UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science
Software Engineering

Mikk Pavelson

The deficiencies in the Apple iOS SDK with the example of third party frameworks usage

Master's Thesis (30 EAP)

Supervisor: Marko Peterson, M.Sc

TARTU 2014

The deficiencies in the Apple iOS SDK with the example of third party frameworks usage

Abstract:

The objective of this thesis is to find weak spots in the iOS SDK, a programming library used to develop applications for the iOS operating system running on Apple's mobile devices. The method to finding these deficiencies is to first index the most frequently used third party frameworks used by iOS developers and analyze the reasons for their popularity. The first step to achieve this will be looking at the most liked frameworks in the GitHub hosting service. The second step will conduct a survey among iOS programmers with questions regarding these very same frameworks. The results will then be analyzed and summarized.

Keywords:

iOS SDK, frameworks, Apple, software development

Apple'i iOS SDK puudujäägid kolmanda osapoole teekide kasutatavuse näitel.

Lühikokkuvõte:

iOS SDK on Apple'i iOS operatsioonisüsteemile mõeldud rakenduste programmeerimisel kasutatav tarkvaraarenduskomplekt. Käesoleva magistritöö eesmärk on leida iOS SDK silmatorkaimavad puudujäägid. Selle tulemuse saavutamiseks tuleb kõigepealt leida iOS-i programmeerijate poolt kõige enimkasutatavad kolmanda osapoole vabavaralised teegid ja analüüsida nende populaarsuse põhjuseid. Esimene samm selliste raamistike leidmiseks on vaadata võõrustusdomeeni GitHubi kõige laialtrakendatavaid tarkvaraprojekte. Järgnev samm näeks ette iOS-i arendajate seas küsitluse läbiviimist, mille päringud põhineks eelmisel etapil saadud teadmistel. Viimase osana tuleks saadud tulemusi sügavamalt analüüsida ja neist järeldusi teha.

Võtmesõnad:

iOS SDK, teegid, Apple, tarkvaraarendus

Table of Contents

Introduction

With the introduction of the iPhone in 2007 and the subsequent revealings of other mobile operating systems and the devices running them, including the Android in 2008 and Windows Phone in 2010, the world saw a pivoting change in the usage of mobile hand-held devices [1]. It was no longer a complex and exclusive commodity for the enterprise market, used only by executives in a need of a personal assistant and a portable e-mailing device but rather an everyday utility device used for socializing, photography, day-to-day planning and a million other trivial tasks, recognized by everybody from teenagers to CEO's.

The last years have seen a lot of competition and tug-of-wars between the developers of these operating systems and different hardware manufacturers, which has laid ground for a rapid progress in technological development and the number of users as well as the number of software developers [2]. All the major mobile operating systems of today are surrounded by open environments used by millions of software developers in order for them so create their own applications. While Android remains to be an open source platform, Apple's iOS and Microsoft's Windows Phone maintain a proprietary nature [3]. This makes Android open to altering for everybody but makes it impossible for third parties to change the appearance of iOS or Windows Phone. Although the operating systems themselves cannot be changed, applications, or so-called "apps", can be built by anyone, albeit for a small fee [4], [5].

The three most popular operating systems all have their own software development kits (SDKs), environments, tools, and frameworks. Apple came out with their own object-oriented programming language Objective-C, which was meant for building applications for their Mac OS X and iOS operating systems. It was expanded from a mixture of ideas from Smalltalk-80 and the C languages [6], [7]. During the iPhone era, the main tool used by developers for building Apple products has been Xcode, Apple's own integrated development environment for the Cocoa and Cocoa Touch frameworks, used respectively by Mac OS X and iOS. While Cocoa is a set of basic frameworks used for Mac OS X development, Cocoa Touch adds the abilities to use hardware tools and features that are only available on Apple's touch screen devices, such as animations,

multitasking and gesture recognizers [8], [9]. These two sets of frameworks together make up the iOS software development kit (SDK) that is used for creating all of the native iOS applications currently available in the Apple App Store. The iOS SDK offers a huge variety of possibilities from database management to multi-finger touch screen manipulations, but third party libraries and frameworks are still almost a daily appliance for many developers and plenty of their projects. Ever since the iPhone OS SDK was made public in 2008 it has gone through a great number of updates, but even today there are a lot of deficiencies in the iOS software development kit [10]. The need for external frameworks directly derives from these deficiencies with developers eagerly constructing new and better tools for making tedious tasks easier and more efficient. The software development community decidedly advocates for an open source mentality, making most of the available libraries free to use and rebuild in any way other developers see fit. According to the popularities of certain third party frameworks it should be rather visible what are the current weak spots of the iOS SDK.

The objective of this thesis will be to map the influential updates made to the iOS SDK by Apple and find third party frameworks that are currently most used by iOS developers. There are two ways of finding this out:

- looking for the most downloaded frameworks for iOS;
- asking developers' personal opinion via a questionnaire.

Apple's previous updates to the iOS SDK can give an interesting insight to which areas Apple themselves has redeemed most update-worthy in the past. Put together and cross-referenced, these two sources of information should provide for a deep base of knowledge.

The first part of this thesis will cover the iOS SDK with its most significant updates and the background of external frameworks. It will be followed by an overview of the most popular currently available third party iOS frameworks.

The second part of this thesis will concentrate on finding the most popular third party frameworks. This will be done by looking through one of the most popular source on the web for software development libraries – GitHub, and by creating a survey among iOS programmers in order to find out the preferences of iOS developers in regard to external libraries.

In the final part of this thesis, these two aspects will be analyzed and compared to each other. The criteria for a deficiency in a framework will be defined, so that the deficiencies could be established and dissected. The outcome of this thesis will hopefully be to find an area in which there is still a need for enhancements, which could be a good basis for further work for the author as an iOS developer.

This thesis has been written with the assumption that the reader has some basic understanding of programming. Although not necessary, it helps to give a more comprehensive understanding of the code examples, which in turn makes it easier to make sense of the surrounding arguments.

1 Background

Apple usually releases a new SDK concurrently with every major and minor iOS update. The latest version of the iOS developer's library at the moment of writing this thesis on the 28 of October 2013 is iOS SDK 7.0.3. It was released alongside iOS 7 [11]. Many beta SDKs are released to developers before publicly releasing the Golden Master (GM) edition.

1.1 iOS SDK Version History

The first iPhone was officially unveiled on the 9 of January 2007 together with its new operating system, the iPhone OS. It was a closed environment, where the users were not meant to develop native applications, but rather come up with web pages customized for the smaller mobile screen [12]. Soon Apple changed their mind and made the first beta iPhone OS SDK available for developers in March of 2008, called the iPhone OS 2.0 [13]. In July the same year the tech giant officially made the new version of iPhone OS public, together with the new iPhone 3G and the App Store, which was to become the channel for developers to share their applications to the public. The name did not change to iOS until version number 4. [14]

1.1.1 iPhone OS 2.0

The first publicly obtainable iPhone OS SDK came with the following frameworks ready to be used [Table 1].

Table 1. The short summaries of each framework from the first public iPhone OS SDK [15].

Framework	Summary of Contents
AddressBook	Address Book is a framework that enables the use of the device owner's contact list, the same that is used by the native Mail and Messages applications.
AddressBookUI	Controllers that facilitate displaying, editing, selecting, and creating records in the Address Book database.
AudioToolbox	Interfaces for recording, playback, and stream parsing. iOS specific: additional interfaces for managing audio sessions.
AudioUnit	Interfaces for using built-in and custom audio processing plug-ins, known as audio units, the lowest programming layer in the iOS audio stack. In iOS, your application can use the built-in audio units.
CFNetwork	A framework in the Core Services framework that provides a library of abstractions for network protocols. These abstractions make it easy to perform a variety of network tasks.
CoreAudio	Interfaces for implementing audio features in applications you create for

	iOS and OS X. Under the hood, it handles all aspects of audio on each of these platforms. In iOS, Core Audio capabilities include recording, playback, sound effects, positioning, format conversion, and file stream parsing.
CoreFoundation	The most basic services used commonly in most applications. It also includes abstractions the most common data types, facilitates internationalization with Unicode string storage, and offers a suite of utilities such as plug-in support, XML property lists, URL resource access, and preferences.
CoreGraphics	A C language based API that is based on the Quartz advanced drawing engine. It provides low-level, lightweight 2D rendering.
CoreLocation	The framework that deals with the device's physical location, using the available hardware. Can be used to deliver location-based events.
Foundation	As the name says it is the framework that provides the API to satisfy the most basic needs with primitive object classes, utility classes, adds functionality and conventions not covered by the Objective-C language. It can also be looked at as a wrapped for Core Foundation classes.
MediaPlayer	An Objective-C framework adding the ability to play video and audio files, also with built-in classes for UI elements and controls. Also facilitates access to the device's iPod music library.
OpenAL	An API for the use of the cross-platform standard known as OpenAL, which is used for rendering multichannel three dimensional positional audio, which is a common way of manipulating with the origin of a sound.
OpenGLES	A C language based framework to help draw 2D and 3D content. It is often used for high quality graphic renderings in games.
QuartzCore	An advanced low-level 2D drawing engine, independent of screen resolution. Part of the Core Graphics framework [20].
Security	A C language level framework that defines an interface for information protection and software access control. Part of the Core OS Layer.
SystemConfiguration	An interface for establishing and keeping network connections.
UIKit	The main infrastructure for building an application, among which are the main elements in the UI, handling the events those elements may fire and interacting with the rest of the system.

The overview shows that there are several frameworks dealing with audio technology. Although there might be some overlapping, each framework has a focus. Core Audio is the digital audio infrastructure of iOS and OS X. In iOS it is optimized for the specific computing resources available on the mobile platform. There are additional services in iOS not present in OS X. The Core Audio framework, which itself provides data types and low level services, is considered to be a peer not an umbrella for the rest. It can be broken down as following: Audio Toolbox lets you manage the audio behavior of your application on a device that works as a telephone and an iPod; Audio Unit works with audio plug-ins and codecs and the OpenAL provides interfaces to work with the open-source cross-platform audio technology. Another thing to take away from this summary is that 14 out of the 18 listed frameworks are all C-based.

