Aalto University
School of Business

Open Sourcing Programming Language - Case Apple Swift

Information Systems Science

Master's thesis

Eero Halmetoja

2016

Department of Information and Service Economy
Aalto University
School of Business

Aalto University
School of Business

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi

Abstract of master's thesis

Author Eero Halmetoja

Title of thesis Open Sourcing Programming Language – Case Apple Swift – Case Apple Swift

Degree Master of Science in Economics and Business Administration

Degree programme Information and Service Management

Thesis advisor(s) Matti Rossi

Year of approval 2016 **Number of pages** 51 **Language** English

Abstract

Companies, that are formerly considered to be closed and leaning towards proprietary software solutions, are increasingly more involved in open source software projects, and introducing more open source projects as part of their business. The open source paradigm is widely researched topic but the increasing involvement of some of these major technology companies has happened in the very recent history, and therefore deserves attention.

This thesis will aim to answer the question: why did Apple open the source code of their programming language Swift? In order to answer this question a case study was conducted, in which earlier literature and publicly available data was used to create a conversation, with a goal to reveal the underlying aspects of the phenomena. The literature review goes from historical evolution of open source software to more recent research and compares it to the steps that Apple has taken in its process of open sourcing programming language Swift.

Apple's statements for open sourcing decision argue the improvements in software development, and the importance of third-party software developers for the company. The earlier literature both supports and disagrees with some of the aspects of Apple's open sourcing decision. Furthermore, the company is able to utilize some of the tools that the earlier literature points out to be beneficial for the success of an open source project. Lastly, the company's history in OSS usage give arguably some indicators for the strategic approach that the company will aim to maintain with its OSS projects.

Keywords open source, open sourcing, open sourcing programming language, Apple, Swift

Table of Contents

1 Introduction

Open source (OS) movement has grown steadily over the recent years and the related research has evolved with the movement. In some forms, OS development paradigm has been around as long as personal computing, and behind the open source lies concepts like peer production, shared code, and software as a public good. The term "open source" in the other hand is relatively recent phenomena (Aksulu & Wade, 2010). In the most recent past, the movement has drawn increasing attention from some of major technology and software firms. In this paper I will examine why Apple, a company that has been accused to be a control-freak (Fortt, 2007), has opened the source code of their relatively new programming language Swift. Apple is not the only major player that opens or at least experiments OS in their programming languages; Facebook, Google, and Mozilla have also done it in recent years (Derballa, 2015).

Although open source software (OSS) and related topics are widely researched, the topic of major technology companies in increasing amounts opening their source codes raises questions. What makes the topic particularly interesting is the structure of some of these companies. As I will showcase later in this thesis, the case company in hand, Apple, has been known for its closed ecosystem and the preference towards proprietary software. I'm highly skeptical that a company that is well known for taking a full control over all the aspects of its business, hardware, and software, would open their source code without a well argued business reason. This reason is, why the company opened their source code, is my main concentration in this thesis. In order of answering this question I have to create a holistic view of the open sourcing phenomena, and examine Apple in the light of the earlier OSS literature.

The goal of this research is to have conversation between existing literature and the case company in order to understand the reason why the case company open sourced their programming language. Some of the aspects that I will hope to tackle are the direct impacts of open sourcing: community involvement, software quality, and indirect impacts: brand recognition, and demand of complementary products. Negative aspects associated with open sourcing in a business environment will be discussed also in this paper, some of these aspects are: the lost control and lost revenues of giving the software away for free. The discussion reaches also some of the more philosophical views of the issue, including the differences

between Open Source Software (OSS) and Free/ Libre Open Source Software (FOSS), and also to the tension between the OSS community and the established technology developers (Gary et al., 2009a). I want to also consider, how much a company actually relinquishes control over their software with open sourcing.

In order of answering my research question, I will also have to address some of the themes related direct and indirect revenue models related to open source software. The view for tackling this issue is concentrated on the complementary product view of OSS business models (Kort & Zaccour, 2011). This point of view looks at the OSS business models in relatively high level, and thereby helps me to discuss this case where all revenues seem to come from indirect usage of open sourced software. In order of creating a holistic view of the indirect revenue generation by the case company, I feel it is important to have some discussion about the platform strategies and related intellectual property rights (IPR) (West, 2003). IPR plays also a major role of the creation of initial OSS creation and therefore will have a significant part dedicated towards it in this paper.

