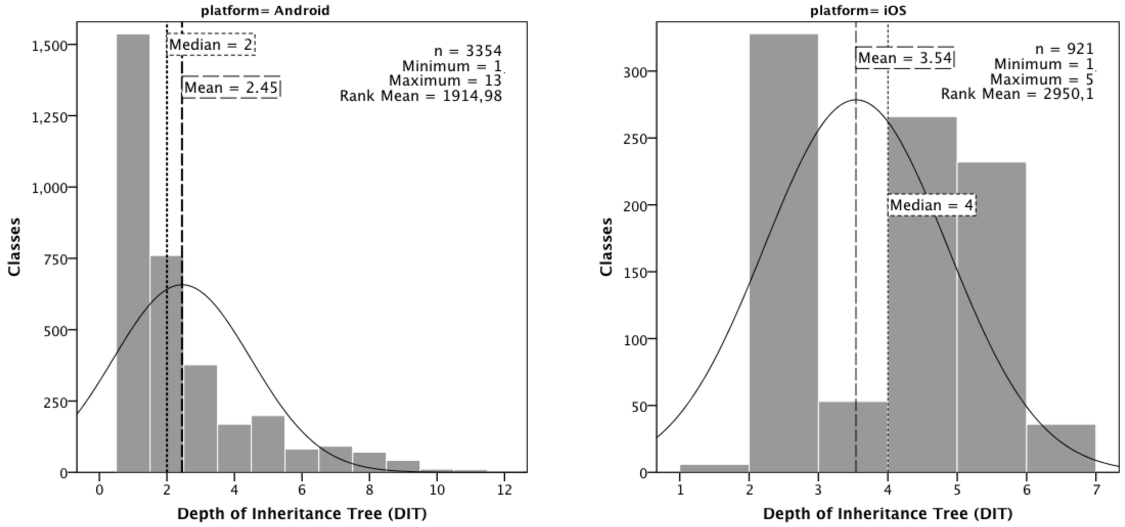
iOS. In this, a mean of 14.6 and a median of 15 is measured. The lowest value is found in the iOS Boobalyzer app with a mean of 2.25 and a median of 2.

4.4.3 Information Flow

Since the information Flow metric is computed with the formula $Length \cdot Fan \cdot Fan$, the Fan weights much more than the length. This leads to the assumption that an Android method is more complex than an iOS method despite the fact of shorter length. Because of serious departures from normality, a non parametric Mann-Whitney test is performed. The results show that the Information Flow complexity is with statistical significance higher on Android(16991.43) than on iOS(16308), with U = 130254008, Z = -6.391 and p = 0.000. A look at Table B.5 shows that there is almost no difference in means. An other interesting fact show the comparison of the kurtosis and skewness values. The skewness differs only 2.4% and kurtosis even only 0.05% between the two platforms. This means that the distributions of the platforms are almost the same. From the results of the Mann-Whitney test, hypothesis H3, that says that there is no difference in information flow between the two platforms, is rejected.

4.5 Depth of Inheritance Tree

The measurement of the depth of inheritance tree results in big differences in the distributions between the two platforms. As in Figure 4.6a can be seen, the classes in Android have the same shape as most metric distributions with a maximum of 13 and a median of 2. Little less than half of the classes have no super class. On the other side, in iOS are almost no classes without a super class. Most classes have two, four or five super classes. And there are no cases measured with a DIT higher than 6. Median value is twice the value on Android. Because the big differences in distributions, a Mann-Whitney is performed to test the difference. The results show that the depth of inheritance tree on iOS is statistical significant bigger than on Android. The mean rank ratio between iOS and Android is 1.541. As in subsection 2.3.4 described is the result as expected and there is no evidence to reject hypothesis H4, that states that the DIT for the Android framework is smaller than that for the iOS framework. The iOS results indicate that most base classes inherit from the same classes in the iOS SDK. To test this assumption,



(a) Android

(b) iOS

Figure 4.6: Distribution of the measured Depth of Inheritance

the DIT of the iOS Schiphol apps is analyzed. This is the biggest iOS app that is measured. The outcome is presented in Figure 4.7. Every box shows a class in the SDK and the percentage of how many base classes inherit from this class.
Every class in the Schiphol project has a minimal DIT of 2 with NSObject as super class. This is the root class of most Objective-C class hierarchies. Through NSObject, classes inherit a basic interface to the runtime system and the ability to behave as Objective-C objects. 64% inherits not further from the SDK, the rest inherits from the UIResponder. This class defines an interface for objects that respond to and handle events. Interestingly, UIResponder only has subclasses in the SDK and no custom subclasses. That's an explanation for the low value of classes with a DIT of 3. The direct subclasses of UIResponder are UIViewController and UIView. The first one provides the fundamental view-management model for all iOS apps, UIView defines a rectangular area on the screen and the interfaces for managing the content in that area. At runtime, a view object handles the rendering of any content in its area and also handles any interactions with that content. UITableViewController or UITableViewCell are extensions in a defined context. Beside NSObject, custom base-classes inherits from either UIView or UIViewController or one of the subclasses of them. This explains also the high bars with a DIT of 4 or 5 in Figure 4.6b. Table A.1 shows that the analyzed iOS Schiphol app has a lower DIT mean than the equivalent app on Android. This again shows that developers on Android make more use of inheritance than iOS developers.

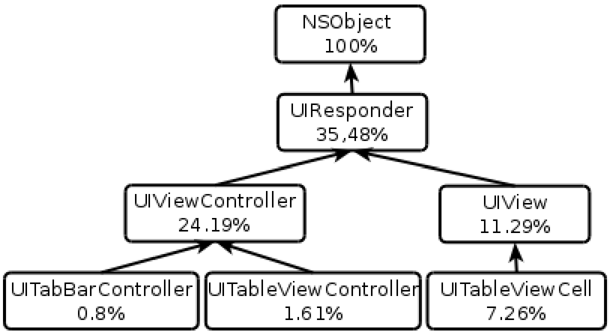
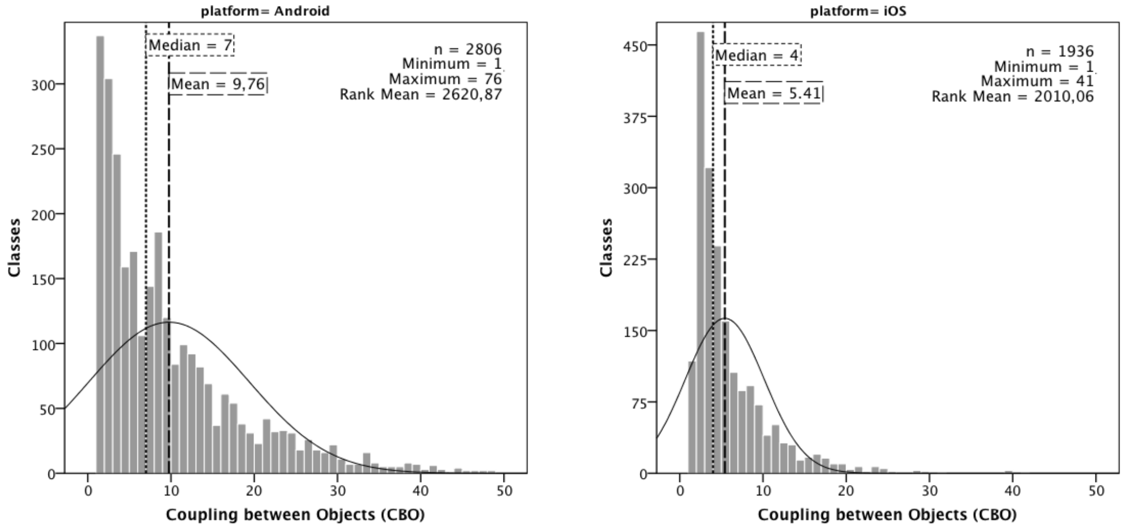