Since Objective-C is based on the C language and they can both be used to develop iOS apps, these frameworks can be used interchangeably. The remaining 4 frameworks can

be looked at as wrappers that were created in order to have common basic data structures and services available in Objective-C.

1.1.1.1 iPhone OS 2.1

While no new frameworks were added with the final release of iPhone OS 2.1 on the September 12th in 2008, there were additions to some existing ones such as Audio Toolbox, Audio Unit and UI Kit [15].

1.1.1.2 iPhone OS 2.2

The next small release was the iPhone OS 2.2. It marked the addition of the iOS specific AV Foundation framework, which provides a simple Objective-C programming interface for audio playback [Table 2].

Table 2. Frameworks added during the release of the iPhone OS 2.1 SDK [15].

Framework	Summary of Contents
AVFoundation	An easy to use Objective-C framework for creating and playing time-based audio and video files. It also includes manipulating live video streams from the device [24].

It also featured extensions of the Audio Toolbox, Core Foundation, Core Location, Foundation and UI Kit frameworks. Most of these updates were modifications or inclusions of new variables, new methods calls and constants.

1.1.2 iPhone OS 3.0

The next major release of the iPhone OS was unveiled together with Apple's new hardware release of the iPhone 3GS. The software came a year after the previous one, along with an updated SDK. The new operating system had such new features as copy – paste, setting up a WiFi hotspot and MMS [25]. Along with these feature updates for the iPhone OS came out the updated version of the SDK [Table 3].

Table 3. Frameworks added with the release of the iPhone OS 3.0 SDK [15].

Framework	Summary of Contents
CoreData	An Objective-C framework that provides an API for object graph management and data persistence for Foundation and Cocoa applications.
ExternalAccessory	A framework that enables the usage of external devices through the 30-pin dock connector or wirelessly using Bluetooth.
GameKit	A framework added to support Apple's Game Center functionality, along with such social features as peer-to-peer connectivity, multiplayer systems and leaderboards.
MapKit	This framework added the possibility to add maps to views and use annotations, overlays and coordinate lookups.
MessageUI	Supplies the presentation of views for sending e-mails and SMS messages without leaving an app.
MobileCoreServices	A framework that defines the low-level types used in uniform type identifiers (UTIs) [26].
StoreKit	An interface for allowing the acceptance of payments from users for additional services or products.

1.1.2.1 iPhone OS 3.1

Yet again there were no new frameworks in the 3.1 version of the SDK, but Apple updated the AV Foundation, Audio Toolbox, Media Player, Open GLES, Quartz Core and UI Kit frameworks.

1.1.2.2 iPhone OS 3.2

The third major version of the iPhone OS 3 contained one new framework [Table 4].

Table 4. Frameworks added with the release of the iPhone OS 3.2 SDK [15].

Framework	Summary of Contents
CoreText	A low-level high performance framework that provides text layouts and font handling.

Together with the new Core Text framework there were updates to the CF Network, Core Foundation, Core Location, Foundation, Map Kit, Media Player, Quartz Core and UI Kit frameworks.

1.1.3 iOS 4.0

The first of the Apple mobile operating system to bear the name iOS, it came out with a heap of new features. The biggest of which was multitasking, the functionality that gave the users the ability to switch between apps without closing them. The new software also included orientation lock, FaceTime calling, the iAd advertising network, folders on the home screen and so on. The iOS SDK had grown by 11 new frameworks that were designed to support the programming for these added features [Table 5].

Table 5. Frameworks added with the release of the iOS 4 SDK [15].

Framework	Summary of Contents
Accelerate	A C framework with the API for complex math operations and graphics processing.
AssetsLibrary	A framework that allows access to the device's images and videos.
CoreMedia	A C framework for playing and managing audio and video media.
CoreMotion	The framework to use when access to the device's gyroscope or accelerometer is needed.
CoreTelephony	Gives access to everything related to the device's cellular service provider, from information about a current call to whether VoIP is allowed.
CoreVideo	A C-level framework for managing videos frame by frame in the application.
EventKit	A gateway to the device's calendar events and reminders.
EventKitUI	A framework providing the necessary customizable UI elements for accessing the calendar.
iAd	The tools necessary to add ads to an application and earn revenue through them.
ImageIO	Reading and writing most image formats, also includes access to image metadata and color management.
QuickLook	Adds the possibility to preview files in formats which the application doesn't support.

As is visible from this table there were 11 new tool kits added to the iOS SDK, 4 of which were based on C, a clear sign of Apple moving away from C and more towards creating Objective-C wrappers around the existing functionality. But it is worth mentioning that iOS 4 was the biggest major release from Apple to its mobile software so far: every framework besides External Accessory, Mobile Core Services and Store Kit received some updates and new tools.

1.1.3.1 iOS 4.1

The next smaller release of the iOS SDK saw a lot of bug fixes and some new features such as moving an event from one calendar to another, adding support for peer-to-peer apps, TV show rentals for the iTunes US store and so forth. The following libraries were further polished: Accelerate, Address Book, Assets Library, AV Foundation, Core Data, Core Text, Game Kit, iAd, Security, System Configuration and UI Kit. Unlike the next minor release, there were no new frameworks added.

1.1.3.2 iOS 4.2

This version of the operating system SDK introduced one new framework [Table 6] and promised a longer battery life, added a text search to web pages, some new fonts and allowed .ics files to be imported to the calendar. This was also the first version with the possibility of playing music and print using a wireless network, features possible thanks to AirPlay and AirPrint. Also added was the free use of the Find My iPhone location service, which was meant for people who's device got stolen, so that they could lock it and initiate a single-sided communication with whoever finds it.

Table 6. Frameworks added with the release of the iOS 4.2 SDK [15].

Framework	Summary of Contents
CoreMIDI	A framework for communicating with musical hardware devices through the dock or a network.

1.1.3.3 iOS 4.3

On March 9 in 2011 the next minor version of the iOS was released, among other things it featured video playing through AirPlay, improved Safari performance, the personal WiFi Hotspot for the iPhone 4 and added a full screen iAd banner format. The affected frameworks were Audio Toolbox, Audio Unit, AV Foundation, Core Audio, Core Foundation, Core Media, Core Telephony, Core Text, Core Video, Foundation, iAd, Image IO, Media Player and UI Kit.

1.1.4 iOS 5.0

On October 11 Apple released their new iPhone 4S mobile phone alongside the new version of their mobile operating system, the iOS 5. It witnessed a big set of new

features, such as Siri, the voice activated intelligent personal assistant feature [27]; the Notification Center; a free messaging system called iMessages; the Newsstand special folder, which was to became the center point of all periodicals and the Reminders application, which supplies the adding of tasks and alerts. The social aspect of iOS was also greatly improved with Twitter integration. Also the camera app could now be easily accessed from the lock screen. But probably the biggest new feature was iCloud, Apple's own built-in cloud syncing service for users to keep tabs on their music, images, pictures and more. The new operating system also allowed for wireless syncing with iTunes. These new features prompted for 7 new frameworks in the iOS SDK [Table 7].

Table 7. Frameworks added with the release of iOS 5 SDK [15].

Framework	Summary of Contents
Accounts	Gives access to the user's accounts stored in the device so that the developer would not have to be responsible for storing account login credentials.
CoreBluetooth	A framework that provides access to devices that are connected through Bluetooth.
CoreImage	Advanced features for image processing.
GLKit	A helper framework meant for creating shader-based apps.
GSS	A standard set of services dealing with security.
NewsstandKit	A framework responsible for the client side of a Newsstand application.
Twitter	A framework that sends Twitter requests on the applications behalf and also manages authentication.

Since iOS 5 was a relatively notable release, very few of the existing frameworks in the SDK were left untouched, the only ones being Core MIDI, Core Telephony, External Accessory, Mobile Core Services and System Configuration.

1.1.4.1 iOS 5.1

While iOS 5 saw many new features and frameworks, its minor successor only saw a minimal amount of existing libraries being changed: Assets Library, Audio Toolbox, Audio Unit, AV Foundation, Core Audio, Core Foundation, Core Video, Foundation and UI Kit; and these were introduced to just one or two new constants and methods each.

1.1.5 iOS 6.0

Three months before the unveiling of the iPhone 5, Apple gave a preview of the iOS 6 by releasing it and its corresponding SDK to registered developers. The sixth major version if iOS introduced users to the long awaited Facebook integration; the Passbook app that manages a users boarding passes, tickets, coupons etc.; also the Photos app was given a social feature, which allowed sharing photo streams; the Maps app started using Apple's own technologies instead of Google's; FaceTime was given the freedom of making calls over a cellular network; the App Store, Weather and iTunes' UI got polished and so on. Four new frameworks were inserted [Table 8]:

Table 8. The frameworks added with the release of the iOS 6 SDK [15].

Framework	Summary of Contents
AdSupport	This framework provides applications with an ID for serving advertisements.
MediaToolbox	Contains the interfaces for playing audio content.
PassKit	Provides access to the user's pass library.
Social	This framework grants the user access and a view for posting to social media without the developer worrying about creating the necessary requests.

One of the biggest changes prompted by the numerous new social features were the security and privacy settings. Until then the only thing configurable under the Privacy section of the Settings app was Location Services, but in iOS 6 that section grew with Photos, Contacts, Calendars, Reminders, Bluetooth sharing, Twitter, Facebook and Sina Weibo (a Chinese equivalent of a hybrid of Facebook and Twitter). This version of the SDK was followed by probably the smallest update to the tool kit in the iOS 6.1.

1.1.5.1 iOS 6.1

This version of the Apple operating system, released on the 28 of January 2013, was relatively unremarkable. There were no new added frameworks and the only current frameworks updated were the UI Kit and Map Kit – the latter being pressured to be updated by the general public's opinion that the Maps app was severely flawed and of inferior quality to its previous version that used Google's map engine.

1.1.6 iOS 7.0

This version of the iOS was seen as the biggest change to the operating system since it first came out, mostly thanks to the total revamping of the user interface. But the design was not the only thing that changed; it also introduced a lot of new functions. Alongside a lot of utilitarian features such as the Control Center and AirDrop, there were also updates in the Camera, Multitasking, Music and many other apps' user interfaces. As for the SDK, the number of added frameworks remained relatively modest with 6 [Table 9].

Table 9. The frameworks added with the release of the iOS 7 SDK [15].