The conversation with the literature and the case company is done in iterative manner like Edmondson and McManus (2007) suggest. Meaning that the literature review takes a broad perspective from the start narrowing down towards the most suitable reasoning that may answer the research question (Edmondson & McManus, 2007). Before starting to dive in to my literature review I state few assumptions that I will consider for the reasons for why Swift was open sourced, these include product branding, platform battle, and the developer attraction. I assume these are the three most important reasons for Apple to open source their programming language.

2 Methodology

2.1 Case study

Case study is a careful study of a single case that leads researcher to see new theoretical relationships and question old ones (W. Gibb Dyer, 1991).

My research question aims to answer the question: "why apple open sourced their programming language Swift". In order to answer this question, the case phenomena being the move from proprietary piece of software to open source software, I have to combine earlier literature for deductive theory testing and present my findings in inductive manner. Iteration process between inductive theory development and deductive theory testing is argued to advancing organizational phenomena (Edmondson & McManus, 2007). And I argue that the deductive theory testing in my case study helps me to identify possible research gaps in the earlier research and allows me to answer the research question. Even if there are no significant gaps in the earlier research the deductive theory testing allows me to draw a holistic view of the case situation, and thereby grasp at least on some level, all the possible causes that have lead to the open sourcing decision.

My goal is to have fluent conversation between the earlier literature and the case, which allows, and requires, me to go back and forth with theoretical models and evolve the design of the study as I proceed. Edmondson and McManus (2007) suggest that the methodological fitting process is a funnel that narrows down giving greater latitude and choice in the early phases of research. The funnel then narrows down and decreases options that can be made in the research. As the options diminish the research design evolves to project that is feasible and viable (Edmondson & McManus, 2007). I feel that this is the most feasible approach for my research as I have limited initial understanding of programming language related business models to start with. With the wide scope of possible reasons to open the source code, I will be required to look in to the phenomena from multiple different angles, and with this iteration process to come up with the final scope of the study in hand.

