DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Cross-Platform Smartphone Application Development with Kotlin Multiplatform

Possible Impacts on Development Productivity, Application Size and Startup Time

ANNA-KARIN EVERT

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Cross-platform Smartphone Application Development with Kotlin Multiplatform

Possible Impacts on Development Productivity, Application Size and Startup Time.

Anna-Karin Evert

Master in Computer Science
Date: July 5, 2019
Supervisor: Håkan Lane
Examiner: Elena Troubitsyna
Swedish title: Krossplattformutveckling av mobilapplikationer med Kotlin Multiplatform: möjliga effekter på produktivitet samt applikationers storlek och starttid.
School of Electrical Engineering and Computer Science

Abstract

The objective of this master's thesis has been to evaluate the Kotlin Multiplatform feature for developing cross-platform mobile applications for Android and iOS. This has been done in comparison to natively developed applications for the two platforms. The method of evaluation has been to develop a sample application natively for Android and iOS, respectively. The same sample application has then been developed using the Kotlin Multiplatform feature. Finally, the multiplatform Android application has been compared to the natively developed Android application, and the multiplatform iOS application has been compared to the natively developed iOS application. The evaluation has focused on measuring the startup time of the applications, as well as the application size of the installed sample applications, comparing the native ones to the multiplatform ones. An attempt has also been made to try to determine if there can be any productivity gains in using the Kotlin Multiplatform feature instead of doing the development natively for each of the studied platforms. For productivity, the evaluation has included measuring number of lines of code, and build time (compilation time) for the applications. The results indicate that it is possible to write less code if making use of the Kotlin Multiplatform feature. However, the results also indicate an increased build time with Kotlin Multiplatform, for both Android and iOS, and an increased startup time for Android. No indication of an increased startup time for the multiplatform iOS application could be found. As for the application sizes, the results show an increased size for the multiplatform applications.

Sammanfattning

Att skriva kod separat för varje plattform är kostsamt - tidsmässigt och ekonomiskt - och innebär att arbetet med att underhålla och testa koden behöver göras separat för varje plattform. Krossplattformsverktyg för mobilapplikationer är ett sätt att minska sådana kostnader. Syftet med detta examensarbete har varit att studera och utvärdera ett sådant verktyg - nämligen Kotlins feature för krossplattformsutveckling av mobilapplikationer - för plattformarna Android och iOS. Metoden som använts har varit att utveckla en exempelapplikation, först plattformsspecifikt för Android och iOS, och sedan med hjälp av Kotlin Multiplatform för båda plattformarna. En jämförelse mellan de plattformsspecifika apparna och krossplattformapparna har sedan gjorts för respektive plattform. De apsekter som undersökts är de färdiga apparnas starttid, apparnas storlek, samt om det går att se någon indikation på att produktiviteten ökar om Kotlin Multiplatform används för utveckling istället för att utveckla applikationen separat för varje plattform. Produktiviteten har mätts i antal rader kod, samt i kompileringstiden för projekten. Studiens resultat visar att det är möjligt att skriva färre rader kod med Kotlin Multiplatform, än med plattformsspecifik utveckling. Resultaten visar dock också en ökning av kompileringstid för krossplattformsprojekten jämfört med de plattformsspecifika, samt en ökad starttid för Android. Resultaten visade ingen signifikant skillnad i starttid mellan iOS-apparna. Applikationernas storlek är större för de båda krossplattformsapparna än för de plattformsspecifika.

Contents

Chapter 1

Introduction

This chapter begins by introducing the subject of cross-platform smartphone applications. Then the problem and research questions are defined, and the motive behind the study presented. Finally, this chapter outlines the project's scope and delimitations.

1.1 Introduction

Smartphone application development is a fast-changing field, with new technology and frameworks frequently introduced, and new areas of use for the smartphone invented continuously. The two most common operating systems on smartphones today are Android and iOS. For Android, Java was the main programming language used, but since Google announced official support for Kotlin as a programming language for Android applications, in october 2017, using Kotlin to develop applications has grown in popularity. For iOS, the development is done in either Objective-C or, increasingly so, in Swift.

A large part of the development of applications is done separately for each individual platform. That is, if an application is meant to target multiple platforms, such as Android and iOS, one application is developed natively for each platform. This means approximately twice the amount of work to make an application targeting two platforms, than if the same application could be developed only once and then run on both platforms, i.e. with cross-platform development.

Sharing of some or all code between platforms, can be done with cross-platform tools. The objective of using such tools to develop applications targeted at multiple platforms, is to minimize development time and costs, and to make it easier to maintain the code base [26].

Cross-platform tools have existed for a number of years, but when studied within academia, it seems that it is difficult for a cross-platform tool to produce applications that reaches the performance, or offers the same access to platform-specific features, of those from native development [2, 23, 31]. However, the application performance of multiplatform applications depends completely on the functionality of the cross-platform development tool used.

Kotlin Multiplatform is a fairly new tool, that has been a part of the Kotlin programming language since version 1.2, that was released in late 2017 [14]. There exist a number of other cross-platform tools. One example is Xamarin, for which the code is written in C#, and then compiled for each of the different platforms Android, iOS and Windows phone [24]. A problem with such tools, from an application developing company's point of view, is that if they currently employ Kotlin and Swift/Objective C developers, this means a transition into working in a new programming language and environment, which can be costly. In such a case, the Kotlin Multiplatform feature could be an easier transition, making use of the specialized knowledge in Kotlin and Swift/Objective C that already exists at the company.

This thesis aims to study the Kotlin Multiplatform feature, since this, to the best of our knowledge, has not yet been done within academia.

1.2 Problem Statement

According to IDC, the market shares for smartphone operating systems were 86.8% for Android and 13.2% for iOS, in the third quarter of 2018 [12]. At many companies today, applications are still developed natively for each targeted platform. This imposes economical challenges, as well as challenges regarding developer productivity and code maintainability [26].

The aim of this thesis is to investigate if there are any trade-offs of using Kotlin Multiplatform instead of native development, and if the advantages are greater than the disadvantages.

Cross-platform development with the Kotlin Multiplatform feature will be evaluated, for Android and iOS applications. Firstly, it will be explored if and how it is possible to use the Kotlin multiplatform feature to create an application that can be run on both Android and iOS. Secondly, these applications will be compared with their equivalent native applications.

One focus of the evaluation will lie on comparing the user-perceived performance of applications developed with the Kotlin multiplatform tool with that of native development. Another comparison that will be made is that of application sizes, to see if using Kotlin Multiplatform will increase the sizes of the applications. A further aspect of cross-platform development is how much work can be saved by abandoning native development in favour of cross-platform development. Thus, the second focus of this thesis is to study possible productivity gains of using Kotlin multiplatform. The aim is to answer the following questions:

1. What are the effects of using Kotlin Multiplatform, when it comes to the user-perceived performance of the applications?
2. What are the effects of using Kotlin Mulitplatform, when it comes to the size of the installed application?
3. What are the effects of using Kotlin Multiplatform, when it comes to developer productivity?

1.3 Motivation

Since the cross-platform tool Kotlin Multiplatform has, to the best of our knowledge, not yet been studied within academia, this projects aims to contribute with an evaluation of said tool. Since no studies at all could be found on Kotlin Multiplatform, there is a wide choice of how and what to evaluate. This thesis will evaluate Kotlin Multiplatform regarding user-perceived performance, application size and development productivity. User-perceived performance and application size of cross-platform applications have been studied before, but in evaluations of other cross-platform tools than Kotlin's. Development productivity does not seem to be a common evaluation criteria for cross-platform tool evaluations, which adds to the news value of this study.

1.4 Scope and Delimitations

The metrics to be studied in this thesis is delimited to lines of code, build time, application size and application startup time. Thus, there are several measurements that will not be included, such as pure performance metrics.

