UiO : Department of Informatics
University of Oslo

Native applications on modern devices

Master's thesis

Simen Hasselknippe

4/22/2014

This page is intentionally left blank

DEVELOPING NATIVE APPLICATIONS ON MODERN DEVICES: A CROSS-PLATFORM APPROACH

A Xamarin product research

Keywords
Xamarin, MvvmCross, MonoGame, Mono, Native, App, App-development, C#, devices, cross-platform, iOS, Android, Windows Phone, Windows 8

Simen Hasselknippe
Simen.hasselknippe@live.com

This page is intentionally left blank

Abstract

This paper is a part of my master thesis at the University of Oslo. It introduces concepts, theories, and patterns related to cross-platform development on mobile devices. It discusses what applications programmers can develop, what devices are important to target, and how to develop mobile applications with a cross-platform approach. During this thesis, I have used various tools and products, including Xamarin.iOS, Xamarin.Android Xamarin Studio, Visual Studio, MvvmCross, MonoGame, iOS, Android and Windows Phone.

This page is intentionally left blank

Table of Contents

Preface

Developing applications is both challenging and fun. I choose this topic because I am interested in developing applications on many platforms. I wanted to learn about methods, architectures and tools that could remove some of the pain related to app development. This paper is intended to educate in application development as well as provide a good overview of applications and devices. I started my work by defining a few research questions. These questions were divided into theories. I then started planned experiments to check if they were valid. As time has passed, the experiments have led to more theories, which required a periodic re-evaluation of the research questions to keep them updated and in-line with path of the research.

Special thanks go to both of my supervisors—Dag Langmyhr at the University of Oslo, who has given me great advice in terms of spelling, content and information, and Tomas Jansson at Bekk Consulting AS, who has given the perfect combination of editing and technical assistance.

Finally, big thanks to my family for endless encouragement and support through all my years of studies.

This page is intentionally left blank

List of figures

List of tables

This page is intentionally left blank

Introduction

Developing applications on today's modern devices seems easy at first glance. You create an application that lives in a sandbox; it lives there protected and isolated. You also utilize a set of APIs that can request access to other parts of the device, for instance, the camera. This creates a safe environment both for the consumer and for the developer. However, there is more to application development than just creating one app on one device.

There are hundreds of different mobile devices. Many of them even run on different operating systems with different programming languages and different screen resolutions. This is a problem for developers. Once they have finished developing an application on one platform, they must restart the process to target another platform. Creating products multiple times is both time consuming and expensive.

Most companies want to give as many users as possible access to their products. In addition, customers expect apps to be available. The solution to this is might be cross-platform development. Performing a search on "cross-platform development" generates over 64 million hits on Google. Wikipedia defines cross-platform development as "a method and concept that are implemented and inter-operate on multiple computer platforms".

Sun Microsystems introduced "Write once run anywhere" (WORA) to illustrate the benefits of cross-platform development. However, cross-platform development is difficult to achieve. This document attempts to look at the options a developer has today to create native, modern applications on modern devices with a cross-platform approach.

What is a native application today? Is it any application that can be installed through a store, or is it just the ones that are developed using native tools and languages? Further, what is native speed and performance? Moreover, how can we measure the nativeness of an application? These are issues I will explore in this paper.

Creating a mobile app is both challenging and fun if you have the right tools and motivation. This paper is for those of you who want to explore different options to develop an application but also for those who want to learn more about the mobile platform or code sharing.

Many development examples here are based on Mono and Xamarin, but as will be discussed later, much of the terminology, tools, and patterns can easily be converted into other tools and languages.

This paper explores the following research question:

"How can we build apps that have a native look and feel by using cross-platform technologies, and when should we use it?"

This leads to number of underlying issues:

- What tools can be used to develop cross-platform apps?

- What is a native look and feel?
- Can cross-platform development suites compete with native?
- What architecture pattern is best suited for cross-platform development?
- What options exist for sharing code in cross-platform development?
- How much code can we expect to share in cross-platform development?
- Can cross-platform development save development time?
- What cases are best suited for cross-platform development?
- Does cross-platform development make put any limitations on our project?

And that is essentially what this paper will try to answer.

To try many of the experiments here you need some tools. They are described in the setup chapter inside each experiment. Following below is an overview on what will be contained in each chapter.

In this paper, I will start with some general background material. In chapter 1.1, I will walk through the history of cross-platform development. How it has moved from being software supporting different hardware to software supporting different devices and operating systems.

I will then detail the current device landscape in chapter 1.2 to identify what makes a device modern. I will discuss the different operating systems and which ones should be targeted for an app to be available to consumers all around the world.

In chapter 1.3, I will show why companies should target different devices on different platforms. I will dig into mobiles and tablets to see why they are so important and why we need to build an app for our application. Most importantly, I will argue why you should have an application on a wide variety of devices and why devices mean so much to users.

I will then move on to chapter 1.4 and introduce modern apps. What is a modern app? What does it look like? How does it feel? I will also discuss the differences between modern apps on different devices and operating systems.

I will then examine how mobile applications can be packed in order to ship them through one of the application stores. We will also see what demands the owners of the store have for us before they allow our app into the system.

Then in chapter 1.6, I will give a short overview of what an application lifecycle is and how it works. We will see that our apps need to interrupt different events to function normally as other apps do. We will also identify how apps differ from each other.

I will also introduce you to the game lifecycle, often referred to as a game loop, in chapter 1.7. Games have enormous potential in cross-platform development since developers usually want a game to look identical on different platforms.

Then I will examine an interaction-based taxonomy of mobile applications. Before experimenting with our apps, examining how users are using their apps and understanding who we could benefit from this knowledge is essential.

Chapter 1.9 will briefly discuss the different approaches you can take to develop an application. I will go through the application types and discuss the differences of native, shell, and web applications. I will explain the differences between them as well as pros and cons about each.

I will finish the background material by forming a research question. I will then argue how I will do it and how it should be done.

Chapter 2 is the general discussion portion of this paper. Here I will go through some general assumptions and explain different methods. This chapter contains four sections.

In chapter 2.1, I will try to generate a feature-based taxonomy in mobile applications. I will see which features a cross-platform development tool needs to support in order to be complete and reliable. This taxonomy will be used for comparing and evaluating different approaches to cross-platform development.

To develop applications with a cross-platform approach, the appropriate tools, such as frameworks, libraries, and IDEs (integrated development environments), are nececcary. Chapter 2.2 gives an overview of what types of tools are out there. I will discuss how different tools can help in different situations. I will also pick the most promising tools to continue using in this experiment.

Chapter 2.3 introduces different code-sharing strategies. I will look at how code can be shared and what tools developers have to make code sharing as pain free as possible.

Then I will finish the general discussion by looking at two common architectures: the MVC (model-view-controller) and the MVVM (model-view-view-model). I will discuss how they can be used in order to be able to share code as well as all there pros and cons.

Chapter 3 introduces the experiments and the apps developed using my assumptions in the general discussion.

First, I will try to build a native application that keeps control wishes. The product is called LittleWishList and is built with MVC and linked classes. I will also dig into a few extensions, such as components, cloud services, and development environment.

I will then move on to a new project called Quizter. Quizter is a quiz app based on open questions that users can create themselves. Here I will explore the MVVM pattern as well as additional components and pushing data using SingalR.

The final experiment is in chapter 3.1 is where I try to make a game with the information learned from the previous experiments. The game is called Bricasa and is a puzzle game involving a log of trigonometry and formulas.

In chapter 4, I will analyze the tools used in the experiments as well as the patterns and architecture. I will see if my assumptions are correct and map the experiments together with the research questions.

I will start by analyzing code-sharing strategies. I will discuss the different approaches I have used and argue about the pros and cons for each of them.

In chapter 4.2, I will look at the ecosystem, how the development ecosystem is fit for cross-platform development, and how it can be improved.

I will then go back to architecture and evaluate the different architectures used in the experiment. I will re-examine MVC and MVVM and identify how developers can build apps that are true to a design paradigm but also easy to work with.

Chapter 4.4 analyses the apps performance. I will look at the look and feel and give my opinion on whether I have managed to build a fluid native app that can be used in cross-platform development. I will conclude by determining whether the apps are as good as native apps, and if not, then why?

Chapter 4 will finish with chapter 4.5. Here, I will go through some general development results, and I will give my true opinion on how the process of developing the apps, on how I as a developer feel about the technology, and on whether this is a viable option against native development.

In Chapter 5, I will look at some ideas I did not have time to try and will give some guidance to other master students wanting to follow up this topic.

In chapter 6, I will try to summarize the results of all my work as well as answer my research question. I will then look back upon this project at give you a little information about the ups and downs of it in chapter 7.

1 Background material

To understand "cross-platform" and "cross-platform development", it is important to understand why we need it. The platform marked is complicated; iPhone, Android, Mac, and Windows are only a small selection of available devices consumers can buy,

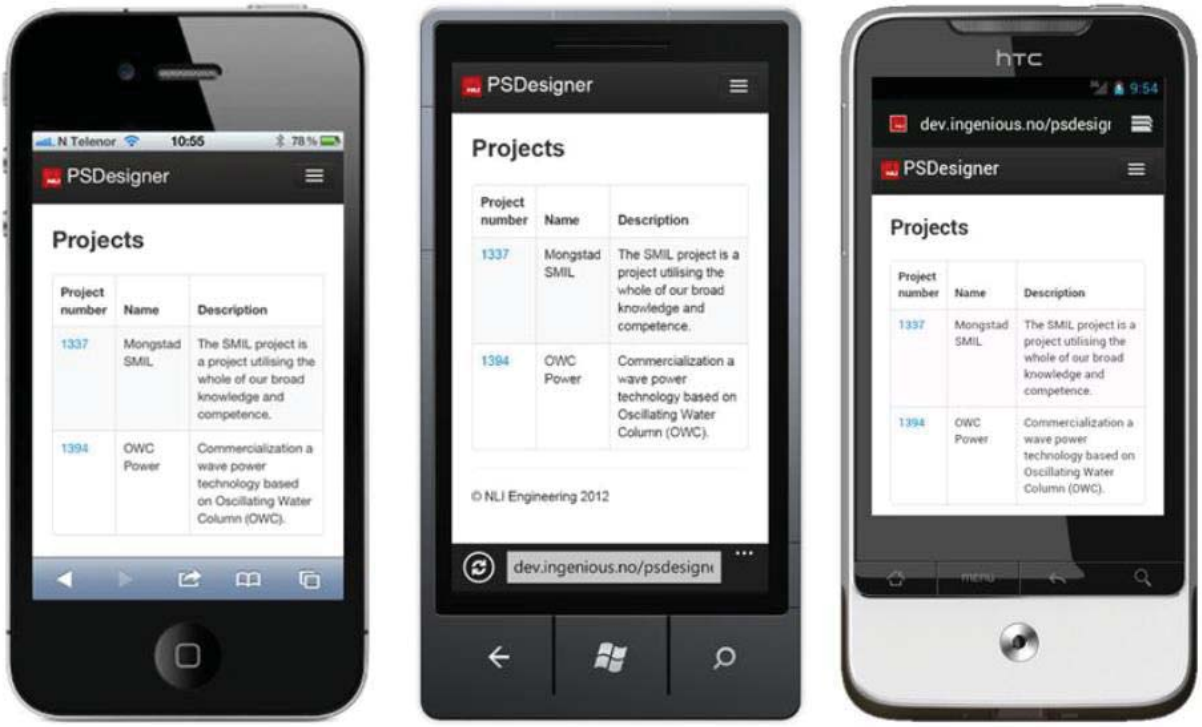

FIGURE 1 - WEB APPLICATION RUNNING ON IPHONE, ANDROID, AND WINDOWS PHONE

1.1 The history of cross-platform development

In 1976, there was no such thing as cross-platform software or cross-platform development. Before this, the Unix software was ported to new platforms and hardware. This was a heavy process requiring all packages to be ported every time there was a change to the hardware. That is why Unix introduced Make and Makefiles in 1976. This was the process of smart-compiling source files and pass command-line arguments to the compiler.

As software and hardware grew increasingly faster, more and more software packages were compatible with different hardware. Developers used separate development branches that contained only the source code necessary for a specific platform, thereby reducing the number of conditional checks done by the preprocessor. (Stuart, 2005)

Cross-platform desktop user interface has also been around a long time. Back in the 1980s this problem was mostly solved by X Window systems, also known as X11. X11 provided the basic framework for a GUI environment, such as visuals and interactions. X11 was developed at the Massachusetts Institute of Technology (MIT). Motif, one of the first widget tools, was built on top of X11. The combination of X11 and Motif provided support for cross-platform

development on all popular operating systems at that point, such as Microsoft Windows, and various flavors of Unix, such as Sun, HP-UX, and IBM's AIX.

Developers could target multiple platforms by developing one codebase, usually written in C or C++. In doing this, they would recompile the program for use on different platforms. Motif has been replaced by newer technology, but popular tools, such as Qt (1991) and GTK+ (1998), are still built on top of X11. (Coder, 2012)

Swing UI Library (Sun 1997) and Silverlight (Microsoft 2007) were both attempts at "write once, run anywhere" technology. Later developers joked about it actually being "write once, run nowhere", which illustrates how they failed in their attempts to become a technology that developers would use to create native UIs. The UI attempted to replace the native look and feel on a platform and rather sought to give a good experience with the same UI on different platforms. As Johannes Fahrenkrug says, "People choose a platform for a reason (at least sometimes): They prefer the UX of one platform to the UX of another; they don't want everything to be the same, they like that their platform is different (sometimes unknowingly). They get used to how things look and feel and they shun applications that feel odd and alien, even out of place" (Fahrenkrug, 2011).

Microsoft's Silverlight is still used by developers. It is now the native development technology for building apps on Windows 8 and Windows Phone. It is used together with C# and Xaml or Web technologies, such as HTML5, CSS, and JavaScript, to create metro apps, but it failed in becoming a leading cross-platform desktop UI technology.

The rise of the Web made cross-platform development easier. Developers could now build web applications that ran in the browser and later on that could run on smartphones and tablets. It was not up to the browser-manufacturers to follow the standards of the Web and display apps in the correct way accordingly to the developed web standards. Unfortunately, they sometimes go their own way. Thus, making a Web application look appealing can be a challenge. The Web also has the disadvantage of not being as powerful as a native program or application; thus, you cannot run tasks in the background, and you cannot save data locally.

The era of smartphones and tablets has come. Operating systems such as Apples iOS, Google's Android, Google Chrome OS, Windows Phone, Windows 8, Bada, Symbian, and Meego have introduced the concept of apps. These apps are like small programs that can take advantage of device technologies, such as camera, accelerometer and GPS.

Later, the wearable device mark has exploded. Every vendor has released or is working to release a smart watch. These watches work as an extension to your mobile phone. Pebble smart watch is a great example; Google Glass, Nike personal trainer, and Fitbit are other examples of wearable devices to which your app can respond.

Today, the art of cross-platform development is the concept of developing to all of these different devices using modern technology.

1.2 Modern devices

On January 9, 2007, Steve Jobs presented the first iPhone at "Macworld Conference & Expo". He introduced what was to become the start of the touch-based devices era. The iPhone was the first successful hand-touch-based phone. Today, the iPhone is still one of the most successful and sold phones, but it is not alone. On May 20, 2010, Google released Android version 2.2, and Microsoft released Windows Phone 7.5 in 2012. Touch-based phones introduced the concept of apps, which I cover in chapter 1.4 Modern applications.

Modern devices are more than mobile phones. In addition, there are tablets to consider. Apple was the first company to release a tablet. Its product, also known as the iPad, was released 27 January 2010. Today, all the big companies have released a tablet. A tablet is not just a big mobile device without phone capabilities. However, it is very close, at least to non-technical people.

These new operating systems, however, are not limited to handheld devices. They appear in all forms and shapes. One example is the Android Stick. This is the Android OS placed on a memory stick. Android Sticks are designed for TVs. It is great to have the possibility to play a favorite game or watch Netflix on a TV, even if the TV is a few years old. All of these growing capabilities are things to consider when building apps for an operating system.

Handheld devices are simply devices that you hold in your hand while using them. This study will focus on mobile phones and tablets. For a handheld device to be modern, it must run a modern operation system. iOS by Apple, Android by Google, and Windows Phone (WP) by Microsoft are all examples of this. Other modern operating systems include Bada, Symbian, Meego, BlackBerry10, Ubuntu, Jolta, and the upcoming Firefox OS.

When companies build applications for customers, they must target platforms that their customers use. However, simply grabbing the most popular platforms will not cut it. Imagine a company building an online message app. What good is the app if users can message their friends on other devices? If we look at Gartner's worldwide smartphone sales to end users by operating system in 4Q12 (Gartner, 2012), we can see what kind of operating systems are the most popular right now.

	Sales in thousands	Market share (%)
Android	144720,3	69,7
iOS	43457,4	20,9
RIM	7333.0	3,5
Microsoft	6185.5	3,0
Bada	2684.0	1,3
Symbian	2569.1	1,2
Others	713,1	0,3

TABLE 1 - WORLDWIDE SMARTPHONE SALES TO END USERS BY OPERATING SYSTEM IN 4Q12

This table shows how important it is for a commercial app to run on Android and iOS, with RIM and Microsoft as runners-up. These are the numbers today, but in recent years, we have seen a great shift in this marked from month to month.

The situation is similar when it comes to tablets. The following table shows the top tablet operating systems and market share in 2013 1Q13. (IDC, 2013)

	Unit Shipments (millions)	Market share (%)
Android	27,8	56,5
iOS	19,5	39,6
Windows/Windows RT	1,8	3,7
Others	0,1	0,2

TABLE 2 - TOP TABLET OPERATING SYSTEMS AND MARKET SHARE, 2013 1Q13

Android is again the big leader, but iOS buyers have download more paid apps per user. So developers should definitely target one of these. Windows has put great effort into its line of products so developing for Windows is something that also should be considered.

In addition to handheld devices, many other interesting platforms exist. Gartner says, "By 2016 85 percent of all smart-TVs will be internet connected smart TVs" (Gartner, 2012). The reason is that people are so used to their smartphones that the concept of apps running on the TV feels familiar. This also interests the manufacturers. As it is not just who can bring most of the phones any more, it is about who can develop the best chain of smart products.

Wearable devices, such as Google Glass and Samsung Gear, are very popular. Collectively, this makes the device landscape hard to determine. To limit the scope of this study, I will target building apps for phones and tablets with a focus on Android, iOS, and Windows operating systems.

1.3 Devices are a big deal

A few years ago, the device landscape was much cleaner than it is today. Companies could build their websites targeted for a regular-sized screen. However, this is unfortunately not the case anymore. Mobile is the next big thing; everybody has a mobile device.

In December 2011, there were 700,000 registered Android activations daily. In the last three months of 2011, Apple sold over 37 million iPhone units. That is more than 400,000 units daily. (Apple makes more iphones than humans make babies, 2012). For perspective and in contrast, there are only 300,000 babies born in a day. (Android activations outpacing baby births, 2012).

Mobile phones sell even more than computers. In 2011, Apple sold more iOS devices than the total number of Macs over 28 years. (Asymco, 2012) This clearly proves that mobile is a big deal. In addition, tablets are expected to sell more than computers in the fall of 2013. (When will the tablet market be larger than the pc market, 2012)

People often use their mobile devices for a large number of things. Facebook reveals that 78% of US users are mobile (Facebook-mobile-user-count, 2013). Amazon made mobile sales

totaling a billion dollars in 2010 (1 Billion via mobile devices, 2010). In addition, the most popular camera on Flickr is the iPhone camera (Flickr Cameras, 2013).

Clearly, mobile phones are the most popular device today with tablets as a good option. In addition, the personal computer is not that personal anymore, but the same cannot be said about mobile phones. Our phones contain nearly all our information: all our mail, past and present locations, personal images, schedules, messages, apps. A phone is always with you at work, at home, and even at the gym. The average person looks at his phone 150 times per day, which equals around 10 times an hour (Textually, 2012).

Mobile is important, but to create a good experience for the user, you must ensure the user can use your application at any time with any device, including phones, tablets, consoles, computers, and wearable mobile devices, such as glasses or watches. The world is changing, and the apps must as well to stay relevant.

Since a phone contains so much information about us, it is almost as if they are an extension of our body; 65% of iPhone users says, “I can’t live without my iPhone” (Künzler, 2013). In addition, 18% say, “I’ll stop bathing every day before I give up my iPhone” (Künzler, 2013). This clearly proves that devices are a big deal, and companies cannot afford to ignore devices if they want to take a big market share in the information industry.

1.4 Modern applications

A mobile application (app) is a software application designed to run on a modern device. The vendor that made the platform usually distributes them. Some applications are free, while others have a cost. The vendor usually takes 20-30% of the sale, while the rest goes to the distributer.

The term “app” has become popular, and in 2010, The American Dialect Society listed app as “Word of the Year”. (Word of the year, 2011) Mobile apps were originally offered for general productivity and information. The first apps to see the world were apps made by Apple and included email, calendar, and contacts apps. Later companies have developed apps for all sorts of things, including games, weather, and banking.

Following is a list of the most common distributors today:

- Amazon Appstore – A mobile application store for Android operating systems. Opened in March 2011.
- Apple App Store – The first app distribution service, which set the standard. Opened on July 10, 2008.
- BlackBerry App World – Apps for BlackBerry mobile devices. Opened in April 2009
- Google Play – Formerly known as the Android Market, provides apps for Android devices. Opened in On March 6, 2012.

- Microsoft Marketplace – Microsoft has its own marketplace supporting xBox, WP, and Windows 8 Metro apps.
- Samsung Apps Store - An app store for Samsung mobile phones. Founded in September 2009. Supports Windows Mobile, Android, and Bada applications.

Venturebeat estimates that we will download 70 billion mobile apps in 2013 (50% Android, 41% iOS) (Venturebeat, 2013).

Applications come in all forms and shapes, but what applications should be classified as modern? Again, we will turn to our platform providers to answer this question. Apple, Amazon, Microsoft, Research in Motion (RIM), and Google have all created their own unique design guidelines. Based on those guidelines, they can each decide which apps they will approve into their respective marketplace. The first sentence in iOS Human Interface Guidelines reads as follows: "People appreciate iOS apps that feel as though they were designed expressly for the device. For example, when an app fits well on the device screen and responds to the gestures that people know, it provides much of the experience people are looking for." (Apple developer, 2013) Clearly, a modern Android application differs from a modern iOS application device.

User interface guidelines created by vendors do not necessarily demand that an app should follow them. Guidelines are just what they prefer. We will later see about different technologies that have approaches to build single, highly branded applications that work on all popular devices. The most important aspect is how it feels and that it behaves, as the user would expect.

In addition to different design rules, devices and platforms ship with a lot of unique technology. For example, Android has unique widgets, and Windows Phone has "live tiles". For apps to be considered modern, you must be aware of their unique technology. Using this technology might encourage your users, and they may spend more time on your app. Notifications are important to keep your users up to date in real time. Below is a table with the unique technology of several devices that you might use when developing apps.

iOS	Android	Windows Phone
Glossy effects	Widgets	Live tiles
Round corners	Options menu	Panorama
Tabs	Tabs	Pivot
Social services	Action menus	Metro
Split view	Action Bar	Snap view
Air Play	NFC	NFC
Face Time	Fragments	Pinning

TABLE 3 - UNIQUE FEATURES OF DIFFERENT DEVICES

Applications can have different purposes. Many application vendors divide their apps into categories. In the Apple Store, there are currently 23 app categories, and in Windows 8, there are 20. They can all fit quite neatly into seven types (How many app types are there, n.d.).

- Utilities: Simple and handy tools to perform tasks; examples include calculators, flashlights, and note apps.
- Entertainment: Apps made to kill time and keep you entertained; examples include TV-guides and music and video apps.
- Games: Apps that are multiplayer and single-player games; examples include single player games and multiplayer games.
- Informational: Apps created to feed the user with news or share educational information, often one-way information sharing; examples include transport applications and first aid applications.
- Productivity: Apps to help you be more productive; examples include calendar and mail apps.
- Lifestyle: Apps created to enhance our lifestyle; examples include fitness, travel, and shopping apps.
- Social Networking: Social apps that give us easy access to our social services. Examples include Facebook and Twitter.

Again, all of these apps can come with different features. The following features are some examples that developers can include in their apps:

- Social Integration: Can we use social services as Identity providers, or can we connect with our friends?
- Multimedia: Can we play video and audio if we need to?
- Internet: Can we get real-time data from the internet?
- Communication: Can we communicate with others? (SMS, calls, email)
- Navigation: Can we access GPS and maps?
- User interaction: What kind of input controls can we create?
- Display information: How can we display information?

You can do most of these things with a laptop, but it is not nearly as easy as many mobile applications can do it. If we think about it, the personal computer is not very personal. However, the mobile phone is quite personal. The phone is full of personal information, such as images, contacts, and messages. Using this information in a mobile application makes it important to create a personal and meaningful app.

Being aware of the restrictions of these devices is also vital. The biggest problem for many developers is the screen size. Developers must learn how to adapt their apps and only highlight information that the user must see. Technical aspects, such as latency and bandwidth, are additional concerns. These small devices do not have power like a computer.

As stated previously, when Apple introduced the iPhone, it completely changed everything. The most important change was hand-touch-driven navigation, but that was just the start of interesting new technology developers can use in their applications. Even though this new

technology in itself does not define a modern device, it is important to be aware of what the technology can do, so we can create interesting, content–rich, and modern apps.

One of the most-used technologies in our devices today is the camera. Mobile cameras have evolved rapidly in both usability and performance. Popular mobile phones, such as the Samsung Galaxy S3 and iPhone 5, capture pictures in 8 megapixels. Many devices even have two cameras, one backend and one frontend. The main reason for using a mobile camera is that it is with you all the time, giving users a chance to take pictures whenever they feel the need. Instagram is a good example of an app built on the camera technology. Our mobile cameras cannot replace a professional camera for a professional photographer, but they are perfect for regular use.

Another interesting service is geolocation and maps, commonly referred to as location. It is easy to show the user where places are and even give them driving locations. It is common to show a map if the user wants to know where an address is located. The location service is very accurate. The fact that people prefer to use a phone rather than a map proves this.

Mobile phones and tablets also ship with an accelerometer. This is usually used in games and to control the behavior of apps. Other technology developers should be aware of included audio capture and video capture. Users no longer have to buy recording equipment. They can use applications for almost any trivial recording task. Likewise, users can use barcodes to scan images and read more information about something.

Another great feature when it comes to modern devices is that they are usually online. Push technology enables us to interact with users. Users no longer have to search for information; rather, it arrives when it is available. Cloud storage is another advantage of the internet. We can backup, sync, and even share user data without them thinking about it.

Additionally, Bluetooth has been around for a while, and it is still important to take advantage of the technology. Bluetooth is the preferred way of communication between devices that are in the same place. SMS, MMS, and calling are other capabilities we can use in our application.

Gyroscope is another new feature. Your app can respond to what orientation the user feels is more comfortable. The layout can adapt to fit the users need. If a user can browse your movie catalogue in portrait mode and then watch the movie in landscape, it might make the user happy.

Another thing to think about is the “app stores”. When you build a native app, you must submit it to the native store. Then you must decide what versions to target and which stores to publish in, and finally, you must agree to the terms of service with each vendor. This process can be time consuming and expensive, and the terms of service are usually not in your business favor. In addition, the vendor usually takes 20% of your sales profit.