Framework	Summary of Contents
GameController	A framework used to receive inputs from game controllers form inside a game.
JavaScriptCore	A framework used to evaluate JavaScript programs from an Objective-C app.
MediaAccessibilty	A framework with functions to access the users preferences on captioning.
MultiPeerConnectivity	Support for finding and communicating with services provided by other iOS devices nearby via WiFi or Bluetooth.
SafariServices	Support for adding URL's to Safari's Reading List and other services.
SpriteKit	The rendering and animations tool kit that can be used to animate textured images.

It is visible from this list that the newly appended frameworks had very little to do with the vastness of visual updates in this iOS version.

1.1.6.1 iOS 7.1

The currently latest version update of the SDK was not very big, but notable in the fact that it was surrounded by a rather vocal media attention: a noticeable security flaw was found in the mobile operating system that made the software vulnerable to main-in-the-middle attacks [16]. There were no new frameworks added to the SDK, but just small changes to AVFoundation, CoreBluetooth, CoreMedia, iAd, OpenGLES, StoreKit and SpriteKit, which had the infamous bug fix; and some more substancial updates to MapKit and MediaPlayer.

1.2 Conclusions

The operating system currently known as iOS and its development kit both have had seven major versions with numerous smaller updates. The latest bigger version was released on September 18 2013. During almost seven years of continuous development, Apple has created a total of 55 frameworks for the developers to use as they please, although they do attach some restrictions by setting rules, which every app has to comply with when it goes through a verification process before being allowed to the App Store. The total list of currently available iOS SDK frameworks can be seen in Appendix 1.

In the beginning of the iOS SDK's lifetime it was a relatively low-level C package with some initial rather basic Objective-C introductions. One of the main emphases of the SDK development has so far been on trying to escalate the coding level to a more iOS-specific Objective-C level, a feat that would incrementally lessen the role of C in application development. The reason for this is that Objective-C is a higher-level programming language than C, which in software development usually means that it is easier to obtain and use, albeit with some performance drawbacks. This has most likely been imperative to make the learning and coding of applications simpler to developers, which in turn would make it more popular and help increase Apple's revenues.

Another aspect visible from these observations is that the number of added frameworks is declining almost with every major update. This signals the common notion of maturing, which means further improvements are more detailed, specific and rather performance-oriented.

Before continuing with further investigative work with these frameworks it seems appropriate to find the most applicable methods on doing so, considering the objective and preferred outcomes. This will be the subject of the following section of this thesis.

2 An overview of related and similar works

There are several aspects to the field of software development frameworks and component-based software engineering; some of these areas and subjects have been covered by earlier literature. A key problem it seems is the methodologies and criteria on how to choose the most efficient frameworks to an existing software system.

The paper “Maintenance-oriented selection of software components” written by P. Ardimento, A. Bianchi, G. Visaggio from the University of Bari [28], stresses the need for a systematic strategy for choosing components when building new software systems. Since the system itself and the components it comprises of can take different routes in evolving, maintenance becomes the key concern when adopting third party frameworks to your software. Also stressed is the competence of the developers on knowing the system as a whole, not just the parts that come into contact with the integrated components.

They execute the study by using four principle characteristics: adequateness of the component with respect to the target system, the costs associated with the implementation and usage of the component, the familiarity of the team towards the new software and the level of support provided by the component’s provider. Since this is a very close subject to the one of this thesis, there are elements that can be learned from, such as the detailed dissection and description of the basis that was used to select the frameworks to be analyzed.

A similar topic of study was covered in “A Framework for Systematic Evaluation of Software Technologies” by A. W. Brown and K. C. Wallnau from the Carnegie Mellon University [29]. These two works overlap in the solution area that tries to find a best practice for choosing software from a financial and pragmatic point of view. The part where they start to differ is that the former deals with the problems of integrating software to existing systems, the latter has to deal with integrating a complete software solution to an existing eco-system of employees and technologies. The authors of this study introduce a solution called the Feature Delta Framework that tries to find a solution by emphasizing the differentiating features of each proposed software.

The main outline of this method would be a good example of a way to measure how a third party framework is different from its closest counterpart from an iOS SDK framework it is trying to enhance.

The adoption of new technologies from an economical and managerial point of view is also further covered in a study by P. M. Herceg called “Defining Useful Technology Evaluations” [30]. Since the aim of this thesis is directed more towards the technological aspects of decision-making, the administrative facets of these papers serve less meaning than the aspects of component integration.

When discussing the more specific technological methods on determining the right frameworks for a solution domain, several different approaches can be scrutinized. An example of a process of choosing iOS apps for a detailed analysis can be found in an article called “Status and trends of mobile-health applications for iOS devices: A developer’s perspective” written by C. Liu, Q. Zhu, K. A. Holroyd and E. K. Seng from the Ohio University [31]. Thanks to the large number of available apps in their area of study and the accessibility of different user-based statistics, they were able to use very simple but convincing criteria for filtering out just the right applications to be used in their further work. This is an approach with characteristics that could be implemented in the thesis at hand.

When the phase of choosing the right frameworks from an extensive pool is completed, the next phase will have to deal with evaluating the remaining libraries in a way to find out the specific areas these very frameworks were built to enhance. The methods for doing this can also be inspired from earlier works. Such as the “Development Frameworks for Mobile/Wireless User Interfaces: A Comparative Case Study” by Simon Pestina from the Concordia University [32]. It conducted a comparative survey of mobile web development frameworks using subject-relevant criteria such as data interactivity, the cross-platform or device specific availability, scalability, extensibility, learnability, etc. It continues by showing the strengths and weaknesses of each software library and pointing out the fields for potential future improvements. A list of questions are highlighted in the beginning to which the author tries to answer by analyzing each of the frameworks separately according to the aforementioned criteria and then comparing them to each other in a later, conclusions chapter of the paper. The advantage of such a

method is the impartiality reflected towards each object under assessment, a mentality that can also be used in the current thesis. A comparison between the examined frameworks can be done via more generic and abstract measures afterwards.

In conclusion, the field of integrated software libraries and related problems is not a very studied one, but in a relatively strong need of being so. The process of finding the best new technology to use is still a problematic one; there are no standards for it -- most of these situations have been addressed with an ad hoc solution and it still heavily relies on subjective experience-based opinions and expert advice. This paper will deal with choosing a selection of third party iOS frameworks that will be analyzed and compared to their iOS SDK counterparts to find the most deficient areas of the SDK. This will be a procedure that will somewhat rely on the strategies found in these previous related works. The process itself and its results will be thoroughly explained in the following paragraph, which will explain how the most popular third party frameworks were found.

3 External frameworks selection

This chapter will focus on the methods used to select the open source public external third party frameworks written in Objective-C and used for iOS development. Filtering out the non-essentials and less popular ones will leave the libraries that will then be used in a successive analysis.

3.1 Selection and evaluation

The evaluation process for selecting frameworks for further analysis will comprise of three parts. In the end of the first phase there will be a list of the selected most popular external frameworks currently available. The second phase will determine the libraries most appropriate for further analysis and the third part will consist of a public survey among iOS developers to find out their experience-based opinions on what are the frameworks most frequently used by themselves.

3.1.1 Initial listing

The objective of the first stage of the selection process was to find the most popular frameworks currently available. In order to do this, some statistics were necessary and it was the opinion of the author that one of the best places to find such numbers is to use an almost industry-standard portal that is used most widely for downloading public third party frameworks: GitHub. It contains a feature called *Star*, which GitHub themselves describe as following: “Starring a repository allows you to keep track of projects that you find interesting, even if you aren't associated with the project” [33].

Every user can star every GitHub project exactly once and a project can also be *un-starred*. Thanks to this feature GitHub can sort projects by popularity according to their *Star* count. In order to do this a custom query string has to be written in the GitHub search field [34]. The strategy of using GitHub and its *Star* feature carries several positive qualities: it is always up to date, it is based on user-created statistics and it allows for the same criteria for describing each available framework. For reasons explained in the next section, 84 of the most starred libraries were initially chosen.

3.1.2 Filtering

The 84 most starred frameworks chosen from GitHub were given a short summary of their contents and a type, which would describe the field of expertise. The available types were as follows:

- "Network" – includes all the libraries that dealt with network connectivity, data transfer and different format parsing;
- "View" – libraries that were related to views, view hierarchies and view animations;
- "Utilities" – frameworks that consisted of helper methods/classes, frequently used boilerplate code and modular components, collections of small enhancements etc.;
- "Graphics" – everything to do with low-level graphics and drawing: plots, 2D and 3D physics, visualizations, games;
- "Data Store" – libraries that dealt with Core Data or other storing, fetching and sorting of data in general;
- "Debug" – tools that help with debugging in the development process;
- "Testing" – frameworks used for writing unit tests for apps in development;
- "Non-framework" – frameworks that are no longer useful or projects that were not frameworks for iOS development or frameworks at all (iOS apps, Mac OS apps, other utilities).

The number of initial frameworks to be chosen stemmed from the number of categories listed here. Excluding the "Non-framework" type, there are seven groups, so to give each group an equal chance to accumulate the same number of popular frameworks it made sense to select at least 70 frameworks. A later revelation made it clear that some additional libraries would have to be tallied in order to give each of the planned framework types a sufficient number of frameworks as options to be included in the questionnaire.

The goal was to have at least three frameworks in each category, but since this became a problem with some types, an extra measure was needed. In the "iOS Frameworks" web page there is a catalogue of almost seventy iOS frameworks available [35].

After categorizing all of these with the above labels, it was possible to collect at least the required three frameworks under each type.

The next step for further segmentation was to give each of these frameworks an estimation that would characterize their difficulty and size. One of three evaluations was assigned:

- "Easy" – small sized frameworks, low complexity; enhancements to one class or object; templates;
- "Medium" – medium sized; packaging functionalities from different development areas, while not covering a whole subject area; full functionality of a small area;
- "Difficult" - frameworks with a large size, great complexity or covering a whole subject field.

It was decided that only libraries assigned with the "Medium" and "Difficult" label were going to be used for further comparisons. The final list of included libraries with their corresponding name, star count, type, evaluation and brief summary can be found in Appendix 2.