The methodology of using a sample application to evaluate the Kotlin Multiplatform feature, entails a delimitation to studying the impacts of using the feature, for that particular sample application. The sample application is chosen in a way such that it should contain characteristics of real-world applications, so that some general conclusions can be drawn from the results. These characteristics are sending and fetching data over the Internet, parsing fetched data, graphical user interface components, multiple views and some underlying business logic. Due to the limitation in time of this project, the sample application only covers a part of the most common features of real-world applications, while others are left out.

Lastly, a comparison to other cross-platform tools will not be included in this thesis, since the objective is not to investigate how Kotlin Multiplatform measures up to other cross-platform tools, but how it measures up to native development.

Chapter 2

Background

This chapter presents background knowledge necessary for this thesis. First, cross-platform development and its paradigms, with their advantages and disadvantages, are described. Following this, the Kotlin Multiplatform feature is presented. Then, related work with respect to cross-platform smartphone applications and development productivity, is summarized. Finally, basic statistical theory is presented.

2.1 Cross-platform Development

Native mobile application development is when applications are developed separately for each platform, using the platform-specific architecture and development support. Native applications allow for full access to all device hardware and platform specific features [27]. However, the downside of native application development is that all code needs to be written once for each platform the application is targeting, since it will only work for this one platform.

A cross-platform mobile application is an application that is targeted to run on more than one platform, and for which a part of its code is shared between platforms [21]. The objective of using cross-platform tools to develop applications targeted at multiple platforms, is to minimize development time and costs, and to make it easier to maintain and test the code base [26].

2.1.1 Paradigms

2.1.1.1 Web Applications

A web application is an application that is not installed on the device but instead executed in the device's web browser. All business logic of the application is run on a server, and the device only hosts the user interface and user validation logic [27].

One advantage of this approach is that the user interface can be reused across platforms. Others are that a web application does not need to be installed, it can be run on any device with a web browser, and updates of the applications do not need to be installed on the device. However, this also means that a web application is not distributable through application stores such as Play store or App store, which can be a disadvantage in the way that it might be more difficult to reach potential users [27].

One of the main drawbacks of the web approach is that such applications do not have access to platform-specific features and hardware, such as the GPS and the camera [11]. Another disadvantage of the web paradigm is that, due to the reliance on network and connections, the performance might be insufficient. A further challenge of this approach is the need to adopt the application for a variety of screen resolutions, and that the developer has less control of the look of the application since the rendering is dependent on the web browsers [27].

2.1.1.2 Hybrid Applications

A hybrid mobile application is, as the name implies, a hybrid between a native and a web application. An application within the hybrid paradigm, is developed with web technologies, but executed inside a native container on the device [11]. The web browser of the device renders and displays the application, and the application can access the device-specific features through APIs in an abstraction layer [27].

The main advantage of the hybrid approach is that applications can use the computational power of the device and access native platform features, while also reusing the user interface. A hybrid application needs to be downloaded, and is thus distributable through application stores [27].

One of the main disadvantages of this approach is that the performance is not on the level of native applications. Another is that although the user interface can be reused, it will lack the native look and feel, and work will still have to be put into adjusting it specifically to each platform [27].

2.1.1.3 Interpreted Applications

Interpreted applications use native execution, and are interpreted at runtime using an interpreter. As with the web approach, the platform-specific features are accessed through APIs in an abstraction layer. This accessibility of the device features is an advantage of the interpreted approach. Another advantage is that interpreted applications are accessible through application stores [27].

A disadvantage of the interpreted approach is that the performance might not be on par with that of native applications, since the interpretation occurs at runtime [27]. Furthermore, the success of multiplatform applications within this paradigm, is dependant on the tool used for development, and what feature set that tool provides [32].

2.1.1.4 Generated Applications

As with the interpreted approach, applications developed with the generated approach are executed within the native environment of the platform. The idea of this approach is that a cross-compiler is used to compile the application into native binary code for each targeted platform [27, 32]. Depending on which native binaries are supported by the tool (i.e. the compiler), this means that code could be written once and then used on several platforms.

As with interpreted applications, the success of this approach of cross-platform development, is entirely dependent on the tool used. Does the compiler provide efficient code for all supported platforms? Is it reliable? Theoretically, however, there can be important advantages with this approach. The first is that it provides access to all platform-specific features such as device hardware and native interface components. The second is that performance can be on par with the performance of native applications, and it is superior to the interpreted approach in that sense [27].

One drawback of the cross-compiled approach is that some code cannot be reused. The kind of code that cannot be reused across platforms are the user interface and the code for the platform specific features, such as camera access and local notifications [27].

2.2 Kotlin Multiplatform

Development of the Kotlin programming language started in 2010, and the first stable version was released in 2016 [30]. Kotlin is developed by Jetbrains, and is officially supported by Google, as a language for Android developement [19], since October 2017 [7]. According to GitHub [6], Kotlin was the fastest growing programming language of 2018, in terms of usage in GitHub repositories.

The Kotlin Multiplatform feature has been a part of the Kotlin programming language since version 1.2, that was released in late 2017 [14], and it belongs to the Generated Paradigm (see section 2.1.1.4). The Kotlin compiler has support for compiling code into native binaries. Consequently, Kotlin applications can be run in environments that do not support virtual machines to run applications, for example smartphones that run iOS. It is upon this technique the multiplatform feature is built [15]. The supported platforms for native binaries are Android NDK, iOS, Linux, MacOS, Windows and WebAssembly. Kotlin also has support for compiling to Javascript [13].

In order to preserve the ability to use platform-specific features, the Kotlin Multiplatform feature does not aim for all code to be shareable between platforms [14]. This is in accordance with the earlier mentioned drawbacks of the generated paradigm.

A Kotlin multiplatform project consists of the building blocks *targets*, *compilations* and *source sets*. A target is the part of the project that is responsible for building, testing and packaging the application for a certain platform. In a multiplatform project, there are generally more than one target, since such projects target more than one platform. A compilation is, as the name implies, the compilation of the Kotlin source code. A target has at least one compilation, but can also have multiple ones; for example, one compilation for production, one for testing, etc. The source files, together with their associated resource files, dependencies and language settings, are grouped into source sets. The source sets are platform-agnostic, which means that the set of sources are not bound to be specific for one platform. However, the entire source set does not have to be shared either. Instead, some parts of the source set can be shared between platforms, and some, that are to be compiled for only one platform, can contain platform-specific dependencies. A source set can have common dependencies, i.e. libraries or other projects that the set of sources are explicitly stated to need access to [13].

A Kotlin Multiplatform targeting native, for example to be run on iOS, is compiled to an artifact of the type *.klib. The *.klib artifact can be resolved by Kotlin itself as a depency, but it cannot be executed, nor can it be used as a library. To compile the executable native files, Kotlin Multiplatform provides factory methods that can declare which binaries are to be created, and configure the compilations of those native binaries [16].

In a Kotlin multiplatform project targeting iOS and Android, for Android the code is compiled into Java bytecode to be run on the Java Virtual Machine [19]. For iOS, on the other hand, the code is compiled into a binary Objective-C framework, which is then interoperable with Swift.

When sharing code between platforms using the Kotlin Multiplatform feature, the idea is to create a shared library of functions and classes, written in Kotlin, which the different platform-specific projects have access to. This library includes one part with common code, and one part for each platform, with platform-specific implementations if and when needed. The common code is exposed to the platform-specific projects, for example an Android project and an iOS project. A schematic description of this can be seen in Figure 2.1.