Figure 4.7: Chart that represents from which classes in the iOS SDK is inherited in the Schiphol app

4.6 Coupling Between Objects

The curve of the iOS and Android distribution with data from all apps looks like a steep F-Distribution. A transformation with a log_{10} doesn't make the data suitable to test it with t-statistics. Because of that, a non parametric independent sample Mann-Whitney U test is done to analyze the hypothesis. This revealed a statistically significant difference in CBO values, U = 1854438.50, Z = -15,425, p = 0.000 where iOS files have a mean rank of 2010.06 and Android files 2620.87, which results in a quotient of 0.7669. Table B.7 shows that median values and range are with almost the same ratio higher on Android than on iOS. From these results, there is strong evidence that hypothesis H5, which states that objects in iOS have higher CBO values than objects in Android, is not correct and thus rejected. An inspection of the import statements in iOS show that most classes only import either UIKit.h or Foundation.h from the SDK. The UIKit framework provides the classes needed to construct and manage an application's user interface for iOS. The foundation framework provides a set of primitive object classes and introduces several paradigms that define functionality not covered by the Objective-C language such as deallocation.



(a) Android

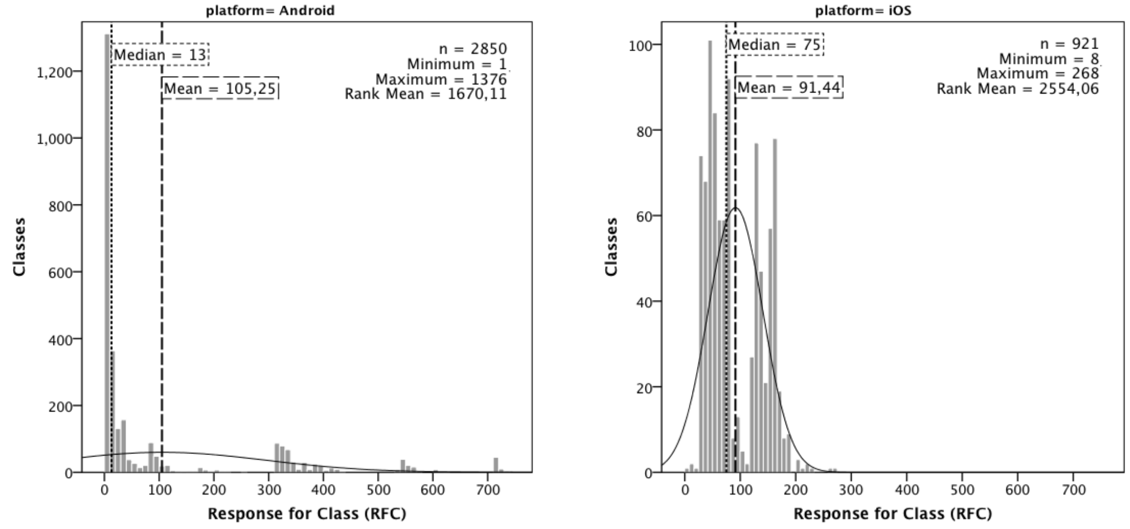
(b) iOS

Figure 4.8: Distribution of the measured Coupling between Objects

The analysis of the import statements on the Android Schiphol app show a more difficult structure. Every app has an automatically generated R.java file. It contains unique identifiers for elements in each category(drawable, string, layout, color, ...) of resources available in the application. This file is imported in every class. And in comparison to iOS, on Android are not frameworks included, but classes. This generates an extra level and introduces more import statements. A look at Table A.1 shows that every Android app has a higher CBO value than his equivalent on iOS.

4.7 Response For Class

Figure 4.9a looks totally different from Figure 4.9b. Almost half the classes on Android have a RFC value that is smaller than 10. That is why the median value is 13. On iOS, there are almost no classes that have a smaller RFC value than 30. From there on, there is a rapid increase. Most classes have a RFC between 50 and 60, but the difference to other values is much less extreme than on Android. On iOS, the median value and the mean are not far away from each contrary to median and mean values on Android. The distribution on Android looks in no way normally distributed. That is the reason why a non parametric independent sample Mann-Whitney U test is performed. The results show that iOS classes have statistically significant higher RFC values than Android classes, U = 697145.5, Z = -21.431, p = 0.000. The quotient of the two mean ranks results in 1.529. Although the RFC is statistical significant bigger on iOS, a look at Table B.8 shows that the range is 5 times smaller. The statistical tests show that there is no evidence to reject hypothesis H6, that states that Objects in iOS have higher RFC values than objects in Android. The inspection of the five highest values on Android show that all these classes inherit from View.java. This class represents the basic building block for user interface components and consist of almost 19k lines of code and has 490 public methods. The block with a RFC between 300 and 420 in Figure 4.9a comes from the activities. The two major super classes for every activity are Activity.java and Context.java. The first class has 145 public classes and the second one has 109 public classes. The total DIT for an activity is bigger than 5, hence there are three more classes that contribute to the RFC.



(a) Android

(b) iOS

Figure 4.9: Distribution of the measured Response for Class

4.8 Closing the circle

For every metric, a mean rank factor between the two platforms is calculated. In this section, an attempt to map this values back to the sub-qualities is given as described in subsection 2.4.6. Table 2.3 shows that every sub-quality is addressed by one or more metrics. The results are in a linear scale and values above 1 say that Android perform better, below perform iOS better.

Adaptability is influenced by the Depth of Inheritance Tree with a ratio of 1.541 and Coupling Between Objects with 0.767. Because higher DIT has a positive effect on adaptability, the reciprocal DIT value is taken for the calculation. This results in 0.706, that indicates that iOS is more adaptable than Android.

Modularity is influenced by Source Lines of Code in Files(1.115), Information Flow(0.96) and Coupling Between Objects(0.767). The geometric mean is 0.936, a very small difference in favor of iOS.

Modifiability is influenced by the Information Flow(0.96) and Coupling Between Objects(0.767). This results in 0.858. This states that iOS apps are more modifiable.

Testability is influenced by Cyclomatic Complexity(0.958), Coupling Between Objects(0.767) and Response for Class(1.529). The calculation of the mean results in 1.034, this shows that Android apps are slightly more testable.

Understandability is influenced by the Source Lines of Code(0.86), Information Flow(0.96), Depth of Inheritance(1.541) and Response for Class(1.529). This results in 1.18, a factor in favor of an Android app.

Self descriptiveness is influenced by the Information Flow(0.96) and Depth of Inheritance Tree(1.541). With a score of 1.216, also for this sub-quality perform Android apps better.