Some of these GitHub projects had to be excluded from subsequent consideration because they were redundant, deprecated, no longer supported, because they were not frameworks or they were built for the Mac OS X system – these were all labeled "Non-frameworks". The reason for the presence of Mac OS X libraries was that GitHub differentiates projects by programming language and Objective-C is a language designed for the development of both iOS and Mac OS X.

3.2 Conducting a survey

A survey among iOS programmers was conducted in order to draw comparisons between statistically popular frameworks and the ones that developers themselves empirically consider the most useful and why. The objective of this questionnaire was to get more details about the most popular frameworks previously selected with the help of the GitHub *Stars* feature.

The introductory questions were targeted at getting some background of the responders. The easiest and most anonymous way to comply with the goal was to ask about the length of their tenure as a software developer in general as well as an iOS developer.

The main content of the inquiry was meant to get some insight on why developers use specific frameworks. To achieve this, every previously mentioned framework type was examined separately. A list of frameworks was offered for each type for the respondent to choose from, marking the ones he or she had used. Each of these questions had an "Other" option, offering the respondent the possibility to add unlisted libraries. Each of these questions was followed by an open-ended inquiry asking them to explain in more detail the reasons for using the frameworks they did.

The closing questions of the survey were also open-ended, targeting the iOS SDK in general. These questions were added to get insight on the developers' subjective opinions of the iOS SDK, with the prospect of adding context to their previous answers.

The full list of questions can be found in Appendix 2.

3.2.1 Results

Given the very specific segment of people who are qualified to answer this questionnaire it was relatively complicated to find a sufficient number of respondents. Looking for developers with at least some experience in developing with the iOS SDK was made difficult by several factors. Firstly, the iOS developer community in Estonia is not as coherent and active as the international ones, which makes finding the developers somewhat difficult. Secondly, although the iOS developer society is very large on the world wide scale, finding people who would comply with helping was made more challenging by the fact that posting this sort of content in corresponding forums and other sites is rather hastily regarded as spam and thereafter deleted or moved to an according segment of said web page.

After exhausting the author's personal acquaintance circle, the number of replies was only 11. The next round of answers was compiled by posting to thematic forums like the "iPhone Dev Forums" (www.iphonedevforums.com) and other sites such as "reddit" (www.reddit.com). This raised the number of respondents to 14.

It was clear by this time that the expectations on the number of replies should not exceed 20. Finding developers from stackoverflow (www.stackoverflow.com) and writing them personally produced 8 of the last responses.

While a considerable number of replies were informative and helpful, there were also entries that only contained answers to the optional and none of the open-ended questions. Nevertheless, they revealed some concrete insights.

Out of the 20 people who answered the survey, 9 had been software developers for more than 6 years, 10 had been doing it for 3 to 5 years and one for only 1 to 2 years. This then made for a rather experienced set of developers, but only one of the respondents had been developing for iOS for more than 6 years, 12 of them for 3 to 5 years and 7 had been doing it for 1 to 2 years. The latter numbers are being backed by the fact that the iOS development environment has been available for only about 6 years now. But since Objective-C can also be used to program for the Mac OS X operating system, its possible that people considered these two things interchangeable.

The answers to the open-ended questions were mostly positive, praising Apple's good documentation, its responding to developers' feedback, the iOS SDK's unified patterns, the general design and implementation maintenance. Inevitably some of the answers surfaced comparisons to the Android SDK, with all of the mentions leaning more towards the iOS SDK, citing a more coherent and polished API. Also mentioned were the tools used with the SDK, such as the iOS Simulator, which avoids building on the ARM architecture, a fact that in turn makes the process faster. The Xcode IDE is said to be more pleasant and has more built-in development tools than the ones in any other platform's development domain. People also seem to feel that the iOS development community is very large and usually helpful.

One of the neutral notions repeatedly brought out was the closed source mentality and the big number of black box components. The negative aspects included the lack of unit and beta testing capabilities, bug reporting options, some documentation errors and the somewhat complex nature of delegating and protocols. The full list of questions and answers can be seen in Appendix 3.

The most relevant part of the survey was of course the questions targeted at the third party frameworks usage. Out of the 7 categories listed, the most mentioned one was "Networking"; it was indicated on 17 responses out of 20. The "Data Store" and "View" with each of them being noted 12 and 10 respectively. The least noted group turned out to be "Debug", with only 3 people acknowledging it. The full distribution is visible in figure 1.

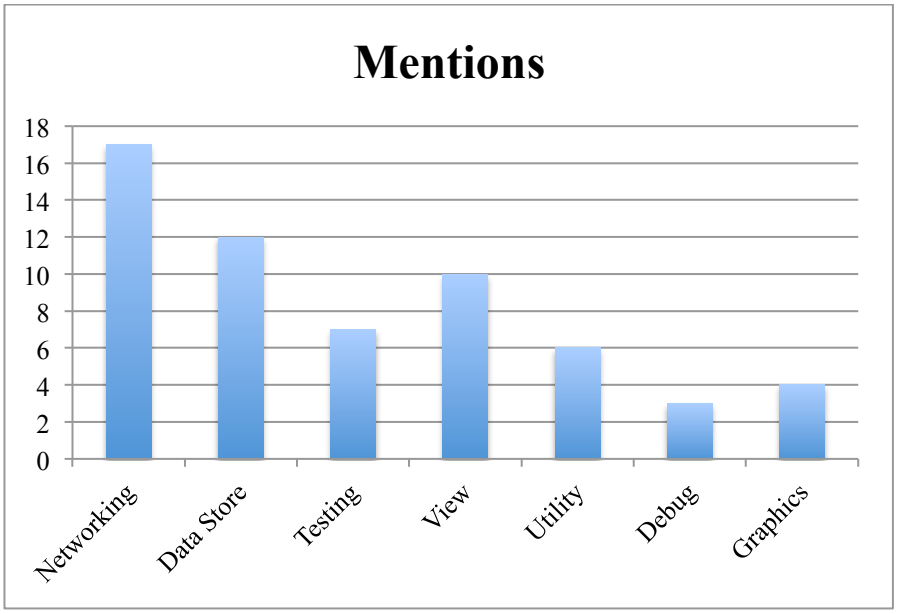

Figure 1. The number of times each category was mentioned in the conducted survey's 20 responses.

Since the next step of this thesis is going to be the deeper analysis of the most widely used third party frameworks, it was decided that the two most popular framework types would be selected to provide the frameworks. But before the next step, a short review of reasons for using some of the other libraries will be listed.

The testing framework usage was mainly explained with the reasoning that Apple themselves have yet not provided any good testing frameworks so far. The most popular libraries were GHUnit and Kiwi, with the latter repeatedly being said to have a more readable syntax than any of Apple's own testing libraries.

The most-used view-related frameworks seem to be the ones that implement the side menu functionality, a feature visible in many mainstream applications such as Facebook, Foursquare, YouTube, etc.

For example the ECSlidingViewController framework, which can be seen here: https://github.com/ECSlidingViewController/ECSlidingViewController. It is a good example of how the iOS development community has adopted a feature to a point where it can almost be seen as an industry standard solution. And since Apple themselves have not yet offered a solution, a large number of developers produce a solution themselves. Another good example of such behavior would be the "pull to refresh" feature, often seen on top of a table view functionality. This feature first grew popular among developers who very often built their own custom software to implement it, then a large number of libraries were being made available in open source communities such as GitHub and finally Apple made their own version of this component available [36]. This also supports the opinion that Apple listen and react to their developers' feedback. Other popular view libraries included DTCoreText [37], which allows editing the appearance of strings in the user interface and iCarousel [38] – a library offering a different way to present views in a scrollable way.

The most frequently mentioned utility framework by far was Appirater, accumulating 4 out of the 6 total times anyone admitted to using a utility-related library. It is used to remind users to go and rate the app in the App Store with an alert [39]. In the "Debug" section only 3 different frameworks were mentioned by the respondents altogether and all of them were related to crash reporting. Since they were all different, any single one will not be highlighted here.

Among the graphics libraries brought up were GPUImage, Box2D, and Chipmunk2D. The first of these works with image and video processing, citing faster handling times compared to Core Image. The two other frameworks are used for drawing 2D graphics, mostly utilized in game developments.

As for the selection of the most popular frameworks from the most popular categories, two libraries stood out the most: AFNetworking from the "Networking" and MagicalRecord from the "Data Store" category. The "Networking" type was noted 18 times and AFNetworking in turn was mentioned in 16 out of these 18 times. The "Data Store" type was acknowledged 12 times, 6 of which had mentions of MagicalRecord. These two then seem to provide a legitimate and sufficient foundation for further analysis – a process described in the next paragraph.

4 External frameworks analysis

This part of the thesis will focus on a deeper analysis of the most popular libraries that emerged from the preparative process described in the previous paragraph. Two libraries were chosen for this for three reasons. Firstly, choosing three or more seemed redundant; secondly, having two options seemed to provide more possibilities of comparison than one; and thirdly, there were coincidentally exactly two frameworks from two different categories that stood out as the most prevalent: AFNetworking and MagicalRecord. Looking at the initial list made up of the most *Starred* open source codes from GitHub, AFNetworking held the topmost position with 10375 *Stars* and with MagicalRecord not far along, boasting 4047 *Stars* (as of the 19th of January 2014).

4.1 AFNetworking - usage reasoning and comparisons with the iOS SDK

As previously mentioned and obvious from the name, AFNetworking was listed under the "Networking" category. Its GitHub page summarizes the framework as follows: "AFNetworking is a delightful networking library for iOS and Mac OS X. It's built on top of the Foundation URL Loading System, extending the powerful high-level networking abstractions built into Cocoa. It has a modular architecture with well-designed, feature-rich APIs that are a joy to use" [40].

It was visible from the survey that among reasons why iOS developers like to use this library were the facts that it provides a simple interface for making requests; heavy usage of blocks instead of delegation methods; having powerful serialization capabilities, which come in handy with RESTful services; and the overall ease of use. Analyzing these arguments in detail is the objective of the next sub-sections.

4.1.1 Simple interface

The interface of the AFNetworking library consists of four main types of classes: the ones implementing NSURLConnection-related functionality, NSURLSession-related functionality, classes regarding serialization and additional components.