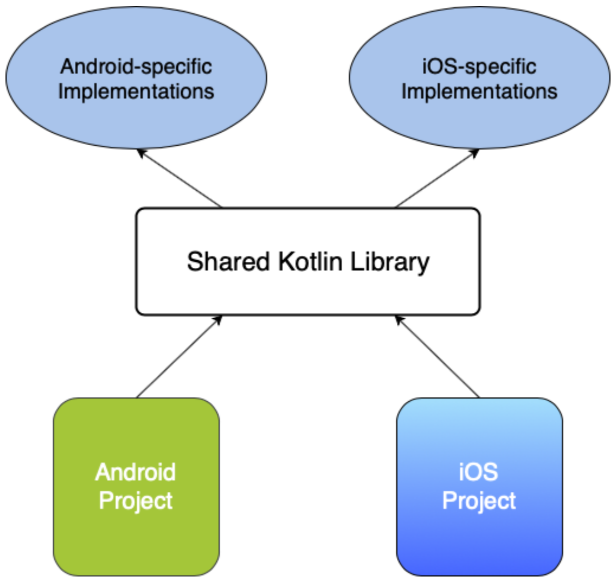

Android-specific Implementations
iOS-specific Implementations
Shared Kotlin Library
Android Project
iOS Project

Figure 2.1: The structure of a Kotlin multiplatform project targeting Android and iOS.

In the common code of a Kotlin Multiplatform project, there can be entire functions and classes that are shared by different platforms. However, Kotlin Multiplatform also enables the common code to define functions and classes that depend on platform-specific implementations. This is done with Kotlin's *expect* and *actual* declarations. The *expect* declaration is used in the common code, to let the compiler know that we expect a corresponding *actual* declaration and implementation, which should be platform-specific, for each targeted platform [17]. Figure 2.2 shows an example of the expect/actual declarations for a simple function that returns a string containing the platform name. Kotlin Multiplatform provides access to a native standard library for the targeted platforms. This can be seen in the iOS-specific implementation in the example in Figure 2.2, where an import is made to gain access to functionality from the Swift standard library [18].

```
// Common code
expect fun platformName(): String
```

```
// Android-specific declaration
actual fun platformName(): String {
    return "Android"
}
```

```
// iOS-specific declaration
import platform.UIKit.UIDevice

actual fun platformName(): String {
    return UIDevice.currentDevice.systemName() +
        " " +
        UIDevice.currentDevice.systemVersion
}
```

Figure 2.2: An example of expect/actual function declarations for Android and iOS. (Example taken from [14].)

Figure 2.3 shows an example of how the common code can declare an expected class, which then has its actual implementation for each platform. In the mentioned example, the cross-platform problem of UI elements, such has images, often being platform-specific, is solved for iOS and Android, by using the *actual typealias* declaration. This declaration is of use when there exist platform libraries containing a class with the desired properties, for one or all the targeted platforms. If a programmer wishes to implement her own platform-specific classes, this is done with expect class/actual class declarations, much similar to the functions seen in Figure. The compiler verifies that each *expected* declaration has a corresponding *actual* declaration, for each targeted platform [17].

```
// Common code
expect class Image
```

```
// Android-specific declaration
import android.graphics.Bitmap

actual typealias Image = BitMap
```

```
// iOS-specific declaration
import platform.UIKit.UIImage

actual typealias Image = UIImage
```

Figure 2.3: An example of expect/actual class declarations for Android and iOS. (Example taken from [14].)

Kotlin provides some multiplatform libraries to make it easier to develop software with Kotlin Multiplatform. These libraries include implementations of common programming tasks, such as HTTP communication, serialization, coroutines and server/client programming [14]. If these libraries are used, it means that the programmer can just use the functionalities supported by the library, and does not need to write the platform-specific implementations herself (as has been done in Figure 2.2).

2.3 Related Work

2.3.1 Cross-platform Development of Smartphone Applications

Since Kotlin was released in 2016, it can still be considered a young programming language. The Kotlin Multiplatform feature is even younger, and still in an experimental phase, which could be the reason why no studies covering Kotlin multiplatform have been found. In fact, mobile development as a whole is also a fairly new phenomenon, and the study thereof is as well. As is the study of cross-platform mobile development. Yet, a fair amount of studies covering related topics have been found during the literature study for this project.

Some studies focus, at least partly, on describing the existing paradigms of cross-platform development [3, 5, 11, 27, 32], and list the Web, Hybrid, Interpreted and Generated paradigms as the four main categories. Although the naming of paradigms differ between the studies, the definitions are mostly the same, with the exception of El-Kassas et al. [5] that introduce three new paradigms. These do not seem to have been adopted by other scientists yet, though.

Most studies aim to evaluate either cross-platform paradigms [5, 23, 27], cross-platform tools, for example Xamarin or Phonegap [3, 4, 26, 29], or both paradigms and tools [2, 31, 32]. But the evaluation criteria differ, and can be divided into two main categories, which are evaluation of the *paradigm or tool itself* and evaluation of *applications* developed using a certain paradigm or tool. Evaluating a tool or paradigm is more qualitative, stating pros and cons of the different paradigms, and describing the tools in terms of license costs, supported programming languages, development experience, etc. [3, 5, 11, 26, 27, 32]. Evaluating applications is more quantitative, focusing on the final product. In those studies, a sample application is developed, and that application is then evaluated in terms of, for example, quality measurements such as CPU usage, energy consumption, memory usage, application size and/or response time [2, 3, 4, 11, 29, 31].

Evaluation of cross-platform development tools and paradigms is usually done in relation to something else. That is, the qualitative features or the quantitative measurements, are compared to those of other paradigms or tools [2, 3, 4, 11, 23, 29, 31, 32], and/or to those of native application development [2, 4, 11, 23, 26, 29, 31]. Furthermore, some studies measure the time to complete certain tasks, such as sorting a list with quicksort [2, 3, 4, 29, 31], while others focus on developing a sample application that has a more general purpose [3, 11, 31, 32]. Although, if it is the latter case, there are almost always some measurements of specific tasks as well, such as startup time, or memory usage in different stages of the application life cycle.

In studies that adopt the method of developing a more general application with different techniques and then comparing those applications, the chosen sample application is different for each study. To our knowledge, there exists no state-of-the-art sample application. Dalmasso et al. [3] have developed an application with a screen displaying three buttons, each of which uses a different technology to send HTTP requests. When the response for the request is received, the application parses the JSON results and displays them to the user. Xanthopoulos and Xinogalos [32] have chosen a sample application showing an RSS feed with news, and thus also including HTTP requests, and then parsing and displaying the data. Neither Dhillon and Mahmoud [4] and Heitkötter et al. [11] describe their choices of sample application functionalities in detail.

A few studies have the objective of creating some sort of framework for choosing the most appropriate paradigm or tool for cross-platform development, depending on functional or non-functional requirements [3, 4].

For mobile cross-platform tools that have been around for a while, there is the possibility of evaluating existing code bases when comparing the quality of the tools and applications. Mercado et al. [23] use existing code bases to evaluate cross-platform tools, and Habchi et al. [9], Reimann et al. [28] and Hecht et al. [10] use existing code bases to evaluate the quality of mobile applications.

Evaluation of software can be done dynamically by measuring performance such as response time, or statically, by looking at the source code. A number of studies within the area of mobile development, have used the latter way of evaluating cross-platform applications. Habchi et al. [9], Hecht et al. [10], Martinez and Lecomte [21] and Reimann et al. [28] all search repositories for mobile applications, for so called code-smells, or anti-patterns. Reimann et al. [28] have defined anti-patterns for Android applications, and developed a tool that can detect them automatically. Habchi et al. [9] are focused on comparing the occurrences of anti-patterns in iOS applications, with those of Android applications. Hecht et al. [10] track the occurrences of anti-patterns in Android projects over time. Martinez and Lecomte suggest a method for comparing the quality of Android applications written in Java and Kotlin, respectively, using occurrences of anti-patterns as the measurement [21].

Mercado et al. [23] use Natural Language Processing to classify reviews of native, hybrid and interpreted/generated applications, with the conclusion that hybrid applications are more criticised than applications of the other two approaches.

Table 2.1 summarizes the main focus areas of previous studies, and shows the contributions of this study.