Perhaps the most important question focuses on what your users expect. Many people expect an app to be in the store. This might be because users expect a modern app with better usability

than a website. As Facebook CEO Mark Zuckerberg said, "When I'm introspective about the last few years, I think that the biggest mistake we made as a company is betting too much on HTML5 as opposed to native, because it just wasn't there" (Grannell, 2012).

Mobile development is a constantly moving target. Every six months, there is a new mobile operating system with unique features only accessible with native APIs. The containers bring those to hybrid apps soon thereafter with the Web making tremendous leaps every few years.

1.5 Application packages

To get an app into an app store, you need to create a project and then compile your app into a native application. These application packages differ from platform to platform. In addition, you have to pass a series of tests to get your application approved. The process of validating an app is unique to the vendor.

If you want to publish an app into Apple app store, you need to obtain an iPhone Development Certificate. To obtain this, you need to generate a Certificate Signing Request, and then download the Development Certificate. You also need a Distribution Provisioning Profile. This profile contains important information about your app, such as the Development Certificate, an App ID, and a unique identifier.

When you have obtained a profile, you can build your application into a binary file. The package is verified using xCode build tools. You can then upload this file into the store.

To release an app into Google Play, you also have to sign your application with a key. However, you can use a self-signed certificate to sign your application. Windows phone also requires signing this certificate, which you can download after you give up some important information about the publisher. Both Google and Windows have made the signing automatic with Eclipse and Visual Studio, where a mac must access the Web to get their information.

All of these vendors will test your application before approval. First, your app runs through a series of automated tests that check for memory leaks and visuals. User testing is performed to prevent spam from hitting the market. The testing also ensures that the app follows the user interface developer guidelines and that it does not have too many bugs.

You also need to add some info about your app. This is mostly what users see when they look at an app in the store. This is information such as icons, preview screens, descriptions, and ratings. Due to different countries having different policies for age rating, it is hard to determine how to rate your app. Some countries force you to have your app before a panel that then determines an age rating.

If your application fails to satisfy the vendor’s demands, you get a detailed review of what is wrong. This can be information about bugs, design, or even the idea of the app itself. You will then need to resubmit the app after you have made improvements.

All vendor's charge you for publishing your apps. Apple takes 99$ a year and then 30% of every app you sell. This is a huge income for the vendor, and this is why many companies have their own stores.

1.6 Application lifecycle

Users can obviously start and stop an application, but an application actually has several different states. This chapter will examine those states. We will also go into details about how different operating systems handle these states differently.

First, we have iOS. iOS has five different states:

- Not running: The app has not been launched or was terminated by the system.
- Inactive: The app is running in the foreground but is currently not receiving events (may execute other code).
- Active: The app is running and is displayed by the user.
- Background: The app is in the background executing code.
- Suspended: The app is in the background not executing code.

The relationship between these states can be seen in the following figure:

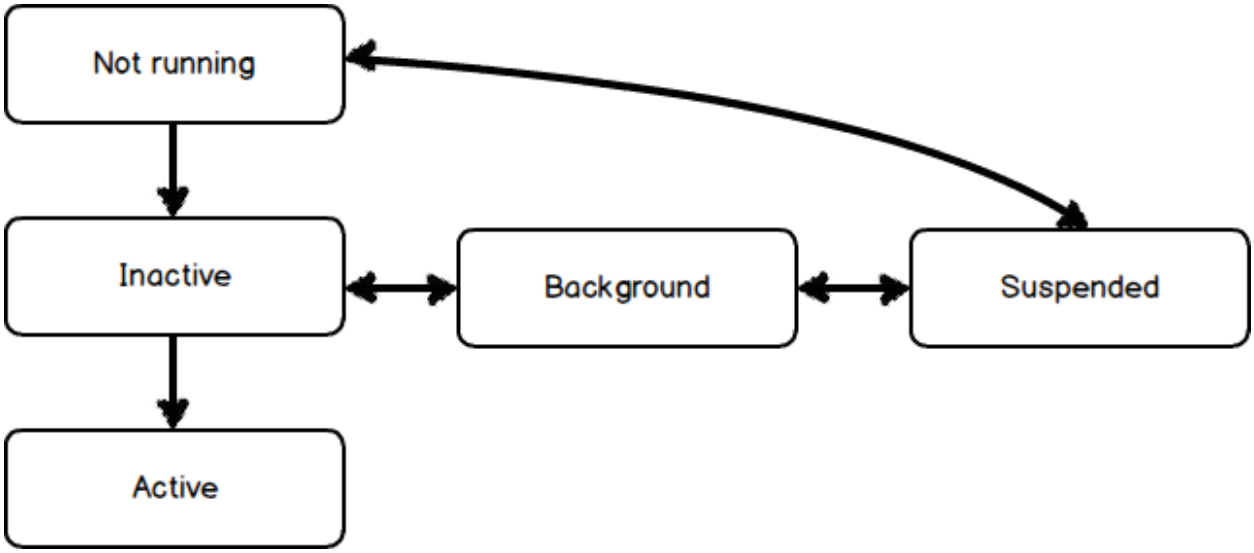

FIGURE 2 - IOS LIFECYCLE

It is important to be aware of these states because we need to handle them. Imagine an iOS user getting a call while using your app. He takes the call, and your app will be sent to the background. When he hangs up, he would like to resume doing what he was doing before the call. As a developer, it is your job to keep his previous data. In addition, it is you job to return the app to the state it was in pre-call as well.

Android, on the other hand, does not use an application lifecycle; instead, it uses an activity lifecycle. An android activity is a single thing that the user can do. Activities take care of creating windows for you, and you can build a UI for it. The activities are managed by an activity stack. When a new activity is started, it is placed in the stack, and if a user presses back, it is popped up from the stack. An activity below another activity in the stack will not come to the foreground until the top activity finishes.

An activity has four main states in Android:

- Active: The activity is in the foreground at the top of the stack.
- Paused: The activity is visible but hase lost focus.
- Stopped: The activity is obscured by another activity. It is hidden but has the same state.
- Finished: the activity is finished being used by the operating system and has lost its state.

With an activity, you can intercept state changes. The following image displays a few of the available features.

```
public class Activity extends ApplicationContext {
    protected void onCreate(Bundle savedInstanceState);

    protected void onStart();

    protected void onRestart();

    protected void onResume();

    protected void onPause();

    protected void onStop();

    protected void onDestroy();
}
```

CODE 1 - ANDROID STATE CHANGES

Windows phone also has its own application lifecycle. In consists of four states:

- Not running: The application is not started or terminated by the system.
- Running: The application is running in the foreground.
- Dormant: All threads in the application are stopped, but the application remains intact in memory.
- Tombstoned: A tombstoned app has been terminated, but the operating system keeps track of the navigation state.

All of these states are tied to different events, such as an “Application launching” event, “Page on navigated to” event, “Page on navigated from” event, “Application deactivated” event, “Application activated” event, or “Application closing” event.

The app can keep track of the application events to do global tasks, such as saving, and individual pages get intercept navigation events. This makes it possible to manage program flow, such as loading and unloading the correct data.

1.7 Game lifecycle

Many popular applications are games. Therefore, it is valuable to know how a game works if we are to create one. Even though a game lifecycle also implements the application lifecycle, it also contains a game lifecycle, often referred to as a game loop. In this chapter, we will examine that loop to be better prepared on how to make games using cross-platform technologies.

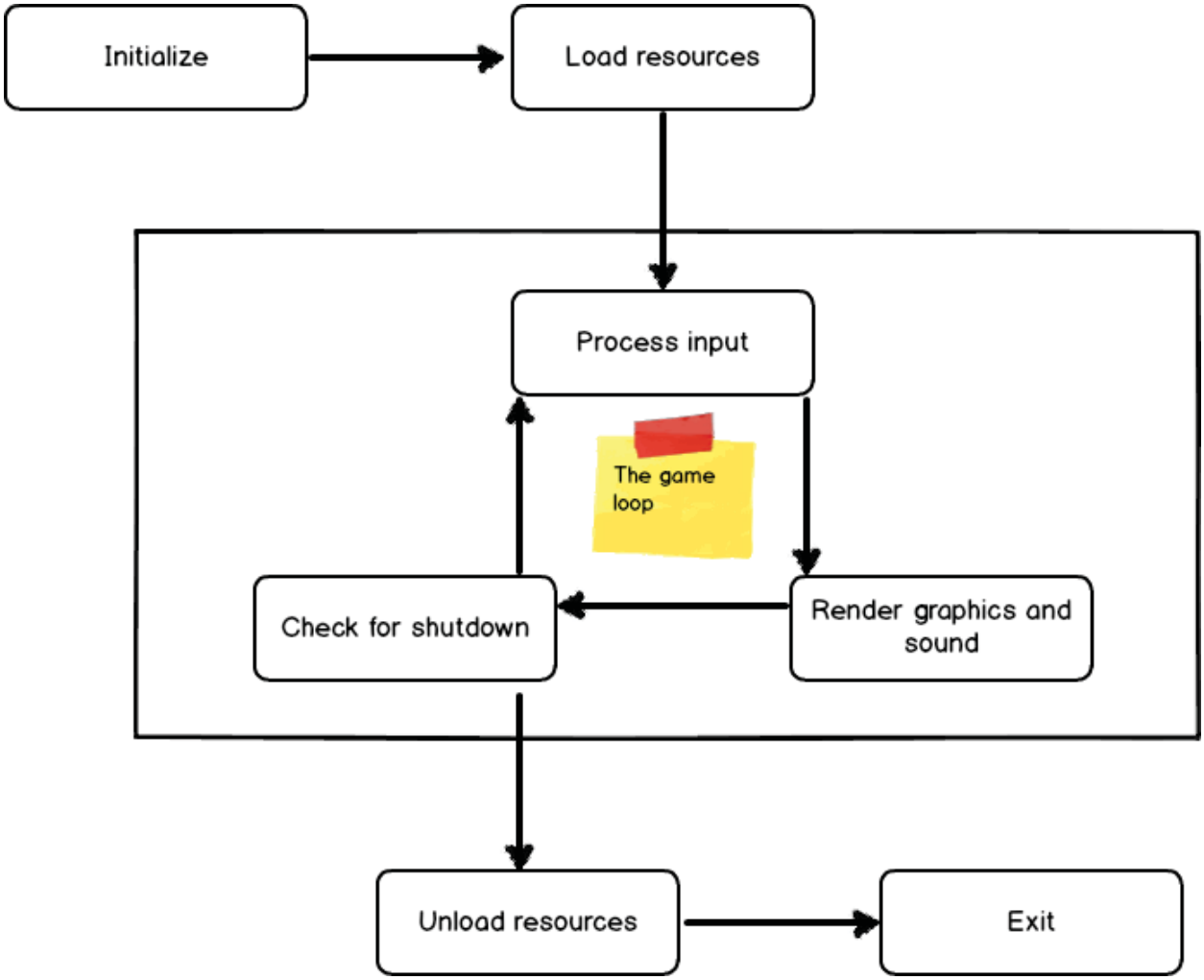

FIGURE 3 - GAME LOOP

Let us go through the different states in order of appearance.

One a user starts to play your game, the initialize method will be called. This initialize method is typically where you will configure the screen of the game. This includes decided what size it should be, and you can also define where assets, resources, and content files, such as images, sounds, and fonts, are located.

After the game is initialized, you will run load resources. In this statement, it is typical to load all resources into the games. This is done to prevent having to load resources when your game is running, which could make the game lag.

After all resources have been loaded, the game loop will start. The game loop is often configured by the user, but the simple pseudo code can look like this:

```
while( user doesn't exit )
  check for user input
  run AI
  move enemies
  resolve collisions
  draw graphics
  play sounds
end while
```

CODE 2 - GAME LOOP PSEUDOCODE

There are two important methods to register here. The update method is where you will check for user input, such as moving, taping, dragging, scrolling, etc. This is also where you would check if any exit symbols are pressed. You can actually exit the game as well. You would then perhaps follow with a set of calculations based on how the game responds to user input, and you would calculate the new positions. You would also solve any collisions made by the graphics. It is best to do everything but the drawing of the textures here. This is because drawing slow will decrease the FPS (frames per second), and your game will lag.

The other method is the draw method, which will draw all textures or sprites on your screen. As most of the heavy calculation is defined in the update, this should not take long to run. As long as you have implemented this right and do all the hard work in the update calls, mobile games typically draw every second; this is ideal.

The common FPS on Windwos Phone is 60. This means that the draw method will be called 60 times in a second.

If a game interrupt signal is cast, we typically unload resources. While this was frequently used earlier in game development, modern mobile ecosystems take care of much of this for you so that you are dependent on platform, which means there might not be too much to unload.

Finally, the game is exited and resumes back in the state of where it was before the user tapped your game.

1.8 Interaction-based taxonomy of mobile applications

Users can use an application in many different ways. (Nickerson, Varshney, Muntermann, & Isaac, 2007) Has suggested a taxonomy based on interaction between the user and the mobile application. This taxonomy consists of dimensions where each dimension has categories. All categories are collectively exhaustive and mutually exclusive, meaning that all mobile applications fall into only one of the categories within a dimension. Below is a summary of the taxonomy:

Temporal dimension: Connectivity

- Synchronous: User and application interact in real time.
- Asynchronous: User and application interact in non-real time.

Communication dimension: Input

- Informational: Information flows only from the mobile application to the user.
- Reporting: Information flows only from the user to the mobile application.
- Interactional: Information flows in both directions between the user and the mobile application.

Transaction dimension: Purchase

- Transactional: User can purchase goods or services through the application.
- Non-transactional: User cannot purchase goods and services through the application.

Public dimension: Users

- Public: Application can be used by any user.
- Private: Application can only be used by a pre-selected group of users.

Multiplicity (or participation) dimension: Group

- Individual: One user; user experiences the application as if he/she were the sole user.
- Group: Multiple users; users view use of the application as part of a group.

Location dimension: Place

- Location-based: Mobile application uses the user's location to provide customized information.
- Non-location-based: Mobile application does not use the user's location.

Identity dimension: User

- Identity-based: Mobile application uses the user's identity to provide customized information.
- Non-identity-based: Mobile application does not use the user's identity.

1.9 Application Types

Companies and individuals all around the world have built millions of apps. This section explains what approaches developers can take to develop modern apps on modern devices.

1.9.1 Native mobile app

Perhaps the most common approach to mobile development today is to make a native mobile app. To build native apps, developers must learn a native language. This means that for building an iOS or OS X app, a developer must understand Objective-C, Cocoa touch framework, and the Cocoa interface. The designer must also learn the iPhone design guidelines. Then when a company decides to build an Android version, its developers must learn to code JAVA. Learning how XML represents interfaces is also essential. In addition, the designer must rethink the design to match the Android design guidelines. This is the downside of native development. Developers must restart the process for every mobile platform they want to target.

On the upside, native apps are fast and responsive. They provide the best usability and support all features. They provide the best experience for users. When developing in the native language, there is still full access to a rich set of APIs, such as the camera or the gyroscope. Information your user has always given the phone, such as contacts or pictures, can be accessed. Having this strength allows developers to create truly modern apps that will give users a unique and personal experience.

1.9.2 Web-app

Another option for making apps is to make a Web-app. A Web-app is a website that looks like an app. Companies publish their apps on their own servers, and they do not get the benefits of being in the app stores. Common technologies, such as HTML5, CSS, and JavaScript, are standard Web technologies typically used to build Web-apps. Web-apps support is a kind of write-once-run-anywhere approach to cross-platform development.

Web-apps cannot access device-specific features, which makes them unfit for many applications. However, Web-apps are important to many companies, and it is common to provide an alternative homepage to mobile users. Important challenges when it comes to building Web-apps are to provide a good experience on many different resolutions and to support many of the old Web browsers.

Even if the latest browsers support hardware-accelerated CSS3 animation properties for transitions between screens, they cannot match the power and flexibility of native apps.

1.9.3 Shell-app

A Shell-app is much like a Web-app and sometimes referenced as a hybrid app. A Shell-app is a Web-app wrapped in a native shell, usually by using common native components, such as webview or webcontrol. This lets Web developers publish their apps to stores. In addition, the wrapping also comes with some form of bridge to the native API, making it possible to create a personal experience and use native APIs.

Some people claim this empowers the best of both worlds. You can write the Web-app as you like, and now, you can publish it on your servers and in the stores. You even receive access to APIs to create a personal user experience. While some say it is the best of both worlds, shell-

apps come with many problems. They are hard to test on many devices. You rely on a third party, and performance can be an issue. In addition, the users expect apps they download to be modern and follow the design rules.

Plus, how do we design the user interface? If we develop a shell-app, how do we avoid the uncanny valley? Do we want it to look like a native app? Do we want to make our existing site responsive? Is a simple UI with big buttons and the most essential information enough?

Shell-apps have grown a lot in the last years, and they will continue to grow since phone manufacturers keep improving hardware. The performance gap between native apps and shell-apps will decrease more and more. Shell apps have a great future, but at the moment, they are not suited if you want your app to be industry leading, which many apps rely on happening to be a success.

1.9.4 Compiled-app

A compiled-app is much like a native app. You compile your code into bundles and deliver them through app stores. The difference from a native app is that it supports some cross-platform development. Usually, the backend, and sometimes the frontend, can share code across operating systems. Compiled-apps are often built on top of the native language, which makes it possible to integrate the compiled code with the native compiled code.

Since the compiled code essentially compiles into the same code as native language, it has the ability to perform at the same speed. A compiled-app is written in a compiling language, such as Java or C#. However, compiled apps also have a few disadvantages; for instance, having a layer between the native code and the development code may introduce bugs. The compiling language is kept up to date, and it has support for most features.

1.9.5 Overview

Below is a comparison of the various application types commonly developed today.

	Native	Web	Hybrid	Compiled
Language	Native	HTML5, CSS, JavaScript	HTML5, CSS, JavaScript	A Compiling language
Performance	Fast	Slow	Slow	Fast
Look and feel	Native	Emulated	Emulated	Native
Distribution	Appstore	Web	Appstore	Appstore
Graphics	Native APIs	HTML, Canvas, SVG	HTML, Canvas, SVG	Bridged Native APIs
Device specific Featurs	All	Few	All/Moderate	All/Moderate
Access to Native APIs	All	None	All/Moderate	All/Moderate
Gestures	All	Few	All/Moderate	All/Moderate
Notifications	Yes	No	Yes	Yes
Connectivity	Online and offline	Online	Online and offline	Online and offline
Offline storage	Secure storage	None	Secure storage	Secure storage
Code portability	None	Full	High	Moderate
Upgrade flexibility	Low	High	Moderate	Low/Moderate

TABLE 4 - COMPARISON OF APPLICATION TYPES

1.10 Research question

Defining a research question about cross-platform development is not trivial. With new frameworks added every month and new operating systems released every year, it is impossible to test them all.

This paper will focus on researching how native applications can be built using cross-platform technologies. Therefore, my research question will be as stated below:

"How can we build apps that have a native look and feel by using cross-platform technologies, and when should we use it?"

This question, of course, also leads to several underlying or more minor questions.

What tools can be used to develop cross-platform apps? I will examine the current content of apps available to developers today and compare them to each other. While it may be impossible to cover all relevant tools, the discussion should provide a good overview.

What is native look and feel? What is a native look and feel, and how can we measure it?

Can cross-platform development suites compete with native? What development tools is it possible to get if choosing a cross-platform approach. How do these tools compare with native tools?

What architecture pattern is best suited for cross-platform development? I will examine what architectures are best suited to develop an application.

What options do we have for sharing code in cross-platform development? Which tools do we have for code sharing, and how do we use them?

How much code can we expect to share in cross-platform development? When developers are creating a cross-platform application, how much code can they expect to share, and how can they get this number as high as possible?

Can cross-platform development save development time? Is cross-platform mobile development useful? Do developers save time on it, or will it create obstacles that are hard to overcome? How good is a cross-platform ecosystem?

What cases is cross-platform development suited for? Are cross-platform apps suited for everything, or is it only a small portion of apps that are eligible? Does cross-platform development combine well with all features?

Does cross-platform development make any limitations to our project? What other hiccups we have when creating cross-platform apps?

In the following chapters, I will try to answer those questions based on different methods. Discussion, development, and testing are essential for this to be a successful project.

2 General discussion

2.1 Feature based taxonomy of mobile applications

To evaluate cross-platform frameworks, I propose the following taxonomy. This is based on what you can do with the application and what part of built-in APIs you can use. The following taxonomy summarizes the most important features in modern applications. The taxonomy consists of dimensions where each dimension is divided into specific categories. A mobile application may fall into none or all of a category within each dimension.

Capture dimension:

- Image capture: Application can use built-in camera(s) to take and use images.
- Video capture: Application can use built-in camera(s) to record and use videos.
- Audio capture: Application can use built-in microphone(s) to record and use audio.

Location dimension:

- Geolocation: Application can determine the real-world geographic location of the device.
- Compass: Application can determine stationary directions relative to the surface of the earth.
- Native Maps: Application can use the built-in visual representation of an area.

Storage dimension:

- File system: Application can use the file system to save information.
- Database: Application can use an internal database to store data.
- Cloud: Application can use built-in cloud systems to store data.

Movement dimension:

- Gyroscope: Application may determine the orientation of the device.
- Accelerometer: Application may monitor and determine the acceleration of the device.

Phone dimension:

- SMS: Application can read and send SMS.
- Call: Application can make calls.
- MMS: Application can read and send MMS.

Personal dimension:

- Contacts: Application can read, create, and delete contacts.
- Pictures: Application can read, create, and delete images.
- Mail: Application can read, create, and delete mail.
- Calendar: Application can read, create, and delete events from the calendar.
- Identity: Application can read and update personal information on the device and user.

Gesture dimension:

- Tap: Application can register tap events on the screen.
- Double-tap: Application can register double-tap events on the screen.

- Flick: Application can register flick events on the screen.
- Rotate: Application can register rotation events on the screen.
- Swipe: Application can register swipe events on the screen.
- Pan: Application can register pan events on the screen.
- Hold: Application can register long press events on the screen.
- Spread/Pinch: Application can register zooming events on the screen.

Input dimension:

- Keyboard: Application has access to keyboard and can register input.
- UI Widgets: Application has access to a rich set of UI widgets.
- Online: Application may asynchronously retrieve information from other sources.

Unique dimension:

- In-app purchase: Application can use native-app purchases.
- Push technology: Application can use device-specific in-push notifications.

2.2 Types of tools to enrich mobile development

To build cross-platform applications, we need tools, frameworks, and libraries. Since the beginning of 2009, there has been an explosion of new frameworks and tools to address mobile development. To cover everything out there would be an impossible task. I have chosen the most popular and powerful frameworks and will give a small review of them and their place in the landscape.

I have tried to divide the tools into different categories depending on their capabilities and features. It can be argued that this list could be separated even more as some products include visual designers and some products build on other products. However, the following classification gives the best overview.

2.2.1 HTML Interface

An HTML interface is simply a JavaScript library. It consists of JavaScript, HTML, and CSS. The libraries exist so that developers do not need to start coding from scratch. You usually get a set of UI widgets that looks native. Common components, such as Buttons, Textboxes, Labels, Checkboxes, and Maps, are easy to implement, and usually, you only have to use a few lines of code. It is also common to have some sort of responsive design, simplifying the process of making sites for many resolutions.

The JavaScript framework itself has nothing to do with building native apps, but it is a rather good tool to create web applications that look native. It is common to use HTML interfaces in combination with shell-wrappers to simplify the app development process.

During the process of reviewing the top HTML interfaces, I have come across many different projects. The ones represented here are those I found to be best in terms of documentation, community, and general impression.

2.2.1.1 jQuery Mobile

JQuery Mobile has a broad base of users. Built on top of jQuery, it is a very popular tool for creating Web-apps. JQuery Mobile has broad support in all modern browsers. JQuery Mobile looks upon itself as a "touch-optimized web framework for smartphones and tablets" (About jQuery, 2012). It comes with a broad set of documentation formats that includes books, examples, tutorials, and a gallery. The approach is to build a single, highly branded application that works on all popular devices. In other words, JQuery is not about designing different UIs for different devices; rather, it is about creating a portable UI that looks good on all of them. This gives the ability to share 100% of the code.

On the downside, jQuery apps do not look really impressive. I would argue that the jQuery Mobile is built for iOS as the app looks native on iOS, thus making it look unnatural on Android and Windows Phone. In jQuery mobile, developers can build themes with a web application called Theme Roller. Theme Roller is a drag-and-drop widget to tailor the templates as you need them.

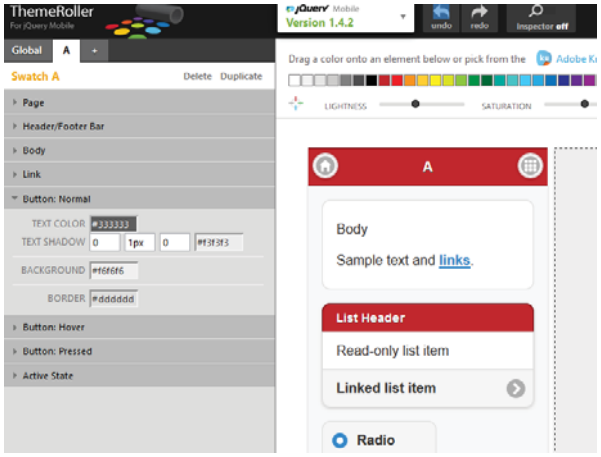

FIGURE 4 - JQUERY MOBILE THEME ROLLER

jQuery Mobile also comes with a great demo app where you can explore all widgets. This is available at the jQuery documentation website[1]. While jQuery is easy to use and has a large community and a great set of documentation, jQuery Mobile is a little too much of an iOS lookalike and does not work very well on other platforms.

2.2.1.2 Dojo Toolkit

Dojo Mobile is a JavaScript library that includes 25 mobile widgets, such as audio, video, view, and buttons. Dojo is free and supports any html5-based browser. The JavaScript library has a size of 4KB zipped. The documentation on the Web is good as it features demos, tutorials, examples, and a support forum. One important feature of Dojo is the CSS themes. These themes provide different styles for iOS, Android, and Blackberry-based devices, making the app look

[1] http://demos.jquerymobile.com/1.4.2/

truly native. Dojo also ships with CSS3-based animations designed to look native. In addition, it responds to both orientations as well as orientation changes.

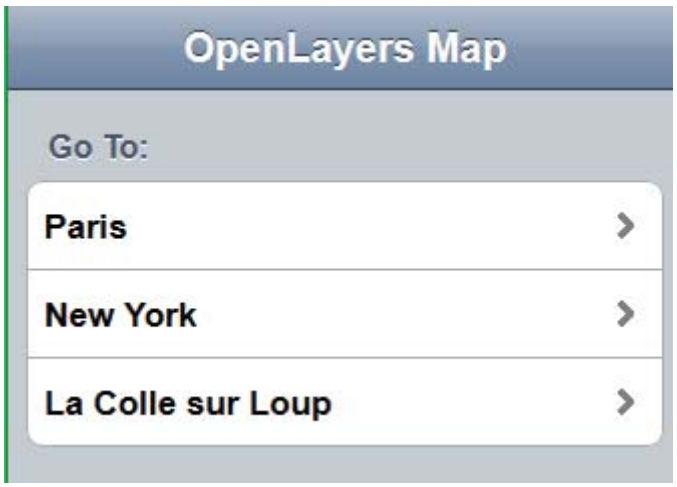

FIGURE 5 - DOJO TOOLKIT DEMO

Digging deeper into Dojo Toolkit, we clearly see some drawbacks. Dojo does not have a theme for Windows Phone, and it seems to lack widgets designed for tables like the popular iOS split view.