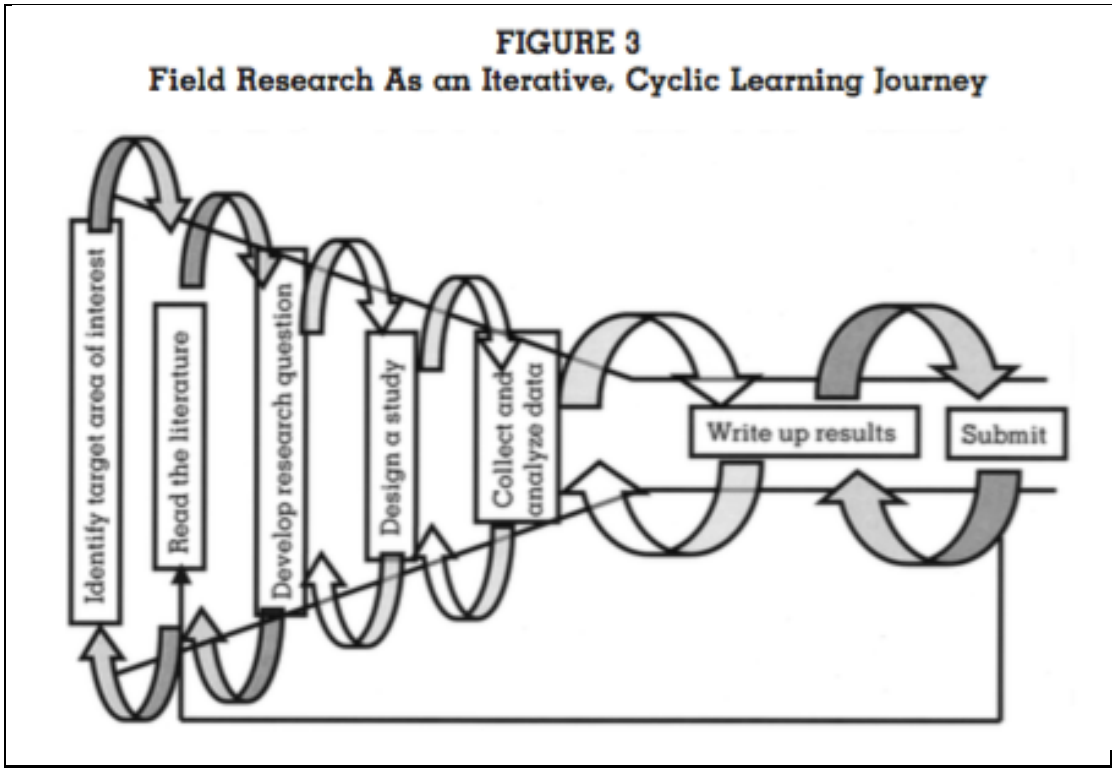


Figure 1. Iterative Research ((Edmondson & McManus, 2007)*).*

Moreover, Edmondson and McManus (2007) suggest that the management research falls in a continuum, from mature to nascent. Nascent theory proposes tentative answers to novel questions of how and why. Where as mature theory presents well-developed constructs and models that have been studied over time. Between these two lies the intermediate theory, which presents provisional explanations of phenomena, giving a new construct and presenting new relationships with proposed construct and already established constructs. My research will move along this continuum as the aim is to answer the research question by revealing possible gaps in the earlier research, and at the same time creating a holistic view of the situation. The research compares the case to well established OSS-research in order to answer the question "why", and aims to connect the results with already existing literature. Thereby focusing on the nascent end of the spectrum, but at least partly covering the entire spectrum suggested by Edmondson and McManus, as my research takes advantage of already established theoretical constructs as well (Edmondson & McManus, 2007).

I have chosen a single case as it allows me to convey a complete story with unbroken narrative, in a manner that allows me to go back and forth with the case company and earlier literature (W. Gibb Dyer, 1991). The case company in hand makes this study extremely interesting, as I will convey in the next chapter, but also limits my access to primary data. The data collection methods I will use in this research are limited to publicly available data, I will utilize news articles, developer blogs, and company's release notes in order to shed light to the phenomena. I consider that the limited access to company informants may also convey

a less biased results as the initial view of the company's actions will not only result in positive image in the company's OSS usage (Eisenhardt & Graebner, 2007). The use of publicly available data creates challenges and has to be considered as a restriction for this research, this is especially true as the case company is deliberate for releasing information, and also due to the fact that the research phenomena is happened in very recent history. Nevertheless, I feel t that the overall picture that I am able to create with the data and earlier literature will give insights for some of the main reasons behind the open sourcing decision, and give possible implications for further research. Also it has to be considered again, that as many other OSS researches this thesis takes a look on a very recent phenomena and thereby there is a risk of generalization of still emerging process (West, 2003). The case company may not be successful with its open sourcing project, or may turn back to a proprietary solution. My hope is that holistic view of the case will provide some reasons why this may happen as well.

Furthermore, even if this research will not be able to point out any gaps in the earlier research it should be able to draw a holistic picture of the phenomena and explain some of the historical reasons for the OSS development and case company's earlier decisions to open source software. This may point out some of the similarities to this more recent open sourcing decision, and possibly create a view of the case company's attitude towards open source solutions.

3 The case company

Apple is the case company in hand, the company is chosen for number of reasons: it has strong hold over its brand and products (Montgomerie & Roscoe, 2013), it can be considered the number one tech company in the world (Chen, 2015), and it has received some negative feedback from the more ideological side of OSS community ("Apple ' s Operating Systems Are Malware," n.d.)

Apple can be considered to have strong hold over its products, to the point that some argue that the company's key to business success is its ability to 'own the customer'. According to Montgomerie and Roscoe (2013) Apple's business model is to drive the consumer into its ecosystem and hold them there with high switching costs. Moreover, the company maintains this multi-channel platform integration with legal and technological means reaching the control from customers to all the way to suppliers and manufacturers. Essential part of the high switching costs and control over the customers was the introduction of iTunes Music store and iPod in 2003. This was the starting point for Apple's multiplatform integration, where the company chose to control the interface between the hardware and content, realizing the first opportunity to 'own the consumer' (Montgomerie & Roscoe, 2013).

Apple was the most valuable brand in 2015 according to Forbes. Already in April 2012, Apple's market capitalization surged to $570 billion, making it more valuable than Google, Microsoft, Hewlett-Packard, Dell and Yahoo combined (Russolillo & Cheng, 2012). Apple's hardware products include smartphones, tablets, personal computers, smart watches and portable music players. Where as the company's consumer software is mainly dedicated to the company's own hardware, including OS X and iOS operating systems, web browser, and iTunes media player. Company's online services include iTunes Store, App Store's for iOS and Mac, and iCloud ("Apple Inc.," n.d.). Apple notes that increasing portion of the company revenues comes from "internet services", which ,for example, include purchases made in the App Store (Apple Inc., 2015). Already this information ties in together the relevance of multiplatform integration and revenue streams, and shows the importance of Swift, as it is projected to be the main programming language for all software in the Apple platform infrastructure.