Focus area	**References**	**Addressed in this thesis**
Evaluation of cross-platform paradigms	[2, 5, 23, 27, 31, 32]	No
Evaluation of cross-platform tools	[2, 3, 4, 26, 29, 31, 32]	Yes, Kotlin Multiplatform
Evaluation of Kotlin Multiplatform	-	Yes
Qualitative evaluation of cross-platform tools	[3, 5, 11, 26, 27, 32]	No
Quantitative evaluation of cross-platform tools	[2, 3, 4, 11, 29, 31]	Yes
Creating a framework for selecting cross-platform approach or tool	[3, 4, 27]	No
Evaluation of performance of cross-platform applications	[2, 3, 4, 29, 31]	Partly, by measuring startup time
Evaluation and comparison of different cross-platform tools and/or paradigms	[2, 3, 4, 11, 29, 31, 32]	No
Evaluation of cross-platform applications or tools in comparison with natively developed applications	[2, 4, 11, 23, 26, 29, 31]	Yes, comparing Kotlin Multiplatform applications for Android and iOS with natively developed applications for Androis and iOS
Evaluation of cross-platform tools using a sample application	[3, 4, 11, 32]	Yes
Evaluation of cross-platform tools with respect to development productivity	-	Yes

Table 2.1: Summary of previous studies, showing the focus areas for this thesis.

2.3.2 Development Productivity

Developer productivity can be a vague concept, and to measure and quantify it a difficult task. Lines of code is a metric that has been commonly used to measure productivity. It is then either used by itself [22], or as a variable in a formula quantifying productivity, such as:

$$Productivity = \frac{Lines\,of\,code}{Effort} \tag{2.1}$$

(p. 256, [25])

2.4 Statistical Theory

According to Lilja [20], there are two main formulas for calculating the means of software performance experiments. They are the arithmetic mean and the harmonic mean. The latter is preferably used when the metric of the measurment is some sort of rate, while the arithmetic mean is appropriate to use on for example execution times.

The arithmetic mean is defined as

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{2.2}$$

(p. 27, [20])

where $\overline{x}$ is the mean value, and the x_i:s are the measured individual values.

Another important metric is standard deviation. Standard deviation shows how the measured values are distributed; that is, how much they vary from the mean value. The standard deviation is defined as

$$s = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \overline{x})^2}{n-1}} \tag{2.3}$$

(p. 39, [20])

Two-sided confidence intervals are appropriate when comparing two alternatives, such as performance before or after a new hardware is introduced. A two-sided interval for a random variable x of Gaussian distribution, with $\mu = 0$ and $\sigma = 1$, is calculated as

$$c_1 = \overline{x} - z_{1-\frac{\alpha}{2}} \frac{s}{\sqrt{n}} \tag{2.4}$$

$$c_2 = \overline{x} + z_{1-\frac{\alpha}{2}} \frac{s}{\sqrt{n}} \tag{2.5}$$

(p. 49, [20])

The interval between the values c_1 and c_2 is the confidence interval. The probability that the mean value lies within this interval, is $1 - \alpha$ [20].

Chapter 3

Methodology

This chapter presents the methodology of the study. It begins with describing the sample application that will later be analyzed. From this, it moves on to describing the metrics used. Finally, the setup and realisation of the data collection is described.

3.1 Sample Application

To evaluate the Kotlin Multiplatform feature, a sample application will be developed for Android and iOS. First, it will be developed natively, and then with the use of the Kotlin Multiplatform feature. The multiplatform application for Android will then be compared to the natively developed one for Android, and the same will be done for the iOS applications.

When choosing the functionality of the sample application, three main aspects were taken into consideration. These were: the context of earlier studies, the properties of the Kotlin Multiplatform feature, and application complexity. The first aspect was the importance of conducting a study in the context of earlier research within the field of cross-platform application development. The literature study for this thesis, showed no consensus regarding a state-of-the-art sample application. The earlier research found in the literature study, that compared native application development to application development using one or more cross-platform tools, all used different sample applications. Also, they seldom described the sample application in detail. The conclusion drawn from this, was that the exact functionality of the sample application, was not the most important factor. However, for those studies that did describe the functionality of their sample application, some similarities were found. They were similar in complexity; all were smaller projects with limited functionality, but still large enough to contain more than one view, and some business logic. Another similarity was that they in some way fetched data from the Internet, parsed that data, and made it visible to the user.

The second aspect considered when choosing the sample application, was the properties of the Kotlin Multiplatform feature. The foundations of Kotlin Multiplatform are described in Section 2.2, and the wish was to make use of all of them. Using one or more of the multiplatform libraries, having business logic with shared data objects and having at least one class or function that required platform-specific implementations were considered the most important properties for the sample application, with respect to the properties of the Kotlin Multiplatform feature.

The third aspect was deciding upon the complexity of the application. To be able to draw more general conclusions regarding the size and startup time of the apps, and the development productivity, the intention was to make a sufficiently complex sample application, that shared characteristics of real-world applications, but still simple enough for the delimitation in time for this project.

All these considerations taken into account, the chosen sample application for this project, is an application in which a user can search for users on GitHub. The application shows the search hits in a list, where each item contains a user, presented with the username and the avatar image of the user. This information is retrieved from the GitHub API [1], via a HTTP request. The responses for this request, i.e. the search results on the given user name, is in the format of JSON data. Thus, the application also needs to parse JSON data, to show the user information. The user items are clickable, and when clicked, a new view is shown. The new view contains a list of the user's repositories. Each repository item contains the repository name together with the number of forks, stars and watchers that the repository has. The repository information is fetched from the GitHub API, using the same techniques as when fetching the user data. The navigation flow and functionality of the sample application can be seen in figure 3.1.

Using HTTP request, meant that the application code could make use of at least one multiplatform library provided by Kotlin. Internet communication and parsing of JSON data, was also seen in sample applications of earlier studies, and are in addition to that a fundamental functionality in many real-world smartphone applications. Users and repositories were considered suitable data objects for the business logic. The avatar images were user interface items, which usually needs platform-specific implementations.

To summarize, the functionality of the sample application was chosen so that it would have similarities with sample applications of previous studies and with

[1] `https://developer.github.com/v3/search/`

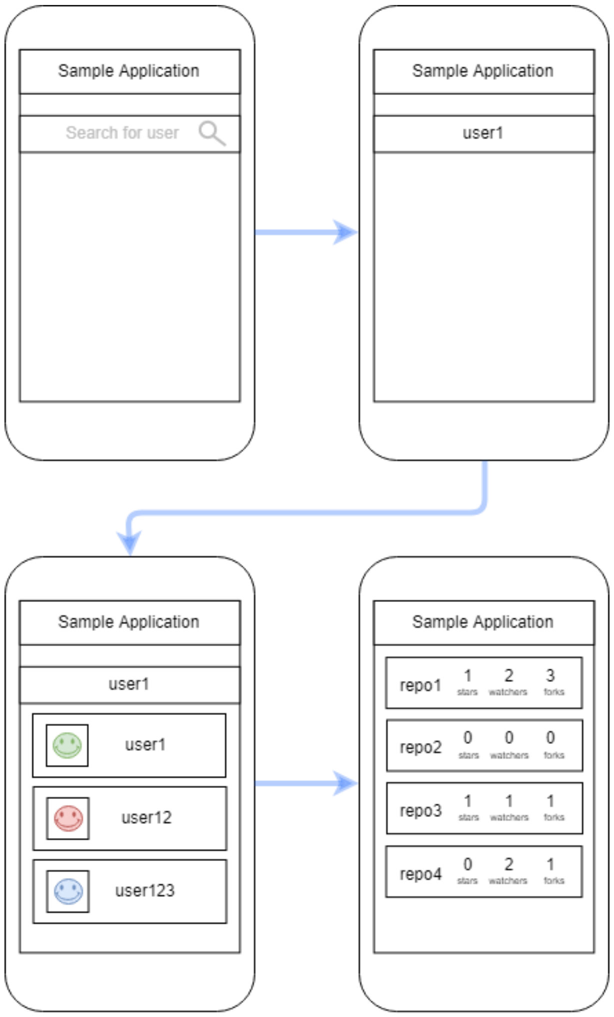

Sample Application
Search for user
Sample Application
user1
Sample Application
user1
user1
user12
user123
Sample Application
repo1 1 stars 2 watchers 3 forks
repo2 0 stars 0 watchers 0 forks
repo3 1 stars 1 watchers 1 forks
repo4 0 stars 2 watchers 1 forks

Figure 3.1: Functionality of sample application.

real-world applications, and so that it could be built making use of central properties of Kotlin Multiplatform. These aspects were considered crucial to be able to draw more general conclusions regarding results.