To give an index which platform offers better secondary user quality, the geometric mean is also computed with all these sub-qualities. This results in 0.976.

Chapter 5

Discussion, Threats and Future work

5.1 Discussion

The results of this research show that the iOS framework as well as the Android acts like a grey-box framework. Although both are grey-box like, there are differences. The iOS framework is more based on white-box. This conclusion is made from the DIT metric, where mean and median values are much higher than on Android. Also the lower CBO values supports this conclusion. A reason for the higher DIT value is the background of Objective-C, the Smalltalk programming language. But not only the programming language, also the Model-View-Controller pattern could be a reason. Since the beginning, all frameworks that Apple build are based on the MVC. Roberts and Johnson states that the MVC originally is completely based on a white-box framework [45].

Gamma et al. says that modifiability in white-box framework is better than in black-box frameworks because it is possible to define the implementation of one class in terms of another's [20]. The calculation of the sub qualities in this research shows indeed that the iOS framework performs better in the sub quality modifiability. Adaptability does have some correlation with modifiability, so it is obvious that adaptability also scores better on the iOS platform. The modularity is influenced by the amount of objects and the size of the objects. The size of the objects is higher on iOS but there are much more objects on Android. The ratio on how many objects is higher than the ratio of the object size, this makes iOS score better on modularity. But Fayad and Schmidt states that modularity is higher by encapsulating volatile implementation details behind stable interfaces [18]. This is a contradiction to the result acquired in the sub-quality modularity. Snyder says that inheritance break encapsulation[48] and white-box frameworks are based on inheritance. The reason for the difference is that Fayad and Schmidt only takes encapsulation as a factor for modularity while this research takes other factors but no encapsulation.

The Android framework is more based on a black-box framework compared to the iOS framework. This is the conclusion from the results of the CBO that are higher on Android. Also DIT values are lower, this means less inheritance. A reason that Google tries to implement the Android framework more like a black-box framework could be that experts prefer black-box frameworks over the white-box frameworks because it helps to keep each class encapsulated and focused on one task. Classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters. On the other hand, a design based on object composition will have more objects (if fewer classes), and the system's behavior will depend on their interrelationships instead of being defined in one class [20]. This behavior is also measured in the *SLOC in file* metric. In this, files on Android are significant smaller than on iOS. But there are also way more files in a Android project compared to an iOS project.

According to Markiewicz and Lucena are black-box frameworks easier to use because developer only need to know the interface of the object to use it[36]. On a white-box framework, the

developer needs to know the internals of the super class for proper use. Result for the sub qualities understandability and self-describtivness show that Android perform better in these two than iOS. This qualities can be mapped to the statements from Markiewicz and Lucena. If something is easier to understand and it is more self-descriptive it should be easier to use.
According to Roberts and Johnson, a framework development should be an evolving process where the initial framework starts as a white-box framework and should evolve into a black-box framework [45]. The fact that the iOS framework is older than the Android, could be an indication that it is more difficult to change the framework and/or Apple implements their changes more conservative.

5.2 Threats of validity

Validity of research is concerned with the question of how the results might be wrong, i.e. the relationship between results and reality. This is a quantitative research, and threats come from the interpretation of the existing theories and implementation of them. Below, threads of validity are divided into the three main types of validity[19].

5.2.1 Construct validity

Construct validity focus on the relation between the theory behind the experiment and the observations. The theory of this research is based on existing research with certain reputation. But none of these researches or theories specifically deals with mobile apps, which are most of the time small projects. For this reason and to get a broader acceptance, the factor-criteria-metrics model is taken and multiple researches are searched for factors. Every further step is supported with empirical research.

5.2.2 Internal validity

The Internal validity focus on the certainty that the treatment actually causes the outcome. These are the biggest threats in this research. In almost all statistical tests are significant results found. Sometimes with very small differences. The reason therefore lies in the very big data set that is created. The bigger the data set is, the more accurate the statistical calculations get. But on the other side, a very small difference in measuring the metrics can be fatal for the results. Therefore, results with big differences get more attention than those with small differences.
The tool with which the quality is analyzed has significant influence on the results. The nicest solution would be to implement all measurements in one tool that parses exactly the same AST so that metrics can be computed with the same piece of code, regardless the programming language. Because of time considerations, this was not possible. A new tool, with the highest priority of measuring the same between the two platforms, is made with Rascal and Clang. Unfortunately, there are some differences in the AST construction. Especially in measuring the fan-in and fan-out. The closest approximation of this metric resulted in a combination of fan-in and fan-out into fan. This differs from the original definition and consequences are not clear.
Front-end compilers, as used in this project to create AST's are very complex projects. It is possible that those can contain bugs, since clang 3.5 as well as Rascal are not official release versions. This could also have influence on the results. An other thread is the way of implementing the metrics. Most metrics leave the user some space for own interpretation. Thus, comparing these results with other results that are measured with an other tool can lead to very different results.

5.2.3 External validity

External validity is concerned with the possibility to generalize the results outside the scope of our study. As in every peace of software, it is possible that it contains bugs. Things of which a developer is unaware. Such things often get visible when the software is used through different people. In this project, theory, implementation and testing is done by one person. The measurements are done by the same person. To minimize bugs, every metric is tested first with the Ranger Dierenjournaal, a small and clear app. After that, random samples are taken from different apps and tested if the results corresponds with the visible perception. After that, the results are analyzed with a statistical software package and extreme values are inspected.
One of the most important factors for external validity are the developers that build the app. In this research, only apps from M2Mobi are evaluated. Most developers in this company are Junior/Medior developers. No app is completely made by a senior developer. The company exists of a very young group of developers. The oldest developer at the moment is 34 years young. Since mobile app business started in 2007, it is assumed that almost no company has many senior developers. Hence, if the experiment is conducted in an other company with a different experience level then the results could differ from these.

5.3 Future Work

This is the first research that compares secondary mobile app quality between the two major platforms. To confirm these results, more research is needed. Thus research that has a similar setup as this.
Further work could also be the research of the mapping from the metrics to the sub-qualities. In this research, only the geometric mean is used to do this and is not based on scientific articles. A weighting of the different factors is also left out of the consideration. Beside that, this research used some very specific measurement methods. It would be interesting what the outcome is, if for example also the code in the SDK is analyzed with all the metrics.

Chapter 6

Conclusion

This research had as goal to inspect the secondary user quality on the Android platform and on the iOS platform and make a comparison of these two. These are the findings of this research:

- In the method metrics cyclomatic complexity and information flow is very little difference measured between iOS and Android.
- For the volume of the apps, the only project metric that is measured, is no evidence found that they are smaller on the iOS platform than on the Android platform.
- The biggest differences are measured with the object oriented metrics. As expected has iOS a higher depth of inheritance tree than Android. In contrast to the expectations, Android has higher coupling between objects than iOS.
- The calculation of the geometric mean for the sub-qualities show that iOS scores better on Adaptability, Modularity and Modifiability. Android scores better when it comes to Testability, Understandability and Self descriptiveness.
- The calculation of the geometric mean for the secondary user quality show that there is almost no difference in secondary user quality between the two platforms. Only the approach of how to achieve this differs.