Free Software Foundation (FSF) considers Apple's operating systems to be malware. The foundation considers malware as any piece of software that is designed to function in ways that mistreat or harm the user. FSF lists the issues that it considers making the Apple's software mistreatful towards the customer. The key here is not considered to be the price, but the control over the software and what does the program do when it runs. FSF acknowledges that in most cases non-free software is considered to be malware by their standards as the user is powerless to fix any malicious functionalities developers have imposed towards them. So in this sense the argument is towards Apple's "malwarus" software is strict, and something that only few companies will avoid, but this will set the scene for the more ideological discussion of Open Source (OSS) and Free/ Open Source Software (FOSS) ("Apple ' s Operating Systems Are Malware," n.d.).

These are just some of the reasons for the initial pick for the case company, and the reason why the case situation can be almost considered to be the 'talking pig'-scenario, meaning that the small sample size is reasonable as the case presents a phenomenon that almost justifies a purely descriptive study (Siggelkow, 2007). Further more I argue that the case company's market position will give its actions such a broad impact, that it may shape the future of the software creation and consumption. The importance of future implications of the case in hand will be more apparent when the discussion moves towards the platform standards in this paper.

3.1 Swift

"On December 3, 2015, the Swift language, supporting libraries, debugger, and package manager were published under the Apache 2.0 license with a Runtime Library Exception, and Swift.org was created to host the project." The source code is hosted on GitHub, and the project is governed by *"a core team of engineers that drive the strategic direction by working with the community, and a collection of code owners responsible for the day-to-day project management"*. Apple has set "community guidelines", which includes detailed information how the Swift community is managed ("Community Guidelines," n.d.).

Before Swift, the main language for creating content to Apple's platforms has been Objective C, which was the default language for NexSTEP, OS X and iOS. In 2010, Apple started developing Swift programming language to replace the Objective-C. Swift 1.0 was released in September 2014 (Bohon, 2016). Swift is backed by the Cocoa and Cocoa Touch frameworks; these are Apple's own application programming interfaces (APIs) that were

already in use with the Objective C. The frame surrounding the programming language itself, is created by advancing the already existed complier, debugger, and framework infrastructure. Apple claims that the Swift is friendly to new programmers, and that *"it is the first industrial-quality systems programming language that is as expressive and enjoyable as scripting language"*. The company also claims that the programming language is designed to scale up all to way for operating system creation (Apple Inc., 2014).

Swift has some big shoes to fill as the predecessor Objective C, dating back to 1983, has been the primary language used for developing iOS apps. By updating the language to a new modern version, Apple hopes to make the adaptation of the language easier for new developers and also to help experienced coders to avoid making serious programming mistakes. Apple has open sourced some of its projects before, like Darwin and WebKit, which have worked in the background of OS X operating system and Safari web browser, respectively. Objective C was never open sourced by Apple, and furthermore the lack of support from apple has typically made the cross platform coding difficult. This has created a market for third party companies that may utilize the inability to cross-platform program development. For example, a company called Xamarin has created tools that allow developers to use Microsoft languages to built software that will run on Windows, Linux, iOS, and Android environments. With Swift, Apple makes their own programming language available on other operating systems for the first time. In theory, the downside for Apple is the fact that this could allow developers to use Swift in competing markets, for example in order to built apps for Android (Derballa, 2015).

The claim is that now when the Swift is open source, all software written in this language will be easier maintained and kept up to date, with fewer bugs and crashes than ever before (Timmer, 2016). This claim is arguably based on some of the fundamental ideas associated with OSS usage, like the improvement in software quality (see e.g. Caulkins et al., 2013; Kort & Zaccour, 2011; Lindman, Juutilainen, & Rossi, 2009). Community involvement in the supporting tools, and the language itself, should allow the software developers to create cleaner code that is less prone to errors. But at the same time it is important to note that the OS programming language does not make the software created with it open, and that the software developer is responsible for the quality of software created. Software quality is thereby depended on development choices and for example the number of community supported components have effect on the ease of maintaining the created software.

Swift is a general-purpose programming language, meaning that it is not restricted to a specific application domain. According to Apple, the goal of Swift is to create the best available language for uses ranging from systems programming, to mobile and desktop apps, scaling up to cloud services. The aim is to allow all this to be done while easing the maintenance and writing of programs easier for the developer. Other goals that the company have for the programming language include: safety, speed, and expressiveness. What some of these mean in programming language domain are explained in the following paragraphs (Swift.org).

Safety, refers to type-safe language, and according to Apple it means strictly defined out bounds, or limits, for the code writing. By this Apple means that the language eliminates entire classes of unsafe code (Swift.org). More over type safety should help the developer to keep up with the values of the code under work, and also catch and fix errors in the development process. To help convey the type-safety there is a type-check process integrated in the complier, in other words Swift performs type checks when compiling the code and flags any mismatched types as errors (Apple Inc., 2014). The question that arises from this is whether the stricter out bound limits will complement Apple's software testing process, allowing the company to streamline the evaluation that proceeds the acceptance in the App Store. The level of technicality leaves this question out of the scope of the research. But another factor that I will aim to investigate is whether this language is more attractive to the developers than Objective-C, this will be done in the "measuring programming language popularity"-section of this paper.