3.2 Metrics

3.2.1 Application Startup Time and Size

One of the two main focuses for evaluation of the Kotlin Multiplatform feature in this thesis project, is on measuring the startup times of the applications developed using it. As mentioned in section 2.3, evaluating cross-platform mobile applications, is usually done either with dynamic analysis of the application in question, or with static analysis of the code of the application. For this project, the choice is to measure the **startup time of the application**, since this is something that has been shown to be negatively affected by using cross-platform tools [4, 31]. We have also chosen to compare the **size of the installed application**, to evaluate if there is any application size overhead in using Kotlin Multiplatform for development of Android and iOS applications. Also application size has been shown to be affected negatively by using other cross-platform tools [31]. Of these two metrics, the first belong to a static program analysis, and the second one to dynamic analysis.

The startup time is the time from when the user starts the application by pressing the icon, until it is fully launched and ready to be used. The first reason to study this, is that a too long startup time might disappoint the user, and make him/her not want to use the application, or give it bad ratings [8]. Xanthopoulos and Xinogalos [32] describe the user-perceived performance as an important factor when evaluating cross-platform development tools, and the launch time can be viewed as an example of such a performance measurement. Willocx et al. [31] study application startup time in their work with evaluating cross-platform tools.

The second reason to study application startup time is that it could give an indication of if there are any possible performance losses of Kotlin Multiplatform, compared to native development.

Regarding application size, its importance mainly concerns low-end devices with limited resources. The importance of this will be discussed further in Section 5.5.

3.2.2 Development Productivity

In this project, only one person, namely the author, will perform the development of the sample applications. With this arrangement, and with the prerequisites of this project, counting and comparing the number of lines of code for the native projects with the multiplatform project, is considered most straightforward. Thus, **lines of code**, is one of the chosen metrics to measure productivity. However, one should keep in mind that this metric builds upon the assumption that less code entails less development time and costs.

Furthermore, the entire idea with cross-platform development is to be able to share code between platforms, and to not have to write the code for each piece of functionality several times; that is, once for each platform. Thus, it is of interest to measure how much code can actually be shared, when evaluating a cross-platform tool. Lines of code is a useful metric for this.

The other metric used to measure developer productivity is **build time**. Build time is the time it takes to build, i.e. compile, the application project. A long build time means a long wait time for the developer, which in turn can lead to a longer development time [33].

3.3 Data Collection

3.3.1 Hardware

For the static analysis of the applications, the hardware has no importance. However, for the dynamic analysis, it is very important to use the same conditions in every experiment, and to run it on the same hardware, with the same preparations.

For the build time measurements the following machine was used.

MacBook Pro (2016)
Processor: 3,3 GHz Intel Core i7
Memory: 16 GB 2133 MHz LPDDR3
Graphics: Intel Iris Graphics 550 1536 MB
Operating system: macOS Mojave, version 10.14.4

For the startup time measurements of the Android applications, the following device was used.

Google Pixel 3XL (2018)
Processor: 2.5GHz octa-core
Memory: 4 GB RAM
Graphics: Adreno 630
Operating system: Android Q Beta

For the startup time measurements of the iOS applications, the following device was used.

Iphone XS Max (2018)
Processor: A12 Bionic
Memory: 4 GB RAM
Graphics: Apple GPU (4-core graphics)
Operating system: iOS 12.2

3.3.2 Tools

3.3.2.1 Android Startup Time

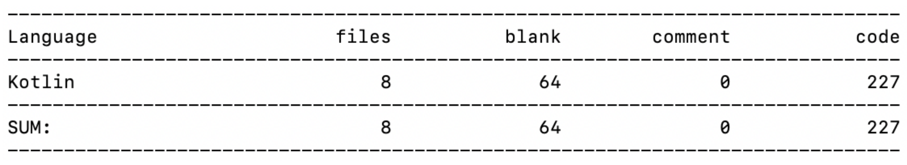
For Android, application startup time is the time between initialization of starting the application (e.g. a user tapping the application icon) until the main view is visible on the screen [8]. Google defines three different types of startups, of which the *cold* startup time will be used as the metric. Cold startup means that the system is starting the entire application from scratch, which happens either when the application is started for the first time when the device has been restarted, or when the application is started again after the system has killed the application. Android Studio has built-in measuring of cold startup time. The result is printed to Logcat and visible when all filters in the Logcat view are disabled.

3.3.2.2 iOS Startup Time

For iOS applications, as well, the cold startup will be measured. There are settings you can configure in Xcode, to allow for statistics to be printed at each build and run. These statistics print the so called pre-main time. To get the post-main time, which together with the pre-main time makes up the app's startup time, some timing has to be done from within the code [1].

3.3.2.3 Lines of Code

To count lines of code, the open source tool cloc is used[2]. It has support for counting lines of code for a number of programming languages, including Kotlin and Swift.



```
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Kotlin                           8             64              0            227
-------------------------------------------------------------------------------
SUM:                             8             64              0            227
-------------------------------------------------------------------------------
```

Figure 3.2: An example of the result of using cloc

As shown in figure 3.2, the tool counts lines of code, lines of comments, and blank lines. For consistency, the number of characters allowed on a single row, is set equally in the corresponding IDEs for Android and iOS.

3.3.3 Preparations

When measuring build times, all programs except the IDE (Xcode or Android Studio, depending on platform), were turned off, as was the Wi-Fi. When measuring startup times, all other applications on the devices were turned off. The devices were also set to Airplane mode, and for Android the Wi-Fi was turned off. On iOS, however, turning off the Wi-Fi makes it impossible to run the sample application from Xcode, since every installation requires the Iphone to control that the developer is trusted, a process that requires Internet connection. To eliminate the disturbance from background Internet data, in the experiments for the iOS applications, settings were made to disallow background data fetching for all applications on the test device.

3.3.4 Statistical Method

For the metrics used for dynamic analysis in this project, i.e. startup time and build time, statistical method needs to be considered. The reason for this, is to prove the significance of possible differences between results for the native applications and the multiplatform applications.

As described in section 2.4, the two main metrics for evaluating performance are the harmonic mean and the arithmetic mean. Since no rates will be measured, but instead execution and build times, the arithmetic mean is the one that is applicable.

[2] https://github.com/AlDanial/cloc/blob/master/README.md

To determine if the possible difference in execution or build times are statistically significant, two-sided confidence intervals will be used. Formula 2.4 and Formula 2.5 can be summarized into one, as in Formula 3.1 below.

$$c = \overline{x} \pm z_{1-\frac{\alpha}{2}} \frac{s}{\sqrt{n}} \tag{3.1}$$

The sample size n, i.e. the number of executions or builds in the experiments, are set to 100. It is always desirable to have as many measurements as possible, but in reality one also has to consider limitations in time. The chosen sample size is perceived a balanced compromise of these two aspects.

The choice is to set $\alpha = 0.05$ to get a confidence level of 95%. This gives $z_{1-\frac{\alpha}{2}} = z_{0.975}$, which in turn is approximately equal to 1.96. If the intervals for natively developed and multiplatform developed applications do not overlap, we can say that no evidence was found against a statistically significant difference in performance or build time not being [20]. If so, there is a difference in performance or build time, with a confidence level of 95%. If the intervals *do* overlap, there is no indication of a significant difference in performance or build time. Any perceived differences could in this case be caused by random fluctuations.

Chapter 4

Results

This chapter presents the results of the study, in terms of application size, application startup time, lines of code and build time.