This is the first research that compares secondary user quality between iOS and Android. Further research should be done to verify this results.

Chapter 7

Bibliography

[1] KK Aggarwal et al. “Measurement of software maintainability using a fuzzy model”. In: *Journal of Computer Science* 1.4 (2005), p. 538.

[2] Android. *Android Developers.* Feb. 2014. URL: `http://developer.android.com/`.

[3] Victor R Basili and Barry T Perricone. “Software errors and complexity: an empirical investigation0”. In: *Communications of the ACM* 27.1 (1984), pp. 42–52.

[4] Donnie Berkholz. *Programming languages ranked by expressiveness.* Mar. 2013. URL: `http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/`.

[5] Barry W Boehm et al. *Characteristics of software quality.* Vol. 1. North-Holland Publishing Company, 1978.

[6] 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.

[7] Tim Bray. *What Android is.* Nov. 2010. URL: `http://www.tbray.org/ongoing/When/201x/2010/11/14/What-Android-Is`.

[8] Lionel Briand et al. “Empirical studies of software maintenance: A report from WESS’97”. In: *Empirical Software Engineering* 3.3 (1998), pp. 299–307.

[9] Manfred Broy, Florian Deissenboeck, and Markus Pizka. “Demystifying maintainability”. In: *Proceedings of the 2006 international workshop on Software quality.* ACM. 2006, pp. 21–26.

[10] Jones Capers. *Programming Languages Table.* Mar. 1996. URL: `http://www.cs.bsu.edu/homepages/dmz/cs697/langtbl.htm`.

[11] Shyam R Chidamber, David P Darcy, and Chris F Kemerer. “Managerial use of metrics for object-oriented software: An exploratory analysis”. In: *Software Engineering, IEEE Transactions on* 24.8 (1998), pp. 629–639.

[12] Shyam R Chidamber and Chris F Kemerer. “A metrics suite for object oriented design”. In: *Software Engineering, IEEE Transactions on* 20.6 (1994), pp. 476–493.

[13] Don Coleman et al. “Using metrics to evaluate software system maintainability”. In: *Computer* 27.8 (1994), pp. 44–49.

[14] Clang Community. *Choosing the Right Interface for Your Application fffdfffdfffd Clang 3.5 documentation.* June 2014. URL: `http://clang.llvm.org/docs/Tooling.html`.

[15] LLVM Community. *The LLVM Compiler Infrastructure Project.* June 2014. URL: `http://llvm.org/`.

[16] B Terry Compton and Carol Withrow. “Prediction and control of ada software defects”. In: *Journal of Systems and Software* 12.3 (1990), pp. 199–207.

[17] R. Geoff Dromey. “A model for software product quality”. In: *Software Engineering, IEEE Transactions on* 21.2 (1995), pp. 146–162.

[18] Mohamed Fayad and Douglas C Schmidt. “Object-oriented application frameworks”. In: *Communications of the ACM* 40.10 (1997), pp. 32–38.

[19] Robert Feldt and Ana Magazinius. “Validity Threats in Empirical Software Engineering Research-An Initial Survey.” In: *SEKE*. 2010, pp. 374–379.

[20] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[21] Marcela Genero et al. “Building UML class diagram maintainability prediction models based on early metrics”. In: *Software Metrics Symposium, 2003. Proceedings. Ninth International*. IEEE. 2003, pp. 263–275.

[22] Marcela Genero et al. “Using metrics to predict OO information systems maintainability”. In: *Advanced Information Systems Engineering*. Springer. 2001, pp. 388–401.

[23] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[24] Khairuddin Hashim and Elizabeth Key. “A Software Maintainability Attributes Model”. In: *Malaysian Journal of Computer Science* 9.2 (1996), pp. 92–97.

[25] Péter Hegedűs et al. “Source code metrics and maintainability: a case study”. In: *Software Engineering, Business Continuity, and Education*. Springer, 2011, pp. 272–284.

[26] Ilja Heitlager, Tobias Kuipers, and Joost Visser. “A practical model for measuring maintainability”. In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*. IEEE. 2007, pp. 30–39.

[27] Sallie Henry and Dennis Kafura. “Software structure metrics based on information flow”. In: *Software Engineering, IEEE Transactions on* 5 (1981), pp. 510–518.

[28] IEC ISO. “IEC 25010: 2011: Systems and software engineering–Systems and software Quality Requirements and Evaluation (SQuaRE)–System and software quality models”. In: *International Organization for Standardization* (2011).

[29] ISO/IEC et al. “ISO/IEC 9126”. In: *Information Technology, Software Product Evaluation, Quality Characteristics and Guidelines for their Use* (1991).

[30] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. “Measuring software product quality: A survey of ISO/IEC 9126”. In: *Software, IEEE* 21.5 (2004), pp. 88–92.

[31] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. 2nd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002. ISBN: 0201729156.

[32] Even-André Karlsson. *Software reuse: a holistic approach*. John Wiley & Sons, Inc., 1995.

[33] Paul Klint, Tijs van der Storm, and Jurgen Vinju. “Rascal: A domain specific language for source code analysis and manipulation”. In: *Source Code Analysis and Manipulation, 2009. SCAM’09. Ninth IEEE International Working Conference on*. IEEE. 2009, pp. 168–177.

[34] Rüdiger Lincke, Jonas Lundberg, and Welf Löwe. “Comparing software metrics tools”. In: *Proceedings of the 2008 international symposium on Software testing and analysis*. ACM. 2008, pp. 131–142.

[35] Matinlassi Mari and Niemelä Eila. “The impact of maintainability on component-based software systems”. In: *Euromicro Conference, 2003. Proceedings. 29th*. IEEE. 2003, pp. 25–32.

[36] Marcus Eduardo Markiewicz and Carlos J. P. de Lucena. "Object Oriented Framework Development". In: *Crossroads* 7.4 (July 2001), pp. 3–9. ISSN: 1528-4972. DOI: 10.1145/372765.372771. URL: http://doi.acm.org/10.1145/372765.372771.

[37] Thomas J McCabe. "A complexity measure". In: *Software Engineering, IEEE Transactions on* 4 (1976), pp. 308–320.

[38] Jim A McCall, Paul K Richards, and Gene F Walters. *Factors in software quality. volume i. concepts and definitions of software quality.* Tech. rep. DTIC Document, 1977.

[39] Douglas C Montgomery and George C Runger. *Applied statistics and probability for engineers.* 2009.

[40] Vu Nguyen et al. "A sloc counting standard". In: *COCOMO II Forum.* 2007.

[41] David E. Peercy. "A software maintainability evaluation methodology". In: *Software Engineering, IEEE Transactions on* 4 (1981), pp. 343–351.

[42] Longyi Qi. *OCLint.* Dec. 2013. URL: http://oclint.org/.

[43] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. "A systematic review of software maintainability prediction and metrics". In: *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement.* IEEE Computer Society. 2009, pp. 367–377.

[44] SWA Rizvi and RA Khan. "Maintainability estimation model for object-oriented software in design phase (memood)". In: *arXiv preprint arXiv:1004.4447* (2010).

[45] Don Roberts and Ralph Johnson. "Evolving frameworks". In: *Pattern languages of program design* 3 (1996).