These details of the basic concepts show how the programming language works, and declares a view on the ideology on what Apple may aim with this language. This scope will help me to discuss the boarder lines that Apple have set for the OSS-community as well as how this language is perceived to work in the future. All helping me to answer the question: why did Apple open source this language. Next I will go through some of the other major factors associated with open source programming language, these include the community, and compatibility of the language.

Swift community guidelines identify five different roles inside the community: project lead, core team, code owner, committer, contributor. Apple Inc. works as the project lead, and interacts with community through its representative. Core team is responsible for steering the project in the right strategic direction, and consists from a small group of engineers appointed by the project lead based on their technical expertise and community involvement. At the current stage the core team is composed solely of Apple employees. Apple states that this is due to the fact that Swift has its origins at Apple, and that in the future exceptional community

members will be appointed in to the core team. Code owners are responsible for contribution revision, community feedback gathering, and tending of approved patches into the product, for the assigned Swift's sub-projects. Any active community member can offer to be a code owner, and the nomination is done by an another community member. In addition, for these core roles, Apple has set a 'code of conduct' for community members, possible violations are addressed by a code of conduct working group, which consists from community members. The table bellow summarizes the roles inside the community ("Community Guidelines," n.d.).

Table 1: Summary of Swift Roles

Role	Members	Appointed by	Primary responsibility / Role
Project lead	Apple Inc.	-	Arbiter of project
Core team	Exceptional community members (currently all Apple employees)	Project lead	Approver of evolution proposals
Code owner	An active community member	Community members	Code quality in sub-project level
Committer	A member with commit access to the code base	-	Code commitment
Contributor	A community member	-	Contribution and code review

("Community Guidelines," n.d.).

The roles inside the Swift community showcase an example of community motivation, which is, at least partly, peer recognition in this situation. Community motivation is highly researched topic related in OSS usage. Okoli and Oh state that this type of peer recognition inside the community is considered to be the one of the notable motivational tools for OSS participation. The informal praise and acknowledgement, in form of granting an administrative position, can be compared to a manager status promotion in traditional organization (Okoli & Oh, 2007). The roles inside the Swift community have clear implications of the fact that the strongly involved individuals will be recognized with non-monetary, intrinsic, rewards. The

community motivation and individual involvement will be further discussed later on in the paper.

Swift programming language is aimed to have compatibility with wide range of platforms. The main platforms that software created with Swift is aimed are Apple's own: iOS, OS X, watchOS, and tvOS. But from early phases of open sourcing the company has made it clear that the language development and software creation will be aimed for other platforms as well, in the beginning these platforms include the Linux kernel (Swift.org, n.d.). One factor of interest for Apple may be web apps. Some claim that there is a reason to believe that the ability to port Swift with Linux means that Swift apps can now run on low-cost, low-maintenance Linux servers that are already the cornerstone for existing web APIs and servers (Tofel, 2015).

The usage of Swift didn't take long for competitors either. IBM introduced "IBM Swift Sandbox" just shortly after the Apple open sourced the programming language and made it available on Linux environment. IBM solution allows the developers to run the Swift programming language on a cloud, using a Docker container. Also the benefit for Apple for cloud based Swift solution is the fact that it allows any device that runs a modern browser to use the language (Tofel, 2015). Some other tools that have already been developed around Swift are: perfect.org, IBM Sandbox, VAPOR @ GitHub, and VMware.

4 Open Source Software

So previously I showcased why the Apple is considered closed, software and business wise. So why company like this would introduce Open Source Software in their architecture? In order to answer that question, in this chapter, I will first go through the history of OSS and some of the reasons why companies choose open source software. Next I will use earlier literature to explain what is OSS and why companies choose to use OSS, while comparing it to the case situation.

The intellectual property rights (IPR) arguably create some of the essential issues related to OSS usage and therefore I have dedicated large portion of this chapter for IPR related discussion. The IPR discussion also helps me to explain where the OSS has come form and possibly where it is heading, as the smart usage of IPR has been the factor that made OSS possible in the first place (Raymond, 1999). Firstly, the licensing allows the company to utilize external programmers in their software development. While the traditional software development grants the right of use to the end-user for a piece of software through a license, and transfers the rights to the developer on the employment contract. In OSS projects' license is used for both the community, which includes employees working on the software, and the end-user (Lindman, Paajanen, & Rossi, 2010).