4.1 Application Size

For both Android and iOS, the multiplatform application was larger than the natively developed one. For Android, the difference was 0.68 MB, which corresponds to the multiplatform application being approximately 13% larger. Regarding iOS, the difference was 7.8 MB, which corresponds to an approximately 18% larger application than the natively developed one.

Size Android	
Native	*Multiplatform*
5,32 MB	6,00 MB

Table 4.1: Sizes of Android applications

Size iOS	
Native	*Multiplatform*
44,2 MB	52,0 MB

Table 4.2: Sizes of iOS applications

4.2 Application Startup Time

For the Android applications, there appears to be a substantial difference in startup time between the natively developed application and the multiplatform one. The mean startup time for the native application is 587.4 ms, and the mean startup time for the multiplatform application is 675.6 ms. The confidence intervals are (585.0, 589.8) ms, and (673.4, 677.8) ms, on the confidence level 95%.

The results for the iOS applications regarding startup time differs, albeit slightly, between native and multiplatform applications. The mean startup time for the native application is 147.9 ms, while the mean for the multiplatform application lies only 1.2 ms higher, at 149.1 ms. For the iOS applications, the calculated confidence intervals are (145.95, 149.85) ms for the natively developed application. The corresponding confidence interval for the multiplatform application is (147.20, 150.00) ms. Since the intervals overlap, no indication can be seen of a significant difference in startup time, for the iOS applications.

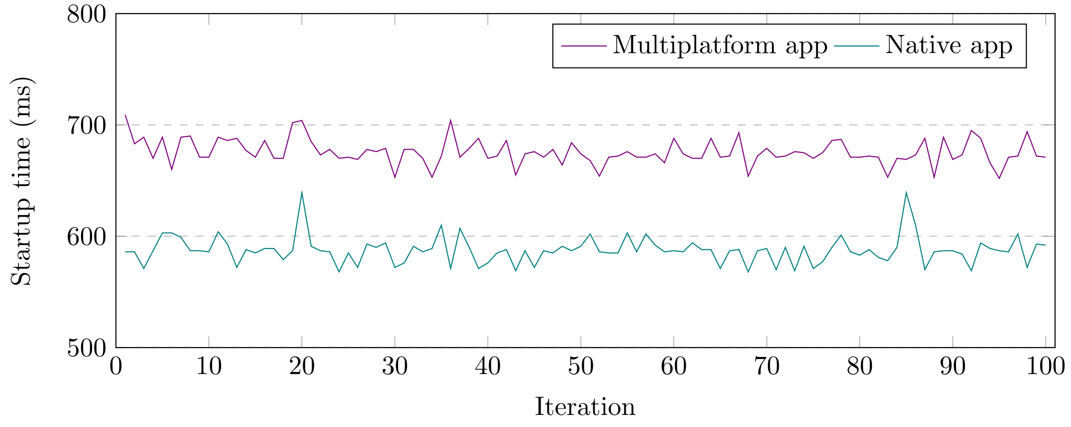


Figure 4.1: Startup time for Android applications

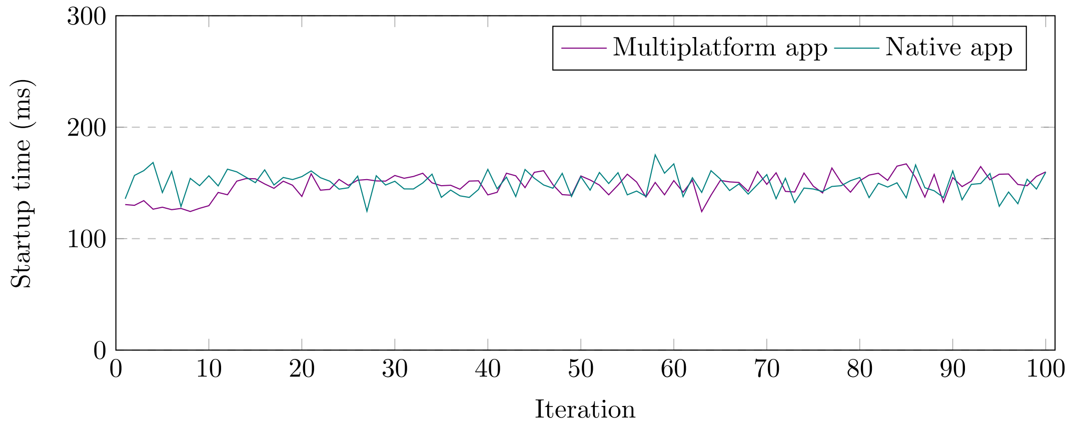


Figure 4.2: Startup time for iOS applications

4.3 Lines of Code

Table 4.3 shows the number of lines of code in the native projects and the multiplatform project. The results in the table show that it was possible to share a fair amount of code in the sample application of this project. If the code for both native applications are added together, there is a total of 494 lines of code. If the same is done for the multiplatform project, there is a total of 383 lines of code for the two multiplatform applications. Thus, 89 lines of code have been saved, by using the Kotlin multiplatform feature. In percentage terms, this corresponds to approximately 18% fewer lines of code.

	Native Projects		**Multiplatform Project**	
	Android	**iOS**	**Kotlin**	**Swift**
LoC	227	267	255	128
Total	**494**		**383**	

Table 4.3: Number of lines of code

Furthermore, in the multiplatform project, 66 lines of code are shared by the Android and the iOS platforms. This corresponds to approximately 17% shared code, and 83% platform specific code.

4.4 Build Time

When compiling a Kotlin Multiplatform project targeting Android and iOS, the regular builds for each platform are performed. But these builds are not the only compilations for a multiplatform project. The compilation of the library of shared code (see section 2.1 about how Kotlin multiplatform projects are structured) into native binaries for iOS, is performed separately. For the sample application for this thesis, that build takes an average of 41 seconds.

The usual builds for each platform is shown in the figures below. The first measurements for each platform were discarded since the first build always takes substantially more time. Both figures show a difference in build times. This difference can be supported by two-sided confidence intervals, for both platform builds. The natively developed Android application has a build time within the 95% confidence interval $(6.839, 7.213)$ s. The confidence interval for the multiplatform Android application is $(10.299, 10.645)$ s. The two intervals do not overlap, which indicates a statistically significant difference between Android build times for native and multiplatform applications.

The same reasoning goes for the non-overlapping confidence intervals of the iOS builds, which for the natively developed application is $(3.479, 3.567)$ s, and for the multiplatform application is $(4.733, 4.835)$ s.