[46] Scott L Schneberger. "Distributed computing environments: effects on software maintenance difficulty". In: *Journal of Systems and Software* 37.2 (1997), pp. 101–116.

[47] Harry M. Sneed and András Mérey. "Automated software quality assurance". In: *Software Engineering, IEEE Transactions on* 9 (1985), pp. 909–916.

[48] Alan Snyder. "Encapsulation and Inheritance in Object-oriented Programming Languages". In: *SIGPLAN Not.* 21.11 (June 1986), pp. 38–45. ISSN: 0362-1340. DOI: 10.1145/960112.28702. URL: http://doi.acm.org/10.1145/960112.28702.

[49] Tiobe Software. *TIOBE Index for February 2014.* Feb. 2014. URL: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[50] Barbara G Tabachnick, Linda S Fidell, et al. *Using multivariate statistics.* Allyn and Bacon Boston, 2001.

[51] Kurt D Welker, Paul W Oman, and Gerald G Atkinson. "Development and application of an automated source code maintainability index". In: *Journal of Software Maintenance: Research and Practice* 9.3 (1997), pp. 127–159.

[52] Stephen G West, John F Finch, and Patrick J Curran. "Structural equation models with nonnormal variables: Problems and remedies." In: *Structural equation modeling: Concepts, issues and applications* (1995), pp. 56–75.

[53] Stephen S. Yau and James S. Collofello. "Design stability measures for software maintenance". In: *Software Engineering, IEEE Transactions on* 9 (1985), pp. 849–856.

Appendix A

Summary of Metric data

App Name	Build Date	Android Version	iOS version	Valid Class cases	Valid Method cases	SLOC	CC mean	CC median	IF Length mean	IF Length median	IF Fan mean	IF Fan median	Info Flow mean	Info Flow median	DIT mean	DIT median	CBO mean	CBO median	RFC mean	RFC median	SLOC in File mean	SLOC in File median
AMC - Navigation iOS	2013		x	73	683	13055	4.17	2	24.38	14	5.37	3	5789.06	103.5	3.63	4	6.08	5	109.13	125	169.25	95
AMC - Navigation Android	2013	x		117	757	9897	3.45	3	21.19	17	6.57	5	3381.67	351	1.71	1	11.4	9	63	8	92.22	75.5
Amstel - Teamlink iOS	2010		x	114	901	34757	6.59	4	41.45	19	6.89	5	12371.6	436.5	3.35	4	7.94	6	112.59	126.5	511.82	304
Amstel - Teamlink Android	2010	x		121	824	23276	4.46	3	34.52	25	9.24	7	11851.19	1008	5.21	5	12.16	10	185.21	321	240.73	150
Amsterdam Museum iOS	2013		x	50	343	4222	3.71	3	22.81	15	5.55	4	2327.09	252	4.11	4	5	3	58.21	66	83.37	55
Amsterdam Museum Android	2013	x		62	344	4421	3.19	2	16.13	13	5.01	4	1487.46	144	2.35	2	9.62	8	84.6	22	77.13	46
BelCompany - Customer App iOS	2011		x	61	579	8683	3.66	3	19.25	11	4.4	3	1961	90	4.27	5	4.23	2	132.54	158	92.19	42.5
BelCompany - Customer App Android	2011	x		82	440	9399	3.62	3	30.51	25	9.36	6	9013.09	882	4.89	5	13.84	13	225.08	183	127.91	83.5
Boobalyzer iOS	2009		x	7	46	694	4.5	2	18	10.5	2.25	2	170	58	3.6	4	3.6	4	113	126	87.8	92
Boobalyzer Android	2009	x		9	51	1185	3.2	2	20.68	17	7.32	4	4955.98	325	4	4	11.5	10	389.13	388	136.63	77.5
EELogic iOS	2011		x	70	631	14195	4.9	3	37.8	23	6.49	4	8758.89	400	3.97	5	4.77	3	114.7	130.5	193.1	77
EELogic Android	2011	x		372	1995	36797	4.56	3	26.24	20	7.72	6	5006.58	2383.21	2.07	2	9.75	8	51.98	5	120.04	78.5
Nulaz - BvM iOS	2010		x	102	908	26436	5.95	3	51.5	26	6.86	4	18256.72	525	4.14	4	5.32	4	135.09	157	301.91	93
Nulaz - BvM Android	2010	x		189	2156	41571	4.39	3	36.96	26	7.83	5	15277.69	576	2.79	2	13.32	9.5	132.93	35	240.22	134.5
DeliXL iOS	2011		x	26	215	4194	4.46	2	29.62	20.5	5.75	4	3656	250.5	2.75	2	4.88	3	82.5	57	165.5	67
DeliXL Android	2011	x		247	1252	14026	5.29	4	29.83	21	8.09	6	23650	832	2.1	2	7.65	5	43.27	5	107.88	57.5
DeliXL - Food Service Fair 2012 iOS	2012		x	44	242	7291	4.82	3	35.54	17.5	4.81	3	3892.49	216	3.39	3.5	4.44	3	97.83	98	166.06	83.5
DeliXL - Food Service Fair 2012 Android	2012	x		35	309	5335	4.29	3	27.43	19	6.78	5	4877.1	576	2.82	1	13	8.5	159.07	14	175.21	104.5

DeliXL - Horecava iOS (Horecava)	2012		x	28	294	4022	3.68	2	19.2	13	4.83	4	1503.83	200	4.23	4	4.31	4	143.23	142	177.23	131
DeliXL - Horecava Android (Horecava)	2012	x		259	1350	24190	5.06	4	27.7	19	8.14	7	6876.46	792	2.15	2	7.69	5	39.11	5	94.09	53
Heineken - e-app iOS(Eprogram)	2010		x	74	634	21298	6.17	3	43.97	22	5.95	4	19678.31	375	3.96	4	6.21	4	68.33	70	306.72	127
Heineken - e-app Android(Eprogram)	2010	x		261	1550	34217	5.41	4	34.06	24	7.2	6	8961.7	2483.32	2.41	2	7.25	4	78.19	5	157.62	65
Heineken - Experience iOS	2013		x	39	264	7262	5.62	3	47.79	21	5.31	3	4458.6	256	3.29	2	5.86	3	63.43	50	120.86	105
Heineken - Experience Android	2013	x		71	377	4968	3.14	2.5	16.64	14	5.01	4	1352.51	156.5	2.37	2	10.19	8	85.71	21	78.34	50
Heineken - Extra Cold Compass iOS	2011		x	27	207	4844	4.17	3	25.04	16	4.14	4	1287.5	224	3.14	2	4.86	4	94.86	55	175.29	157
Heineken - Extra Cold Compass Android	2011	x		27	271	4788	3.91	3	23.59	16	7.81	5	7085	375	2.3	1	11.3	6	132.7	7	199.52	75
KRO - Rekenkamer iOS	2012		x	43	549	11008	5.48	3	26.77	16	5.86	4	5499.58	300	3.13	3	6.38	4.5	123.19	127	438.75	136.5
KRO - Rekenkamer Android	2012	x		36	225	3118	3.58	3	19.89	15	6.21	5	2295.9	425	3.41	3	10.56	7	229.47	83	90.65	63.5
KLM - Shaker iOS	2011		x	3	15	1179	12.2	12.5	83.4	75.5	8.6	9	7999.5	4604	3.67	4	3	3	155.5	155.5	566.5	566.5
KLM - Shaker Android	2011	x		10	56	981	2.53	2	20.8	15	6.47	4	2682.13	240	3.9	5	6.67	5	310.56	313	108.33	56
Lets Meet - Multi Event app iOS	2012		x	227	536	27346	4.61	3	30.48	18	7.04	5	8949.17	493.5	3.38	4	5.46	4	61.32	54	185.54	67
Lets Meet - Multi Event app Android	2012	x		445	2691	52076	4.73	4	30.08	21	7.95	6	7393.82	675	2.02	2	9.86	7	40.34	7	149.42	91.5
Lets Meet - Multi Event app New iOS	2014		x	49	243	2905	2.69	2	18.78	14	6.39	6	1801.01	576	3.44	3	2.96	3	55.32	53	35.16	29
Niehe - Ranger Dierenjournaal iOS	2013		x	16	51	807	2.62	2	11.86	9	3.71	3	314	81	3.33	2.5	3	3	52.83	49	37	19