With number of examples of voluntarily started OSS products outperforming commercial software with similar functionalities, it is clear that the interest towards OSS solutions is increasing among commercial companies. Examples of these outperformers include: Apache web server, My SQL database, and Linux operating system (Lindman et al., 2009). In the recent years some of the major tech companies have started utilizing or at least experimenting with OSS solutions; Facebook, Google, and Mozilla all have had OSS projects in the recent years (Derballa, 2015). One of the Google's OSS projects include the programming language Go, which was announced in 2009 and was targeted to various platforms including Linux, OS X, Windows, and multiple different BSD and Unix versions ("Go (programming language)," n.d.).

A topic that will come up in the evolution of OSS is the separate paths of OSI certified OSS and Free Software Foundation's (FSF) Free/ Libre Open Source Software (FOSS). The philosophical differences between these two will be indicated in the upcoming chapter.

4.1 Intellectual Property Rights of OSS

The early software solutions were always tied to the hardware and therefore no protection of software was needed. In the 1970s this package was unbundled creating two individual products. In the 1980s personal computers created a vast business for software solutions. Earliest software protection tools included usage of object code, which did not allow the user to interpret nor change the software in a feasible manner. As a second option came the legal manners to keep the software secret and the first legal actions were the introduction of trade secrecy laws (de Laat, 2005).

Trade secrecy laws were not seen as best option for the companies as the nondisclosure agreements did have negative effect on companies if they led to court cases. Reason for this is the effect on the public image if the company aims to pursue a legal action against its leaving employees. The other option was to pursue legal actions against the users. So the secrecy act aimed to silent first the employees and then the consumer side of the software market. For the developer side the secrecy tactic was hard to execute as the nondisclosure agreements and their enforcement in the court could lead the company in negative light. For the customer side there were two main customer categories, the customers with tailor-made solutions and the mass-distributed solutions. Tailor-made customers were made to sign confidentiality clauses, which may not be suitable for the customers who were aiming to require the source code in order for modifying the software in the future. For the mass-distribution side firms invented the "shrink-wrap" license; customer would automatically comply for the license terms upon unwrapping the software (de Laat, 2005).

Copyright has been developed to protect literary works like novels, plays and poems, and other art works like paintings and sculptures. It grants the creator of the work all the rights of publication and distribution of the product in its literal form. Thereby the form of expression is protected not the underlying ideas. It is important to note that the copyright will be granted automatically once the original piece of work is compiled on a physical medium. Once the scope of the copyright reached to software the object code and source code both came to enjoy the protection. But still the protection applied just for the literal text, meaning that the underlying ideas and algorithms weren't protected. (de Laat, 2005). For my limited IPR knowledge the copyright usage for software creates two underlying ideological problems. Firstly, the copyrighted software can be read but somehow the clause changes in the case of running the software. In order to run the software, you now have to ask permission from the

copyright holder, which is often times the developer, or the 'author' of the program. The second issue to me is the fact that programming language is a set of predefined orders, thereby creating somewhat strict boundaries for textual expression in software creation.

For the companies using copyright laws in their software protection, the issue was twofold in the early phases of copyright protected software. For the company, the question is whether to sue a supposed infringer or not? Cost of the suing may not be in line with the gained results and, as stated before, the copyright protects the text itself not the underlying ideas, which are the most valuable part of the software, and therefore still remained unprotected (de Laat, 2005).

The other option, patents, are usually granted for inventions to protect the results of research and development. In order to be obtain a patent the invention must be useful, novel and non-obvious. Patents require also the statutory subject matter, and thereby the developers started pursuing patents as "implemented in the software" (de Laat, 2005). Abstract concepts are not patentable and thereby it is not usually possible patent software. In cases where the software is patented as a part of hardware it will cover not only the source code of the software but also the underlying idea of the software. Furthermore, the patent protects the innovation in a manner, that even the "accidentally" similar product is banned from the market.

The issue behind patenting software is that if the software is considered as an idea created by mental process, the patent would limit the freedom of thought. And in the other hand the algorithms behind the software would lead to limitation of mathematical language. Gradually when the software patenting generalized they were claimed both for the process and for a machine. Machine embodied solutions being more likely to pass the statutory test. For the processes the requirement is that the software requires physical steps before or after the process, or optionally the process has practical applications within technological arts. Copyright and patent usage in the software has led to situation where the software is "double protected", copyright protects the written text in the programming language and the patent protects the process the program performs (de Laat, 2005).