2.2.1.3 Kendo UI

Teleriks Kendo UI is a very popular tool to create Web sites. Kendo UI takes the approach that different platforms use different CSS files. This makes the app look native. Kendo UI supports Android, iOS, Windows Phone, and Blackberry. Kendo UI cost 199$. Kendo UI has a rich set of tools and widgets and seems to be easy customizable.

Looking at Kendo UIs demo project is impressive; you can find it at their demo website[2].

[2] http://demos.telerik.com/kendo-ui/.

FIGURE 6 - KENDO UI ON IOS, ANDROID AND WINDOWS PHONE

Kendo UI also ships with a theme roller that you can use to customize your applications. This looks really promising to anyone building shell apps.

2.2.1.4 Topcoat

Where many HTML-interfaces prioritize looks before performance, Topcoat is one that does not. Topcoat is developed to be fast. Topcoat also has support for MVVM architecture. This makes it possible to combine Topcoat with popular MVVM-based frameworks, such as angular, so it is possible to use this with an angular-based shell wrapper, which is a huge benefit if your Web application uses that, meaning this might be a good product for you.

Topcoat ships with four themes, a mobile and a desktop in dark and light colors. Unfortunately, Topcoat does not offer you any native look. This is the interface for angular builders who are looking for a performance boost.

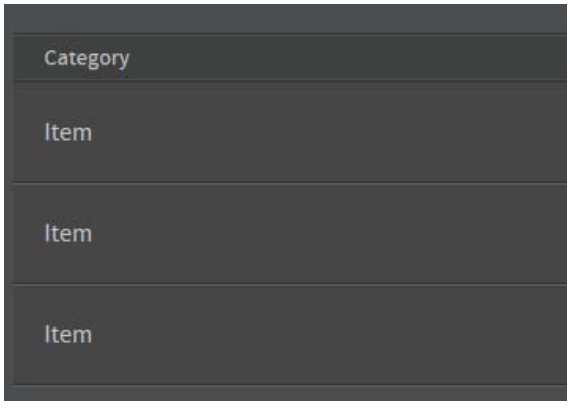

FIGURE 7 - TOPCOAT LIST VIEW

2.2.2 Shell-wrapper

A shell-wrapper is a tool that lets you create shell-apps. A shell-wrapper wraps a website into a native app using common native components, such as webview or webcontrol. Some shell-

wrappers also come with a Visual designer or simple HTML interface widgets. There are many different shell-wrappers out there today. Shell-wrappers rely on JavaScript libraries to provide a good user experience. Sometimes an HTML interface ships with a Visual designer built on an IDE. The IDE has drag-and-drop functionality that lets you create many of the widgets without the need to code. This is also a very common feature in native development IDEs, such as xCode, Eclipse, and Visual Studio.

2.2.2.1 Application Craft

Application Craft is an interesting UI builder. It is cloud based, so you can do drag-and-drop UI design in the browser. The price of this service is $14 each month. This technology is good for inexperienced developers. For experienced developers, this kind of technology might be perfect for prototyping. The browser-based IDE seems clean and consistent; Application Craft also has a built-in plugin with PhoneGap.

Most developers prefer to actually have an IDE to build applications; building code in a webview is uncommon for many. Therefore, Application Craft has not yet gained the popularity it might deserve.

I am also very concerned that this product will not perform well during speed tests, and by the look of the project, it is not native. If this platform keeps growing, it might be very usable for prototyping.

2.2.2.2 Appcelerator Titanium

Appcelerator Titanium is a little bit different from other shell-wrappers as much of the code is actually compiled into native code, and only parts of the code are embedded inside a Web view. Titanium ships with an open-source SDK with over 5000 device and mobile system APIs, an eclipse-based IDE, a custom MVC framework called Alloy and a mobile backend called cloud services. Titanium does not have support for Windows Phone.

2.2.2.3 DHTMLX Touch

DHTMLX Touch is a JavaScript library. It ships with a visual designer, a skin designer, and touch UI Inspector. It is targeted towards iOS and Android, but in theory, it can be used in many browsers. The last update from this project is from 13 September 2012. Since the mobile landscape is changing swiftly, I doubt that DHTMLX Touch has enough features to be useful. The community seems rather small and a lot of documentation is lacking.

2.2.2.4 PhoneGap

PhoneGap is a shell-wrapper where developers can build applications using HTML5, CSS, and JavaScript. The framework wraps the final webpages into a native “Web View” with ability to access many device-specific APIs. PhoneGap can deploy to a wide variety of platforms, such as iOS, Android, Bada, Windows Phone, BlackBerry, webOS, and Symbian. PhoneGap has

been around since 2008. It has a broad user base, and it is therefore easy to obtain documentation and examples. Adobe owns PhoneGap, and PhoneGap is open source,

PhoneGap is perhaps one of the most popular shell apps, and many other shell apps are built on top of it, not without a good reason.

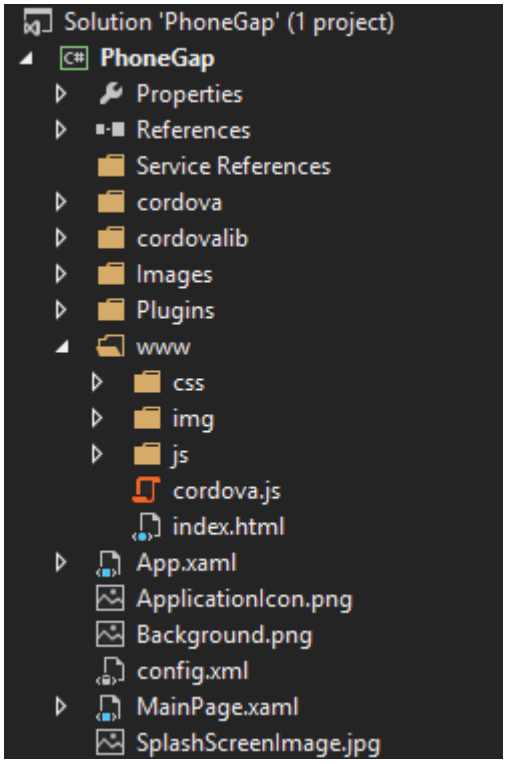

FIGURE 8 - PHONEGAP PROJECT RUNNING IN VISUAL STUDIO

PhoneGap offers templates for Visual Studio, xCode, and Eclipse. It is also possible to use the native debugger to debug your app. PhoneGap brings the power of the Web to mobile. You can install a Web framework like knockout, backbone, or angular into it to improve the performance. While PhoneGap is at the top in shell apps. It still is remarkably slower than a native app, but for small apps with small user groups, such as an in-house application, PhoneGap might be the best path to follow.

2.2.2.5 Corona

Corona is another shell-wrapper built on technologies like OpenGL, OpenAL, and Lua, making it optimal as a game engine as well. It is also possible to extend with Corona Enterprise that allows developers to call native libraries. Unfortunately, Corona can only build apps on iPhone and Android.

2.2.2.6 Adobe AIR

Adobe AIR is a combination of WebKit, Adobe's own flex technology, and ActionScript. It can be a game engine, a media producer, or simply a tool to create html shell apps. Adobe recently decided to discontinue further support for Linux. Adobe can create native apps for

Windows 8 classic, Mac OS X, Android, iOS, and some televisions. It is widely used with Adobe Creative suite, but that is not required. Adobe has made a huge commitment to the mobile experience. Adobe AIR does not yet have support for Windows Phone.

2.2.2.7 AppBuilder

AppBuilder is another product from Telerik. It is based on Icenium (a mobile CMS system). AppBuilder has support for many IDEs but also a cloud client that you can access online. AppBuilder uses Apace Cordova to wrap apps in a Web layer. It has support for Windows 8, iOS, and Android and has a rich set of features. By default, AppsBuilder apps are installed with Kendo UI.

AppBuilder can be used with Telerik Platform, an end-to-end platform for hybrid apps. You also get access to a backend cloud, app analytics, and a rich set of UI tools. There is a lot of documentation available, and AppBuilder seems like a good product.

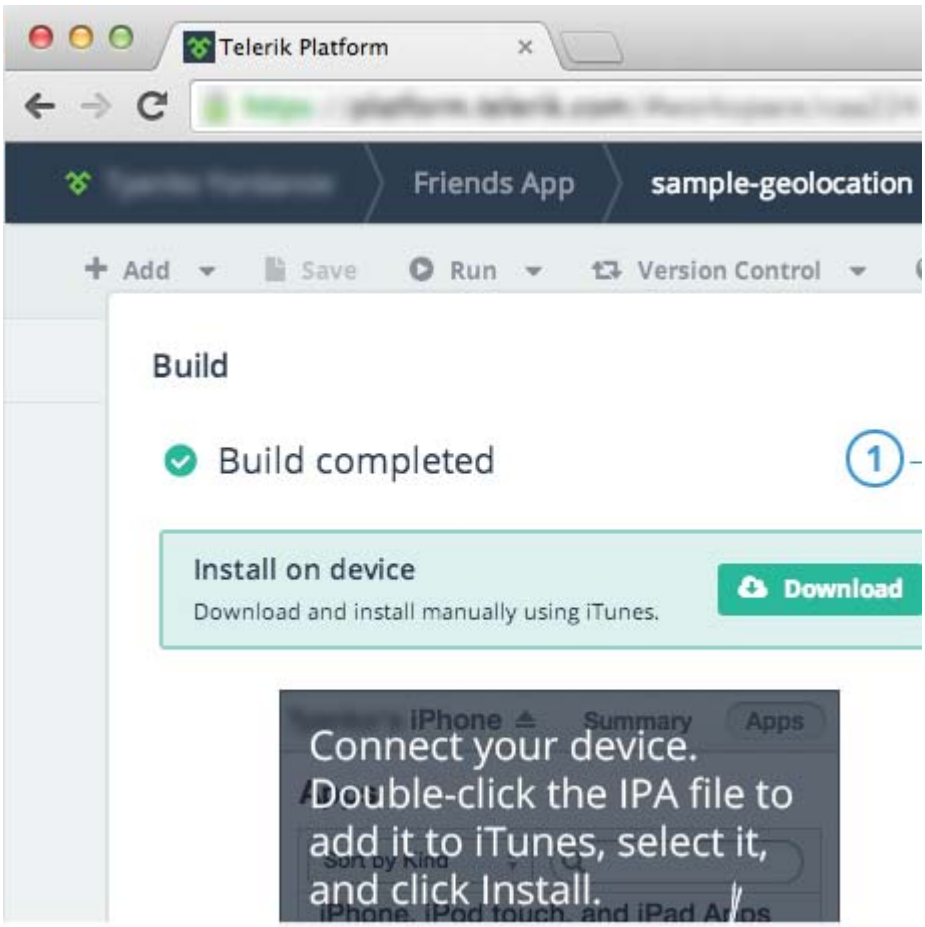

FIGURE 9 - TELERIK APPBUILDER IDE

2.2.3 SDK

Perhaps the most interesting group I named is SDK. These tools provide functionality to create truly native apps with a cross-platform approach. SDKs often ship with their own IDE or rely

on the native IDEs. They are usually not built on HTML5, JavaScript, and CSS, but rather on a native language, such as C++, C#, or even JAVA. Their secret is to compile the source code to different bundles depending on the platforms you target. The bundles can also interact with native bundles, giving a huge advantage for creating modern apps on modern platforms. One drawback is that you sometimes have to create native UIs, which means you might have to learn a new language or framework.

For all SDKs I have invested some time in researching functions and possibilities. They are measured in percent. At the end is the summary of the percentage. This will reflect how good this project is and in the end, it will decide which SDK framework I will continue using.

2.2.3.1 Xamarin/Mono

Mono is an open-source implementation of C# targeted towards running C# code on Linux and Mac. Xamarin has additionally created Xamarin for iOS and Xamarin for Android. In the table below I have done a detailed measure on Xamarin and how well I think it works in different areas.

Function	Score	Comment
Platform support	90%	Initially mono provides support for Linux, Windows Phone, and Windows. Xamarin has ported the product to iOS, Mac, and Android.
Shared Code	70%	Mono works by creating native UIs, so the UI code itself is not shared. Backend code should and can be shared.
API Support	100%	Mono integrates with native binaries so that any native code can also run in mono by adding a shell to it. Mono for iOS was released for iOS 7 at the release date.
Technology	100%	Mono code is written in C#; in addition, native UI code can be written in Native code, such as XAML.
Availability		
- Installation	80%	One installer that is easy to run in Mac and Windows. Setting up iOS support on Windows requires some copy-paste of assemblies between different operating systems. Portable class libraries also require some modifications to the environment.
- Development environment	100%	On Windows, developers can use a plugin for Visual Studio. There is also Xamarin Studio, which can run on both Mac and Windows. Xamarin Studio has support for building iOS user interfaces.

- Support	70%	It is easy to get support from Xamarin forums. You can also buy professional support.
- Documentation	100%	You can find a lot of documentation and samples on the Xamarin document page.[3]
- License	50%	Xamarin is somewhat expensive for a business edition of iOS. For Android and Mac, the price is $999 per platform, per developer. Mono itself is open source, so Windows products are free.
- **Total**	80%	
Maturity		
- Versions	100%	Mono for iOS had support for iOS 7 at the release date. New versions are pushed often.
- Partners	100%	Xamarin has partnership with many big companies, including Microsoft and Sap
- Community	80%	Even though Xamarin is a new company, the customer base is big. There are many user groups for Xamarin, and The Evolve conference is growing in popularity. Stackoverflow has a good community that answers questions fast.
- Components	80%	Xamarin Component store has a lot valuable components, from cloud storage to user interface plugins.
- **Total**	90%	
Sum	90%	

TABLE 5 - EVALUATION OF XAMARIN

Mono seems very promising. It is interesting to build native UIs and then use shared backend code to send and receive data to the viewers. The Xamarin products make this package much more interesting because of the support to iOS and Android.

2.2.3.2 RhoMobile Suite

RhoMobile suite consists of the following:

- RhoElements: (UI elements)
- Rhodes: a Ruby-based framework built to create native
- RhoConnect: connect mobile application to business data
- RhoStudio: an IDE.
- RhoHub: an online IDE
- RhoGallery: a mobile app management studio that lets users upload apps

In the table below are the scores and feature review for RhoMobile suite.

[3] http://docs.xamarin.com/

Function	Score	Comment
Platform support	70%	iOS, Android, and Windows Phone 8.
Shared Code	100%	All code is shared.
API Support	60%	Support all common features. The full list of support can be found at Rhodes API Compatibility: http://bit.ly/1hAVAsz Can be very problematic if the API does not support the feature you want since it does not interact with native binaries.
Technology	90%	RhoMobile is based on Ruby and JavaScript.
Availability		
- Installation	100%	One installer.
- Development environment	50%	RhoHub is a cloud-based Web environment. To develop offline, you must use a simple text editor or RhoStudio, which is not very mature.
- Support	50%	Support from forum and website.
- Documentation	80%	You can find a lot of documentation and samples. Some of the documentation is not updated for earlier versions.
- License	70%	RhoMobile is free if you develop an open-source application under GPL v3. Commercial apps cost $500 per application.
- **Total**	70%	
Maturity		
- Versions	100%	New versions with more API support are released continuously.
- Partners	100%	Motorola has a wide partner base. It is also driven by Google.
- Community	50%	The community seems to be at a middle size. No conference and user groups can be hard to come by in medium cities.
- Components	50%	Not easy to find components, Mototola has no store where browsing through different versions is possible.
- **Total**	75%	
Sum	76%	

TABLE 6 - EVALUATION OF RHOMOBILE

It is a shame that RhoMobile does not provide full support from all vendors to all APIs. Motorola has a promising product, but it does not support all native APIs therefore it seems to be missing a bit of functionality to make it interesting.

2.2.3.3 Qt

The Norwegian company Trolltech started developing Qt in 1991. The first editions of Qt were Qt/X11 for Unix and Qt/windows for Windows. At the end of 2001, Trolltech released Qt 3.0, which added support for OS X. Nokia acquired Trolltech ASA on 17 June 2008. At this time,

Qt supported the Symbian operating system. In February 2011, Nokia announced its decision to drop Symbian. One month later, Nokia sold Qt to Digia with the goal of taking Qt to Android, iOS, and Windows 8

In the following table, the measurements for Qt are displayed and discussed:

Function	Score	Comment
Platform support	70%	iOS, Android, and Windows 8.
Shared Code	100%	All code is shared.
API Support	80%	Supports most features. Minus for not having a supported feature list since it can use native packages.
Technology	80%	Based on C++, many of the newest technologies do not have official support for C++, such as SignalR.
Availability		
- Installation	100%	One installer.
- Development environment	80%	Ships with Qt Creator, but also possible to use Eclipse.
- Support	100%	Unlimited.
- Documentation	80%	You can find a lot of documentation and samples. Some of the documentation is at the Nokia developer site.
- License	60%	Commercial license from $149 per month.
- **Total**	70%	
Maturity		
- Versions	100%	Somewhat long distance between new major releases.
- Partners	80%	Some partners, but relatively small in size.
- Community	50%	The community seems to be at a middle size. No conference, and user groups can be hard to come by in medium cities.
- Components	50%	Not easy to find components. Diaga has no store that can be browsed through for different versions.
- **Total**	70%	
Sum	79%	

TABLE 7 - EVALUATION OF QT

Qt seems promising. Qt was released with support for iOS and Android on December 12, 2013, so it is a relatively new framework. Qt can also be used as a game engine.

2.2.3.4 Codename One

One is an SDK built in Java; it has support for iOS, Android, and BlackBerry. One transfers Java bytecode to native C/Objective C code and is compiled using xcode. Large parts of Codename One were inspired by the design of Swing.

Codename One and the possibilities are discussed and analyzed below.

Function	Score	Comment
Platform support	70%	iOS, Android, Windows Phone, and BlackBerry10.
Shared Code	90%	All code is shared. UI is built using a UI builder. You can apply different themes to different devices.
API Support	100%	Supports most features. Since it is open source, you can also create missing features yourself.
Technology	100%	Based on Java, which means Android apps will be native apps.
Availability		
- Installation	100%	One installer.
- Development environment	100%	Eclipse, Netbeans, or IDEA.
- Support	0%	None.
- Documentation	50%	Some documentation and samples.
- License	100%	Free and open source.
- **Total**	70%	
Maturity		
- Versions	100%	Somewhat long distance between new major releases.
- Partners	80%	No partners, but some JAVA consultants invest time and resources to Codename One.
- Community	50%	The community seems to be at a middle size. No conference, and user groups can be hard to come by in medium cities.
- Components	50%	Not easy to find components. No store that can browsed through for different versions.
- **Total**	70%	
Sum	83%	

TABLE 8 - EVALUATION OF CODENAME ONE

Codename One is relatively young. Once developers start to embrace this technology, it will be a very good option for cross-platform development.

2.2.4 Game engine

A Game Engine is a system designed to create games. Game engines extend with libraries that make the game development process simpler. They consist of objects such as Vectors2D, Vector3D, sprites, and a coordinate system. In addition, a game engine provides you with a game loop. Games have a big advantage when it comes to design. A game can look similar on all platforms, making cross-platform development an interesting topic.

2.2.4.1 Monogame

Monogame is a game engine built on top of Mono and the XNA framework. It provides support for many platforms, including iOS, Android, Windows Phone, Windows 8, Xbox, and PlayStation. It is an open-source product designed to make simple, two-dimensional apps.

2.2.4.2 Unity 3D

Unity 3D is the only 3D-based game engine that runs cross-platform today. It has a large community and a very active user group. The documentation is great with books, videos, tutorials, and examples. Unity 3D is a little expensive depending on the add-ons you buy. Unity can build apps for almost any device, including iOS, OS X, Linux, Android, Windows Phone, and Windows 8.

2.2.5 No-coding designer

A no-coding designer is exactly what the words say. It is an IDE or a Web page where you can go through a series of steps designing and implementing your own app without ever touching the code.

2.2.5.1 AppsBuilder.

AppsBuilder is a no-coding designer; it provides many features, such as Facebook integration, QR-code reading, and radio. There is also some support for JavaScript coding. It can compile apps for many devices, such as iPhone, iPad, Android, and Windows phones.

2.2.5.2 AppMakr

AppMakr is a step-by-step app builder. It lets you create apps by entering RSS feeds, and you can customize the app by going through a series of step. It compiles apps for iPhone and Android. It is free, but you can buy a pro version for $79. AppMakr seems great at building RSS feed apps.

2.2.6 Overview of tools

The following table shows an overview of all tools covered in this chapter as well as some others that I did not cover in detail

Name	URL	Type
Monogame	http://www.monogame.net/	Game engine
Unity 3D	http://unity3d.com/	Game engine
Uno	http://www.outracks.com/	Game engine
Batterytech	http://www.batterytechsdk.com/	Game engine
Dojo	http://dojotoolkit.org	HTML interface
IWebKit	http://snippetspace.com/portfolio/iwebkit/	HTML interface
jQT	http://jqtjs.com/	HTML interface
jQuery Mobile	http://jquerymobile.com	HTML interface
KendoUI	http://www.kendoui.com/	HTML interface
AppMakr	http://www.appmakr.com/	No coding designer

AppsBuilder	http://www.apps-builder.com/en/home	No coding designer
Mono	http://xamarin.com/monotouch	SDK
MoSync	http://www.mosync.com/	SDK
Codename One	http://www.codenameone.com/	SDK
Qt	http://qt.digia.com/	SDK
Rhodes	http://rhomobile.com/products/rhodes/	SDK
Sencha Touch	http://www.sencha.com/products/touch/	Shell-wrapper
Adobe AIR	http://www.adobe.com/products/air.html	Shell-wrapper
Appcelerator/Titanium	http://www.appcelerator.com/	Shell-wrapper
Corona	http://coronalabs.com	Shell-wrapper
Intel XDX	http://html5dev-software.intel.com/	Shell-wrapper
PhoneGap	http://www.phonegap.com/	Shell-wrapper
Application Craft	http://www.applicationcraft.com/	Visual designer
DHTMLX Touch	http://www.dhtmlx.com/touch/	Visual designer

TABLE 9 - LIST OF CROSS-PLATFORM TOOLS

2.2.7 Continuing on

While there are many tools that could be examined and tested in this paper, I wanted to focus on one of them and then go a bit deeper into that implementation. Since it is important with native speed and performance, I want to try to use an SDK to develop a compiled app. The chosen SDK is Xamarin/Mono because that it is the most complete product at the time I am writing this.

I will also experiment a bit with monogame since it touches upon many of the same topics as Xamarin. It will give me a chance to develop with and analyse the entire ecosystem, which is one of my research questions.

The rest of this paper will, in general, apply to Xamarin. However, many of the concepts described are valid for other platforms and tools.

2.3 Code-sharing strategies

As Xamarin and Mono are now selected as the developing framework, it is time look at the ecosystem for sharing code. While in reality you could combine many approaches, there are two common ways to execute a code-sharing strategy. In this chapter, we will go through them and see how they work and in which situations you want to apply them.

2.3.1 File linking

The simplest solution apart from copy/pasting files to other files is file linking inside each project. I will set up a few new projects, and then I will go into detail on setting up these projects in experiments in Chapter 4.

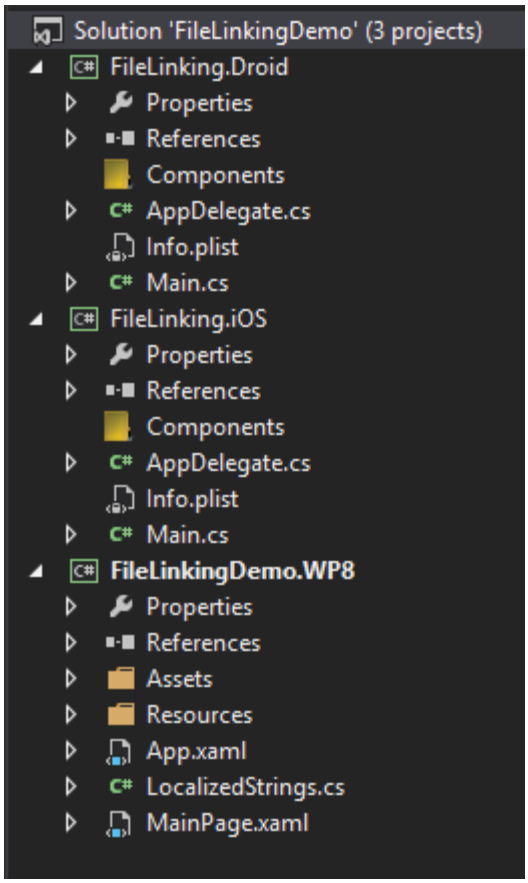

FIGURE 10 - XAMARIN PROJECT STRUCTURE FOR FILE LINKING

Each project can be set up individually, and then, I can add a new file into one of the projects. In the other projects, I can then select 'Add existing files', and navigate to the file just created in the popup menu for adding a select linked file. This is illustrated here:

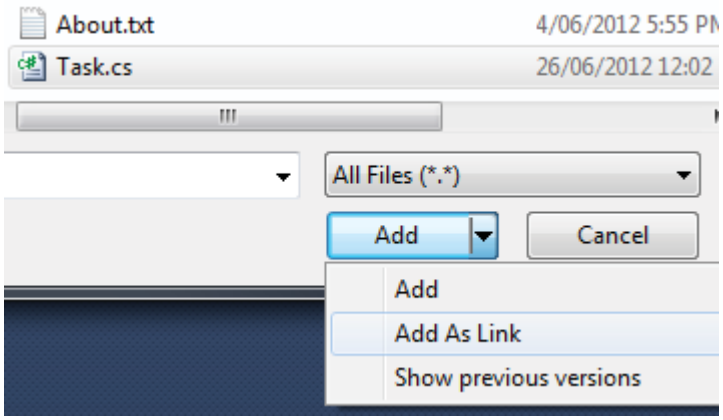

FIGURE 11 - ADD LINKED FILE IN VISUAL STUDIO

The process is much the same in Xamarin studio, but here you get an option after you click the Add button.

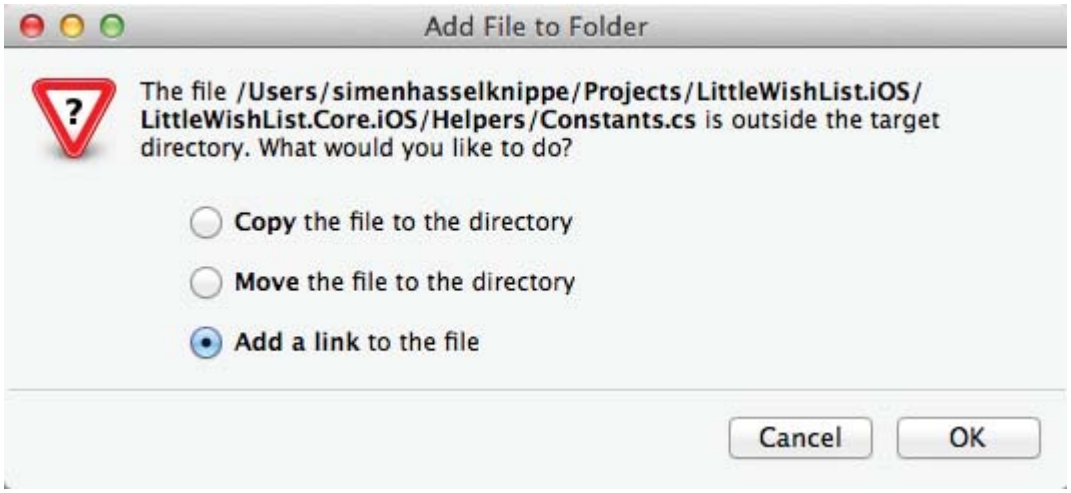

FIGURE 12 - ADD LINKED FILE IN XAMARIN STUDIO

As you can see, the linked file is now created in the Droid project and linked in both the iOS project and Windows Phone 8 project.