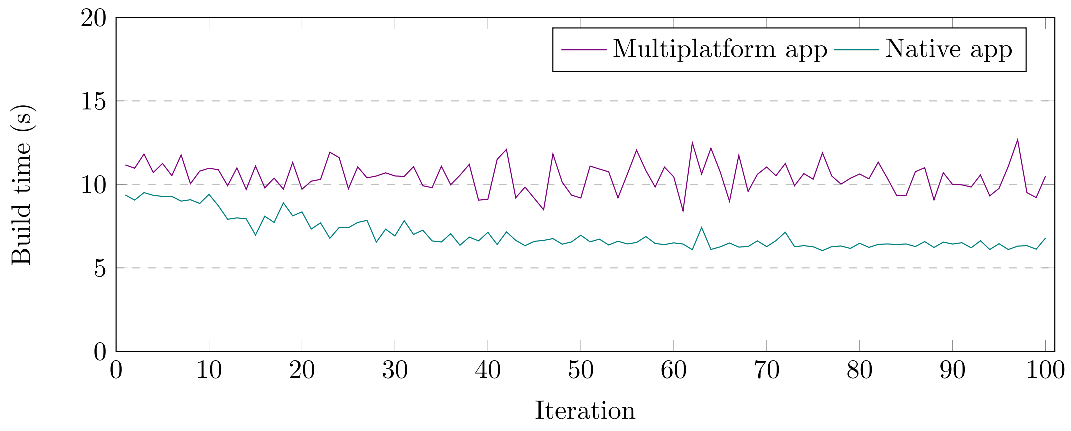


Figure 4.3: Build time for Android applications

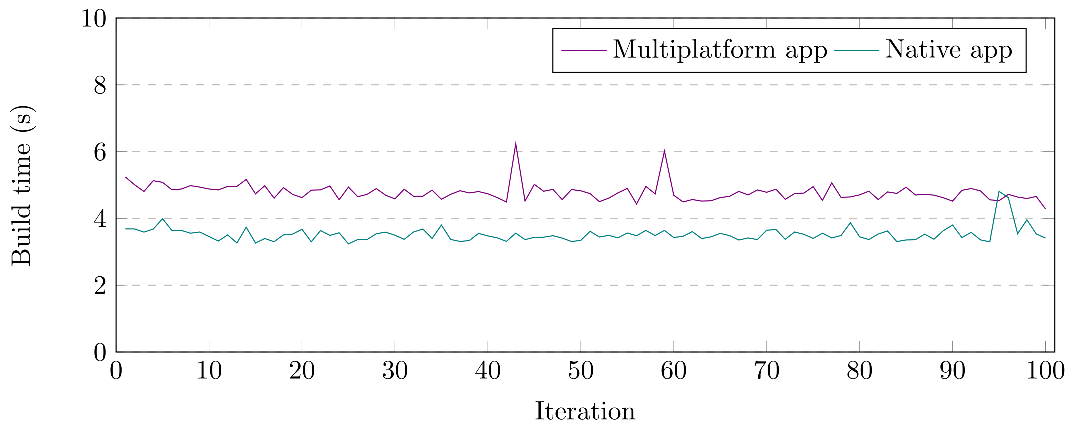


Figure 4.4: Build time for iOS applications

Chapter 5

Discussion

This chapter includes discussions and analysis of the results, the choice of methods and possible sources of error.

5.1 Application Startup Time and Size

For the startup times, the results for the Android application indicated a significant difference between the natively developed, and the multiplatform, application, with the multiplatform application having a slower startup time than the native one. For the iOS applications, however, there was no indication of a significant difference in startup times between native and multiplatform. These results are perhaps a bit surprising, since the implementations for the native and the multiplatform Android applications does not differ nearly as much as the implementations for the iOS applications do. That is, both Android applications are written in Kotlin, and they are both compiled to Java bytecode. Why do the results indicate a difference in startup times there, but not in the iOS applications, where the multiplatform application actually makes use of a library compiled by another (i.e. the Kotlin) compiler? Without actually looking at and comparing the Java byte code for each of the Android applications - which is beyond the scope of this thesis - one can only make a qualified guess. Perhaps there is a difference between iOS and Android in what is actually done and loaded before the startup of an application is completed, which gives the use of libraries less impact on the startup times of iOS applications than for Android ones.

The application sizes for the sample applications showed an approximately 18% larger multiplatform application than native app, for iOS. The corresponding number for Android was 13%. As with other results in this thesis, it would be interesting to investigate if this increase in size will hold also for larger projects, or if it is an initial penalty, that will decrease in ratio for a larger project.

5.2 Development Productivity

The measurement of productivity in terms of number of lines of code, in one project, developed by one single person, can of course not be directly applied to all real-world cases, where the projects are much larger and there are more people involved in the development process. In this project, it was possible to write at least 18% fewer lines of code, using the Kotlin Multiplatform feature, but we cannot know if that would be the case for a larger and more complex application. Nor can we know that it was impossible, for the sample application developed in this project, to save more than 18% of lines of code, only that it was possible to write *at least* 18% fewer lines of code.

The assumption that fewer lines of code really entail less development time and costs is, of course, disputable. Since this assumption might, in fact, not be true, the question of possible productivity gains of using the Kotlin Multiplatform feature is one that should be further addressed in future studies, possibly using other metrics to measure it. One way would be to do a larger study, with more developers involved, and not only measure lines of code, but also the development time. Anyhow, the results of this study concerning lines of code, is at the very least a suggestion that it is of interest to study the possible productivity implications further, since it proves that it is possible to write less code if using Kotlin Multiplatform.

The build time seemed to be significantly larger for the multiplatform than the native application, for both platforms. This could be interpreted as a possibly lower development productivity. Even if no increase could be seen in in the regular build times for the multiplatform applications, the approximately 41 seconds long compilation of Kotlin into iOS native binaries, adds up to an increase of multiple times the one for the native projects. It can be discussed if these 41 seconds should count in the results for the build times. This depends on how the development process is organised. As long as you are not making changes in the shared library, that you want to test in the iOS app, it will not effect the build time. However, as soon as you do want to test some newly developed shared functionality in the iOS application, you need to run this special build.

5.3 Sample Application

Even though no state-of-the-art sample application could be found in the literature study for this thesis, this does not mean that which functionality one chooses for the sample application does not matter. There are impacts on the results, of which sample application is used. The size and the functionality of the sample application can effect the startup times, build times, application sizes and the number of lines of code. There might have been a larger or smaller difference between the native applications and the multiplatform ones, if the sample application was chosen differently. It is also worth to mention that one factor that affects how much code is shareable between platforms, and how extensive the platform-specific implementations will be, is which Kotlin libraries, designed for multiplatform projects, are available. HTTP request is a substantial part of this project's sample application, and for that, there are designated libraries that makes it much easier to share code in a quite compact way. Again, it is proposed to study larger and more complex projects, with extended functionality.

5.4 Sources of Error

When measuring execution or build times, it is important to acknowledge the fact that neither execution nor build times are deterministic. A computer or smartphone have processes that are run in the background, which means that we cannot be sure that the measurements are comparable with each other, since there might be disturbance affecting the results. Efforts were made to minimize such noise in all experiments performed for this thesis (see Section 3.3.3 for details). Because of this, and because of the small fluctuations in measured startup times, we can be fairly certain that there is little disturbance in the results, also for the iOS applications.

The individual who developed the sample applications (i.e. the author of this thesis) had no prior experience in neither Kotlin, Swift nor Kotlin Multiplatform. The sample applications can therefore not be argued to be the best or most compact solutions. Hence, it might be possible for a more experienced developer to write more compact code, both for the native and the multiplatform applications. That being said, the possible lack of compactness in the source codes for all the different platforms, can be considered of approximately equal measures. Thus, the results for the lines of code metrics are deemed usable and not prone to too many errors.

5.5 Sustainability and Societal Aspects

Studying the performance of cross-platform tools, can be viewed in a sustainability perspective. For example, one might think that smartphone memory has improved so to such a high degree lately, that there is no need to study the size of mobile applications anymore. However, many people around the globe use older smartphones, with limited resources, and therefore it is important to take application size into consideration when making decisions regarding the development of smartphone applications. If applications become too big to be used on older smartphones, this could by extension increase the digital divide in the world. Also, manufacturing smartphones produces waste, toxicants and carbon dioxide, and it becomes a sustainability issue if people always need the newest phone to be able to use modern mobile applications.

The motive behind cross-platform tools is to make application development easier and less costly. A study of cross-platform tools that showed the possibility to use less resources in terms of developers, to produce the same application, could make companies more prone to use such tools and hire fewer developers. This, in turn, could have economic and social implications, if it meant more unemployment. Perhaps this reasoning is taking it to its extreme, but it might be worth to consider.

Chapter 6

Conclusion

This chapter summarizes the main results what can be concluded from them. Finally, it presents ideas for future research.

It has been proven that, for a small example application with the functionality described in Section 3.1, it is possible to use Kotlin Multiplatform for development, producing a smartphone application that can be run on both Android and iOS. Moreover, this thesis shows that this can be done by writing at least 18% less code, and sharing at least 17% between platforms.

These possible productivity gains might, however, be hampered by the possible productivity losses of the prolonged build time, which was shown in the results of the experiments.

As for if the startup time is affected by using Kotlin Multiplatform instead of native development, the results speak partly for and partly against. The startup times seems to be significantly longer for the multiplatform Android, than the native Android, application. No significance can be seen in the difference in the startup times for the iOS applications.