4.2 OSS Licenses

Licensing was something that made OSS possible in the first place, but at the same time, some argue that legal risks associated with OSS use has critical influence on the sustainability of open source movement as whole. Legal obligations related to the OSS usage reaches both the producer and consumer, and sets restrictions and rights for the future use of the software (Lokhman, Abdul-Rahman, Luoto, & Hammouda, 2011).

Complexity of a single OSS license may create an issue to start with, the exact terms of the license can be too difficult to understand, and the complexity increases as different software components are tied together. With OSS, unlike with proprietary software, there is no single owner for the software the user may consult with (Shaikh, 2015). The importance of licensing is apparent with the OSS projects and next I will go trough some of the most important OS licenses.

4.2.1 Restrictive Licenses

In 1984 Richard Stallman, a MIT programmer, started writing a free operating system from the scratch called GNU. GNU should be compatible with Unix so it can replace this non-free operating system. The Free Software Foundation (FSF) was set up to help with the project. Stallman's releases were licensed under a General Public License (GPL). GPL turned the tables around for copyright use, pointing a change where the property is considered in a new way. Under GPL license the source code may be freely used, modified and (re)distributed, but modifications and recombinations have to be licensed under the same terms as the original code. In other words, the future recipient of the code will have the same rights as the code creator. This will cause the situation that the evolving code may not leave the public path, and that the GPL-tie in a program will remain there forever, moreover because in case of combining multiple sets of code the upcoming work may be redistributed only under GPL-conditions (Laat, 2005).

Later FSF created Lesser General Public License (LGPL), using the similar key concepts as the GPL, but allowing to link the open source code to a library without a needing to "contaminate" the library with the GPL (de Laat, 2005). LGPL and Mozilla Public License (MPL) are examples of moderately restrictive open source licenses (Välimäki, 2005). The obligations and rights related to moderately restrictive licenses become the most apparent when tying in multiple different OSS licenses together, this linking process will be discussed later on in the paper.

4.2.2 Permissive Licenses

Berkley's approach for creating operating system was different from GPL as associated developers aimed to avoid writing the software form scratch as much as possible. This was done by liberating existing Unix files, libraries, and utilities for the parts that they were sole authors. Other parts of the program had to be rewritten from the scratch. Complete Unix system became available in 1992. The freed and recreated releases came under BSD license, which

was used since 1989. With BSD everyone may freely use, modify and distribute binaries and source code, in original or modified form. The main difference with GPL is the fact that BSD allows redistribution to be done under closed commercial licenses (de Laat, 2005).

Some other permissive widely used permissive licenses include MIT, Apache (ASL) licenses. Apache license is the license used for Swift programming language. Also some of the other projects by Apple tend to lean towards the permissive spectrum of OSS licenses; Apple's Darwin operating system was licensed under Apple Public Source license, and WebKit used in the Safari web browser licensed under BSD license, for most parts. In Darwin's case the use of permissive license caused controversy as the public were afraid that Apple will use the contributions of the OSS community and turn the parts of the operating system back under a proprietary license (West, 2003).

4.3 License compatibility

Legality concerns in open source software intensive systems is not restricted only to the component level of software licenses, but also the implementation, packing, and deployment of the software components have to be taken in the consideration. The vast number of open source licenses are explained in separate chapter, but here I go through some of the compatibility issues related to the licenses. As explained in OSS license chapter, there are three main categories where the OSS licenses can be placed: copyleft, weak-copyleft, and permissive licenses. Also it has to be taken in consideration that this is undefined set of categories as the number of licenses is so great and there are "forks" of the same licenses that may apply to different rules. But using these three main categories we can establish a fundamental linking rules between these licenses (Lokhman et al., 2011).

So as stated before, even the different kind of OSS licenses may not be compatible with each other due to subtle differences in the license terms. There is a vast number of different OSS licenses some having wide spectrum of differences between them, varying in the privileges and requirements that they set for use and distribution of the software. The differences between these license terms make some of these licenses to be incompatible with each other. If the two or more licenses that the software is licensed in, do not have compatible terms it will result in licensing obligations that may not be satisfied at the same time (Hammouda, Mikkonen, Oksanen, & Jaaksi, 2010).

A one practical factor that has to be considered is the fact that a programmer rarely writes an application from scratch and the use of existing code is common and accepted

practice. The newly written code and the other programmers code can be linked and used by the complier. The manner how these links are created have a major impact on the interpretation license terms with some of the OSS. Dynamic linking is created if the executable code draws from a library when it is running. Static link, in the other hand, is created when the complier is instructed to bind the code permanently into the executable for a new program. Hence, it can be argued that the static linking creates a situation where the created work "contains or is derived from" the work of the linked code. Where as the dynamic linking does not create a permanent binding with between the code, and the new program only creates transitory copies of the earlier code, when the program is running. Thereby creating a weaker argument of the use of derived work (Henley & Kemp, 2008b).