The classed related to NSURLConnection contain three different objects: AFURLConnectionOperation, AFHTTPRequestOperation and AFHTTPRequest-OperationManager. The latter is the one that is considered to be the starting point for a programmer; it "encapsulates the common patterns of communicating with a web application over HTTP, including request creation, response serialization, network reachability monitoring, and security, as well as request operation management."[40] The operation manager object is a singleton: a shared single instance is used throughout the application's lifetime. The singleton mechanism has seen a rise in overall usage in recent years, owing most of its thanks to the simplifying effect it has on code quality and usage. Having a single object of a class is good for several reasons:

- an instance of it can be retrieved from anywhere in the code at any time without passing it along from object to object or initiating a totally new one,
- it makes is easy to create or fetch it and use its properties without having to worry about their thread safety – meaning that you do not have to think about decoupling this object,
- it maintains its state throughout the applications lifetime.

The only negative side to a singleton object is its memory usage -- it might take a bit more of it than creating and releasing a new class instance at will and need.

AFHTTPRequestOperation objects are created with every request, with the success and failure events of these operations defined in the method's parameters as blocks. AFURLConnectionOperation is a subclass of NSOperation that implements NSURLConnection delegate methods, used mainly for callbacks that give information about the status of the URL connection. [40]

The request serialization classes are used for creating requests from URL strings; also serializing JSON formatted files, property lists, HTTP bodies, XML strings and images.

The additional functionality classes include AFSecurityPolicy and AFNetwork-ReachabilityManager. The first is used for establishing server trust against man-in-the-middle attacks and the latter for evaluating the reachability of IP addresses.

The latest update to the AFNetworking library added the session classes AFURLSessionManager and AFHTTPSessionManager. These create and operate an NSURLSession object, which are used for downloading content through HTTP. Its API provides many delegate methods to observe and control the content transfer [41].

In contrast, Apple have created a number of libraries related to URL loading in the Foundation framework, which include classes that cover the same area of functionality as the AFNetworking does. The "NS" prefix in Apple's class names has a historic reason: it is an acronym that stems from the company name "NeXTSTEP", which developed the Objective-C language, later purchased by Apple. These can be considered as equivalents:

- NSURL,
- NSURLAuthenticationChallenge,
- NSURLCache,
- NSURLConnection,
- NSURLCredential,
- NSURLCredentialStorage,
- NSURLProtectionSpace,
- NSURLProtocol,
- NSURLRequest,
- NSURLResponse,
- NSURLSession,
- NSURLSessionConfiguration,
- NSURLSessionDataTask,
- NSURLSessionDownloadTask,
- NSURLSessionTask,
- NSURLSessionUploadTask.

NSURLCredentialStorage is the only one that cannot be found in the AFNetworking framework. This list comprises of a total of 16 classes with an additional 10 protocols that define some delegation methods that programmers can implement either optionally or by being required to do so. It must be noted that some of these classes are in themselves and provide for some extra functionality compared to AFNetworking. AFNetworking uses these classes in its architecture – the reason behind it being said

that AFNetworking was built on top of the Foundation URL Loading System. In order to clarify the simple interface argument an example will follow.

4.1.1.1 Example

To make it more obvious how AFNetworking has a simple programming interface, an example of a simple URL request will be outlined. In order for every kind of reader to better understand the delegation mechanism, it will be explained in short. A URL request can be started to fetch data if one needs some from a web service. Delegation is the method used to retrieve responses form URL requests. First you set an object as a "delegate" to a URL request object and then later some delegation methods (which are defined in delegate protocols and implemented by the programmer) get called to notify you of the success or failure and contents of the answer from the web server. The same goal can be achieved using blocks, a feature that it will be examined later in this thesis.

iOS SDK

This iOS example is largely based on and comments are copied from Apple's own sample project "AdvancedURLConnections", available from the NSURLRequest class reference web page [42]. Example 1 shows the creation of a URL request, starting a connection and setting the delegate.

Example 1. The URL request initialization

```
NSURL *url = [NSURL URLWithString:@"http://example.com"];
NSURLRequest *request = [NSURLRequest requestWithURL:url];
self.connection = [NSURLConnection connectionWithRequest:request
                                                 delegate:self];
```

This example illustrates a very common way of creating, altering and using objects in the Objective-C language: first it is created using the NSURL class method *URLWithString:* while passing the web site URL as the only parameter (class methods are the ones that can be called to a class in general, while instance methods can be called to a single object of a class). The NSURL object is then used to create an NSURLRequest object, which is needed to start a connection to the aforementioned

URL. This is done using an NSURLConnection object, while setting self as the delegate callback listener. We can see how the delegation methods are used in example 2.

Example 2. Handling URL request responses. The iOS SDK has defined 4 delegate methods for separate purposes.

```
- (void)connection:(NSURLConnection *)conn
didReceiveResponse:(NSURLResponse *)response
{
  NSHTTPURLResponse *httpResponse = (NSHTTPURLResponse *)response;
  [self handleURLRequestResponse:httpResponse];
}


- (void)connection:(NSURLConnection *)conn didReceiveData:(NSData
*)data
{
  [self handleReceivedData:data];
}


- (void)connectionDidFinishLoading:(NSURLConnection *)conn
{
  [self handleFinishedConnection:conn];
}


- (void)connection:(NSURLConnection *)conn didFailWithError:(NSError
*)error
{
  [self handleURLRequestError:error];
}
```

The first three methods are used to observe the data exchange and the last one is called at most once to notify the listener of a failed request loading. The difference between the first two methods is that while the first one is called once the connection has delivered a sufficient amount of data to build an NSURLResponse object, the second one is called repeatedly as the response information is incrementally being loaded. The third callback is executed when the connection has finished and no further data exchange will occur.

AFNetworking

It is visible from example 3 how the same thing with the same parameters would have to be implemented using the AFNetworking framework by starting a connection and handling the URL request responses. AFNetworking has a single method with two callback blocks as parameters for this purpose.

Example 3. The URL request initialization and handling responses.

```
AFHTTPRequestOperationManager *manager =
[AFHTTPRequestOperationManager manager];
  [manager GET:@"http://example.com"
    parameters:nil
       success:^(AFHTTPRequestOperation *operation, id responseObject)
   {
     [self handleReceivedObject:responseObject];
   }
       failure:^(AFHTTPRequestOperation *operation, NSError *error)
   {
     [self handleURLRequestError:error];
   }];
```

This is a good example of why block callbacks are more efficient than delegate callbacks: instead of handling the URL request response in a different method or even a different class, it can be done in the same place where the request was initiated in, alongside the same object instances or variables surrounding that object in the scope of the initialization.

It is fairly obvious why respondents from the conducted survey considered the AFNetworking syntax simpler than that of the iOS SDK. Not only is the total number of code lines written considerably smaller, but it also does it through a singleton object that encapsulates all the initializations, parameter handlings and callbacks. This being a rather simple example, the differences between the two frameworks' implementations are not as distinct as they would be when dealing with more complex and error prone operations. But then again, one must consider that, like with any simplified tool, flexibility is not its strongest suite. It might happen that when a situation calls for more complex solutions, Apple's URL loading classes can be the more efficient solution.

4.1.2 Blocks

The heavy usage of blocks, also known as callbacks, is considered a convincing reason for preferring the AFNetworking framework to the Foundation's networking classes. There are several reasons why blocks are more economical than delegation, some of which were mentioned earlier as well.

The first advantage would be the obvious fact that one has to write fewer lines of code in order to implement the desired functionality, this aspect is clear from examples 2 and 3. In contrast, when using delegation, the programmer must follow these steps:

1. find the respective delegate protocol,
2. look up and copy the methods he or she wants to use,
3. declare the class in which the request will be used as conforming to the aforementioned protocol,
4. use the delegate methods in a way deemed necessary by the developer.

A total of 4 tedious steps, while using blocks would only require the last step.

4.1.2.1 Example

Another leverage blocks have over delegation is the fact that a variable scope extends inside the contents of the blocks. An example from Apple's own iOS SDK will be laid out to illustrate this point.

Example 4. A method with a callback block from the iOS SDK.

```
int i = 5;
NSNumber *five = [NSNumber numberWithInt:i];
NSDate *today = [NSDate date];

UIViewController *viewController = [[UIViewController alloc] init];
[self presentViewController:viewController animated:YES completion:^
{
  [self udpateViewForCount:five date:today];
}];
```

The method used in example 4 presents a modal view controller and gives the programmer an option to implement some code after the presentation has finished performing its animation. The three lines prior to the presentation are variables that can be used in the block code. If the same functionality were written using delegation methods, these three variables would have to be referenced and later received through instance variables or some other practice in order to be used in the methods found in the protocol. This feature makes coding more efficient for the writer and more readable for the reader.

The final considerable convenience is the absence of plurality. For example, when initializing several URL requests their responses would somehow have to be

differentiated from each other in the delegate callback methods. A problem that cannot occur when using blocks since every request has its own response callback.

Next to these highlighted advantages there are also some things to keep in mind with blocks. One of the things developers have to think about is avoiding retain cycles. A retain cycle is a memory management problem, it occurs when two objects, each with their own allocated memory spot, have a strong reference to each other and this can cause issues in a situation when one of these would have to be released in order to free some memory from under it. An object can be killed and removed from memory and its slot de-allocated only when that object no longer has any strong references to it. Blocks always create a strong reference to every object passed into them, and in a case where an object has a strong reference to a block, neither of them can ever be released because of the two-way strong reference. This problem can seamlessly occur when a block is declared as a property to a class instance. In order to avoid this conundrum, objects passed on to a block should be marked with a special keyword: "__weak", "__unsafe_unretained" or "__block". All of these have specific use cases and implications, which are out of the scope of this thesis. It is enough to know that using any of these is better than nothing in a straightforward block usage. A correct block property usage will be demonstrated in example 5.

Example 5. The correct way to avoid retain cycles.

```
__weak ExercisePlayerView *weakSelf = self;
self.timerFinishBlock = ^{
 [weakSelf setupViewForFinishedTimer];
};
```

While this was a simple solution to a specific situation, there are a lot of nuances to using blocks dependent on the application being developed, but all in all blocks do call for some precaution.

4.1.3 Powerful serialization capabilities

Serialization in software development in general refers to a restructuring of data from one format to another, a necessity in some cases where data is transported between technically different domains. As mentioned earlier, the serialization classes in AFNetworking help with creating URL requests from different available formats. These

are URL query strings, URL Form Parameter Encoding and JSON encoding. This field of functionality is also available in the iOS SDK, but the methods of usage do vary to some extent.