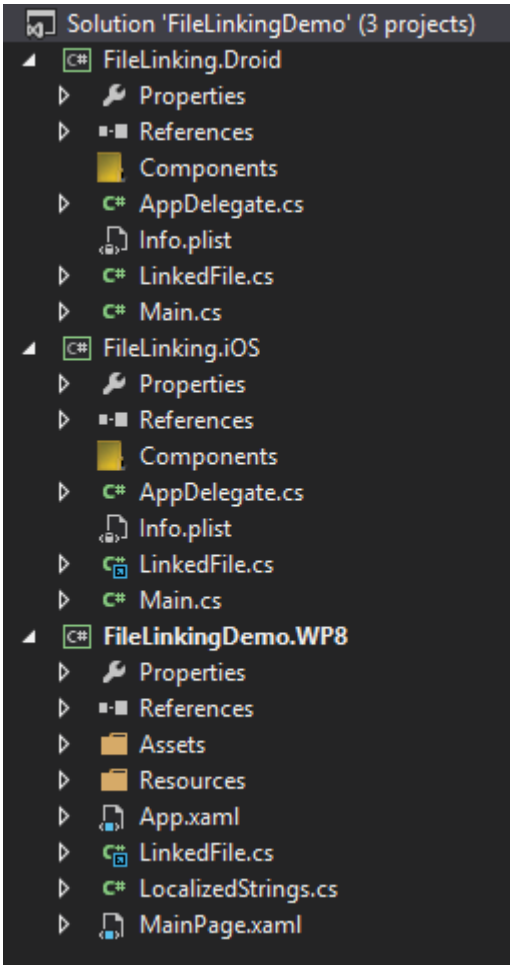

FIGURE 13 - LINKED FILE VISUAL STUDIO

This now means that if you open the LinkedFile.cs in any project and make a change to it all other projects will get the same change as well. Usually, you want to put these shared files inside a class library to better maintain them.

The biggest pro with a linked file solution is that platform-specific projects can include platform-specific references. If you combine this with compiler directives, a class can be shared and tailored for each application. Compiler directive is an option to compile a class depending on user-defined compiled options. Have a look at the following code. The conditional compilation symbol is set in the Android project to "__ANDROID__", and we can see by code highlighting that we have the desired behavior. You can set multiple compiler directives in each solution.

```
public string DeviceType
{
   get
   {
      #if __ANDROID__
         return "Android";
      #elif __iOS__
         return "iOS";
      #elif __WP8__
         return "WP8";
      #endif
   }
}
```

CODE 3 - COMPILER DIRECTIVE

As mentioned, compiler directives allow us to specify what code to evaluate and build on for each project. This makes file linking a good option, but there are downsides to file linking as well.

With the file linking approach there is the need for a lot of maintenance. Every time you want to create a new service, the file needs to be linked in all other core projects. This also applies to deletion. In a small project the number of files should be easy to track but as a project grows and more and more developers take part. It will be messy and someone would on occations forget to add a file.

2.3.2 Portable class libraries

Portable class library (PLC) is a class library that can be used across disparate CLI platforms, such as WPF, Windows Phone, Xbox, Xamarin.iOS, and Xamarin iOS. The library utilizes a subset of the complete .NET framework limited by the platforms being targeted.

FIGURE 14 - PORTABLE CLASS LIBRARY

This approach is easy to maintain, but it does come with some limitations. For example, you cannot use code that is not a part of the sharable .NET subset.

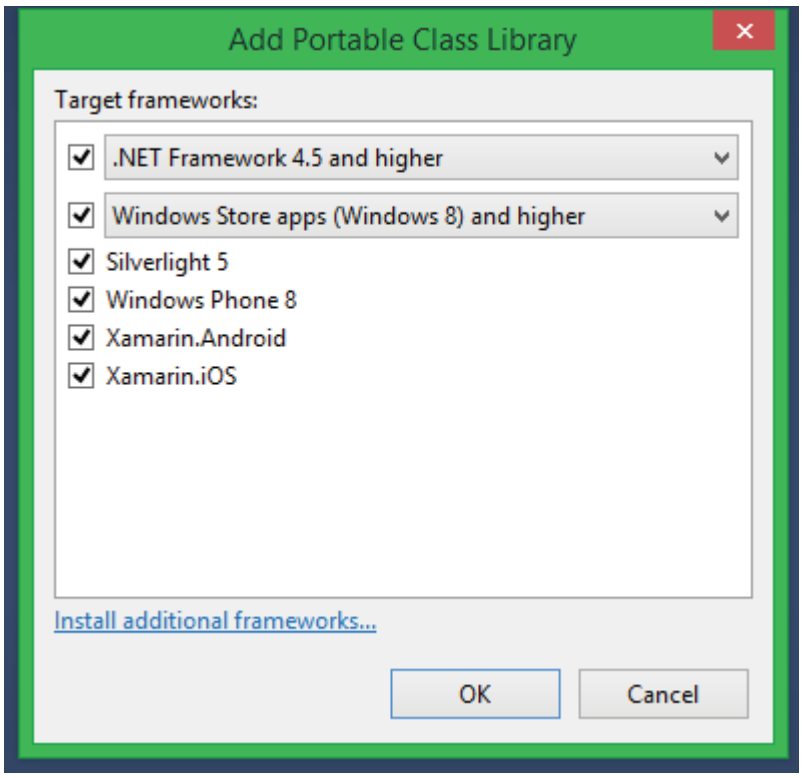

FIGURE 15 - DIALOG FOR CREATING A PORTABLE CLASS LIBRARY

2.4 Cross-platform architecture

In software development, many commonly used architectures allows developers to build both Web and native applications. Perhaps the most common pattern today is the MVC architecture.

When building cross-platform architecture, it is important to have the architecture that enables the developer to share as much code as possible. This chapter will examine two of the most common architectures. I will also discuss in which way they can be used in the terms of cross-platform development.

2.4.1 MVC

MVC was one of the first approaches to describe and implement software constructs in terms of their responsibilities. MVC was introduced by Trygve Reenskaug into Smalltalk-76 at Xerox PARC in 1979. Ten years later, Jim Althoff and others implemented a MVC version for the Smalltalk-80 class library. And in 1988, MVC was expressed as a general concept in an article in *The Journal of Object Technology.*

The MVC pattern has since subsequently evolved. It has also given rice to various child variants, such as HMVC, MVA, MVP, and MVVM. On March 13, 2009, Microsoft released Asp.Net MVC for the Web. Today, MVC is used in most modern Web and GUI frameworks, such as Ruby on Rails, Apple Cocoa, Spring and Apache Struts.

MVC stands for Model-View-Controller and consists of the three layers: model, view, and controller.

The first step in an MVC project is that the user makes a request to a controller. The controller then processes the request and creates and manipulates a model. The model is then passed to the view. And the view transforms the model into the appropriate output format. Response is then rendered for the user.

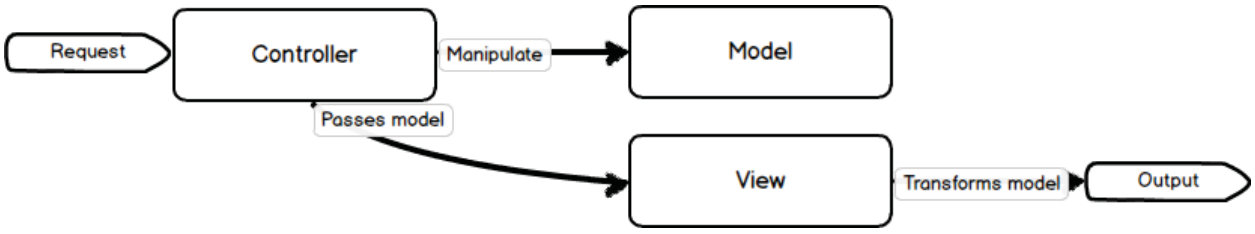

FIGURE 16 - MVC REQUEST

MVC is a compound pattern; the view and controller can have different strategy implementation. In this case, the controller knows about the view. The idea is that a view can switch controller and that the controller can pass data to the views.

MVC in a cross-platform project means the model is definitely sharable. We argued that views should have a native look and feel. Therefore, if we are creating a compiled app, the view is not sharable. However, if we were using a shell app, the view itself might be sharable, but we probably want to have separate CSS files for our different projects.

While in theory a controller could use multiple views, it might be hard to achieve in cross-platform development. An iOS controller has a more specific implementation than an Android view. From application lifecycle, we also know that android activities work a little differently than controllers in terms of handling lifecycle. Summarized, this makes the controller unsharable. However, the controller often uses some services, such as getting data online, storing data, opening the camera module, etc., and they are sharable. I will take a deeper look at this once we get started with experiments. In essence though, if you create a shell app, you can share the controller.

2.4.2 MVVM

In 2005, Microsoft unveiled the Model-View-View-Model pattern. At that time, it was tailor made for the WPF and Silverlight platforms. At the release of .NET Framework 3.0 on 21 November, 2006, MVVM was released stable. As the MVVM pattern kept growing, developers saw that it might be suited for other platforms as well. This gave life to the popular JavaScript library, Knockout.js. It has also been described as MVB (model-view-binder), which is the first reported implementation of the pattern outside the Microsoft stack, using ZK/Java.

In MVVM, a request is made on a view. The view then locates the view model usually through a service locator or singleton, but other possibilities are also present. When the instance of the view model is found, the view binds to it. For example, a view model can contain a property text. You would then bind a textbox to the value of the text. The binding can be two ways. Events are also bound by commands in the view model.

The view model may and may not manipulate a model. The properties in the view model often implement an interface to inform changes.

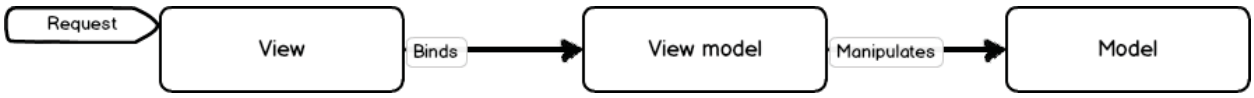

FIGURE 17 - MVVM REQUEST

Whereas MVC is open for interpretation, the MVVM pattern is a little more set. The idea here is that the view knows about the view model, but the view model does not know about the view. The view model also knows about the model, and the model does not know anything about the view model.

For a mobile cross-platform perspective, this gives some advantages when comparing with MVC. This means that the entire view model and model can be shared, which would allow us to share more code. As for the view, it is still the same; we can share it using a shell app but we cannot share it in a compiled app.

3 Experiments

To get a better understanding of and experience in cross-platform development, I have conducted several experiments. Experiments have been done by developing and/or researching different tools, frameworks and products. In the table below is an overview of all experiments done.

Experiment	Framework and tools	Architecture
LittleWishList	Xamarin, Monodialog, Azure mobile SDK	MVC
Quizter	MvvmCross, .Net api	MVVM
Movie Seeker	MvvmCross, json api	Game loop
Brickasa	MonoGame	MVVM
PropertyCross	Xamarin, MvvmCross, and Native	

TABLE 10 - LIST OF EXPERIMENTS

The experiments are essentially apps developed to test theories I have or to learn more about one of my research questions. They are built up by first providing some brief information about the project and then discussing some theories and/or research questions for this development experiment. I then provide an overview about how the application is implemented as well as a few important details. In the last chapter, I discuss the results of these experiments and, if appropriate, lead you toward some resources.

A detailed analysis of the combined results can be found in chapter 4, Analysis.

3.1 Experiment 1 – LittleWishList

LittleWishList is a compiled app build with Xamarin. It runs on iOS and Android. The goal of the app is to allow people to see when friends have birthdays and what their wishes are. It should also be possible to add your own wishes so that your friends know what you want for your birthday. This app was originally built before Xamarin had stable support for async/await; however, it was updated to a stable build when it was released.

3.1.1 Requirements

This app should have the following functional requirements:

- People should be able to log in as users.
- People should be able to register as users.
- Users logged in should be prompted with a home screen.
- The home screen should show all friends of a logged in user.
- The home screen should show all wishes of a logged in user.
- The user should be able to add a wish.
- The user should be able to search for friends.
- The user should be able to add a friend.
- The user should be able to see what wishes a friend has.

Perhaps more important than the functional requirements are the technical requirements:

- The app should feel, look and behave like a native app.
- The framework should have a backend API for saving wishes to the cloud.
- The app should be based on Xamarin.
- The app should be built with the MVC architecture.
- The app should use file-linking as the code-sharing strategy.
- The app should be developed in Xamarin Studio on a Mac.
- The app should run natively on iOS.
- The app should run natively on Android.

3.1.2 Research question

This experiment is tuned to answer several of my research questions as well as some theories I have. In this first experiment, it is most important to figure out if compiled apps are compatible with native apps and how building a cross-platform application influences the development process. I have the following theories about this application:

- A Xamarin built app has a native look and feel.
- The MVC architecture pattern is suited for cross-platform development.
- File-linking is suited as a code-sharing strategy.
- We can share most of our business logic between applications.

- Xamarin cross-platform development saves development time.
- Xamarin studio is an ideal ecosystem and IDE for developing cross-platform applications.
- Xamarin is suited for any application.
- Xamarin-based apps do not have any limits.

With these theories at hand, I will describe the building of LittleWishList.

3.1.3 Setup

Since this was the first time I set up a Xamarin-based application, I will go into some detail about how it was done. Since one of the technical requirements was to build this application using Xamarin Studio, the setup will be from that perspective. However, Xamarin Studio differs little from Visual Studio; it is the same process there.

Xamarin Suite can be downloaded from the Xamarin homepage[4]. One receives a complete installer that checks for dependencies and installs them as well. Once the installation is finished, one should have the following installed: Xamarin Studio, Android SDK, Xamarin.Android and Xamarin.iOS. When the installation is complete, one can run Xamarin Studio and select "new solution" to start a new project.

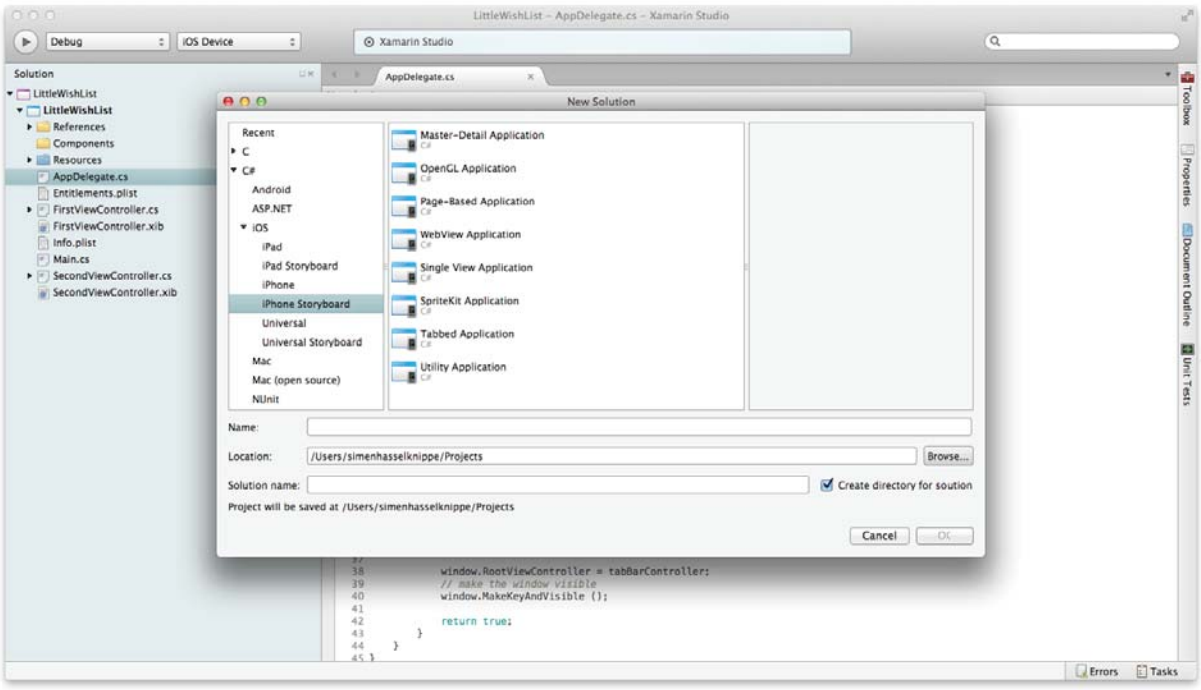

FIGURE 18 - CREATING A NEW PROJECT IN XAMARIN STUDIO

In Xamarin Studio, one can choose between different templates for creating new applications. IOS templates are used to compile an application to the iOS platform. In addition, it is possible to create

[4] http://xamarin.com/download

Mac applications and Android applications. Xamarin supports iOS Storyboard. The iOS Storyboard is an advanced UI editing tool where one can edit the entire UI and the relationship between different frames. This is a feature developed by Apple for native developers. Native iOS developers would benefit from the support Xamarin gives here. For this project, I created four projects.

- LittleWishList.Core.Droid is an Android library project containing the shared code, but all the files would be linked files from the iOS core project.
- LittleWishList.Core.iOS is an iOS library project containing all shared code.
- LittleWishList.Droid is the Android application.
- LittleWishList.iOS is the iOS application.

After the projects are created, the solution structure should look like the image below. The solution window is the window to find, create and view files.

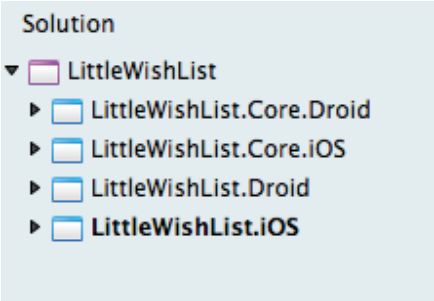

FIGURE 19 - LITTLEWISHLIST PROJECT SETUP

LittleWishList has been set up with four projects where two of the projects (the core projects) have linked files between them. The core projects are divided into three folders, and they are responsible for the following features. The Service folder contains the services connected with the cloud or to do other work one would normally expect a service to do, such as working on the file system. The Models folder holds the basic models or classes representing the objects I used. The Helpers folder holds other code that can be shared across the application.

After the folders and references are set, the project structure should look like the figure below.

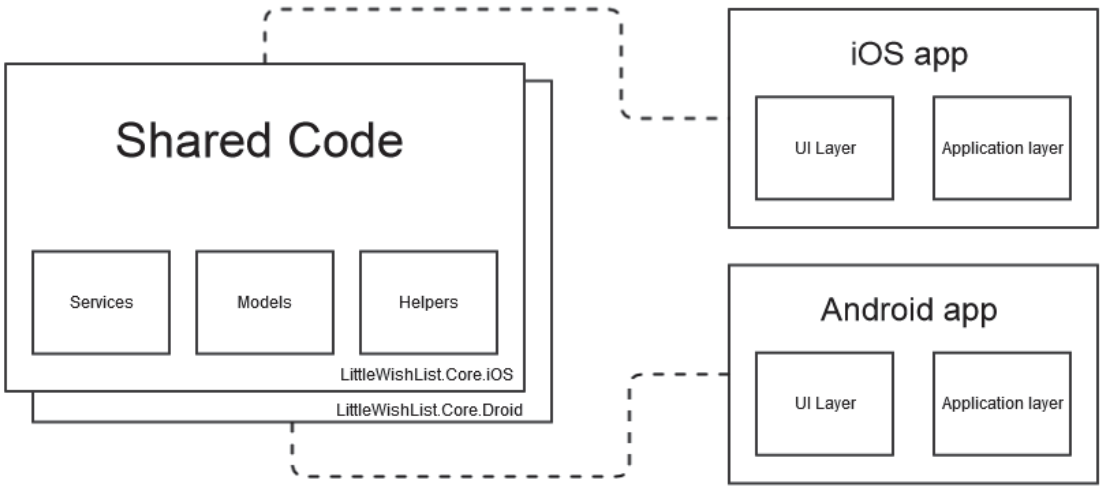

FIGURE 20 - LITTLEWISHLIST CONCEPTUAL ARCHITECTURE

The iOS application is responsible for building and compiling the iOS application. It has all the native features, such as views and their related controllers. It also contains the application layer code that is needed to start the application and manage the lifecycle.

The Android project contains the native Android features, such as views and activities. It also consists of the Android-specific code that allows one to run the Android application.

3.1.4 Development of shared code

As mentioned earlier, the class libraries will consist of shared and common code used in all of the apps. One interesting feature to notice is the component folder. Here, one can add different components to the app.

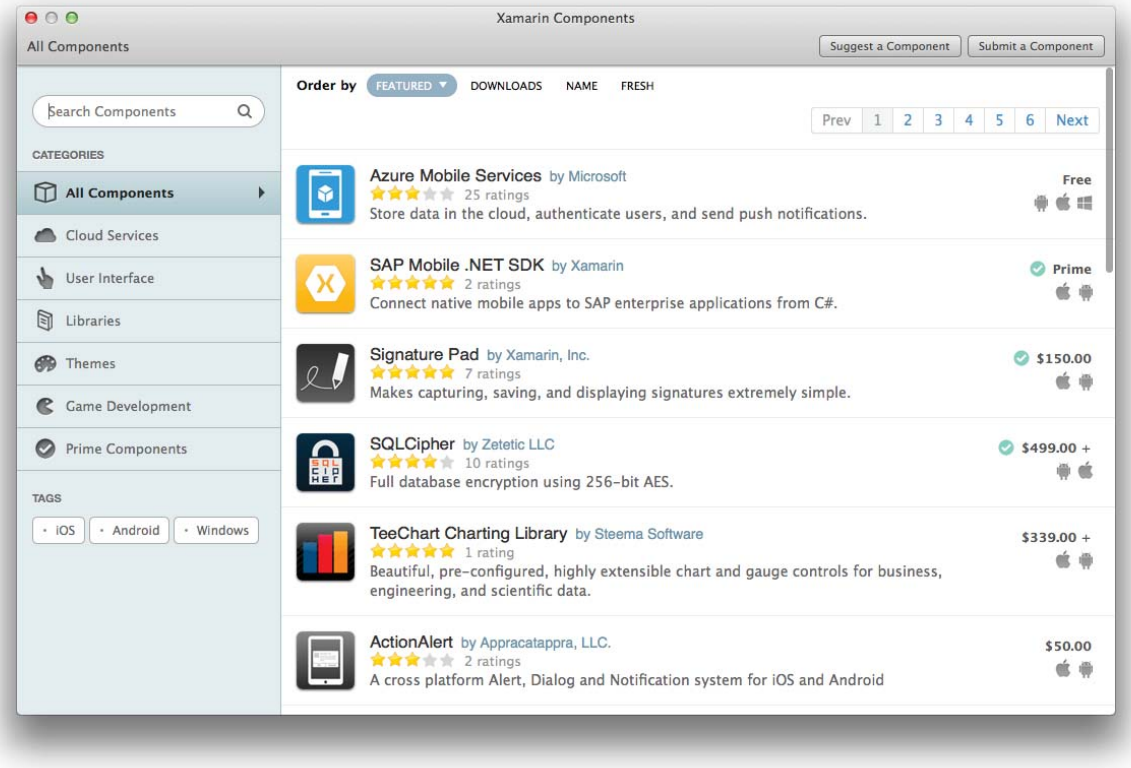

FIGURE 21 - XAMARIN COMPONENT STORE

Components are separated into different categories. I am interested in one component in this project, "Azure Mobile Services," which is for storing data in the cloud, authenticating users and sending push notifications.

Since some of the code might be new to some developers, I will go through it in detail. The models are simple classes with properties. If desired, one can also add constructors or nested properties.

```
public class Wish
{
	public Guid Id { get; set; }
	public string Name { get; set; }
	public string Price { get; set; }
	public string Username { get; set; }
}
```

CODE 4 - A SIMPLE MODEL

The Helpers folder consists of static constants, such as the URL to the Azure service and the colours used throughout the apps. In this way, one is able to create apps that use the same colour theme. In addition, changing one colour would affect the colours of all applications.

The Service folder consists of different services. There are two notable types of services.

AccountService, FriendService and WishService are all Azure Services, meaning they will go to the cloud and retrieve, add, update or delete data. A typical Azure method can look like the example below. Notice the async and await statements. Async/await is a feature in the .NET framework allowing programmers to easily write asynchronous code. In the sample below the service will run the request on a background thread, this makes the UI responsive. Since the await is applied, the method will continue once the request is finished. This is a relatively new feature in the .NET framework that Xamarin has ported to iOS and Android.

```
public class AccountService
{
	public async Task<User> Register(User user) {
		CurrentPlatform.Init ();
		try {
			await Constants.MobileService.GetTable<User> ().InsertAsync(user);
			return user;
		}
		catch (Exception e) {
			return null;
		}
	}
	/*
	  More methods
	 */
}
```

CODE 5 - AZURE MOBILE SERVICE EXAMPLE

The Credentials service is essentially a file-storage server where a user can store his login information in the file system, giving the application the possibility to remember the logged in user. The save method looks like this:

```
public static void SaveLoginCredentials(string email, string password) {
        var documents = Environment.GetFolderPath (
                                Environment.SpecialFolder.ApplicationData);
        var filename = Path.Combine(documents, "AccountInformation.txt");
        File.WriteAllText (filename, email + "\n" + password);
}
```

CODE 6 - FILE SYSTEM

3.1.5 Development for iOS

In Xamarin Studio, there are three different options for creating views. First, one can add a native view. A native view is built with the xCode designer. The xCode designer has a drag-and-drop interface that is easy to use. In LittleWishList, I used the native approach to create views for adding and displaying wishes.