Niehe - Ranger Dierenjournaal Android	2013	x		17	90	1284	2.85	2	15.66	12	6.02	5	2236.32	250	2.94	3	9.81	7	254.81	95	76.94	50
Nulaz iOS	2010		x	149	782	40087	5	3	26.56	16.5	5.82	4	5745.33	325	3.73	4	9	5.5	78.73	75.5	535.64	213.5
Nulaz Android	2010	x		131	640	17069	4.37	3	28.66	17	7.12	4	11997.22	279	3.48	3	8.82	7	209.22	86	134.41	72
OCR - Printer App iOS	2010		x	13	6	1437	8.4	6	78.4	48	14.6	15	21813.4	10800	2	2	2	2	50	50	296	296
OCR - Printer App Android	2010	x		27	181	3461	7.29	4	32.16	22	8.05	6	11259.06	612	1.95	2	9.86	8	65.57	12	153.33	91
Republic-M - Ferinject Ipad iOS	2010		x	9	39	969	5.53	4	29.47	30	5.12	5	1412.18	1080	3.5	4	2.75	2	109.25	126	74.25	38.5
Republic-M - Ferinject iOS	2010		x	16	35	1917	5.53	4	32.32	14	4.53	2	4622.58	72	3.33	4	3.33	2	99.67	123	127.33	32
Republic-M - Ferinject Android	2010	x		8	26	769	5.38	5	43.77	44	6.77	7	3958.15	1274	4.25	5	8.63	5.5	285.38	387	96.13	56.5
Schiphol - MTS iOS	2010		x	175	1585	41323	4.67	3	26.58	17	5.43	4	4700.93	275	3.51	4	5.68	4	62.75	62	254.3	78
Schiphol - MTS Android	2010	x		205	2064	33932	4.5	18	26.11	18	7.06	5	6640.4	432	4.59	4	14.99	12	267.39	144	207.23	96
Schiphol - Safety & Security Zakboek iOS	2012		x	35	321	4792	4.13	3	20.56	13	5.8	5	2800.61	225	4	5	3.07	2	121.71	146.5	54	30.5
Schiphol - Safety & Security Zakboek Android	2012	x		28	296	5084	3.32	3	19.68	15	6.71	5	4223.37	350	2.79	2	11.41	11	162.94	66	145.41	98
The New Motion iOS	2014		x	82	383	5563	3.39	2	16.91	12	4.58	3	1319.6	108	3.86	5	3.1	2	62.51	75	50.78	25
The New Motion Android	2014	x		51	375	5877	3.48	2	23.11	16	6.88	4	5556.64	256	2.76	2	11.89	8.5	123.07	15.5	122.33	68.5
Voedingscentrum - Eetwijzer iOS	2012		x	22	169	3356	4.42	3	26.78	19	7.62	5	5657.06	468	3.69	4	4.08	4	120.62	132	198	148
Voedingscentrum - Eetwijzer Android	2012	x		47	300	5549	4.01	3	26.21	17	7.54	5	6663.23	468	3.86	3.5	9.25	8	228.28	110.5	145	87
VGZ - Mindfulness iOS	2013		x	42	241	3567	3.54	3	23.45	17.5	5.27	4	2140.1	256	5	5	3.88	2	65.12	61	58.24	27
VGZ - Mindfulness Android	2013	x		61	534	7865	3.48	2	20.88	16	6.13	4	2864.58	320	2.91	3	14.95	11	145.51	81	138.35	77
Victron - LED App iOS	2013		x	12	51	832	3.28	2	19.32	15	5.68	5	1636.8	250	3.75	3.5	3.75	2	52.88	52	77.63	37.5
Victron - LED App Android	2013	x		27	223	3027	3.17	2	18.41	15	5.1	4	1347.36	224	3.42	2	13.37	12	157.79	46	151.53	95

Victron - VRM iOS	2012		x	59	321	5505	4.69	3	25.59	16	6.7	4	5800.7	190	4.64	5	4.41	3	72.82	77.5	59.05	32
Victron - VRM Android	2012	x		81	793	9434	3.3	3	17.32	14	5.39	4	2044.17	240	2.76	2	12.1	8	140.92	26	129.96	72
Only for iOS																						
Ahold - iOS	2012		x	37	134	3884	4.03	2	19.62	10.5	3.85	3	796	65.5	3.4	4	3.2	2	103.27	123	68.87	46
Bast iOS	2009		x	7	70	1586	4.85	5	21.79	24	6.4	7	1569.47	1029	4	4	4.5	5	67.17	71	254.17	282.5
Best Pong Ever iOS	2008		x	5	13	285	4	4	29.5	29.5	3.5	3.5	685	685	3.33	4	3.67	4	110	130	57	56
Ditzo iOS	2011		x	17	132	5746	12.64	6	71.07	44	7.45	4	18827.45	684	3.86	4	2.71	2	124.42	134	309.29	272
Funda iOS	2010		x	24	113	2588	4.4	3	29.6	20.5	6.88	6	3417.5	772	4.08	4	4.42	3.5	131.83	134	131.58	86.5
Gemeente Utrecht iOS	2013		x	36	254	4952	3.98	3	23.38	12	6.18	4	5105.39	176	3.6	4	4.53	2	105.93	123	121	52
Heineken - 5s iOS	2013		x	64	173	5546	3.36	3	21.74	18	7.44	6.5	2823.02	684	2.71	2	2.84	2	42.84	31	46.21	17
Heineken - MTBA iOS	2013		x	59	263	6756	3.84	3	22.54	16	5.85	5	4411.09	325	2.17	2	3.72	2	36.83	33	92.25	39.5
Heineken - Virtual Tag iOS	2013		x	45	294	4161	3.46	3	20.37	15	5.93	4	2829.12	312	2.77	2	5.46	2	48.46	54	146.21	63.5
iSoccer iOS	2011		x	3	5	199	18	13	44	42	5	5	1540	1050	4	4	2	2	124	124	171	171
KiloMeter iOS	2010		x	12	100	2079	3.5	2	26.79	13.5	4.92	2	5497.29	82	3.2	4	2.8	3	110	133	107.4	68
Playground iOS	2009		x	6	69	1774	7.53	4	42.37	21	8	4	12147.63	144	4	4	5.75	6	147.5	150	427.5	376.5
Pokerface iOS	2010		x	8	72	1235	3	2	23.45	15	5.1	3.5	1716.05	249	4	4	4.33	4	140	142.5	169.83	159.5
Transavia - Commercial App iOS	2011		x	63	480	8148	4.07	2	21.39	12	4.36	3	2079.76	93	3.35	4	4.18	3	102.09	124.5	115.21	68
Transavia - Crew App iOS	2012		x	96	598	26877	4.19	2	23.64	14	6.47	4	6988.05	240	3.61	4	3.86	3	106.66	129.5	124.36	60
Tracks	2009		x	7	18	414	2.33	2	18.67	18	5	3	900	162	3.75	4	3	3	116.75	130	71.5	37
Only for Android																						
Crap App Android	2009	x		131	640	17069	4.37	3	28.66	17	7.12	4	11997.22	279	3.48	3	8.82	7	209.22	86	134.41	72
Date Picker Android	2009	x		17	132	306	3.69	3	21.07	16	5.69	4	3063	336	2.73	2	7.73	8	60.73	17	27.73	9
Doorbel Android	2009	x		11	34	404	2.67	2	12.67	4	4.67	3	1528.8	36	3	2.5	7.83	5	121.33	35	55.67	47
Heineken - Holland Heineken House Android	2009	x		53	430	12610	4.77	3	33.88	27	7.78	5	6906.77	500	3.85	5	14.51	16	251.46	318	252.29	212
MyLaps - Event Results Android	2012	x		38	239	3007	3.45	3	20.64	15.5	6.73	5	5538.63	398	2.16	2	8.42	9	77.96	11	108.04	76
SMS My Position Android	2009	x		11	150	1754	4.14	3	25.94	19	4.54	3	1904.1	153	1.8	1	12.8	6	84.2	40	283.6	290