Starting from iOS 5, Apple added the NSJSONSerialization class to the Foundation framework. As the name suggests, it creates Foundation objects (an NSString, NSNumber, NSArray, NSDictionary, or NSNull) from JSON data and vice versa.

4.1.3.1 Example

A comparison example will be shown between the usages of the iOS SDK NSJSONSerialization and the AFJSONRequestSerializer. The contents of the illustration will be a POST request to a web service; the body of the request will come from a Foundation NSDictionary object.

iOS SDK Foundation

It is visible from this case that a single NSJSONSerialization class method does all the serializing, which creates a JSON-format object that can be appended to the request body. The rest of the code is the same as in example 1.

Example 6. Creating a URL request with the NSJSONSerialition class.

```
NSDictionary *postDictionary = [NSDictionary
                    dictionaryWithObject:@"value1" forKey:@"key1"];
NSError *error = nil;
NSData *jsonData = [NSJSONSerialization
              dataWithJSONObject:postDictionary
                         options:NSJSONReadingMutableContainers
                           error:&error];
NSMutableURLRequest *request = [NSMutableURLRequest
requestWithURL:[NSURL URLWithString:@"http://example.com"]];
[request setHTTPBody:jsonData];
self.connection = [NSURLConnection connectionWithRequest:request
                                                 delegate:self];
```

The NSJSONSerialization class method *dataWithJSONObject:options:error:* is used to create data from a dictionary, which in turn is later used to create a URL request and start a connection. This is the same process as seen in example 1 with the exception that this time there is some extra data sent along with the request.

AFNetworking

As it will be seen in the next example, the interface of the operation in AFNetworking is quite efficient: the AFHTTPRequestOperationManager object returns a mutable request with a single method call and therefore does not have to be initialized separately.

Example 7. Creating a URL request with an AFHTTPRequestOperationManager object.

```
NSDictionary *postDictionary = [NSDictionary
dictionaryWithObject:@"someValue" forKey:@"someKey"];
NSError *error = nil;
NSMutableURLRequest *request = [[AFJSONRequestSerializer serializer]
                            requestWithMethod:@"POST"
                                    URLString:@http://example.com
                                   parameters:postDictionary
                                        error:&error];
AFHTTPRequestOperationManager *manager =
                                [AFHTTPRequestOperationManager manager];
  [manager HTTPRequestOperationWithRequest:request
                                   success:^(AFHTTPRequestOperation
*operation, id responseObject)
   {
     [self handlePOSTResponseOperation:operation];
   }
                                   failure:^(AFHTTPRequestOperation
*operation, NSError *error)
   {
     [self handlePOSTResponseOperation:operation error:error];
   }];
```

It was mentioned in the survey that an added value of the AFNetworking framework was its powerful serialization capabilities, an aspect that does not necessarily mean that it is in any way better than Apple's own, but rather just good added value to have bundled with the rest of its functionality.

Three of the AFNetworking framework's most relevant advantages over the iOS SDK's URL loading library have been dissected. Looking at the high popularity of this framework on GitHub, it can be understood that these are enough for a great deal of people to prefer this library to Apple's solutions. The next section of this thesis will focus on the second framework that surfaced in the conducted survey and its virtues over the corresponding Apple counterparts: MagicalRecord.

4.2 MagicalRecord - usage reasoning and comparisons with the iOS SDK

MagicalRecord is a framework developed to ease the process of using relational databases in iOS programming. Its name derives from the active record pattern that is used in relational database manipulation. The creators of MagicalRecord describe their library as follows: ” MagicalRecord was inspired by the ease of Ruby on Rails' Active Record fetching. The goals of this code are:

- Clean up my Core Data related code
- Allow for clear, simple, one-line fetches
- Still allow the modification of the NSFetchRequest when request optimizations are needed”. [43]

Apple has its own framework for managing databases called Core Data. Although this framework is considered very comprehensive and capable, it is also seen as slightly tedious and demanding of a lot of boilerplate code. Boilerplate code is the kind that calls for a lot of rewriting the same things with very little alternations to it every time you want to implement it. The expression derives from the makers’ labels that were once put on steam boilers.

Just like AFNetworking is built on top of Apple’s URL loading classes, the MagicalRecord framework is constructed on Apple’s Core Data. This has its benefits that will be analyzed in a later section. But unlike the case with AFNetworking, the carried out survey only revealed one broad upside to using MagicalRecord: its great power to simplify the use of database management by removing the need for boilerplate code. Although MagicalRecord uses the Core Data library, they will be treated as separate entities in the context of this paragraph in order to simplify the styling of text.

4.2.1 The absence of boilerplate code

In order to make this point as clear as possible, the gap between the implementation complexities of MagicalRecord and Core Data will be illustrated. An example of a typical database setup and usage will follow using both of these libraries.

4.2.1.1 Example

A typical database manipulation cycle starts with setting up a database, which includes coming up with an appropriate data model with all of its objects, relationships and attributes. Once the data model is up and running, one can then create, change, delete and fetch instances of objects defined in that data model. The simple example to show the process of implementing this with iOS SDK's Core Data framework and MagicalRecord's library in this thesis will be in the context of an application that shows planned purchases in a list.

Core Data

Since the process of creating a data model is mutual for the operation methods of these two libraries, it will be assumed to be existent beforehand. Although the Xcode IDE could automatically generate a large portion of the boilerplate needed to set up Core Data, it is still something that in principal has to be added manually. An initial setup of a database using Core data is shown in example 8.

Example 8. The initial code required for setting up the Core Data implementation.

```
@property (strong, nonatomic) NSManagedObjectContext
*managedObjectContext;
@property (strong, nonatomic) NSManagedObjectModel
*managedObjectModel;
@property (strong, nonatomic) NSPersistentStoreCoordinator
*persistentStoreCoordinator;
@property (strong, nonatomic) NSFetchedResultsController
*fetchedResultsController;

- (void)viewDidLoad
{
  [super viewDidLoad];

  NSError *error;
  if (![[self fetchedResultsController] performFetch:&error])
  {
      NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
  }
}
```

```
- (NSManagedObjectContext *)managedObjectContext
{
  if (_managedObjectContext != nil)
  {
    return _managedObjectContext;
  }

  NSPersistentStoreCoordinator *coordinator = [self
  persistentStoreCoordinator];
  if (coordinator != nil) {
    _managedObjectContext = [[NSManagedObjectContext alloc] init];
    [_managedObjectContext setPersistentStoreCoordinator:coordinator];
  }
  return _managedObjectContext;
}

- (NSManagedObjectModel *)managedObjectModel
{
  if (_managedObjectModel != nil)
  {
    return _managedObjectModel;
  }
  NSURL *modelURL = [[NSBundle mainBundle]
URLForResource:@"ProductList" withExtension:@"momd"];
  _managedObjectModel = [[NSManagedObjectModel alloc]
initWithContentsOfURL:modelURL];
  return _managedObjectModel;
}

- (NSPersistentStoreCoordinator *)persistentStoreCoordinator
{
  if (_persistentStoreCoordinator != nil)
  {
    return _persistentStoreCoordinator;
  }

  NSURL *storeURL = [[self applicationDocumentsDirectory]
URLByAppendingPathComponent:@"ProductList.sqlite"];

  NSError *error = nil;
  _persistentStoreCoordinator = [[NSPersistentStoreCoordinator alloc]
initWithManagedObjectModel:[self managedObjectModel]];
  if (![_persistentStoreCoordinator
addPersistentStoreWithType:NSSQLiteStoreType configuration:nil
URL:storeURL options:nil error:&error]) {
    NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
    abort();
  }

  return _persistentStoreCoordinator;
}
```

```
- (void)saveContext
{
  NSError *error = nil;
  NSManagedObjectContext *managedObjectContext =
self.managedObjectContext;
  if (managedObjectContext != nil)
  {
    if ([managedObjectContext hasChanges] && ![managedObjectContext
save:&error])
    {
      NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
      abort();
    }
  }
}

- (NSURL *)applicationDocumentsDirectory
{
  return [[[NSFileManager defaultManager]
URLsForDirectory:NSDocumentDirectory inDomains:NSUserDomainMask]
lastObject];
}


- (NSFetchedResultsController *)fetchedResultsController
{
  if (_fetchedResultsController != nil)
  {
    return _fetchedResultsController;
  }

  NSFetchRequest *fetchRequest = [[NSFetchRequest alloc] init];
  NSEntityDescription *entity = [NSEntityDescription
entityForName:@"Product"
inManagedObjectContext:self.managedObjectContext];
  [fetchRequest setEntity:entity];

  NSSortDescriptor *sort = [[NSSortDescriptor alloc]
initWithKey:@"name" ascending:NO];
  [fetchRequest setSortDescriptors:[NSArray arrayWithObject:sort]];

  NSFetchedResultsController *theFetchedResultsController =
  [[NSFetchedResultsController alloc]
initWithFetchRequest:fetchRequest

managedObjectContext:self.managedObjectContext
                                          sectionNameKeyPath:nil
                                                   cacheName:@"Root"];
  self.fetchedResultsController = theFetchedResultsController;
  [_fetchedResultsController setDelegate:self];

  return _fetchedResultsController;
}
```

All of the previous lines were written in order to build and use a database, except the last method, which sets up a way to collect data from the database. Most of these methods are essentially using lazy loading and are in themselves getter methods, meaning that none of them are called and their contents not implemented until necessary. For example, the *managedObjectContext* function is not called until a new object needs to be inserted into the database. The calling of this method can be seen in example 9 as the line *self.managedObjectContext*.

The first method illustrated in example 8 – *viewDidLoad* – is an iOS SDK built-in function that is called every time a view controller's view is loaded for the very first time; Apple describes it in their auto-generated code comments as a place where one can "Do any additional setup after loading the view". In example 8 this functionality is used to set up the fetched results controller. There are several different ways to fetch objects from the database, one of the most common ways of doing so is using an NSFetchedResultsController, which returns objects in a manner that makes it easy to present them in a table view – using an NSIndexPath object, which defines an object's location with a section and a row. This location object is later used in table views and fetched results controllers alike, it will see usage in further code examples as well. The actual creation of the fetched results controller is demonstrated in the contents of the last method. In order to create a fetched results controller, it has to be passed a few arguments: a fetch request, which will describe the object entity that will be returned, and a sort descriptor, which sets up the way these fetched objects will be ordered.