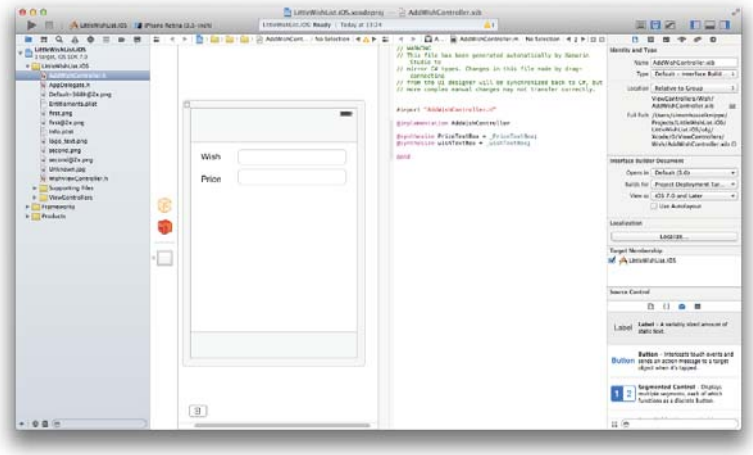

FIGURE 22 - XCODE UI DESIGNER

The second option is to create a view using code. In LittleWishList, this is done by creating tables. One can then add different widgets to the view by using code. The code for the search controller can be seen below. Here, one can see that I created a search bar, and I use it in the header of the table.

```
public class SearchViewController : UITableViewController
{
       public SearchViewController(Credentials credentials)
            : base(UITableViewStyle.Grouped)
       {
            UISearchBar searchBar = new UISearchBar(new RectangleF(
                           0, 0, TableView.Frame.Size.Width, 44));
            searchBar.SearchButtonClicked += async (object sender, EventArgs e) =>
            {
                // code for handeling a search
            };
            TableView.TableHeaderView = searchBar;
       }
   }
```

CODE 7 - ADD A SUBVIEW BY CODE

The third option is to use the mono dialogue framework. Mono dialogue is a simple way to create tables in which one requires the cells in the table to use different views. Mono dialogue is built by using sections and elements. Below is the login window in LittleWishList created with the dialogue view controller.

FIGURE 23 - LOGIN VIEW OF LITTLEWISHLIST USING MONO DIALOGE

Since LittleWishList will use a login view, I will demonstrate how one can manage this in iOS. If one looks at the startup flow chart, one can see the following sequence:

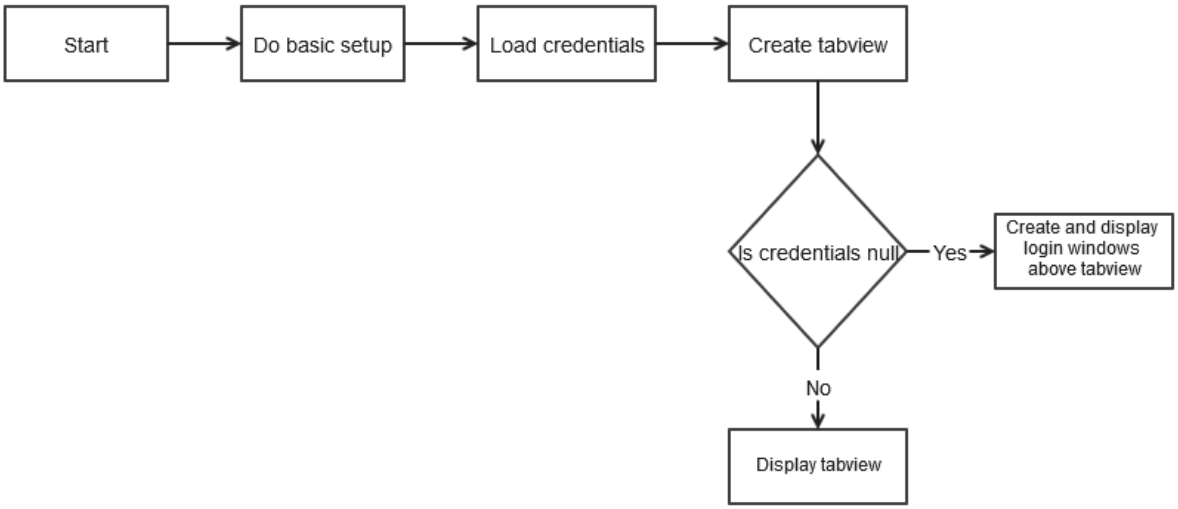

FIGURE 24 - LITTLEWISHLIST STARTUP SEQUENCE

The interesting thing to note here is that the tabview does a check on credentials; if no credentials are checked, it creates and displays a login window in code. This translates to:

```
public override bool FinishedLaunching(UIApplication app, NSDictionary options)
{
	// Initializes the app with some common properties
	window = new UIWindow(UIScreen.MainScreen.Bounds);

	// Tries to get the login credentials will be null if it does not exsist
	credentials = CredentialsService.GetLoginCredentials();

	// create the tab view witch consists of two ther views
	var wishesView = new UINavigationController(/* view */);
	var friendView = new UINavigationController(/* view */);
	tabBarController = new UITabBarController();
	tabBarController.ViewControllers = new UIViewController[] {
		wishesView,
		friendView,
	};

	// displays the tab view
	window.RootViewController = tabBarController;
	window.MakeKeyAndVisible();

	// displays the login windows over the tab view if not credentials are present
	if (credentials.Username == null)
	{
		var login = new LoginViewController(credentials);
		window.RootViewController.PresentViewController(login, false, null);
	}
	return true;
}
```

CODE 8 - THE STARTUP SEQUENCE

The present ViewController ensures that the control I am using gets displayed over the root windows. The second parameter is set to “false”. This means the login view is displayed immediately without any animation.

Xamarin.iOS does a full Ahead-of-Time (AOT) compilation to produce an ARM binary suitable for publishing into Apple's App Store. The AOT compilation is fairly straightforward in most cases, but when it comes to generics, they are not. In those situations, the compiler performs a static analysis of the code and determines all possible instantiations of a type; after this, it is possible to generate the code required.

3.1.6 Development for Android

When developing Android applications in Xamarin Studio, one has only two choices for creating views. One can either do it by code or by using .axml files. Developing user interfaces by code is very much like creating iOS user interfaces by code. Therefore, I will not go into details in that situation. The .axml file type is a special .xml file used in native Android development. A simple view may look like this:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <TextView
        android:text="Name"
        android:textAppearance="?android:attr/textAppearanceMedium"
        android:layout_width="match_parent"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:layout_height="wrap_content" />
    <EditText
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:id="@+id/wishNameEditText" />
    <Button
        android:text="Add"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginLeft="20dp"
        android:layout_marginRight="20dp"
        android:id="@+id/addWishButton" />
</LinearLayout>
```

CODE 9 - ANDROID .AXML CODE

Controllers in Android are called Activities. The following is an example of an activity developed in Xamarin Studio:

```
[Activity (Label = "AddWishActivity")]
public class AddWishActivity : Activity
{
	protected override void OnCreate (Bundle bundle)
	{
		base.OnCreate (bundle);
		var id = Intent.GetStringExtra ("id");

		SetContentView (Resource.Layout.AddWish);

		Button addButton = FindViewById<Button> (Resource.Id.addWishButton);
		addButton.Click += async (object sender, EventArgs e) => {
		};
	}
}
```

CODE 10 – ADDWISHACTIVITY

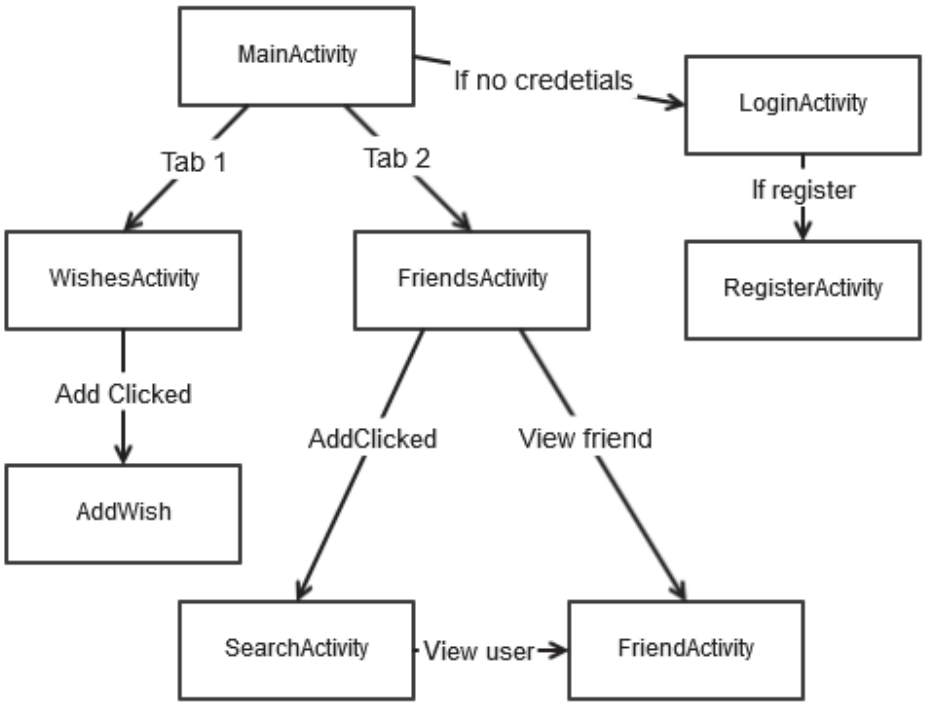

FIGURE 25 - ACTIVITY FLOW IN LITTLEWISHLIST

LittleWishList in Android consists of 8 activities. Above, one can see how they are connected. Below is a short list of their responsibilities and how they work.

- MainActivity is the starting activity. If it can find any credentials by using the credentials service, it acts as a tab activity and loads wishes and friends as two separate tabs. If no credentials are found, it will start the LoginActivity.
- The LoginActivity is responsible for creating and displaying a login view. If a user logs in, the results, i.e. the wishes of the logged in user, are cast back to the main activity so that it can display the views. The login activity also holds a link to the RegisterActivity.
- The RegisterActivity is where users can register. If a user successfully registers, responses are cast back to the login view and then back again to the MainActivity.
- The WishesActivity is a ListActivity that displays all the users' wishes in a list. It has a link to add new wishes. The list also contains a context menu that users can use to delete wishes.
- AddWishActivity is responsible for adding new wishes. If a wish is added successfully, the results are cast back to Wishes so that Wishes can update the list of active wishes.
- The FriendsAactivity lists all one's friends in a list view. One can also go to the SearchActivity to find new friends. Or click on a friend to display his wishes in the friendactivity.
- The SearchActivity consists of a search box and a list view. The list view contains users in the database and is populated as a user fills in data. If a user taps the view, it goes to the friend view.
- The friend view shows a user's wishes. Here, one can also friend or unfriend a user.

Previously, I discussed the AOT pattern; so how does the compiling process work on Android?

Xamarin is based on Mono (an open-source implementation of the CLR), but how does Xamarin actually manage to run Android applications? Well the compiler, which transforms C# .Net into Android, will compile and link all C# .NET code directly into an .APK (Android application package). An APK is also the required format to deploy applications onto Android devices.

Xamarin uses JIT (Just In Time compilation). JIT is a hybrid approach to compiling. It stands in the middle between "interpreted language," where an interpreter translates the language to the underlying machine every time one needs it, and the "compiled" approach, where the whole code is compiled into a machine in one run.

The "interpreted language" approach loads at light speed but comes with poor runtime performance, while the compiled one loads very slowly but offers better performance. The JIT compilation compiles the code when the user needs it, but it also saves the translated code in a

cache. The goal of this mechanism is to avoid performance degradation but also to load at a decent speed. JIT is considered by many as the both of both worlds.

3.1.7 Result

The final screenshots of LittleWishList can be found in chapter 10.1.1, "Screenshots – LittleWishList". The final result seems good and is easy to use. The navigation feels smooth, and it looks native.

The code can be found in chapter 10.2.1, "Code – LittleWishList." The code is simple to read and is organized in a decent manner. Now, I will examine the theories I had for this project versus how it turned out.

A Xamarin build has a native look and feel. By my judgement, this app does have a native look and feel. However, it is hard to decide what a native look and feel is, so I have to do some more tests to validate this theory.

The MVC architecture pattern is suited for cross-platform development. As a developer, I am satisfied with the MVC architecture, but it is not obvious to me that MVC is the best approach to cross-platform development. MVC is suited for cross-platform development, but I think that one can do better. Perhaps another architecture is better suited for cross-platform development.

File-linking is suited as a code-sharing strategy. While file-linking was hard to maintain, it is not impossible. File-linking is a powerful option. I especially liked how I could share the Azure Mobile Service code when two completely different products were used in the class library.

We can share most of our business logic between applications. I do not think this theory holds. While I did share the service code, there was still a lot of business code in the native libraries, such as updating a table with a new wish.

Xamarin cross-platform development saves development time. I think the development process here went smoothly, and I do really think that I saved a bit of time. However, I still had to learn Android APIs and the iOS APIs.

Xamarin Studio is an ideal ecosystem and IDE for developing cross-platform applications. Xamarin Studio is an okay product; I found some glitches that will be explored in the analysis chapter. I am hoping that Visual Studio might be even better.

Xamarin is suited for any application. While I have not yet tried every feature, I have not yet discovered any missing features. This theory still holds.

Xamarin-based apps do not have any limits. I have not found any traces of limits yet.

While developing for Xamarin with the MVC pattern has been great, I am doing the next experiment with MVVM and in Visual Studio to learn more about the different approaches one can take in Xamarin Suite and to continue testing my theories.

3.2 Experiment 2 – Quizter

Quizter is an app for content-based quizzes. Users can upload their own sets of quizzes and play against friends or random opponents. Quizter runs on Windows Phone, Android and iOS. I built it with the Xamarin approach and combined it with an MVVM architecture to see if I could share more code than in experiment 1. I will also describe what one can do with packages that do not support portable class libraries, which is the approach I used for code-sharing.

3.2.1 Required specifications

The functional requirements for the Quizter app include the following:

- A person should be able to log in as a user.
- A user should be able to browse through quizzes and questions.
- A user should be able to play a quiz.
- A user should be able to get live data from an opponent in a quiz.
- A scoreboard should be accessible from a quiz.

From the technical side, the requirements are somewhat similar to those of LittleWishList:

- The app should use the MVVM architecture.
- The app should be able to get live data through an API.
- The app should feel, look and behave like a native app.
- The framework should have a backend API for saving quizzes to the cloud.
- The app should be based on Xamarin.
- The app should be built in Visual Studio.

3.2.2 Research question

This experiment is tuned to improve the results of experiment 1. It is also intended to confirm some theories that are not yet decided. The new theories I have include:

- The MVVM architecture is better-suited for cross-platform development than MVC.
- Using PCLs is a better code-sharing strategy than file-linking.
- I can share more business logic between applications using MVVM than I could have if I had used MVC.
- Visual Studio is an ideal ecosystem and IDE to develop cross-platform applications.

There are also theories that are not yet decided:

- A MvvmCross build has a native look and feel.

- MvvmCross is suited for any application.
- MvvmCross-based apps do not have any limits.

3.2.3 Setup

As discussed earlier, I was not truly satisfied with Xamarin Studio on Mac as an ecosystem. Therefore, this time, I used Visual Studio to see if it gets any easier to develop. Setting up the project in Visual Studio does look a lot like Xamarin Studio; this time, I set up four projects.

- Quizter.Cores is the portable class library and holds all of the viewmodels, models and services.
- Quizter.Phone is the Windows Phone application.
- Quizter.iOS is the iOS application.
- Quizter.Droid is the Android application.

Since I am now using MVVM, I need an MVVM framework to work with. I chose to do this experiment with MvvmCross, a library developed by Stuart Lodge from the company Cirrious. MvvmCross is open-source, and therefore it is easy to create extensions and tailor the framework to one's needs.

To add MvvmCross to the application, we can use Nuget Package Manager. I opened it and searched for "mvvmcross". The package must then be added separately for all projects.

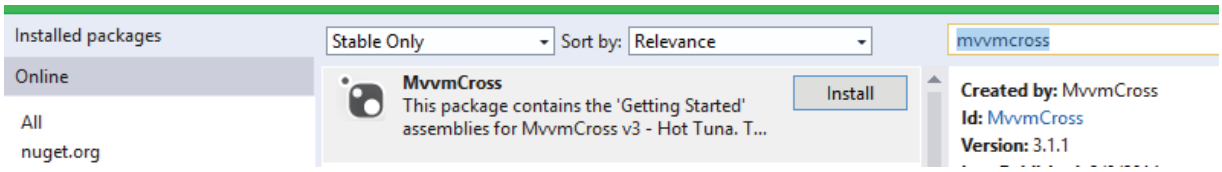

FIGURE 26 - NUGET PACKAGE MANAGER

In Quizter, I tried to achieve the following project structure. I wanted to have one shared class library holding all of my code. This would be the models, the viewmodels, and the services. Then, each application would be responsible for having views and an application layer that runs and manages the lifecycle of the application.

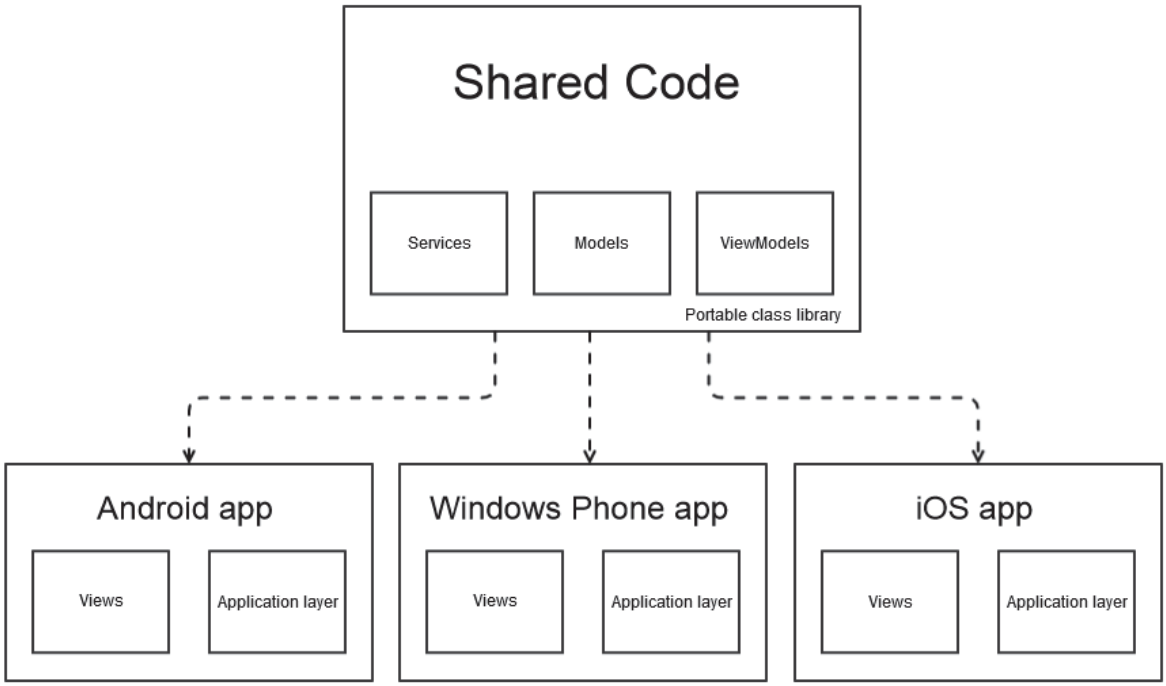

FIGURE 27 - QUIZTER ARCHITECTURE

3.2.4 Development of shared code

The shared code that resides in Quizter.Core contains more code than was present in experiment 1. It also contains the models. These are simple models similar to the ones used in experiment 1. It will also contain all services also similar to what I had in experiment 1.

It will also contain the viewmodels. A viewmodel is a model representing a view. The view binds to the viewmodel properties and can call various events through the command interface. The following code snippet is an example of a viewmodel in Quizter.

```
public class LoginViewModel : MvxViewModel
{
	private readonly IAccountService _accountService;

	public LoginViewModel(IAccountService accountService)
	{
		_accountService = accountService;
	}

	private string _email = "simen.hasselknippe@gmail.com";
	public string Email
	{
		get { return _email; }
		set { _email = value; RaisePropertyChanged(() => Email); }
	}

	// Other simple properties

	private MvxCommand _loginCommand;
	public ICommand LoginCommand
	{
		get
		{
			_loginCommand = _loginCommand ?? new MvxCommand(DoLoginCommand);
			return _loginCommand;
		}
	}

	private void DoLoginCommand()
	{
		// Do action such as calling a service and updating a model
	}
}
```

Code 11 - LoginViewModel in Quizter

The shared code will also contain the converters. A converter in MVVM is used for converting a simple property to a viewable property. For example, if one has a Boolean property called IsLoading, one can convert that into a visible property depending on the value. This makes it possible to view start a spinner in the view. In this way, the user gets valuable feedback if the application is doing something. It is also much easier to unit test the viewmodel. In fact, a well-written viewmodel can be unit tested in the same way a simple model is.

One thing I had to decide was how to use live data in my application. For this project, I selected SignalR as a framework to get live data. When this application was released, Xamarin was not given official support from SignalR, but there were various unofficial projects that it was possible to use. This project did not support portable class libraries, so I had to come up with a way to inject them into my code. I am now going to show how this was solved.

In the core project, I created an interface. The interface describes the various methods the service could have. Then, one can create the implementation of those interfaces inside the products. What MVVM does is to register all singletons automatically. However, it will not instantiate this service as an instance since it cannot find it; luckily, one can register the self-made platform-specific services inside setup.cs in each core library. One can instantiate a singleton by typing

```
Mvx.RegisterSingleton<IScreenSize>(new StoreScreenSize());
```

CODE 12 - REGISTER A SINGLETON

Another feature that comes from the MVVM pattern is messages. A message can be sent to zero or to many viewmodels. The viewmodels can then take appropriate actions, such as updating properties or reloading other data.

3.2.5 Development for Windows Phone

To develop views for Windows Phone, one has to use XAML. Since Windows Phone in general is based on C#, one can actually use truly native development. XAML is a special version of XML where only a special classes in .NET are allowed. The following is an example of the .xaml login view in Quizter:

```
<StackPanel
       Grid.Row="1"
       Margin="12,0,12,0">
       <TextBox
              x:Name="EmailBox"
              Text="{Binding Email, Mode=TwoWay}"
              Foreground="Gray"
              GotFocus="EmailGotFocus"
              LostFocus="EmailLostFocus" />
       <TextBox
              x:Name="PasswordBox"
              Text="{Binding Password, Mode=TwoWay}"
              Foreground="Gray"
              GotFocus="PasswordGotFocus"
              LostFocus="PasswordLostFocus" />
       <Button
              Content="Login"
              Command="{Binding LoginCommand}"/>
       <ProgressBar
              Visibility="{Binding IsWorking, Converter={StaticResource Visibility}}"
              Foreground="White"
              IsIndeterminate="True" />
       <TextBlock
              Text="{Binding Error}"
              Margin="10,0"/>
</StackPanel>
```

CODE 13 - XAML EXAMPLE

On negative thing about Windows Phone development is that sometimes one cannot bind the action through a view, but one instead has to go through the code behind the files. The best example of this is the appbar. An appbar displays a menu on the bottom of the screen. Developers can not bind buttons on the appbar to a command in the viewmodel without going through the code behind the file.

3.2.6 Development for iOS

Developing in iOS using Visual Studio is tricky. First of all, one does not have access to xCode, so one cannot use the interface design builder, which means one has to create the UI by coding. One could, however, open the project in Xamarin Studio on a Mac. It is also possible to run iOS apps in the iPhone simulator on a Mac. To make this possible, both computers have to be on the same network. One must start the program “Xamarin Build Host” on the Mac. It provides a step-by-step solution to build the application over Wi-Fi.

MVVM is supported in iOS by using binding sets. The following code shows an example of how a table is bound to a list in the viewmodel

```
public override void ViewDidLoad ()
{
        base.ViewDidLoad ();

        var source = new DecksTableSource(TableView);
        TableView.Source = source;

        var set = this.CreateBindingSet<DecksView, DecksViewModel>();
        set.Bind(source).For(s => s.ItemsSource).To(vm => vm.GroupedItems);
        set.Apply();

        TableView.ReloadData();
}
```

CODE 14 - IOS BINDINGS IN MVVMCROSS

3.2.7 Development for Android

Developing for Android, on the other hand, is easy in Visual Studio. One has full support for AXML editing. Xamarin has enabled the same UI editor inside Visual Studio as exists in Xamarin Studio. The image below is a screenshot from Visual Studio's Android interface builder.

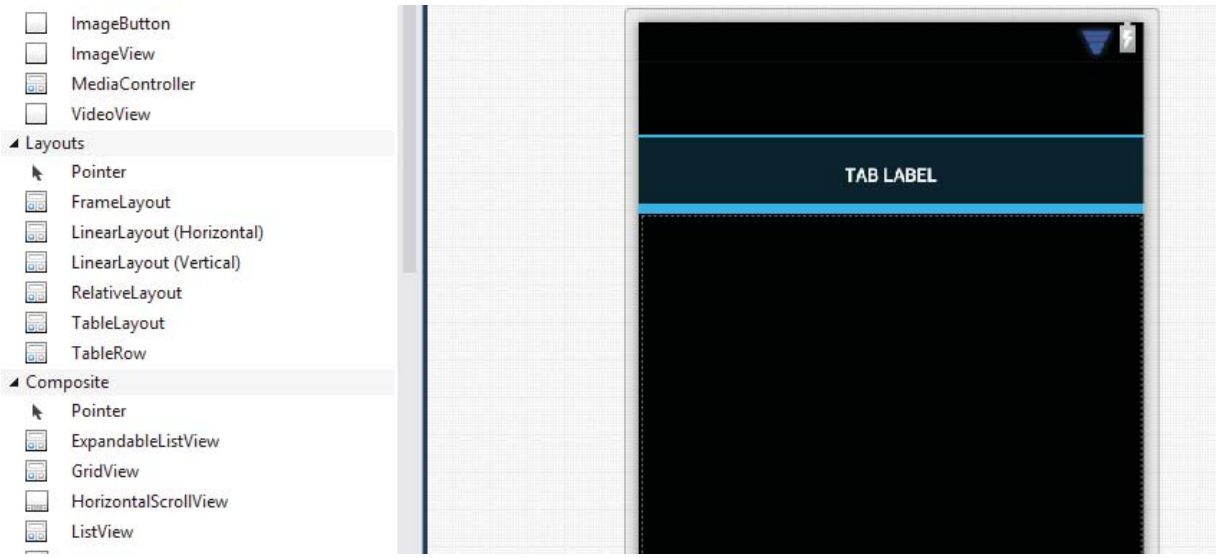

FIGURE 28 - VISUAL STUDIO'S ANDROID UI DESIGNER

To use bindings in an AXML view, one can use the MvxBind command as shown below:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:local="http://schemas.android.com/apk/res-auto"
    android:orientation="horizontal"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <LinearLayout
        android:orientation="vertical"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent">
        <TextView
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:textSize="40dp"
            local:MvxBind="Text Caption" />
        <TextView
            android:layout_width="fill_parent"
            android:layout_height="wrap_content"
            android:textSize="20dp"
            local:MvxBind="Text Description" />
    </LinearLayout>
</LinearLayout>
```

CODE 15 - ANDROID BINDINGS IN MVVMCROSS

3.2.8 Result

Unfortunately, this project got a little out of control. I quickly realized that the project was too big for me to finish in time. I do, however, have a working prototype on all platforms, but they are not nearly complete and require much work to reach completion.

I have long thought about why it became this way. The reason, I think, was that I took on too much work at once. I started trying to build the apps and the API simultaneously. I did not have the prior knowledge of SignalR, and that confused me. I think most of the blame that this experiment did not turn out too well is mine to bear. Now, I will review the theories I had for this development process.

The MVVM architecture is better suited for cross-platform development than MVC. While I do believe that this is still true, it is hard to determine based on just two experiment, where one experiment did not work out the way I hoped. I will need another look at this.