Transavia - WOW Android	2012	x	55	426	6648	3.51	3	20.46	15	6.57	5	3252.92	300	3	2.5	12.3	8	197.9	58	129.9	67

Table A.1: This table represents an overview of the mean and median values for all metrics for all app

Appendix B

Metric Descriptives

Descriptives					
platform				Statistic	Std. Error
iOS	SLOC	Mean		8427.51	1600.538
		95% Confidence Interval for Mean	Lower Bound	5201.84	
			Upper Bound	11653.18	
		5% Trimmed Mean		7124.72	
		Median		4222.00	
		Weighted Average	5%	323.00	
			10%	761.00	
			25%	1680.00	
			50%	4220.00	
			75%	8415.50	
			90%	27064.60	
			95%	38488.00	
		Variance		115277539.437	
		Std. Deviation		10736.738	
		Minimum		199	
		Maximum		41323	
		Range		41124	
		Interquartile Range		6736	
		Skewness		1.922	.354
		Kurtosis		2.826	.695
Android	SLOC	Mean		11879.23	2286.538
		95% Confidence Interval for Mean	Lower Bound	7232.42	
			Upper Bound	16526.03	
		5% Trimmed Mean		10532.47	
		Median		5549.00	
		Weighted Average	5%	445.60	
			10%	703.40	
			25%	3007.00	
			50%	5549.00	
			75%	17069.00	
			90%	35249.00	
			95%	43672.00	
		Variance		182988984.417	
		Std. Deviation		13527.342	
		Minimum		404	
		Maximum		52076	
		Range		51672	
		Interquartile Range		14062	
		Skewness		1.488	.398
		Kurtosis		1.406	.778

Table B.1: Calculated statistical values for the Source Lines of Code in a project

Descriptives					
platform				Statistic	Std. Error
iOS	CC	Mean		4.77	.071
		95% Confidence Interval for Mean	Lower Bound	4.63	
			Upper Bound	4.91	
		5% Trimmed Mean		3.83	
		Median		3.00	
		Weighted Average	5%	2.00	
			10%	2.00	
			25%	2.00	
			50%	3.00	
			75%	5.00	
			90%	9.00	
			95%	14.00	
		Variance		36.335	
		Std. Deviation		6.028	
		Minimum		2	
		Maximum		145	
		Range		143	
		Interquartile Range		3	
		Skewness		7.059	.029
		Kurtosis		93.276	.058
Android	CC	Mean		4.40	.043
		95% Confidence Interval for Mean	Lower Bound	4.32	
			Upper Bound	4.49	
		5% Trimmed Mean		3.81	
		Median		3.00	
		Weighted Average	5%	2.00	
			10%	2.00	
			25%	2.00	
			50%	3.00	
			75%	5.00	
			90%	8.00	
			95%	11.00	
		Variance		19.233	
		Std. Deviation		4.386	
		Minimum		2	
		Maximum		206	
		Range		204	
		Interquartile Range		3	
		Skewness		12.627	.024
		Kurtosis		459.321	.049

Table B.2: Calculated statistical values for the Cyclomatic Complexity

Descriptives					
platform				Statistic	Std. Error
iOS	IFLength	Mean		19.27	.283
		95% Confidence Interval for Mean	Lower Bound	18.71	
			Upper Bound	19.82	
		5% Trimmed Mean		14.19	
		Median		9.00	
		Weighted Average	5%	2.00	
			10%	3.00	
			25%	5.00	
			50%	9.00	
			75%	20.00	
			90%	43.00	
			95%	67.00	
		Variance		1125.434	
		Std. Deviation		33.547	
		Minimum		1	
		Maximum		1012	
		Range		1011	
		Interquartile Range		15	
		Skewness		7.393	.021
		Kurtosis		106.626	.041
Android	IFLength	Mean		16.02	.154
		95% Confidence Interval for Mean	Lower Bound	15.72	
			Upper Bound	16.32	
		5% Trimmed Mean		12.75	
		Median		8.00	
		Weighted Average	5%	2.00	
			10%	2.00	
			25%	4.00	
			50%	8.00	
			75%	19.00	
			90%	38.00	
			95%	55.00	
		Variance		526.532	
		Std. Deviation		22.946	
		Minimum		1	
		Maximum		1041	
		Range		1040	
		Interquartile Range		15	
		Skewness		7.764	.016
		Kurtosis		205.644	.033

Table B.3: Calculated statistical values for the Method Length for the Information Flow

Descriptives					
platform				Statistic	Std. Error
iOS	IFFan	Mean		4.37	.045
		95% Confidence Interval for Mean	Lower Bound	4.28	
			Upper Bound	4.46	
		5% Trimmed Mean		3.61	
		Median		3.00	
		Weighted Average	5%	1.00	
			10%	1.00	
			25%	1.00	
			50%	3.00	
			75%	5.00	
			90%	10.00	
			95%	13.00	
		Variance		28.378	
		Std. Deviation		5.327	
		Minimum		1	
		Maximum		78	
		Range		77	
		Interquartile Range		4	
		Skewness		4.441	.021
		Kurtosis		31.932	.041
Android	IFFan	Mean		5.15	.042
		95% Confidence Interval for Mean	Lower Bound	5.07	
			Upper Bound	5.24	
		5% Trimmed Mean		4.35	
		Median		3.00	
		Weighted Average	5%	1.00	
			10%	1.00	
			25%	1.00	
			50%	3.00	
			75%	7.00	
			90%	12.00	
			95%	16.00	
		Variance		34.571	
		Std. Deviation		5.880	
		Minimum		1	
		Maximum		81	
		Range		80	
		Interquartile Range		6	
		Skewness		3.261	.018
		Kurtosis		18.713	.035