As for the application sizes, they are shown to be larger in the multiplatform applications, than in the native applications. However, the larger size might very well be due to an initial application size overhead, and the difference in size might decrease with a more extensive application.

The scope of this thesis does not entail a comparison to other cross-platform tools. However, it is worth mentioning that the results of this study regarding application size and startup time is in line with earlier studies of other cross-platform tools. Sizes of applications developed with tools such as Xamarin and PhoneGap have been shown to differ even more, in percentages, from natively developed applications [31]. Startup times, as well, have shown slower app startup times than native ones [4, 29].

To summarize, the drawbacks of using Kotlin Multiplatform can definitely be viewed to outweigh the advantages - depending on if application size, build time and Android startup time is of importance, and how great that importance is, in that particular project.

6.1 Future Work

In the sample application, evaluated for this project, the startup times for Android seemed to be significantly larger for the multiplatform application. It would be of interest to see if this difference in performance would be visible for projects of increasing size. The overhead for multiplatform applications might be a fixed penalty, which ratio of the total startup time would decrease with an increasingly large project. There is also a need to study the performance of Kotlin multiplatform applications in greater detail, and investigate more performance metrics.

As mentioned in Section 5.2, another extension of the work of this thesis, would be to study in more depth the productivity of Kotlin multiplatform development in comparison to native development. The suggestion is then to add more metrics, such as development time, but also to study the productivity in larger projects with multiple developers.

Bibliography

[1] Avijeet Dutta. iOS App Launch time analysis and optimizations. https://medium.com/@avijeet.dutta13/ios-app-launch-time-analysis-and-optimization-a219ee81447c. [Online; accessed May 6 2019].

[2] M. Ciman and O. Gaggi. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, 39(C):214–230, 2017.

[3] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein. Survey, comparison and evaluation of cross platform mobile application development tools. In *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 323–328. IEEE, 2013.

[4] S. Dhillon and Q. H. Mahmoud. An evaluation framework for cross-platform mobile application development tools. *Software: Practice and Experience*, 45(10):1331–1357, 2015.

[5] W. S. El-Kassas, B. A. Abdullah, A. H. Yousef, and A. M. Wahba. Taxonomy of cross-platform mobile applications development approaches. *Ain Shams Engineering Journal*, 8(2):163–190, 2017.

[6] GitHub, Inc. Octoverse 2018, Languages. https://octoverse.github.com/projects, 2018. [Online; accessed January 30 2019].

[7] Google. Android Studio Release Notes. https://developer.android.com/studio/releases#3-0-0, . [Online; accessed April 25 2019].

[8] Google. App startup time. https://developer.android.com/topic/performance/vitals/launch-time, . [Online; accessed April 15 2019].

[9] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha. Code smells in ios apps: how do they compare to android? In *Proceedings of the 4th International Conference on mobile software engineering and systems*, MOBILESoft '17, pages 110–121. IEEE Press, 2017.

[10] G. Hecht, B. Omar, R. Rouvoy, N. Moha, and L. Duchien. Tracking the software quality of android applications along their evolution. In *30th IEEE/ACM International Conference on Automated Software Engineering*, Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015). IEEE, 2015.

[11] H. Heitkötter, S. Hanschke, and T. A. Majchrzak. Evaluating cross-platform development approaches for mobile applications. In J. Cordeiro and K.-H. Krempels, editors, *Web Information Systems and Technologies*, pages 120–138, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] IDC. Smartphone market trends. `https://www.idc.com/promo/smartphone-market-share/os`. [Online; accessed March 12 2019].

[13] Jetbrains. Building Multiplatform Projects with Gradle. `https://kotlinlang.org/docs/reference/building-mpp-with-gradle.html`, . [Online; accessed March 18 2019].

[14] Jetbrains. Kotlin multiplatform. `https://kotlinlang.org/docs/reference/multiplatform.html`, . [Online; accessed January 28 2019].

[15] Jetbrains. Kotlin/Native for Native. `https://kotlinlang.org/docs/reference/native-overview.html`, . [Online; accessed March 12 2019].

[16] Jetbrains. Full Kotlin Refernce. `https://kotlinlang.org/docs/kotlin-docs.pdf`, . [Online; accessed May 28 2019].

[17] Jetbrains. Platform-Specific Declarations. `https://kotlinlang.org/docs/reference/platform-specific-declarations.html`, . [Online; accessed May 28 2019].

[18] Jetbrains. Multiplatform Project: iOS and Android. `https://kotlinlang.org/docs/tutorials/native/mpp-ios-android.html`, . [Online; accessed May 28 2019].

[19] Jetbrains. FAQ - Kotlin Programming Language. `https://kotlinlang.org/docs/reference/faq.html`, . [Online; accessed April 25 2019].

[20] D. J. Lilja. *Measuring Computer Performance: A Practitioner's Guide [electronic resource]*. Cambridge University Press, Cambridge, UK, 2004.

[21] M. Martinez and S. Lecomte. Towards the quality improvement of cross-platform mobile applications. In *Proc. of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE/ACM, 2017.

[22] S. Memeti, L. Li, S. Pllana, J. Kołodziej, and C. Kessler. Benchmarking opencl, openacc, openmp, and cuda: Programming productivity, performance, and energy consumption. In *Proceedings of the 2017 Workshop on adaptive resource management and scheduling for cloud computing*, ARMS-CC '17, pages 1–6. ACM, 2017. ISBN 9781450351164.

[23] I. T. Mercado, N. Munaiah, and A. Meneely. The impact of cross-platform development approaches for mobile applications from the user's perspective. In *Proceedings of the International Workshop on app market analytics*, WAMA 2016, pages 43–49. ACM, 2016.

[24] Microsoft. Introduction to Mobile Development. https://docs.microsoft.com/en-us/xamarin/cross-platform/get-started/introduction-to-mobile-development. [Online; accessed March 19 2019].

[25] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi. A case study on the impact of refactoring on quality and productivity in an agile team. volume 5082, pages 252–266, 2008.

[26] M. Palmieri, I. Singh, and A. Cicchetti. Comparison of cross-platform mobile development tools. In *2012 16th International Conference on Intelligence in Next Generation Networks*, pages 179–186. IEEE, 2012.

[27] R. Raj and S. B. Tolety. A study on approaches to build cross-platform mobile applications and criteria to select appropriate approach. In *Proc. of Annual IEEE India Conference (INDICON)*, pages 625–629. IEEE, 2012.

[28] J. Reimann, M. Brylski, and U. Aßmann. A tool-supported quality smell catalogue for android developers. In *Proc. of the conference Modellierung 2014 in the Workshop Modellbasierte und modellgetriebene Softwaremodernisierung–MMSM*. MMSM 2014, 2014.

[29] J. Smołka, M. Kulisz, B. Matacz, E. Łukasik, M. Skublewska-Paszkowska, and M. Szala. Performance analysis of mobile applications developed with different programming tools. *MATEC Web of Conferences*, 252, 2019.

[30] Wikipedia [Internet]. St. Petersburg (FL): Wikimedia Foundation, Inc; 2001 . Kotlin (programming language). https://en.wikipedia.org/wiki/Kotlin_(programming_language). [Online; accessed January 31 2019].

[31] M. Willocx, J. Vossaert, and V. Naessens. A quantitative assessment of performance in mobile app development tools. In *Proceedings - 2015 IEEE 3rd International Conference on Mobile Services, MS 2015*, pages 454–461. Institute of Electrical and Electronics Engineers Inc., 2015.

[32] S. Xanthopoulos and S. Xinogalos. A comparative analysis of cross-platform development approaches for mobile applications. In *Proceedings of the 6th Balkan Conference in Informatics*, BCI '13, pages 213–220. ACM, 2013.

[33] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos. Reducing build time through precompilations for evolving large software. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, volume 2005, pages 59–68. IEEE, 2005.

TRITA -EECS-EX-2019:575