Using PCLs is a better code-sharing strategy than file-linking. I don't think this theory holds. While PCLs are good, choosing between a PCL and file-linking depends on the implementation. If one is using many special products that are not supported by PCLs, one is probably better off with file-linking, and if one knows that most of the services will be sharable, it is easier to manage a PCL rather than file-linking.

One can share more business logic between applications using MVVM than one could have by using MVC. As of experiments 1 and 2, this is the conclusion.

Visual Studio is an ideal ecosystem and IDE to develop cross-platform applications. Visual Studio is a better ecosystem when developing Android, Windows Phone or shared-class libraries. The lack of support for editing views on iOS makes Xamarin Studio a better fit for iOS development.

The other theories I had were:

- An MvvmCross build has a native look and feel.
- MvvmCross is suited for any application.
- MvvmCross-based apps do not have any limits.

These still hold, but it is hard to prove that since this project did not go very well. The final screenshots from the application can be found in chapter 10.1.2, "Screenshots – Quizter" and the code can be found in chapter 10.2.2, "Code – Quizter". In the next experiment, I again try an MVVM solution to examine if I can complete a successful MVVM project.

3.3 Experiment 3 – MovieSeeker

MovieSeeker is an app for searching for movies and reading their descriptions. MovieSeeker uses the Rotten Tomatoes API to search for and display information about movies. MovieSeeker is targeted to run on Windows Phone, Android and iOS. Since the Quizter experiment failed, I will again try to use MvvmCross.

3.3.1 Requirement specification

The functional requirements for the MovieSeeker app include the following:

- A person should be able to search for movies.
- The search should be live, and users should get results as they type.
- It should be possible to click on movies to read their synopses.

Since I am retrying MvvmCross, the technical requirements are almost identical to those of Quizter.

- The app should use the MVVM architecture
- The app should be able to get data through the Rotten Tomatoes API.
- The app should feel, look and behave like a native app.
- The app should be based on Xamarin.
- The app should be built in Visual Studio.

3.3.2 Research question

The theories I have for this application are:

- The MVVM architecture is better suited for cross-platform development than MVC.

- Visual Studio is an ideal ecosystem and IDE to develop cross-platform applications.
- A MvvmCross build has a native look and feel.
- MvvmCross is suited for any application.
- MvvmCross-based apps do not have any limits.

3.3.3 Setup

The setup for this project differs little from that of Quizter. See Figure 27 - Quizter for the architecture of this application.

3.3.4 Development

Most of the development can be executed the same way as that of Quizter. This project does not have any platform-specific services to inject into the viewmodels. It is trivial code that any object-oriented developer should know.

One interesting thing is the search algorithm, so I will provide a quick look on how that is implemented.

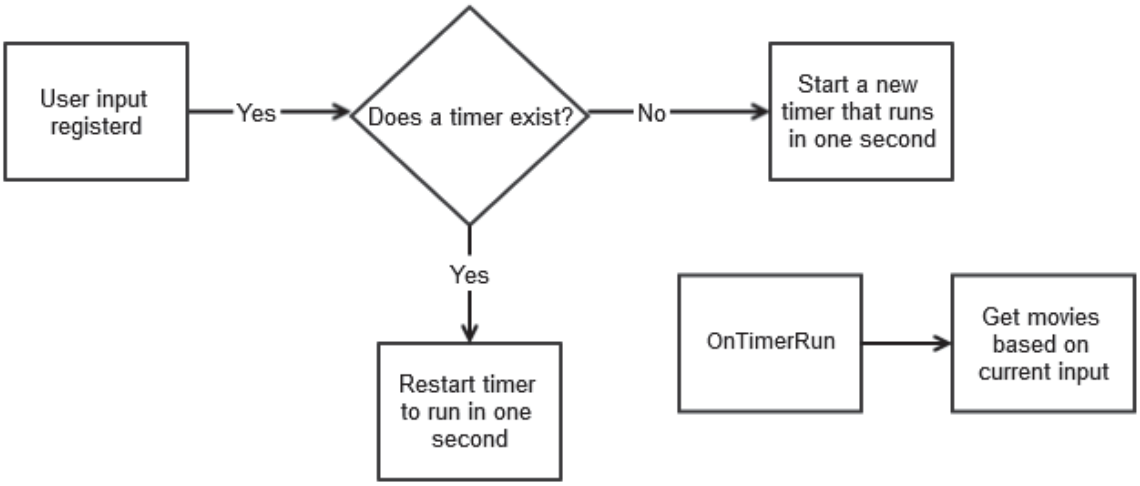

FIGURE 29 - SEARCH ALGORITHM

When the user writes input into the search box, a user input is registered. The program then checks to see if a timer exists; if no timer exists, then it creates a new timer that should tick in one second. Before that one second is finished, another user input is registered, and then the timer is reset so that the tick will happen one second after the user writes the final info. When the timer ticks, it calls the service and reloads all movies based on the current input. Using an algorithm like this is effective, and the application does feel responsive and fast.

3.3.5 How MvvmCross works

MvvmCross is open-source, and one can download the entire project instead of using the Nuget Package Manager. This allows one to see how it works and go through some of the implementation of it.

If one decides to download the source, one could use the debugger to examine how it is built and how it works. When a native app starts, the constructor of the native application will be created. The setup initializes the Inversion of Control (IOC) system. The IoC system registers and resolves all interfaces, such as services.

After the IoC system is set, setup will call the core project and construct an app. This file is found in the PCL named app.cs. The app will register all services with the IoC system as singletons and register which viewmodel the application should start with.

Setup will then configure the UI and bind the UI to the viewmodel. After this is done, the application is started.

3.3.6 Result

Screenshots from the final results can be found in chapter 10.1.3, "Screenshots - Movie Seeker", and the code can be found in chapter 10.2.3, "Code – Movie Seeker". Now, I will look back at my theories.

An MVVM architecture is better suited for cross-platform development than MVC. MVVM allows one to share more of the code; it is a big contribution that the navigation can be shared.

Visual Studio is an ideal ecosystem and IDE through which to develop cross-platform applications. Developing in Visual Studio provides a robust and functional IDE. It still remains unfit for iOS since it is hard to produce nice native views. As one can see in the screenshots, the application does look better in Android and Windows Phone than in iOS.

MvvmCross-built apps have a native look and feel. I still think this is the case, but I cannot say for sure until I have compared a native application with an MvvmCross-based application.

MvvmCross is suited for any application. I have barely scratched the surface of features one can accomplish with MvvmCross, but the theory still holds.

MvvmCross-based apps do not have any limits. As far as I can tell, this is the case.

3.4 Experiment 4 – Brickasa

Brickasa is a simple game app developed with MonoGame. It is a puzzle app wherein a user can drag and rotate triangles, squares, parallelograms and other shapes to try to match the defined parent. Below is a picture of the first tutorial in the game. To complete this project on time, only a tutorial was developed and not the game itself.

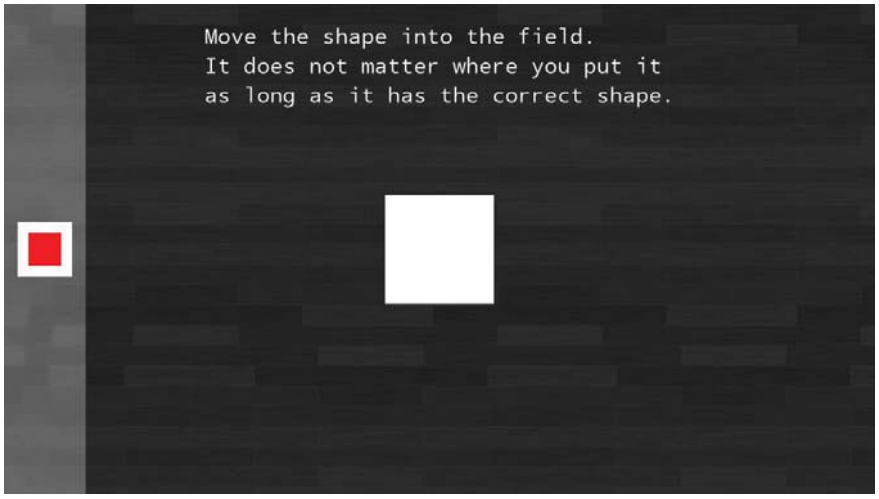

FIGURE 30 - BRICKASA RUNNING ON WINDOWS 8

3.4.1 Required specifications

For Brickasa, I only wanted to develop a prototype running a tutorial. The game was however, built in a way that easily allows me to add levels later. The app runs on Windows 8, iOS, Android and Windows Phone. The following functional requirements apply to Brickasa.

- A tutorial with 3 different levels should be playable.
- Brickasa should support rotation.
- Brickasa should support squares and rectangles.

The technical requirements for this experiment include:

- The app should feel, look and behave like a native app.
- The app should be based on MonoGame.
- As much code as possible should be shared.

3.4.2 Research question

This experiment was done to research the difference between creating a game and a normal app. I would also like to see how much code I can share in a game. I have the following theories for this experiment.

The main theories I want to test include:

- Games are more suited for cross-platform development.
- Games use a lot of different algorithms, and therefore, a lot of the code can be shared.
- Games can share the UI across-platforms.
- A MonoGame-built application has a native look and feel.

3.4.3 Setup

MonoGame has an installer available on their homepage[5].When the installer finishes running developers, have the possibility to create MonoGame projects within Visual Studio. Alternatively, the MonoGame source code can be downloaded from GitHub. MonoGame is open-source, so developers can examine and edit the code themselves.

The MonoGame installer installs several templates into Visual Studio. A screenshot of the available template can be seen below.

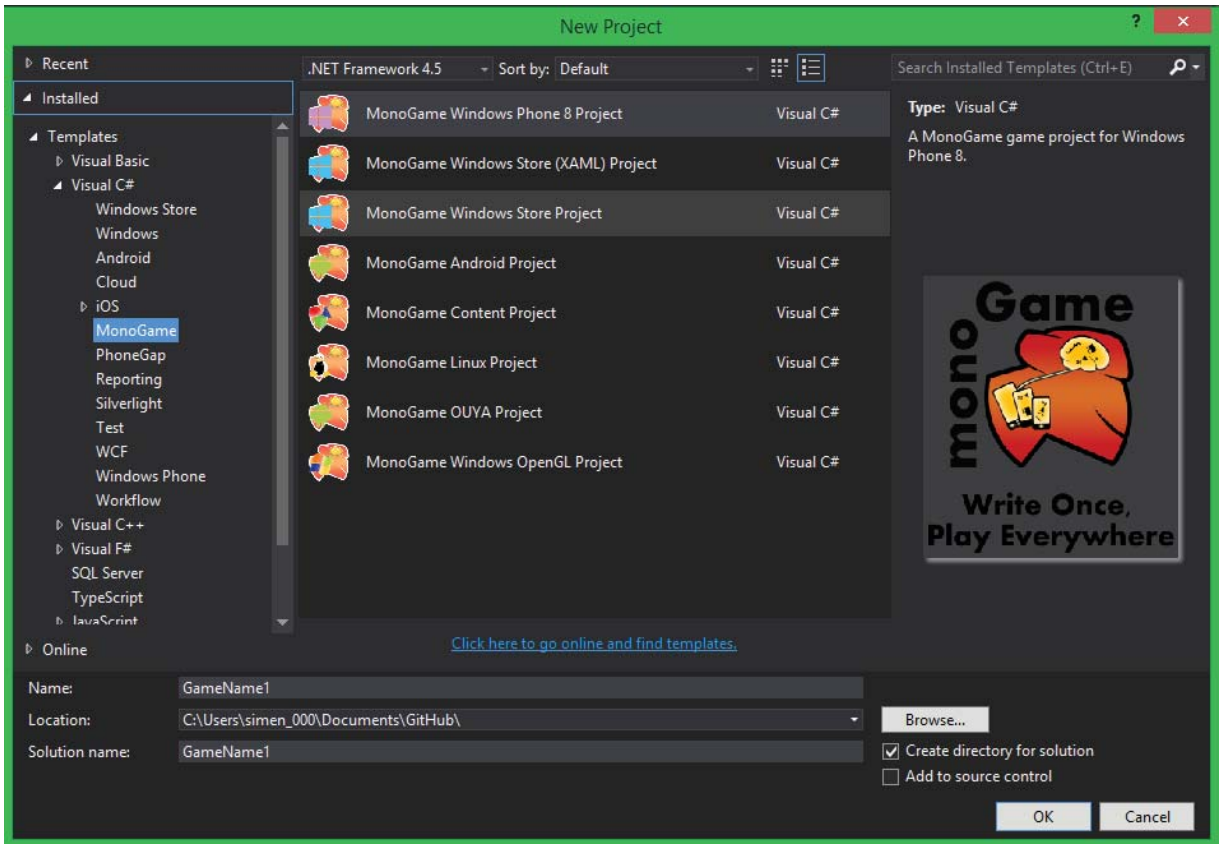

FIGURE 31 - TEMPLATES FOR MONOGAME

Here is a look at the boilerplate code this template provides: First of all, it creates a file called Game1.cs. This is the file that contains the game loop. The game loop consists of a constructer, an initialisation method, a load-content method, an update method and a draw method.

[5] http://www.monogame.net/downloads/

```
protected override void LoadContent()
{
	_spriteBatch = new SpriteBatch(GraphicsDevice);
	_menuScene = new MenuScene(this, _spriteBatch);
	_activeScene = _menuScene;
}
```

Code 16 - LoadContent in a game

The load-content method is responsible for loading all sprites the game uses. These sprites could be figures, fonts or sounds. In the update method, the developer must apply the game logic. This logic parses user input and calculates movement on a screen. Calculations are common to use in popular games. Many of the calculations also use a bit of random numbers to vary the results. Angry Birds is an example of one of those games, and it is the reason why one could never do the same thing twice.

The drawing method will then draw figures in the right positions. In Brickasa, I often keep my objects in a list. I can then iterate through the list and draw all of the figures as demonstrated below:

```
public override void Draw(GameTime gameTime)
{
	_spriteBatch.Draw(_background, _backgroundRectangle, Color.White);
	foreach (var button in _buttons)
	{
		button.Draw(gameTime, _spriteBatch);
	}
	base.Draw(gameTime);
}
```

Code 17 - Draw method in a game

A common mistake done by developers is to have calculations in the draw method. This makes the draw method slower, and the result is a lagging game.

MonoGame, like XNA, depends on game content such as images, sounds and fonts to be processed at build time into a usable runtime use. This process requires some hand-editing of the various game project as well as building a content project.

MonoGame does not yet provide a content pipeline on Mac or Linux. MonoGame depends on XNA content pipeline implementation and a version of Visual Studio to build it. For more information on the content pipeline, one can have a look at the MonoGame website[6].

[6] https://github.com/mono/MonoGame/wiki/MonoGame-Content-Processing

3.4.4 Game mechanics

Brickasa uses a set of different algorithms to make this game playable. Below is an overview of the update method.

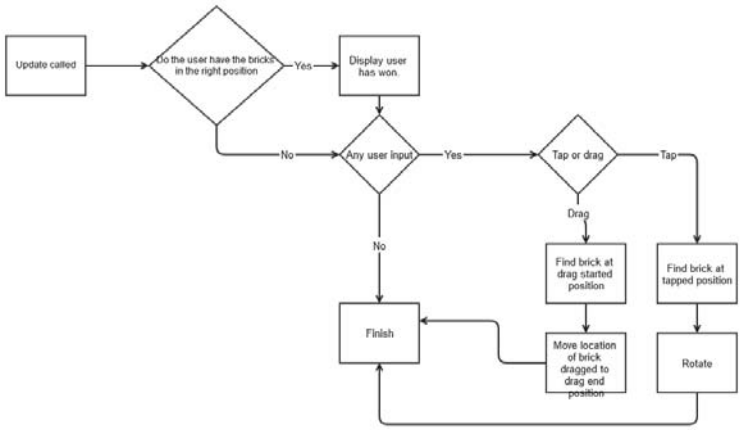

FIGURE 32 - BRICKASA UPDATE LOOP

It starts when the update method is called. First, the application checks that the user has won by putting all shapes in the right position. This is done by checking that each shape matches the result defined in an XML file. Then it checks if there is any input. If there is a touch input, one gets the coordinates and calculates if any shapes cover the tapped location, and then that shape is rotated.

If the user has dragged anything, the shape is moved to the end position. The update loop will run many times a second, and therefore it will constantly keep the game alive.

3.4.5 Development for Windows 8

Brickasa consist of different scenes. A menu scene for selection, the tutorial, and a game screen, which is the game itself. Game1.cs is responsible for keeping a link of these scenes and displaying the active one.

```
protected override void LoadContent()
{
       _spriteBatch = new SpriteBatch(GraphicsDevice);
       _menuScene = new MenuScene(this, _spriteBatch);
       _activeScene = _menuScene;
}
```

CODE 18 - LOADING THE MENU SCENE IN BRICKASA

Load-content is a method I described earlier, wherein the developer sets up the game to load different content depending on what content is available. Here, one can see that the menu scene is initialised and placed as the active scene. The result is displayed below.

FIGURE 33 - THE MENU SCENE IN BRICKASA

The menu screen is a place where one can choose which content one wants to use. The yellow marked "tutorial" is highlighted because the user is hovering on it. This only works on a computer with a mouse, while smartphones and tablets will provide the same effect if a button is tapped or hooded.

The template for Windows 8 contains the code for loading the game; this should be trivial to read and customize.

3.4.6 Development for Android

To port Brickasa to Android, a new project has to be created. Again, one can use the template in Visual Studio. The boilerplate code is very similar to what was created for Windows 8. Naturally, it is an activity that starts the game.

In order to copy all the created files that I wanted to use to the Android project, I also replaced the game1.cs from Android with the same that I have in the Windows 8 project. I could use linked items, or I could simply copy most of them. The content file must be copied to Assets/Content.

On Android, it is important to change all content files build actions. To do this, select the properties window for each file and change the build action to Android asset as shown in the image below:

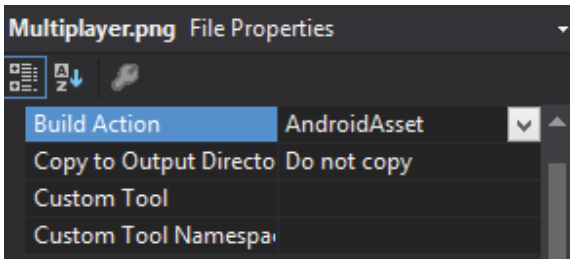

FIGURE 34 - ANDROID BUILD ACTION

Android does not have support for loading resource files without using a stream reader. Therefore, I have to change how the XML files are loaded. See the code snippet below.

```
public GameManager(Game game)
{
	_game = game;
	_plays = new Queue<XDocument>();

	// Not valid in android
	//_plays.Enqueue(XDocument.Load("Results\\Tutorial1.xml"));

	// Valid in android
	using (var sr = new
		StreamReader(Game.Activity.Assets.Open("Results/Tutorial1.xml")))
	{
		_plays.Enqueue(XDocument.Load(sr));
	}
}
```

CODE 19 - LOADING RESOURCES ON ANDROID

Apart from this, all of the code is valid and the product can be built successfully on any Android device.

3.4.7 Development for Windows Phone

Next, the project is going to be ported to Windows Phone. While MonoGame is excellent on iOS and Windows 8, it is somewhat buggy on Windows Phone. Windows Phone requires content processing on all content files pre-build. This means I have to create a new MonoGame content project and copy all of the images before building the project and finding the processed files inside the bin folder. Then, I copy those over to the project along with the classes.

If one runs the application at this point, one will notice some annoying issues. Windows Phone does currently not have support for landscape mode, and it does not recognize any user input. While user input can easily be fixed by applying these two lines to the code:

```
var fieldInfo = typeof(TouchPanel).GetField("_touchScale",
                    System.Reflection.BindingFlags.NonPublic |
                    System.Reflection.BindingFlags.Static);
fieldInfo.SetValue(null, Vector2.One);
```

CODE 20 - FIXING USER INPUT IN MONOGAME

Landscape support is not fixable in this version. The good news is that it is possible to fix it on the development branch of MonoGame. One can download this branch in git by typing

```
git clone https://github.com/mono/MonoGame
```

CODE 21 - GIT CLONE MONOGAME

Inside the development, one can build a fresh branch by running Protobuild.exe. This will generate new DLLs that one can use in the source code. Using these new references fixes the problem.

3.4.8 Development for iOS

The last part of this experiment is to port the project to iOS. Like earlier, I created a new project and copied the files. On iOS, the app delegate is responsible for creating the game. The rest should be very familiar by now.

3.4.9 Result

The final result actually looks great on all platforms, and it does have a nice shared codebase. There is a wrapper around each class. That is mostly created for the user, and it is also easy to maintain. The shared code keeps growing as one adds more screens to the app. Highscores, Multiplayer and Campaign all produce code that should be close to 100% sharable.

The game UI is consistent across devices and should, in many cases, be easy to maintain. This development uses the MonoGame version 3.2 released on 4/7/2014. However, when I started the project, version 3.0 was the current stable release it had been out since 1/21/2013. The old version had far more problems than the current version. Among other things, there was no support for Windows Phone or Playstation. The unofficial support was working okay. There were more problems creating project since references were missing.

For the final live screenshots on all devices, see chapter 10.1.4. Screenshots – Brickasa. To browse the code, see chapter 10.2.4.

3.5 Experiment 5 – PropertyCross

In this final experiment, I evaluate how Xamarin-based apps compare to native ones. To evaluate this, I will use an already existing application called PropertyCross. PropertyCross presents a non-trivial application for searching UK property listings. It is developed on a lot of different platforms, included Native, Xamarin and MvvmCross.

3.5.1 Research question

This is the experiment where I would get really valuable experience in how developing in Xamarin differs from native development. The theories I have for this experiment are:

- Xamarin, MvvmCross and native apps have the same look and feel.
- MvvmCross-based applications are a little bigger than Xamarin-based applications.
- Xamarin-based applications are a little bigger than native applications.

- Native applications should have a faster loading time than Xamarin and MvvmCross-based applications.
- It is possible to share more code using MvvmCross than in Xamarin.

3.5.2 Setup

To do this experiment, I have modified the source code a little. I have added the following code to different parts of the applications that need timing.

```
private static readonly Stopwatch Stopwatch = new Stopwatch();

public static void StartWatch()
{
	Stopwatch.Start();
}

public static void StopWatch()
{
	Stopwatch.Stop();
	Debug.WriteLine(Stopwatch.ElapsedMilliseconds);
	Stopwatch.Reset();
}
```

CODE 22 - MEASURE TIME IN PROPERTY CROSS

In Windows Phone, these methods can be called by writing:

```
protected override void OnNavigatedTo(NavigationEventArgs e)
{
	base.OnNavigatedTo(e);
	App.StopWatch();
}

protected override void OnNavigatingFrom(NavigatingCancelEventArgs e)
{
	base.OnNavigatingFrom(e);
	App.StartWatch();
}
```

CODE 23 - MEASURE TIME ON PAGE NAVIGATION, WINDOWS PHONE

This will allow me to time the startup time for a Windows Phone application as well as the time between frame navigation.

For Android, I have timed the seconds it takes to open a view.

```
protected override void OnCreate(Bundle bundle)
{
	StartWatch();
	base.OnCreate(bundle);
}

protected override void OnResume()
{
	base.OnResume();
	StopWatch();
}
```

CODE 24 – MEASURE RIME ON PAGE NAVIGATION, ANDROID

In addition, I have used a program called CLOC (Counting Lines of Code) to count how much code exists in various projects.

3.5.3 Results

The results of the application can be found in chapter 10.1.5, "Screenshots – Property Cross".

First of all, I ran the program on different devices. On Android, I ran it on an Asus FonePad, on Windows Phone, I ran the programs on a Nokia Lumia 920, and on iOS, I ran the application in the simulator. On the first run, I tested the native look and feel. To me, they almost feel the same, and I could not tell the different programs apart.

I measured the time on starting the Windows Phone application. In the table below, the results are listed.

	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)
Native	710	720	676	695	732
Xamarin	718	705	706	691	740
MvvmCross	1809	1829	1872	1845	1841

TABLE 11 - STARTUP TIME WINDOWS PHONE

We can see that native and Xamarin are opening at almost the same speed; this is because a Xamarin Windows Phone application actually is a native application that sometimes uses the Xamarin APIs. MvvmCross is a little bit slower. This, I expect, is because creating the viewmodels and injecting services does take some time. In the next run, I measured the speed of changing a view; the results can be seen in the table below:

	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)
Native	2	1	0	2	5
Xamarin	3	1	1	2	2
MvvmCross	245	71	134	42	55

TABLE 12 - NAVIGATION TIME WINDOWS PHONE

In addition, I measured the speed of going back; on the native solution and the Xamarin solution, this was 0 milliseconds, and on MvvmCross, it was 80 milliseconds. I also measured the application sizes, which are listed below:

- Native 408 kb
- Xamarin: 417 kb
- MvvmCross: 594 kb

Since Android is built by using Activities I did not find a good way of measuring time between opening frames. I did however measure the time it took to open an application. The table below is a list of how long it took to open the Android application on various devices.

	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)
Native	250	250	250	250	250
Xamarin	280	268	284	295	281
MvvmCross	894	968	915	901	955

TABLE 13 - STARTUP TIME ANDROID

We can see that MvvmCross is somewhat slower than native and Xamarin. It is, however, still less than a second and very acceptable. I would still call this native speed.

I also measured the size of the Android package:

- Native: 628 kb
- Xamarin 6.54 MB
- MvvmCross 10.2 MB

One can clearly see that Xamarin-based products on Android are significantly larger than native-based projects. This is due to the overhead and links between the C# layer and the native Android binaries.

Then, I counted the code using CLOC. In the native solution, the metrics are defined below:

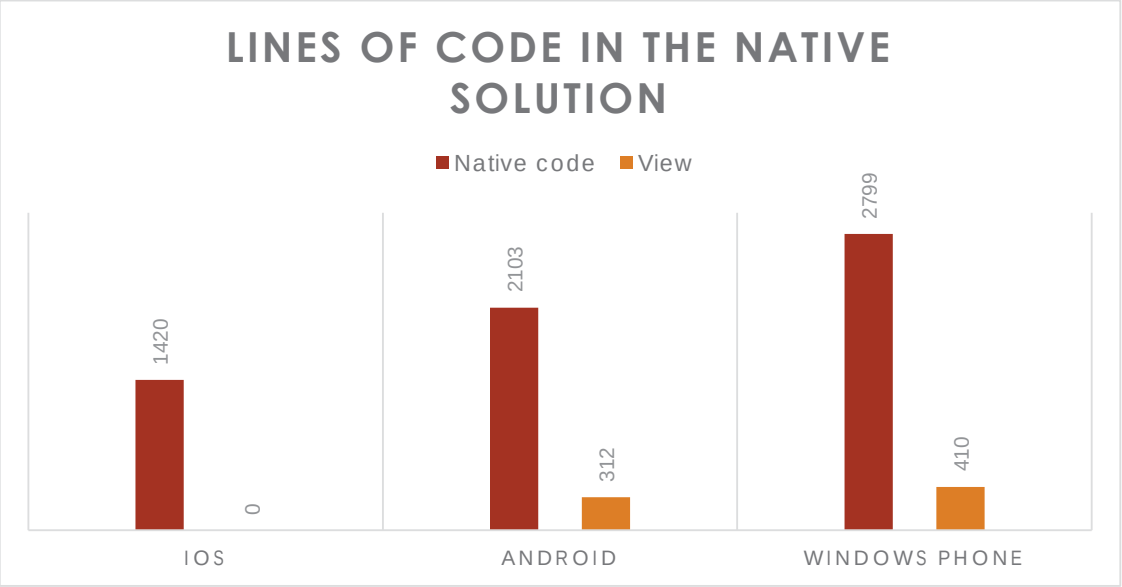

FIGURE 35 - LINES OF CODE, NATIVE SOLUTION

Countin the code in iOS is impossible due to iOS views generating binaries for the views. We also see in that this much code in Android and WP, while the view code is relatively small. This due to the native applications uses coded view. Much of a views behavior is defined in the code behind file, many developers would possibly say that this is not optimal.