The essence of the second method in example 8 is the creation of a managed object context, which is an object that manages the state of a single data model. In Apple's own words, it can be viewed as a "scratch pad" – a space where one can insert, alter and delete objects; all the changes made will be committed to the database when a "save" function is called. In most common situations there will be a single context, but extraordinary situations might call for a larger number of instances. This can create a complex situation where the programmer has to be aware of potential issues that stem from thread safety, concurrency and syncing of contexts, but it is a deeper subject field that deviates from the topic of this thesis.

The following method creates a managed object model instance, which is responsible for describing the schema of the data model that the developer has previously defined, including the entities and relationships in the database necessary for the application.

The succeeding step is the building of a persistent store coordinator. This is where things get a bit more complicated. Since it is possible to have several persistent data stores and several managed object contexts, there needs to be something that facilitates between those two layers of core data management and this is exactly what a persistent store coordinator does. Thanks to the coordinator, the managed object context sees only one persistent store, although there might actually be several of them; and all of the persistent data stores receive objects from a single coordinator object. Again, this object is only directly used on extraordinary occasions with a more complicated data model.

The *saveContext* and *applicationDocumentsDirectory* methods are simply convenience methods, meaning that they are written just to make the code elsewhere a little more clear. The first is used to commit changes done in a managed object context to the database and the second returns the path to the device documents folder.

Once an object is obtained, it can be altered or deleted. Before fetching a class instance, it has to be created and saved. The following code is written in order to create and alter objects in the database.

Example 9. Manipulating the database.

```
- (void)addProductWithName:(NSString *)name shopName:(NSString
*)shopName
{
  Product *newProduct = [NSEntityDescription
    insertNewObjectForEntityForName:NSStringFromClass([Product class])
              inManagedObjectContext:self.managedObjectContext];
  [newProduct setName:name];

  Shop *newShop = [NSEntityDescription
      insertNewObjectForEntityForName:NSStringFromClass([Shop class])
                inManagedObjectContext:self.managedObjectContext];
  [newShop setName:shopName];
  [newShop addProductsObject:newProduct];

  [self saveContext];
}
```

This example is rather straightforward despite its appearance: you create an object by calling an NSEntityDescription class method and telling it which class' object is needed and which managed object context to use. Then any of the newly created object's properties can be changed according to the programmer's wishes and a save call executed to finish the altering process.

It should be mentioned that there was a tool called "mogenerator"[44] used in order to simplify object alterations. "mogenerator" is a command line utility used to create two classes for each database entity. One of them is intended only for machine reading and the other one is where the programmer can insert custom code. Generating these two wrapper classes is actually a disputed topic on its own right, but out of the scope of this thesis. Since it is used with both Core Data and MagicalRecord and it has no impact on the results of this research, it will not be discussed further.

In case an instance of a class is no longer needed, it can be deleted. Example 10 will demonstrate how to delete an object from the database using Core Data. It also demonstrates the fetching of an object through a fetched results controller using an NSIndexPath object.

Example 10. Deleting an object.

```
- (void)deleteProductAtIndexPath:(NSIndexPath *)indexPath
{
   Product *product = [self.fetchedResultsController
                                        objectAtIndexPath:indexPath];
  [self.managedObjectContext deleteObject:product];

  [self saveContext];
}
```

These were the most basic and common operations that can be done with Core Data managed objects. As one could see, it requires quite a large number of lines of code to set up a database using Core Data. A following comparison will highlight the differences between Core Data and MagicalRecord.

MagicalRecord

Starting to code using MagicalRecords requires one to set up a Core Data stack, using on of five potential class method calls, each of which has their own specific advantages. In keeping with this thesis' examples the simplest one will be used.

Example 11. The initial code required for setting up the MagicalRecord implementation.

```
- (void)viewDidLoad
{
  [super viewDidLoad];

  [MagicalRecord setupCoreDataStack];

  NSError *error;
  if (![[self fetchedResultsController] performFetch:&error])
  {
      NSLog(@"Unresolved error %@, %@", error, [error userInfo]);
  }
}


- (NSFetchedResultsController *)fetchedResultsController
{
  if (_fetchedResultsController != nil)
  {
    return _fetchedResultsController;
  }

  NSFetchedResultsController *theFetchedResultsController =
      [Product MR_fetchAllSortedBy:@"name"
                         ascending:NO
                     withPredicate:nil
                           groupBy:nil
                          delegate:self];
  self.fetchedResultsController = theFetchedResultsController;
  [_fetchedResultsController setDelegate:self];

  return _fetchedResultsController;
}
```

The *viewDidLoad* is used again as the starting point to build a database. The *[MagicalRecord setupCoreDataStack];* line creates a persistent store coordinator, essentially the same as the *persistentStoreCoordinator* method in the Core Data example. The succeeding part is the creation of the fetched results controller, the contents of which are exactly the same as they were in the previous example.

It is obvious that MagicalRecord encapsulates a lot of the tediousness of initializing a database using Core Data. Although Xcode offers ways to create the necessary Core

Data setup code automatically, they currently only come as a package when one starts a new project with the "Master-Detail Application" template. So it seems that using MagicalRecords in this situation would give the programmer greater flexibility.

The next example in the previous section was that of creating and altering objects in the database. Example 12 will illustrate how that can be done using MagicalRecord. All of MagicalRecord's method calls use the "MR_" prefix to differentiate them from the rest – a feature that can also be turned off.

Example 12. Manipulating the database.

```
- (void)addProductWithName:(NSString *)name shopName:(NSString
*)shopName
{
  Product *newProduct = [Product MR_createEntity];
  [newProduct setName:name];

  Shop *newShop = [Shop MR_createEntity];
  [newShop setName:shopName];
  [newShop addProductsObject:newProduct];

  [[NSManagedObjectContext MR_defaultContext]
                              MR_saveToPersistentStoreAndWait];
}
```

The difference here with Core Data is that the creation of an object does not require the passing of the desired class name and managed object context used: it knows the class because the class itself is used to create a new object and it just uses the default context created during the initialization of the MagicalRecord database. All of this is followed by a custom "save" method call just like in Core Data.

The last use case in the previous section was deleting an object, a function that will be laid out in example 13 using MagicalRecord.

Example 13. Deleting an object.

```
- (void)deleteProductAtIndexPath:(NSIndexPath *)indexPath
{

  Product *product = [self.fetchedResultsController
                        objectAtIndexPath:indexPath];
  [product MR_deleteEntity];

  [[NSManagedObjectContext MR_defaultContext]
   MR_saveToPersistentStoreAndWait];
}
```

The distinctions between Core Data and MagicalRecord for altering objects are not as contrasting as with the creation of a database, but MagicalRecord does make it a bit less verbose, which in turn makes the method names easier to remember and use. This feat is achieved using a lot of default values, a practice that also has its potential downsides. While this is normally a sign of rigidness and it may make some more complex operations implausible or hard to achieve because it is not possible to modify the contents of black box components, MagicalRecord has worked around this issue thanks to the fact that its whole feature set is built on top of Core Data's libraries.

Since MagicalRecord is an open source framework, it can theoretically be changed by anyone in order to support a programmer's any wish. In software development's good practices it is advised against changing someone else's code, since the threat of rendering some unknown parts of it faulty can only increase. There is an alternative when the alteration of the code base is excluded from possible options. Due to the fact that MagicalRecord is built using Core Data, it also means that Core Data's functionality can be used interchangeably with the one from MagicalRecord, which gives the programmer all the simplified syntax advantages of MagicalRecord as well as all the dynamic lower level feature advantages of Core Data. This is also true in the previous case with AFNetworking and the Foundation framework's URL loading capabilities.

As was mentioned in the introduction of this subsection and visible from the above examples, there is a lot less coding required to use a database when utilizing the help from MagicalRecord. As with any framework, there are of course its own drawbacks, but the most common ways of using a database are far more comfortably and efficiently done with MagicalRecord then they would be when using only Core Data. The next section will focus on summarizing the qualities that make the most wide spread third party frameworks used in iOS development so popular.

4.3 Conclusions of AFNetworking and MagicalRecord's advantages

As it turned out in the survey carried out earlier, the most popular third party frameworks used in iOS development were AFNetworking and MagicalRecord. The advantages noted for AFNetworking were its simple programming interface for making requests, heavy usage of blocks and its powerful serialization capabilities. It was later illustrated how AFNetworking's simple interface is superior to that of Apple's iOS SDK's Foundations frameworks URL loading classes and how they can come up useful during programming.

The widely spread use of blocks was also analyzed and it was demonstrated how and why can using blocks be advantageous compared to using delegation practices. The last aspect of AFNetworking that was considered an upside was its powerful serialization capabilities. Although this field of functionality does also exist in Apple's iOS SDK, these were considered more convenient and thought to give even more extra value to the AFNetworking framework in general.

While the AFNetworking-related results of the survey could boast with three different noticeable positive qualities, there was only one major one regarding MagicalRecord – its capability to avoid boilerplate code. This should not be misunderstood: it can at times be a very difficult feat to accomplish. MagicalRecord is a framework built entirely on the functionality of Apple's Core Data framework, which in addition to its ease of use makes it very flexible and powerful, because all of Core Data's features can also be used in MagicalRecord. This is also true for AFNetworking and the Foundation framework's URL loading library, but not definitely true for all third party frameworks. One example of the opposite would be libraries where the features are built on very low-level C or C++ code, which a lot of developers are not that familiar with. Another case when a third party framework can be very unaccommodating is when the contents of it are closed-sourced, also known as a black box framework, and inaccessible to the programmer.

During the last previous years it has been quite clear that Apple has more and more tried to introduce these listed aspects to its own iOS SDK's frameworks: blocks are

becoming more common in everyday tasks, singletons are seeing increasingly more usage and programming interfaces are constantly being updated in widely varying fractions of the iOS SDK.