Table B.4: Calculated statistical values for the Fan for the Information Flow

Descriptives					
platform				Statistic	Std. Error
iOS	infoFlow	Mean		3871.31	338.560
		95% Confidence Interval for Mean	Lower Bound	3207.69	
			Upper Bound	4534.93	
		5% Trimmed Mean		622.89	
		Median		60.00	
		Weighted Average	5%	3.00	
			10%	4.00	
			25%	9.00	
			50%	60.00	
			75%	504.00	
			90%	3402.00	
			95%	9800.05	
		Variance		1602197200.894	
		Std. Deviation		40027.456	
		Minimum		1	
		Maximum		3.E+06	
		Range		2543111	
		Interquartile Range		495	
		Skewness		36.398	.021
		Kurtosis		1872.221	.041
Android	infoFlow	Mean		3933.83	241.170
		95% Confidence Interval for Mean	Lower Bound	3461.12	
			Upper Bound	4406.55	
		5% Trimmed Mean		912.87	
		Median		90.00	
		Weighted Average	5%	2.00	
			10%	2.00	
			25%	8.00	
			50%	90.00	
			75%	825.00	
			90%	5195.20	
			95%	12600.00	
		Variance		1130224014.699	
		Std. Deviation		33618.804	
		Minimum		2	
		Maximum		2.E+06	
		Range		2021439	
		Interquartile Range		817	
		Skewness		37.291	.018
		Kurtosis		1881.832	.035

Table B.5: Calculated statistical values for the Information Flow

Descriptives					
platform				Statistic	Std. Error
iOS	DIT	Mean		3.54	.043
		95% Confidence Interval for Mean	Lower Bound	3.46	
			Upper Bound	3.63	
		5% Trimmed Mean		3.51	
		Median		4.00	
		Weighted Average	5%	2.00	
			10%	2.00	
			25%	2.00	
			50%	4.00	
			75%	5.00	
			90%	5.00	
			95%	5.00	
		Variance		1.740	
		Std. Deviation		1.319	
		Minimum		1	
		Maximum		6	
		Range		5	
		Interquartile Range		3	
		Skewness		-.011	.081
		Kurtosis		-1.389	.161
Android	DIT	Mean		2.45	.035
		95% Confidence Interval for Mean	Lower Bound	2.38	
			Upper Bound	2.52	
		5% Trimmed Mean		2.19	
		Median		2.00	
		Weighted Average	5%	1.00	
			10%	1.00	
			25%	1.00	
			50%	2.00	
			75%	3.00	
			90%	5.00	
			95%	7.00	
		Variance		4.142	
		Std. Deviation		2.035	
		Minimum		1	
		Maximum		13	
		Range		12	
		Interquartile Range		2	
		Skewness		1.792	.042
		Kurtosis		2.976	.085

Table B.6: Calculated statistical values for the Depth of Inheritance Tree

Descriptives					
platform				Statistic	Std. Error
iOS	CBO	Mean		5.41	.108
		95% Confidence Interval for Mean	Lower Bound	5.20	
			Upper Bound	5.62	
		5% Trimmed Mean		4.84	
		Median		4.00	
		Weighted Average	5%	1.00	
			10%	2.00	
			25%	2.00	
			50%	4.00	
			75%	7.00	
			90%	11.00	
			95%	15.00	
		Variance		22.506	
		Std. Deviation		4.744	
		Minimum		1	
		Maximum		41	
		Range		40	
		Interquartile Range		5	
		Skewness		2.393	.056
		Kurtosis		8.536	.111
Android	CBO	Mean		9.76	.182
		95% Confidence Interval for Mean	Lower Bound	9.40	
			Upper Bound	10.12	
		5% Trimmed Mean		8.65	
		Median		7.00	
		Weighted Average	5%	1.00	
			10%	1.00	
			25%	3.00	
			50%	7.00	
			75%	13.00	
			90%	23.00	
			95%	29.00	
		Variance		92.507	
		Std. Deviation		9.618	
		Minimum		1	
		Maximum		76	
		Range		75	
		Interquartile Range		10	
		Skewness		1.958	.046
		Kurtosis		5.172	.092

Table B.7: Calculated statistical values for coupling between objects

Descriptives					
platform				Statistic	Std. Error
iOS	RFC	Mean		91.44	1.631
		95% Confidence Interval for Mean	Lower Bound	88.24	
			Upper Bound	94.64	
		5% Trimmed Mean		89.71	
		Median		75.00	
		Weighted Average	5%	31.00	
			10%	36.00	
			25%	48.00	
			50%	75.00	
			75%	136.00	
			90%	161.00	
			95%	166.00	
		Variance		2449.373	
		Std. Deviation		49.491	
		Minimum		8	
		Maximum		268	
		Range		260	
		Interquartile Range		88	
		Skewness		.499	.081
		Kurtosis		-.972	.161
Android	RFC	Mean		105.25	3.550
		95% Confidence Interval for Mean	Lower Bound	98.29	
			Upper Bound	112.21	
		5% Trimmed Mean		78.25	
		Median		13.00	
		Weighted Average	5%	1.00	
			10%	2.00	
			25%	3.00	
			50%	13.00	
			75%	86.00	
			90%	361.00	
			95%	548.00	
		Variance		35923.524	
		Std. Deviation		189.535	
		Minimum		1	
		Maximum		1376	
		Range		1375	
		Interquartile Range		83	
		Skewness		2.359	.046
		Kurtosis		6.446	.092

Table B.8: Calculated statistical values for the Response of a Class

Descriptives					
platform				Statistic	Std. Error
iOS	SLOCinFile	Mean		187.28	8.356
		95% Confidence Interval for Mean	Lower Bound	170.89	
			Upper Bound	203.67	
		5% Trimmed Mean		136.05	
		Median		79.00	
		Weighted Average	5%	9.00	
			10%	16.00	
			25%	32.00	
			50%	79.00	
			75%	215.00	
			90%	434.40	
			95%	650.40	
		Variance		141392.303	
		Std. Deviation		376.022	
		Minimum		1	
		Maximum		7627	
		Range		7626	
		Interquartile Range		184	
		Skewness		8.993	.054
		Kurtosis		124.311	.109
Android	SLOCinFile	Mean		124.59	3.380
		95% Confidence Interval for Mean	Lower Bound	117.97	
			Upper Bound	131.22	
		5% Trimmed Mean		97.26	
		Median		64.00	
		Weighted Average	5%	8.00	
			10%	13.00	
			25%	28.00	
			50%	64.00	
			75%	149.00	
			90%	295.20	
			95%	429.10	
		Variance		38116.802	
		Std. Deviation		195.235	
		Minimum		1	
		Maximum		3439	
		Range		3438	
		Interquartile Range		121	
		Skewness		6.407	.042
		Kurtosis		72.335	.085

Table B.9: Calculated statistical values for the Source Lines of Code in a file