In MvvmCross, the metrics were the following:

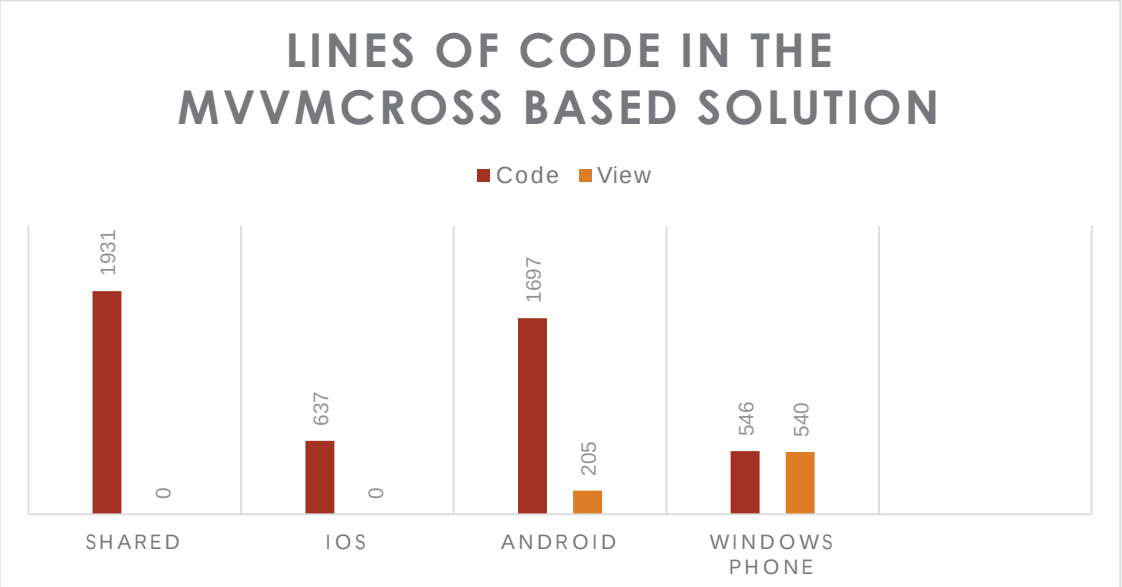

FIGURE 36 - LINES OF CODE, MVVMCROSS

We can see that 1931 lines of shared code. There is only 637 lines of code in iOS, but again iOS views are compiled to binary code and it is not possible to count. In WP the metrics are as I would expect them to be. Android has 1697 lines of code and only 205 lines of view code (.axml). By browsing the code, one should quickly see that bindings are done in code and not in the view. Witch evidently leads me to the conclusion that the Android implementation is not ideal.

In Xamarin:

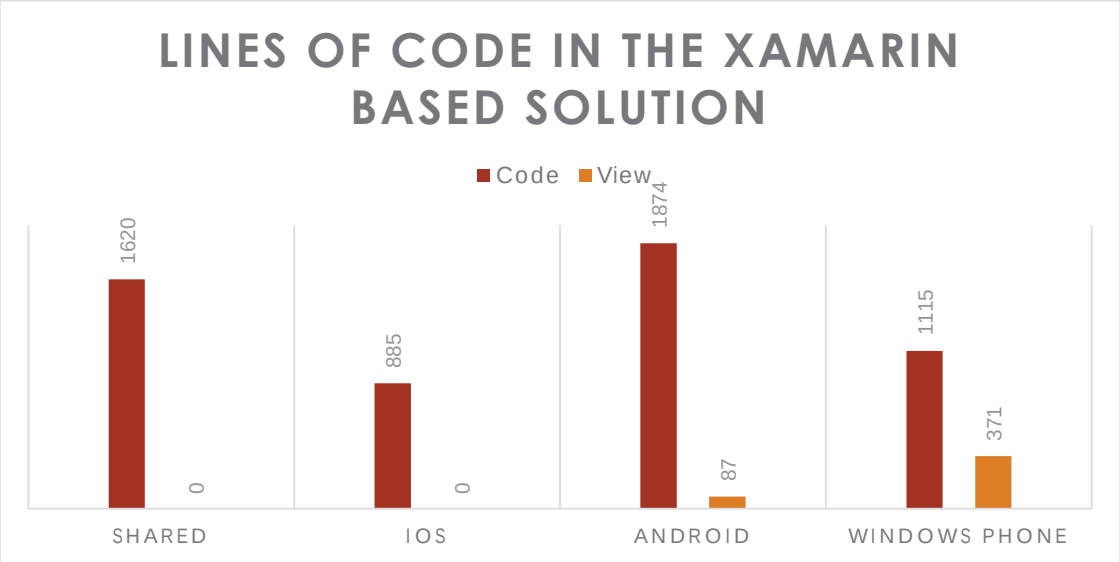

FIGURE 37 - LINES OF CODE, XAMARIN

In Xamarin there is 1620 lines of sharable code. This number is somewhat high due to many interfaces and abstractions created in this project. The runnable project is on average somewhat over the solution we had in MvvmCross as one could excpect.

4 Analysis

This chapter will concentrate on analysing the experiments in terms of the research questions as well as the theories I had during the experiment. I will go thought all of my smaller research questions and map them to the findings I have made in my experiments. As mentioned earlier, my research questions were:

- What tools can be used to develop cross-platform apps?
- What is a native look and feeling?
- What architecture pattern is best-suited for cross-platform development?
- Can cross-platform development suites compete with native development?
- What options do developers have for sharing code in cross-platform development?
- How much code can one expect to share in cross-platform development?
- Can cross-platform development save development time?
- What cases are cross-platform development suited for?
- Does cross-platform development create any limitations for this project?

4.1 Tools

In Chapter 2.2, I introduced a small part of the landscape of tools available to achieve cross-platform development. There are new tools released every month and new operating systems released every year, which makes this landscape complicated and large. I put a great deal of effort into classifying these tools. Two of the categories were extra-exciting for this paper since they allowed me to build cross-platform applications. The shell-wrappers wrap a web-based application in a native web view, and the SDKs could build native apps by translating code. In this research, I chose to dig into SDKs and especially the Xamarin Suite, but let me quickly explain a bit about shell applications.

A shell-wrapper wraps a webpage into a native app by embedding web code such as JavaScript, html and CSS into a web view. In addition, it provides a few special JavaScript functions to request access to the native APIs so that developers can access native functions, such as loading an image from the image gallery.

This technology has, for a long time, been said to be very slow compared to native development. While this is true, it is catching up as hardware gets better and better. Shell wrappers will always perform more slowly than native apps. I do, however, think that in a few years' time this gap will shorten so much that web apps could be the ideal starting point for any application builder. It might even be possible to create a native look and feel shell app. I think the gap may reduce to the size of what an SDK is today. At that point, it would be really valuable to have another look at shell-wrappers.

In the experiments, I have used a few different tools to achieve cross-platform development. My experiments are built upon Xamarin, and I now have a great overview of what Xamarin can provide in the development process. I also have a good view of the pros and cons one runs into by using a tool like Xamarin.

Xamarin is a powerful tool, but it is hard to share much of the code without using the MVVM pattern. Converting models or services from one object-oriented language, such as C#, to another, such as Objective-C, should be trivial if one knows them both.

One of the big reasons to choose Xamarin is the built-in features one gets by using the .NET framework. While Java is also an excellent programming language, it does not have as many features as .NET. Having the possibility to use Lambda expressions and async/await statements is almost a good enough reason to choose .NET over a native solution. Lambda has been released in Java development kit 8 (JDK8), but is not yet possible to use in native Android development.

MvvmCross was another tool I experimented with. I think that MvvmCross improved the cross-platform experiment. Using MVVM, developers have the possibility to share code such as app navigation and actions.

The last tool I used was MonoGame to develop Brickasa, a puzzle game where users can try to build structures with a given set of shapes. MonoGame works as a game engine on top of Xamarin and Mono. I saw that I could share most of the code with little or no difficulty.

All the tools tested in the experiments are fit for developing apps. Xamarin brings the power of .NET into mobile platforms. MvvmCross allows one to share more code; MonoGame is suited for game development on the Xamarin platform.

While I have been researching Xamarin, the concepts and techniques used by Xamarin may also apply to other SDKs. If one is interested in developing apps in Java, one could take a look at CodenameOne. Ruby developers should look at Rhodes.

I asked the question, “What tools can be used to develop cross-platform apps?” The answer is many; the one a particular developer chooses should not be based on a guess, but a pre-analysis of what one needs and how one wants an app to be. I would recommend Xamarin in many cases.

4.2 Native look and feel

I have talked about a native look and feel in this entire paper. All my experiments have involved some theories about this. I strongly believe that a native look and feel is something for the user and developers to measure. However, that is hard since it may vary on various devices. For example, one may get a native look and feel by developing a Xamarin-based app on the latest iPhone, but if one runs the same application on an older iOS device, such as iPhone 4, then it may not have a native look and feel.

One thing is for sure: while I have developed four applications that all feel native to me, the only way to ensure a native look and feel is by developing a truly native application. In experiment 5, I compared native applications with Xamarin-based applications, and the results were that native applications are faster than Xamarin-based applications, but not by much.

In my opinion, it is quite possible to develop apps that users will experience as native by using Xamarin tools. If developers decide to go for a pure Xamarin solution, they can expect that their apps will be somewhat slower than a truly native application. If they go for a MvvmCross-based application, they can expect the application to be a bit slower than a purely Xamarin-based approach.

4.3 Architecture

In these experiments, I had a look at two different architecture patterns. The first one was model-view-controller (MVC).The MVC pattern is popular for many developers, and that is not without reason. MVC is simple to learn and easy to test. However, in my experiments, I found that MVC was hard to apply on cross-platform solutions.

MVC was used in the LittleWishList experiment. I saw that the controller was hard to share between projects due to the high connection between a controller and the view. Android activities inherit from the activity class or a subclass of activity such as tabActivity or listActivity. On the other hand, iOS inherits from UIViewController or a subclass of UIViewController, such as UITableviewcontroller.

This inheritance problem made it impossible to share code. If someone absolutely wants to share this code, it could be done partly by using conditional build actions. I do not think this is a good approach, and the code will have more lines of conditional build actions than shared code. The code would have been hard to maintain and develop as the app kept growing; this would create annoyances for developers. It would also be tricky to identify bugs that would appear.

The reason it is difficult to share classes that inherit from other classes is because one has to override certain methods in order to get the application working. In an activity, one usually wants to override the OnCreate method, and inside that method, one would have requested different navigation parameters, such as an ID of the selected object. In the iOS environment, however, one would rather send the clicked object as a parameter to the constructor of the new view.

This makes MVC somewhat unfit for cross-platform development. To be able to share only models and services is clearly a limitation.

I also had a look at the model-view-viewmodel (MVVM) pattern. This pattern gave me far more power than MVC. The big difference was the viewmodel. In MVVM, a viewmodel does not know about the view, and it is also platform-independent and thereby easy to share.

It was possible to share simple properties and commands inside the viewmodel; these commands could handle navigation as well as requests to different services, thus enabling me to share much more code than in MVC. It is also easy to inject platform-specific code into the viewmodels, making it easy to rely on a shared codebase while having the flexibility to access platform-specific features.

There were also some issues related to using MVVM. Sometimes, in a Windows Phone environment, I had to call specific commands from the code behind the file rather than calling it straight from the view. This is unfortunately how Windows Phone works, and there is nothing to do about that. A good example is the application bar. The application bar is not bound to the viewmodel. Instead it is bound to a separate data context that cannot be accessed.

For cross-platform development, an MVVM architecture is superior to the MVC pattern despite the fact that one sometimes has to call the viewmodel from a code behind the file. This is because one can share navigation and commands and update the view from the viewmodel. MVVM works that way so that when a property is changed, it casts a property-changed event. The view can listen to this event by using a binding; the result is that the view will update the specific field.

MVVM does possibly have a more intense learning curve, but the results are well worth it. If one decides to make a cross-platform product and wants to share as much code as possible, then the MVVM approach is recommended over MVC.

4.4 Ecosystem

The development environments utilized in this paper are Xamarin Studio and Visual Studio. Xamarin studio is a Xamarin product and can run on both Mac and Windows. On Mac we can develop for both iOS and Android. On Windows, Xamarin studio is one of two options, the other one being Visual Studio.

Xamarin Studio is an acceptable development environment but nothing more. Visual Studio developers get a recognizable view with access to a decent set of features. It has built-in code completions as well as an option to publish your application from the IDE. I have found some bugs with Xamarin Studio that are annoying.

I have found out that the support for lambdas and extension methods seems to be very lacking. Xamarin Studio does not recognize some extension methods, such as linq. This is especially true when using them in an advanced context. Take a look at the image below.

```
try {
    return await Constants.MobileService.GetTable<Wish> ().Where(b => b.UserId == id).ToListAsync ();

} catch (Exception e) {
    throw new Exception ("Failed to get wishes", e);
}
```

CODE 25- XAMARIN STUDIO REPORTING FALSE ERROR

One can here see that Xamarin Studio marks the method as invalid by saying that a wish does not have a property UserId. However, this build is successful because a wish does have a property UserId.

Xamarin Studio is often hard to understand. In the image below, you can see that Xamarin has found a fault with the program. It is hard to understand where and what this fault is based on the given input.

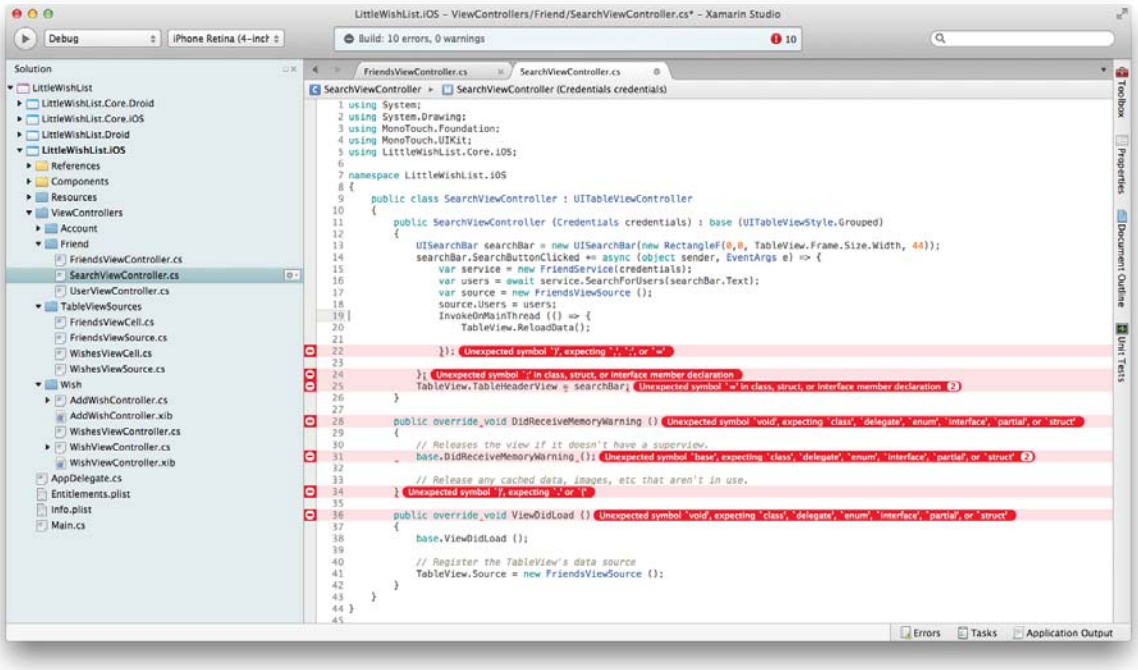

FIGURE 38 - XAMARIN STUDIO ERROR REPORT

Xamarin Studio also has some problems with the build engine. Below is a screenshot of a build that did not compile. It looks like there is faults all over the code. In reality, however, this was a simple code error that should be highlighted by the compiler.

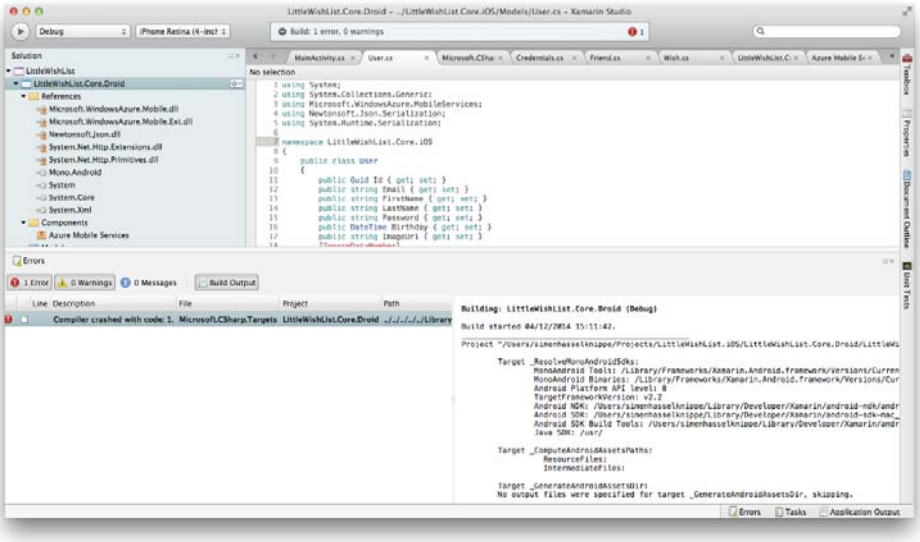

FIGURE 39 - XAMARIN STUDIO BUILD FAILED

Apart from this error, I have found Xamarin Studio easy to learn. Developers should quickly get used to the editor as well as a few commands for building and running projects.

The other option is Visual Studio. In my opinion, Visual Studio is superior to Xamarin Studio. Visual also has a lot of extensions one can apply, such as resharper and stylecop. The downside on using Visual Studio is the lack of support for iOS. While iOS application can be built and run through "Xamarin build host", it is not possible to edit views.

The best option is to use Visual Studio to build the shared code, the Windows projects and the Android projects. I would recommend Xamarin Studio for iOS development. The other parts of the ecosystem are great. I have not talked much about the Xamarin Test Cloud. Xamarin Test Cloud is a web application where one can upload applications and test them on various devices. One then gets an extensive report back saying what went wrong on which platforms.

I think that the entire ecosystem built around Xamarin products is great. I especially like the Android UI editor, the ability to edit iOS views using xCode, and the Windows UI in Visual Studio. I think that Xamarin provides an ecosystem that is as great as or perhaps a little bit better than the native ecosystems such as xCode and Eclipse.

4.5 Code sharing strategies

Earlier, I discussed which code-sharing strategies I had for cross-platform development. The two options I looked at were portable class libraries and file linking. All of the projects have used one of the two methods.

In LittleWishList, I linked files in specific class libraries. This strategy is a bit hard to maintain. There is constantly a need to go into different projects and add new links as more and more files are added. On the bright side, file-linking is powerful. Developers have the possibility to use all of a device's APIs inside the applications. Doing this made it possible to share the code used to connect to Azure Mobile Services. If I had used a portable library in this situation, the code would have to be injected from the projects int the core class library.

In Quizter, I used a portable class library (PLC). PLCs are indeed a unique and cool feature. They allow developers to have the code in only one place, and there is no maintenance. This does, however, create a set of limits by only allowing one to share code that is understandable by subset of .NET code. If one needs to use code that is not possible to run, then that code has to be injected by the use of interfaces.

In Brickasa, I used file-linking inside each and every project. The result was somewhat messy, but it worked. In this approach, it is important to structure one's code in a way that reduces the amount of mess, such as adding folders for all linked code.

I will not say that one is better than the other because they both have different use cases. Portable class libraries are a really good option if one does not have much code that cannot run on the combined .NET framework. Linked files are a good option if one does.

It is also possible to combine the two solutions. Developers could create one PCL that serves as a core and then link class libraries containing code that has to be injected on a class-specific level. The core project would then contain all the code it is possible to share, and the linked class libraries would contain the implementation of unsharable code. This option would give the developers the best of both worlds.

4.6 Sharable code

As for sharable code, one can expect different numbers depending on the project. In a pure Xamarin MVC solution, one could expect somewhere around 25% of the code to be sharable. It does, however, depend on the application one is developing. If one does have a great number of services and does lot of work in the cloud, the amount of shared code would increase, but for an app that does not use many services, such as a flashlight, then the number would decrease.

When developing a cross-platform application using MVVM, one can definitely expect to share more code than before. I would say that in an average project, around 50% of the code could be shared. Again, this number serves as a starting point. If one creates fantastic views that handle a lot of animation, then the number would go down. If one uses simple small views built in the designer, however, the code would go up.

In a game situation, this is quite the opposite. Apart from the boilerplate generated code, I was able to share almost 100% of the code. There was some work related to generating content, and I showed that Android did not support loading resource files the regular way. I would argue that games should be developed with a cross-platform approach since it does, in a great way, give one this option.

Testing sharable code is also trivial, while testing views in a mobile environment can be hard. Pieces of code that are not attached to the views should be easy to test. In cross-platform development, this means that all the code one can share should be tested. This will increase the amount of sharable code. If one is using a native approach, all the services have to be tested natively. This means that for a service, in addition to writing the service x times, it would need to be tested x times. In a cross-platform environment, however, one would only test the service one time.

Some might argue that 25% of the code is not much and perhaps decide to go native. While this is true for some cases, I strongly believe that it is worth it in many cases.

4.7 Development

One of my research questions is, “Can cross-platform development save development time?” In general, I think the answer here is yes. There are definitely possibilities to save time if one uses cross-platform development. This, however, is dependent on different variables such as:

- Which platforms should be supported?
- Which tests do we need?
- What architectures do we know?
- What programming languages do we know?
- What is our application update strategy?

If one already knows all the native programming languages but is not familiar with MVVM, then learning MVVM might take all the time one could spare on going cross-platform.

In my experiments, the development went smooth except for experiment 3, Quizter. In Quizter, I made a huge mistake of starting developing for all three platforms at the same time. I did not have a process structure that helped me, but rather fooled myself into thinking that I would manage to develop three applications simultaneously. This is one thing that should be carefully considered. If one developer should have control over all platforms at once, it will be messy.

Developers have to stay alert if they are supposed to make an application that can run on many platforms at once. I would argue that it would be better to create an application for one product first before porting it over to the new platform. This would reduce time, like I did in Brickasa.

Testing is another thing that might save developers time. Having to write fewer tests for the same methods is a time-saver.

What perhaps saved me most time in my projects was having the opportunity to use Visual Studio and having access to the .NET framework. Having access to async/await and linq is perhaps a good reason to pick Xamarin over native development.

One thing developers should consider before using Xamarin is that they have to learn some new technologies. Xamarin removes the need of learning Objective-C and Java, but developers still have to learn axml, xaml and the iOS designer. On top of this, they have to understand how the different platform lifecycles work and they have to learn many different APIs. If one is to compare this against a shell-app built with web technologies the gap is large. To conclude one could say that Xamarin saves development time versus native, but increased development time compared to shell-apps.

4.8 Features

If one looks at the feature-based taxonomy, Xamarin has 100% coverage on all platforms. I have not directly proven this in my experiments, so I will therefore do it now. The following table is an overview of the dimensions and what APIs one can use to get access to the features.

Dimention	iOS	Android	Windows Phone
Capture	1: MediaPicker 2: AVFoundation 3: AVAudioRecorder	1: MediaPicker 2: MediaRecorder 3: MediaRecorder	1: MediaPicker 2: CaptureSource 3: CaptureSource
Location	1: CoreLocation 2: CoreLocation 3:MKMapView	1: LocationManager 2: SensorManager 3: Google Maps	1: .NET location 2: Sensors 3: Bing maps
Storage	1: System.IO 2: System.Data 3: HttpClient	1: System.IO 2: System.Data 3: HttpClient	1: Files and folders 2: System.Data 3: HttpClient
Movement	1: CoreMotion 2: CoreMotion	1: SensorManager 2: SensorManager	1: Sensors 2:Sensors
Phone	1: UIApplication 2: UIApplication 3: UIApplication	1: SmsManager 2: Android.Net 3: ConnectivityManager	1: Devices 2: Phone.Tasks 3: Devices
Personal	1: Xamarin.Media 2: Xamarin.Media 3: MFMailComposer 4: EventKit 5: *	1: Xamarin.Media 2: Xamarin.Media 3: Android.Content 4: CalendarContract 5:*	1: Xamarin.Media 2: Xamarin.Media 3: Phone.Tasks 4: Phone.UserData 5:*
Gesture	A: *	A: *	A: *
Input	A: *	A: *	A: *
Unique	1: StoreKit 2: UIApplication	1: PlayServices 2: CloudMessaging	1: ApplicationModel 2: Phone.System

TABLE 14 - XAMARIN APIS OVERVIEW

4.9 Limitations

In my experiments, I have not found any limitations in developing an application based on Xamarin. However, there are some limits to Xamarin. These can be found at the Xamarin web page[7].

I would say that these limitations are not something one would want to use in everyday development. They should be researched before one selects Xamarin as a cross-platform development tool.

[7] http://docs.xamarin.com/guides/ios/advanced_topics/limitations/ and http://docs.xamarin.com/guides/android/advanced_topics/limitations/

5 Further work

As this research closes, it also opens up for more research around cross-platform development. This chapter will cover ideas and thoughts I have as this project is coming to the end. I will discuss in detail how others can continue my work and in which directions they should be heading.

This paper has given much attention to Xamarin and related products. I believe that many of the theories and conclusions will stay the same for many other SDKs, but there might be exceptions. I would have enjoyed reading about other SDKs and seeing an evaluation of them based on the same criteria as I have used here. It would also have been valuable to highlight the difference between these SDKs.

Choosing the right cross-platform development tools is perhaps the most important decision a developer could make. Therefore, it should be given high priority, and developers should invest time in it.

That is one way to continue this paper: to examine other SDKs what they could do and perhaps list pros and cons about them.

Another way this research could go is to start evaluating shell-apps and shell-wrappers. It would be interesting to see how they differ from SDKs and native development and what they can provide to cross-platform development. To continue this study by writing about PhoneGap, Appsbuilder or similar tools would be valuable.

I think this study in particular would be valuable in a few years since shell apps decrease the native look and feel gap more and more every year. It would also be valuable to have a look at how these web apps can be built to keep this gap at a minimum.

Another direction it could take is to go even more in depth on measurements, which I thought of as I analyzed PropertyCross. I think there are many more interesting numbers one can extract from this project. These numbers include frame navigation, such as how long does it take to navigate from one frame to another.