The simple fact of the matter is that the iOS SDK is still relatively young – a point also brought up in the conducted survey. As with any new thing, it still has its flaws and deficiencies, but is also keenly calling to be improved. A positive thing about Apple is that they seem to be learning a fair amount from their developers. As it was pointed out in the survey, Apple responds to user feedback and in the author's opinion, they do it directly and as well as indirectly. The direct way being reading and listening to users' complaints and ideas from their feedback submission system and the indirect way being learning from the users' habits, as with the "pull to refresh" example explained in the survey results subsection. So it would make sense to believe that Apple, in order to strive to make its SDK follow the preferences of its users, would accommodate the aspects that make AFNetworking and MagicalRecord so popular, into its own software development kit.

It would make sense to apply the positive aspects of these third party frameworks in anyone's future enterprises as much as possible. But it seems that the biggest deficiency of the iOS SDK is the simple fact that it is still rather new and it is not even close to being fully developed. All the while, it seems to be on the course to being a fully-fledged and capable software development kit – a course that is greatly influenced by lessons learned from its users.

5 Summary

The aim of this thesis was to take deeper look into Apple's iOS SDK, try to map the changes made in it during its lifetime and locate its weak spots if it had any. This was achieved by taking a closer look at the most popular third party frameworks utilized in iOS development.

The first part of this dissertation focused on the changes made in the iOS SDK during its nearly 10 years of existence. It evolved from the initial 17 frameworks in its first release to 55 of them in the latest version. It was revealed that Apple has a history of listening to the opinions of its development community and adapting the SDK appropriately, although it might occasionally take time to implement some new features.

The second part of the thesis was subjected to the finding of the most popular third party frameworks by using the *Star* feature in GitHub and then asking the iOS development community to voice their opinions through conducting a survey. It was found that the most popular third party libraries – AFNetworking and MagicalRecord – were used because of reasons including simplicity of the programming interface, usage of blocks and the removal of boilerplate code.

In the final part of the thesis these two frameworks were analyzed more closely to explain the reasons of their high popularity. Although a specific area of weakness in the iOS SDK was not found, a larger number of smaller ones were highlighted. It was concluded that the SDK's current biggest weakness is its relatively young age and raw - but constantly improving - evolutionary phase.

This thesis has hopefully helped pinpoint the deficiencies of the iOS SDK, which would in turn help current and future developers improve their technical skills and applications by using the observed useful iOS programming practices. The results and observations of this dissertation can be used as a basis for future works in order to even further scrutinize the state of this subject.

6 Bibliography

1. Smartphones to Overtake Feature Phones in U.S by 2011, http://www.nielsen.com/us/en/newswire/2010/smartphones-to-overtake-feature-phones-in-u-s-by-2011.html, (28. December 2013)
2. Samsung Wins U.K. Apple Ruling Over "Not As Cool" Galaxy Tab, http://www.bloomberg.com/news/2012-07-09/samsung-wins-u-k-apple-ruling-over-not-as-cool-galaxy-tablet.html, (28. December 2013)
3. Open Handset Alliance, http://www.openhandsetalliance.com/press_110507.html, (28. October 2013)
4. Apple Developer Programs, https://developer.apple.com/programs/, (28 October 2013)
5. Windows Phone Application Publishing, http://msdn.microsoft.com/library/windowsphone/help/jj206719(v=vs.105).aspx, (28. October 2013)
6. A Brief History of Mac OS X, http://osxbook.com/book/bonus/ancient/whatismacosx/history.html, (28. October 2013)
7. Smalltalk, http://www.smalltalk.org/main/, (28 October 2013)
8. Cocoa Touch Frameworks, https://developer.apple.com/technologies/ios/cocoa-touch.html, (28. October 2013)
9. Cocoa Frameworks, https://developer.apple.com/technologies/mac/cocoa.html, (28. October 2013)
10. Apple releases iPhone SDK, http://www.gsmarena.com/apple_releases_the_iphone_sdk-news-454.php, (28. October 2013)
11. iOS Developer Library Release Notes, https://developer.apple.com/library/ios/navigation/#section=Resource%20Types&topic=Release%20Notes, (28. October 2013)
12. The Evolution Of iOS: From iPhone OS to iOS 7, http://www.cultofmac.com/191340/the-evolution-of-ios-from-iphone-os-to-ios-6-gallery/, (28. October 2013)

13. Live from Apple's iPhone SDK press conference, http://www.engadget.com/2008/03/06/live-from-apples-iphone-press-conference/, (28. October 2013)
14. Apple Unveils iPhone, http://www.macworld.com/article/1054769/iphone.html, (28. October 2013)
15. iOS Developer Library Release Notes https://developer.apple.com/library/ios/navigation/#section=Resource%20Types&topic=Release%20Notes (02. December 2013)
16. Apple Fixes a Bevy of Serious Flaws with Latest iOS 7.1 Update, http://www.infosecurity-magazine.com/view/37399/apple-fixes-a-bevy-of-serious-flaws-with-latest-ios-71-update/, (06. April 2014)
17. iOS Developer Library Frameworks, https://developer.apple.com/library/ios/navigation/#section=Frameworks (29. October 2013)
18. Introduction to CFNetwork Programming Guide, https://developer.apple.com/library/mac/documentation/Networking/Conceptual/CFNetwork/Introduction/Introduction.html, (30. October 2013)
19. Core Audio Overview, https://developer.apple.com/library/ios/documentation/MusicAudio/Conceptual/CoreAudioOverview/Introduction/Introduction.html, (30. October 2013)
20. iOS Technology Overview https://developer.apple.com/library/ios/documentation/miscellaneous/conceptual/iphoneostechoverview/MediaLayer/MediaLayer.html (30. October 2013)
21. Quartz 2D Programming Guide https://developer.apple.com/library/mac/documentation/graphicsimaging/conceptual/drawingwithquartz2d/Introduction/Introduction.html (30. October 2013)
22. Core OS Layer https://developer.apple.com/library/ios/documentation/miscellaneous/conceptual/iphoneostechoverview/coreoslayer/coreoslayer.html (30. October 2013)
23. Core App Objects https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AppArchitecture/AppArchitecture.html (30. October 2013)

24. About AV Foundation https://developer.apple.com/library/ios/documentation/AudioVideo/Conceptual/AVFoundationPG/Articles/00_Introduction.html (30. October 2013)
25. Mixed reaction to iPhone update http://news.bbc.co.uk/2/hi/technology/8090513.stm (30. October 2013)
26. Mobile Core Services https://developer.apple.com/library/ios/documentation/MobileCoreServices/Reference/UTTypeRef/Reference/reference.html#//apple_ref/doc/uid/TP40008771-CH1-SW1 (02. December 2013)
27. Siri, http://www.apple.com/ios/siri/, (07. December 2013)
28. Maintenance-oriented selection of software components, http://www.it.iitb.ac.in/%7Epalwencha/mtp_first/lic/lic/pap1412/maintenance%20oriented%20selection%20of%20aoftware%20components.pdf, (06. January 2014)
29. A Framework for Systematic Evaluation of Software Technologies, http://www.free-conversant.com/mindspill/140/enclosure/brown96framework.pdf, (06. January 2014)
30. Defining Useful Technology Evaluations, http://www.dtic.mil/dtic/tr/fulltext/u2/a476811.pdf, (06. January 2014)
31. Status and trends of mobile-health applications for iOS devices: A developer's perspective, http://www.ohioupsychology.com/files/images/holroyd_lab/Liu,%20C%20et%20al%20%282012%29%20Status%20and%20Trends%20in%20mobile-health%20J.%20Sysm%20Soft.pdf, (11. January 2014)
32. Development Frameworks for Mobile/Wireless User Interfaces: A Comparative Case Study, http://spectrum.library.concordia.ca/1706/1/MQ68478.pdf, (11. January 2014)
33. GitHub Stars feature, https://help.github.com/articles/stars, (19. January 2014)
34. GitHub most starred custom query, https://github.com/search?l=objective-c&o=desc&q=stars%3A%3E1&s=stars&type=Repositories, (19. January 2014)
35. iOS Frameworks, http://iosframeworks.com, (22. January 2014)
36. UIRefreshControl Class Reference, https://developer.apple.com/library/ios/documentation/uikit/reference/UIRefreshControl_class/Reference/Reference.html, (6. April 2014)
37. DTCoreText, https://github.com/Cocoanetics/DTCoreText, (6. April 2014)

38. iCarousel, https://github.com/nicklockwood/iCarousel, (6. April 2014)
39. Appirater, https://github.com/arashpayan/appirater, (6. April 2014)
40. AFNetworking on GitHub, https://github.com/AFNetworking/AFNetworking, (6. April 2014)
41. NSURLSession Class Reference, https://developer.apple.com/library/ios/documentation/Foundation/Reference/NSURLSession_class/Introduction/Introduction.html, (18. April 2014)
42. NSURLRequest Class Reference, https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSURLRequest_Class/Reference/Reference.html#//apple_ref/doc/uid/TP40003762, (18. April 2014)
43. MagicalRecord on GitHub, https://github.com/magicalpanda/MagicalRecord, (01. May 2014)
44. mogenerator, https://github.com/rentzsch/mogenerator, (04. May 2014)

Appendices

Appendix 1. The total list of iOS SDK frameworks

A file named “Appendix 1. The total list of iOS SDK frameworks” is also appended to this thesis: it summarizes all of Apple’s iOS SDK frameworks available to developers, categorized by iOS SDK versions. It contains the name of the framework, the programming language it is written in and a URL to its framework reference web page.

Appendix 2. The final list of selected third party frameworks

A file named “Appendix 2. Third party frameworks” is appended to this thesis, which contains the final and total list of public open source third party frameworks selected to be included in the conducted survey. It consists of the name, its GitHub *Stars* count, a short summary, their GitHub web site URL and the type of the library, assigned by the author. The color marks the level of complexity of each framework – definitions are included.

Appendix 3. The contents of the survey

The final appendix is a PDF file named “Appendix 3. Questions and answers of the survey”, which contains the full list of questions and answers accumulated by conducting it.

Non-exclusive license to reproduce thesis and make thesis public

I, Mikk Pavelson (date of birth: 08.04.1988),

1. herewith grant the University of Tartu a free permit (non-exclusive license) to:
   1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
   1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright, "The deficiencies in the Apple iOS SDK with the example of third party frameworks usage", supervised by Marko Peterson,
2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive license does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **25.05.2014**