As continuation of this study, it would be valuable to check the source code for MvvmCross to see why it is taking longer and perhaps look for optimization possibilities.

I made one game called Brickasa. As stated before, games do really have an advantage when it comes to cross-platform development. It would be very interesting to see someone willingly to go deeper into this. For doing that, I would start by exploring more cross-platform game development tools. I identified a few more; one that looks very promising to me is Unity. Unity is recognized by many game developers and has been used in many projects. I would like a comparison between MonoGame and Unity, both in terms of features and, perhaps even more interestingly, in terms of speed and performance. I had hoped to do some of this myself, but to complete this study within

the given time frame, it was not possible. I therefore hope that someone interested in creating games may follow up on this research.

If one is interested in developing apps, I would kindly encourage going through my work and continuing on with one of the paths I have provided or another path that seems fitting.

6 Conclusion

To conclude this paper, it is time to go back to my research question. The research question for this paper is as follows:

"How can we build apps that have a native look and feel by using cross-platform technologies, and when should we use it?"

I had defined a set of issues to be able to answer this question better. In this chapter, I will go through them in detail.

I asked, "What tools can be used to develop cross-platform apps?" It turns out there are a great number of tools that can be used to develop cross-platform applications. One possible option is to use Xamarin. The Xamarin tool suite is powerful yet simple to use. Xamarin can be built with MvvmCross in order to use MVVM as the architecture. If one wants to build a game, MonoGame is built on top of Xamarin, providing a game loop. As mentioned, there are other tools as well, including PhoneGap, AppsBuilder, CodenameOne, Rhodes and Titanium, to name a few..

Next I asked, "What is native look and feeling?" There is no simple answer to this. The best answer I can come up with is that if it looks and feels native, it has native look and feel. I have demonstrated how Xamarin has a longer load time than native applications. This is very little and almost not visible to users, however. Native look and feel is up to users to decide. One should have a user test session to check if the app has a native feeling.

"Can cross-platform development suites compete with native development?" Yes, they can; while this depends on the product, it certainly is possible. We saw how Xamarin has solved this by adding a plugin to Visual Studio and making it possible to open xCode on Mac. It has developed a rather good UI designer for Android, and in addition, it has the test cloud and its own editor. A cross-platform development suite can definitely compare with native if they are developed right.

In my experiments, I tried to test different architectures to answer what architecture pattern is best suited for cross-platform development. The results were pretty clear. Using an MVC architecture, I could share services and models, while in an MVVM environment, I could share the viewmodel as well. The MVVM architecture is best for developing a code platform application if the goal is to share as much code as possible.

I also invested a great number of resources into finding and testing what options developers have for sharing code in cross-platform development. In turns out there are two possible options. PCLs are ideal because they have no need for maintenance, while file-linking is an option if one's code relies on code that is not supported by the portable .NET subset. I also analysed that a combination could be used if one wants to develop a large project.

I also showed how much code one could expect to share. I argued that this would greatly rely on the implementation. If one has to do a lot of work on the UI and does not have many services, the number will be reduced. If one has a great set of services, however, it will increase. By using the MVVM pattern, one should be able to share over 50% of the code if the application is developed the right way.

I argued that much time can be saved while doing cross-platform development. One of the biggest benefits is that one will not have to test all services multiple times.

I compared Xamarin with the feature-based taxonomy of mobile applications and saw that it had 100% coverage for it, making cross-platform development ideal in many situations. Cross-platform development is very useful in a game situation since nearly 100% of the code can be shared.

I did not find any limits in cross-platform development, except the ones described earlier. I would argue that these limits are so few and one rarely needs them, so in most situations, it will not affect your problem or application.

So, back to the big question:

"How can we build apps that have a native look and feel by using cross-platform technologies, and when should we use it?"

Developers can build apps by using cross-platform tools such as Xamarin. If developed right, they will have a native look and feel, despite the fact that they are a tad slower than truly native application. Cross-platform applications can be used in any situation as it has support for many features. A cross-platform approach should be considered when any of the following conditions apply to a project:

Developers want to create a native looking application.
Developers want their applications to run on multiple platforms.
Developers do not have the resources to develop many separate applications.
Developers do not have many development teams that know all native platforms.
Developers want to publish an application on multiple platforms.

7 Looking back

I have now worked on cross-platform development for well over a year. One thing that hits me now is that the world of cross-platform development moves remarkably fast. When I started, there was no support for portable class libraries, and MonoGame only had unofficial support for Windows Phone. Now PLCs have been released stably, and it has added support for many namespaces such as linq and threading.task. This makes it possible to use async/await. All in all one could say that what I did when I started one year ago is no longer the common way of doing cross-platform development.

In fact, this world of cross-platform development is changing so fast that my conclusion may not hold water for so long. This is a bit sad, but it is also a sign that many great things are going on and cross-platform development keeps innovating, which is essentially good for customers, vendors and developers.

I started my project by researching a few tools with which to experiment. In retrospect, I now see that the difference between tools in the same category is enormous. The differences between codenameOne and Xamarin are huge. I based my choice on features, reputation and community. These factors could have changed while I was doing my thesis.

I wish I had known from the start that picking the right tools is just as relevant as picking a category. I could have spent less time in the Xamarin environment and researched more tools instead. As I said, if one has a small team of web developers and one wants to turn a site into an app as soon as possible, web is the obvious option. That said, there are so many ways to go on the web. What kind of architecture should one use? What UI frameworks should one use? How can one deliver data from servers to the app? The list goes on.

In the experiments, I learned a lot, and I am grateful for getting the opportunity to test a product like this. Looking back, I now have gained much experience on cross-platform development.

One thing that was especially hard for me was to be ability to act neutral in terms of language, IDEs and platforms. We all have our favourite languages, tools and patterns, and it is hard to evaluate them neutrally when one have an opinion on everything. I feel that I have managed to stay neutral and make neutral conclusions. I still think the work should be done with someone with other preferences to ensure that the results from this paper are correct.

Looking back, I also feel like I perhaps have taken too much on my shoulders. I might have been better off choosing not to develop APIs and instead using APIs others have created, like I did in Movie Seeker. Having to manage an API as well as all platforms was one of the reasons I did not manage to successfully finish Quizter. I think the Quizter project is a good example of how things can go wrong if one does too much in a project.

One thing I regret is that I did the Property Cross experiment as my last experiment. I therefore got in a bit of time struggle. There are many more interesting numbers I could have extracted from this project.

As this paper is coming to an end, I must say that I have learned so much. I hope that reading this paper has been useful and that readers now have a good overview of challenges related to cross-platform development but also are interested in the subject and want to find out more.

I think this research shows a lot of the pros and cons with going cross-platform; in the end, one should base the decision on resources such as time and money and the expected results.

8 Appendix

8.1 Screenshots

8.1.1 LittleWishList

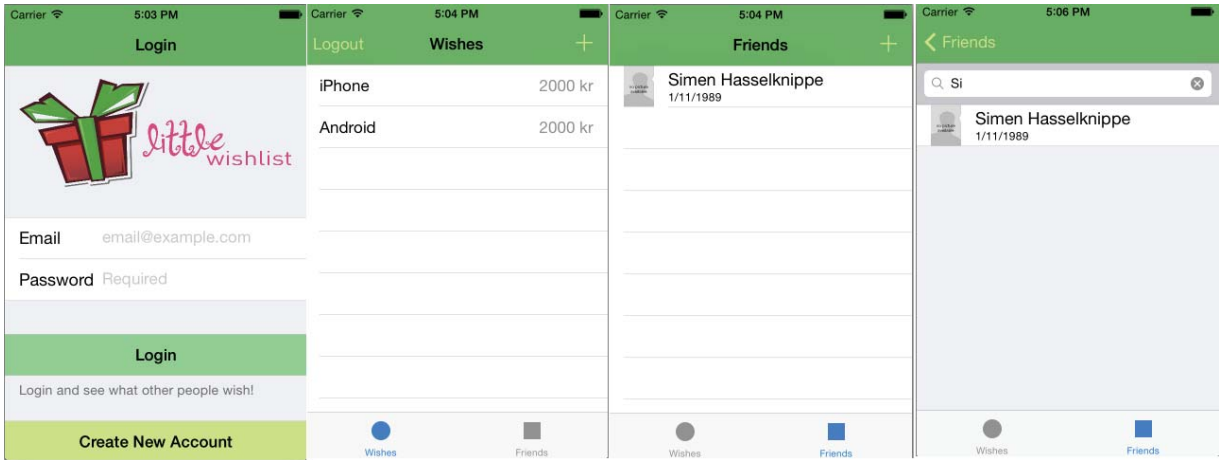

FIGURE 40 - LITTLEWISHLIST RUNNING ON IOS

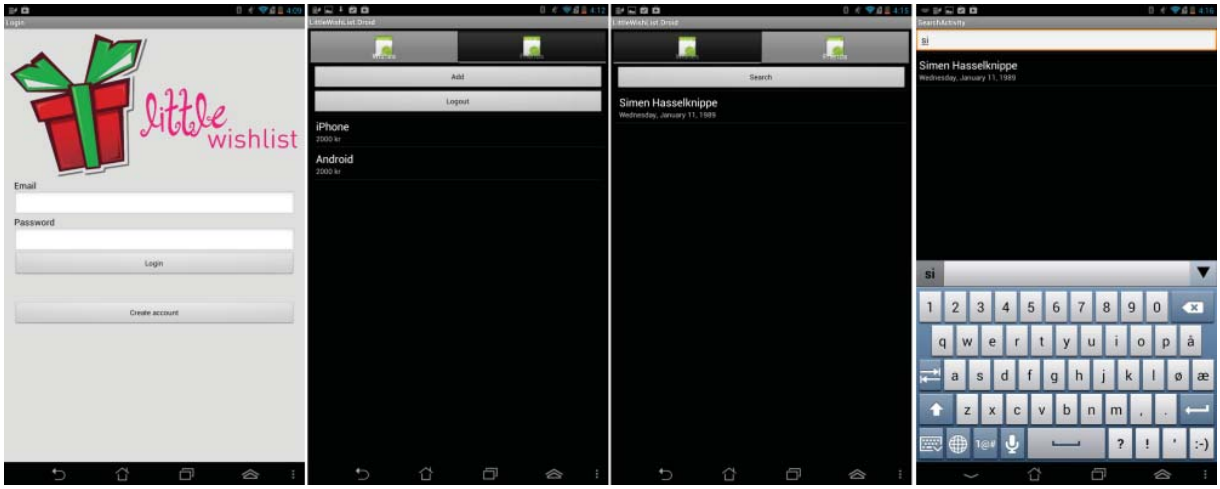

FIGURE 41 - LITTLEWISHLIST RUNNING ON ANDROID

8.1.2 Quizter

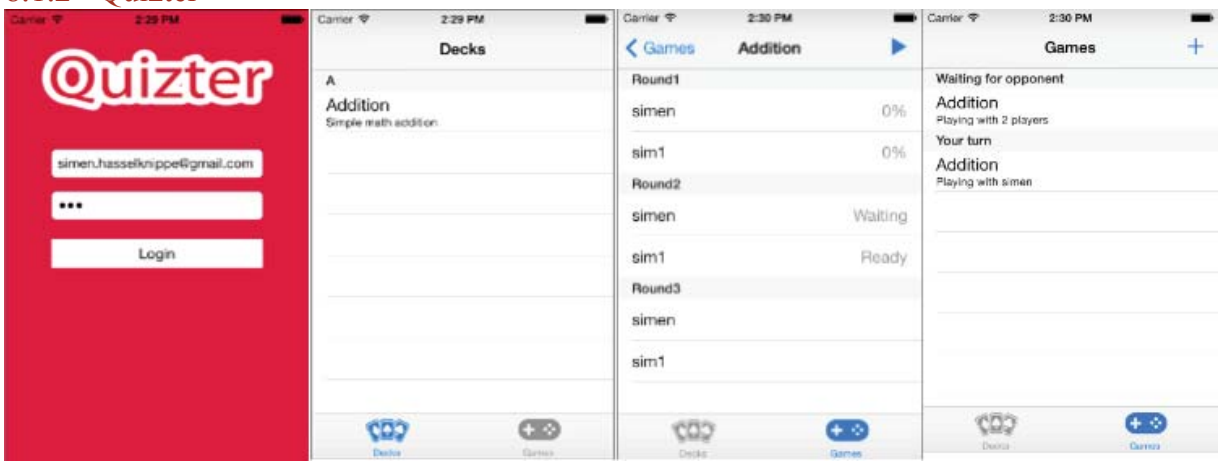

FIGURE 42 - QUIZTER RUNNING ON IOS

FIGURE 43 - QUIZTER RUNNING ON WINDOWS PHONE

8.1.3 MovieSeeker

FIGURE 44 - MOVIESEEKER RUNNING ON ANDROID

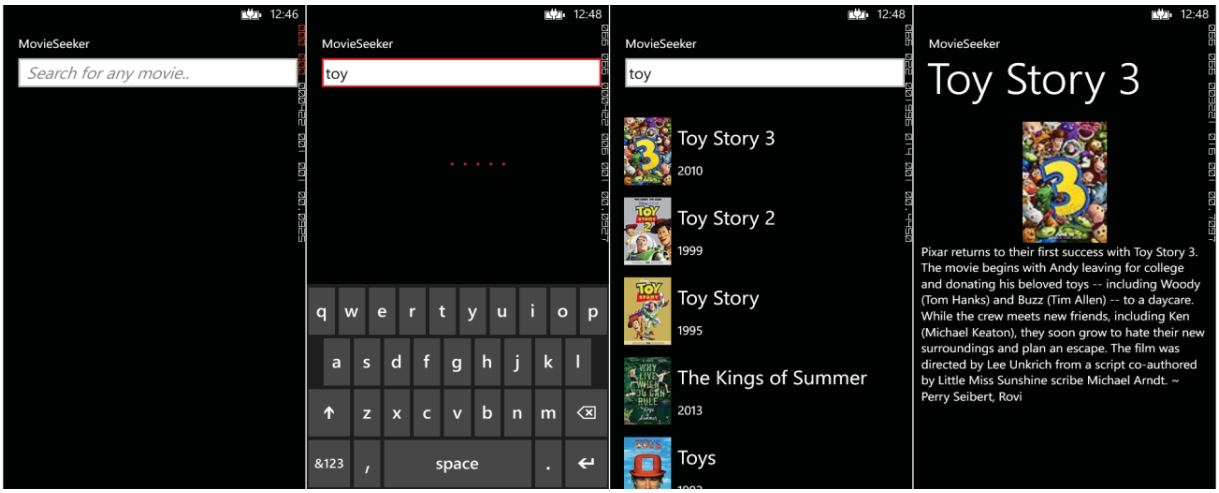

FIGURE 45 - MOVIESEEKER RUNNING ON WINDOWS PHONE

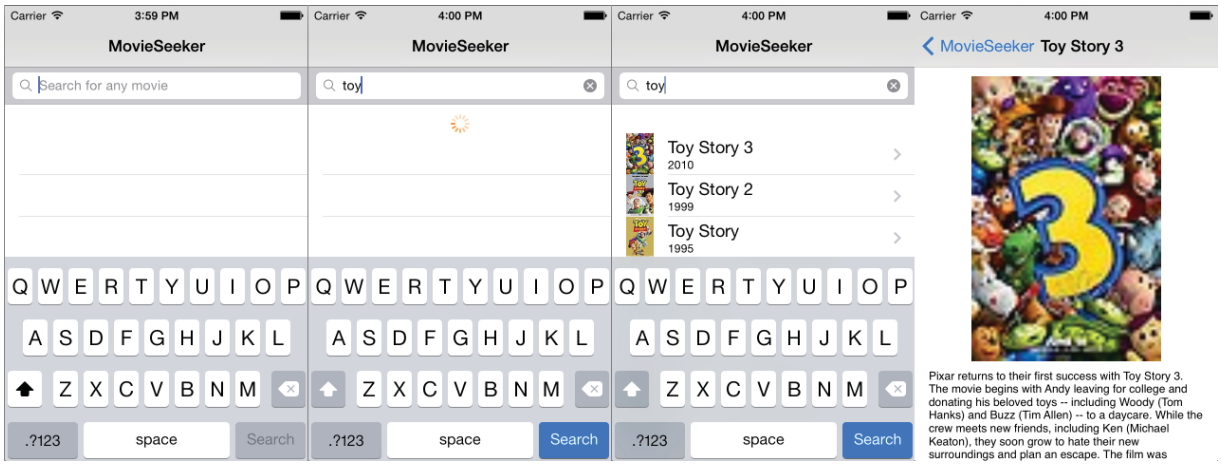

FIGURE 46 - MOVIESEEKER RUNNING ON IOS

8.1.4 Brickasa

Since Brickasa looks identical on all platforms, I will show just one screenshot from each device.

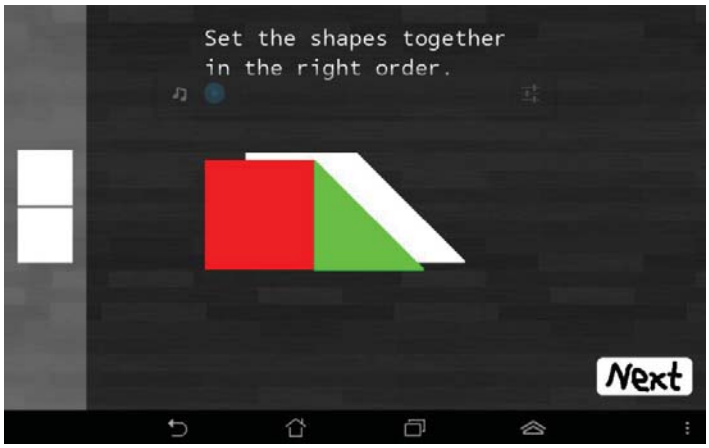

FIGURE 47 - BRICKASA RUNNING ON ANDROID

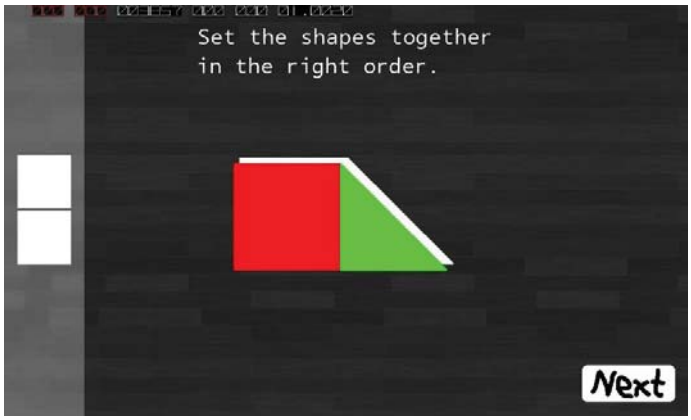

FIGURE 48 - BRICKASA RUNNING ON WINDOWS PHONE

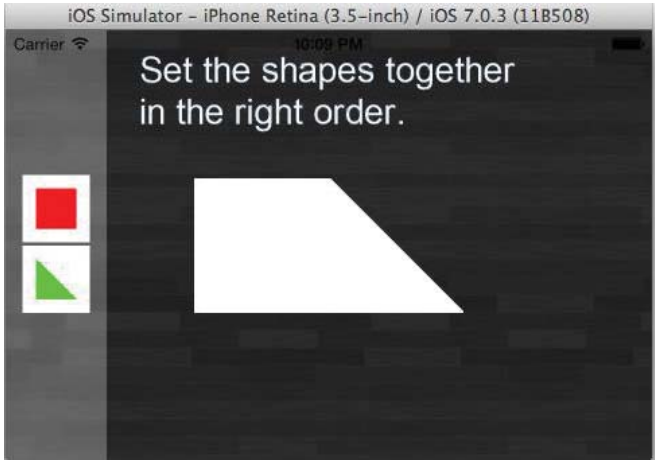

FIGURE 49 - BRICKASA RUNNING ON IOS

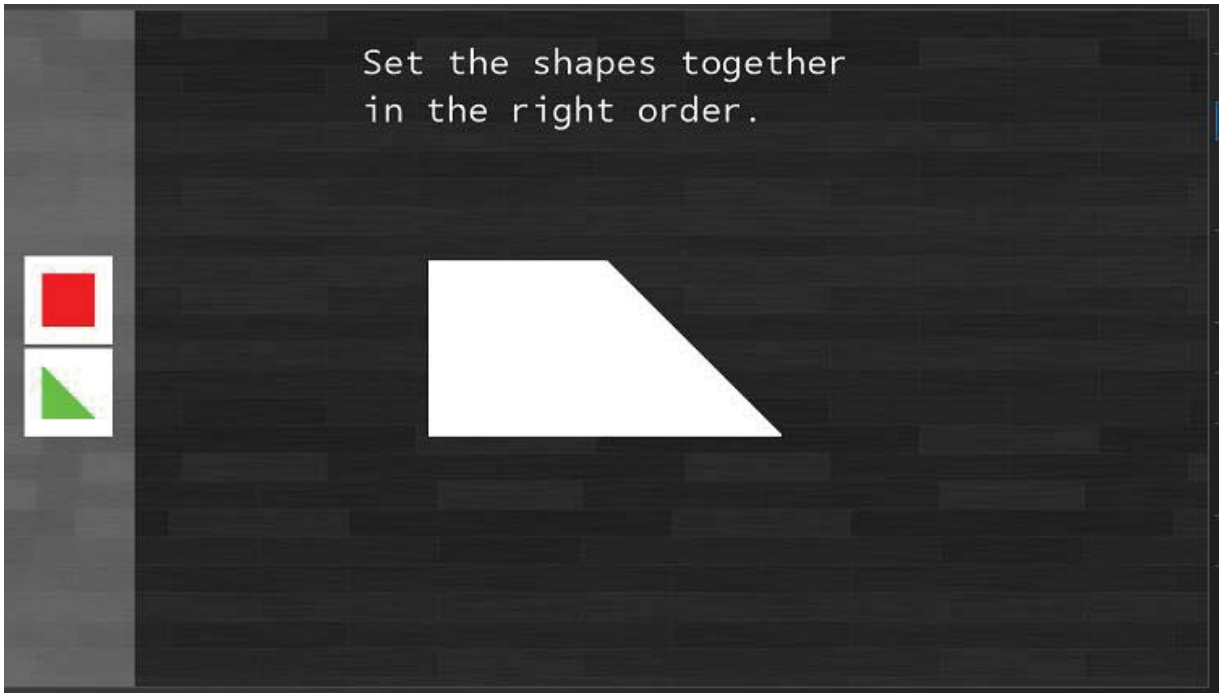

FIGURE 50 - BRICKASA RUNNING ON WINDOWS 8

8.1.5 PropertyCross

Screenshots from PropertyCross can be found on the PropertyCross homepage:

- Native: http://propertycross.com/native/
- Xamarin: http://propertycross.com/xamarin/
- MvvmCross: http://propertycross.com/xamarinmvvmcross/

8.2 Code

The code written in the experiments can be found on GitHub. These experiments are meant to be used as research and is not of production quality. To run the projects you must have the required tools and frameworks, see each experiment for more information. I have only tested the experiments on one computer, so I give no guaranty that they run in other environments than my own. The code for each experiment is listed in the sections below:

8.2.1 LittleWishList

https://github.com/simhas/LittleWishList

8.2.2 Quizter

https://github.com/simhas/Quizter

8.2.3 Movie Seeker

https://github.com/simhas/MovieSeeker

8.2.4 Brickasa

https://github.com/simhas/Brickasa

8.2.5 PropertyCross

http://propertycross.com/native/

http://propertycross.com/xamarin/

http://propertycross.com/xamarinmvvmcross/

9 References

1 Billion via mobile devices. (2010, 6 23). Retrieved from Business-Opportunities: http://www.business-opportunities.biz/2010/07/23/1-billion-via-mobile-devices/

About jQuery. (2012, 10 10). Retrieved from jQuery: http://jquery.com/

Android activations outpacing baby births. (2012, 1 15). Retrieved from The Telegraph: http://www.telegraph.co.uk/technology/ces/9013487/CES-2012-Android-activations-outpacing-baby-births.html

Apple developer. (2013, 11 01). Retrieved from iOS Human Interface Guidelines: https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/MobileHIG/index.html

Apple makes more iphones than humans make babies. (2012, 1 25). Retrieved from cnet: http://news.cnet.com/8301-17938_105-57365767-1/apple-makes-more-iphones-than-humans-make-babies/

Asymco. (2012, 02 16). Retrieved from http://www.asymco.com/2012/02/16/ios-devices-in-2011-vs-macs-sold-it-in-28-years/

Coder, D. (2012, January 4). *Modern Cross Platform Development*. Retrieved from Dodgy Coder: http://www.dodgycoder.net/2012/01/modern-cross-platform-development.html

Facebook-mobile-user-count. (2013, 8 13). Retrieved from Techcrunch: http://techcrunch.com/2013/08/13/facebook-mobile-user-count/

Fahrenkrug, J. (2011, September 12). *Thoughts on Mobile UI Design*. Retrieved from Springenwerk: http://www.springenwerk.com/2011/09/thoughts-on-mobile-ui-design.html

Flickr Cameras. (2013, 10 28). Retrieved from Flickr: http://www.flickr.com/cameras/

Gartner. (2012, 2 13). Retrieved from http://www.gartner.com/newsroom/id/2335616?utm_source=Triggermail&utm_medium=email&utm_term=10%20Things%20In%20Tech%20You%20Need%20To%20Know&utm_campaign=Post%20Blast%20%28sai%29%3A%2010%20Things%20You%20Need%20To%20Know%20This%20Morning

Gartner. (2012, December 18). *85 Percent of All Flat-Panel TVs Will Be Internet-Connected Smart TVs by 2016*. Retrieved from Gartner: http://www.gartner.com/newsroom/id/2280617

Grannell, C. (2012, September 12). *Facebook: bet on HTML5 a 'big mistake'*. Retrieved from .net magazine: http://www.netmagazine.com/news/facebook-bet-html5-big-mistake-122221

How many app types are there. (n.d.). Retrieved from Buzinga: http://www.buzinga.com.au/smartphone-apps/how-many-app-types-are-there/

IDC. (2013, 5 1). Retrieved from http://www.idc.com/getdoc.jsp?containerId=prUS24093213

Künzler, J. G. (2013, 10 8). *MacTrast*. Retrieved from How People Really Use Their Smartphones: http://www.mactrast.com/2013/10/infographic-people-really-use-smartphones/

Nickerson, R., Varshney, U., Muntermann, J., & Isaac, H. (2007, 12 31). Towards a Taxonomy of Mobile Applications.

Stuart, J. A. (2005). *A Unified Approach for Cross-Platform Software.* Nevada: University of Nevada.

Textually. (2012, 2 6). Retrieved from Textually: http://www.textually.org/textually/archives/2012/02/030229.htm

Venturebeat. (2013, 3 4). Retrieved from Venturebeat: http://venturebeat.com/2013/03/04/we-will-download-70-billion-mobile-apps-in-2013-50-android-41-ios/

When will the tablet market be larger than the pc market. (2012, 03 02). Retrieved from Asymco: http://www.asymco.com/2012/03/02/when-will-the-tablet-market-be-larger-than-the-pc-market/

Word of the year. (2011, 1 8). Retrieved from Americandialect: http://www.americandialect.org/app-voted-2010-word-of-the-year-by-the-american-dialect-society-updated