The strong copyleft licenses pull the derivative work with them to the open source world whether or not it is suitable for the creation of the new software. (Hammouda et al., 2010). This so called viral effect, contaminates the proprietary software which is linked to copyleft licensed software, requiring that proprietary software should be licensed under OSS license as well. This can be the scenario when proprietary device drivers are used with OSS such as GNU/Linux. For example, when graphics card is introduced in GNU/Linux operating system the Linux tends to treat it as a kernel module. This communication between the graphics card's software and the operating system creates a link that is considered by many in OSS community requiring the card's software be licensed under GPL. Even so many the developers of the most advanced graphics cards permit their hardware to be used in GNU/Linux environment but decline to publish the source code of their software (Henley & Kemp, 2008a).

So preparing the codebase to be moved in to repository and to be released as open source is not a simple task. The codebase must be checked for proprietary pieces and they and other artifacts that may cause legal issues have to be removed, causing some additional cost in the process (Gary et al., 2009a).

Table 2: Example Open Source Licenses and their Compatibility

	PHP	Apache	IPL	SSPL	Artistic
GPL	3	3	3	1	3
LGPL	2	2	2	1	2
BSD	1	1	1	1	1

1- Mixing and linking permissible
2- Only dynamic linking is permissible
3- Completely incompatible

(Lokhman et al., 2011)

From the table we can see the compatibilities between different licenses. For example, GPL as a copyleft license can be in this scenario linked just with SSPL, this is due to the fact that GPL's strong terms can not be set with other strong or moderately strong license requirements. Whereas LGPL as a weak-copyleft license can be linked with SSPL it can also be dynamically linked with other licenses in the table (Lokhman et al., 2011).

As stated, the licensing requirement get more complex as multiple components with a different licenses are combined. The issue is twofold as the licenses operate within the scope of a legal system, whereas the software itself is deployed in software architecture's scope. Both scopes, possibly extremely complex and reaching to multiple different levels, have to be taken under consideration when creating OSS (Abdul-Rahman, 2014). The licensing and linking requirements create almost a paradoxical situation as the modularity is both a major attraction for developers, and one of the main challenges related to OSS usage.

Modular system can be said to consists from modules and a platform, allowing independent work on modules in a complex system, and still allowing the modules to work to support the system as a whole. Modularity thereby requires that the modules are compatible with each other, and require architectural design rules so this can be accomplished. Modularity is argued to draw more developers to a project due to the fact that developers are usually interested only in small bits of software. (Baldwin et al., 2005)

Modularity is often associated with option value; in process design the outcome is uncertain and thereby creates 'option-like' properties. When a developer decides to change a design he or she will have the option, but not the necessity, to do something in a new way. If something new is introduced the rationality of other developers is tested, as the new design should be only adopted only if it is better than the old design. Option value and modularity combined create an environment where parts of design can be improved without hampering the functionality of the system as a whole. Hence, experimentalism is welcomed in modular environment with rational developer community (Baldwin et al., 2005).

Developers can learn about the modularity a system and option values simply by working on a codebase directly. *"If changes can be made cleanly by contributing small chunks of code, the codebase architecture is—manifestly—modular, and other things equal, the option values will be high"* (Baldwin et al., 2005). Apple states that the project uses *"small, incremental changes"* as the preferred development model, showcasing at least some level of modularity in the project. Furthermore the company states that the long-term development process may prolong the process as the community will be left without a voice during the development process, indicating some of the option-like properties of their OSS project ("Swift - Contributing," n.d.).

Getting back to the IPR issues related to OSS, furthermore, the inability to comply to the licensing terms can lead to public legal disputes that will hamper company's opportunity to draw developers in the company's projects, the disputes may also have direct effect on company's revenue generation, as complementary product tie-ins have to be restated, for example (Henley & Kemp, 2008b). The licensing infringement is a justifiable concern, due to the complexity of OSS licenses and lack of tools that manage the legality concerns at the architectural level. Lokhman et. al. argue that license related issues will be one of the major challenges for the sustainability of OSS, if they will not be addressed by legal experts and software developers (Lokhman et al., 2011).

For businesses perspective the licensing decision is highly disputed area of research. Colazo and Fang (2005) state that restrictive licenses attract more developers with their ideological approach towards software creation (Colazo & Fang, 2009). Fershtman and Gandal (2007) oppose this view by stating that the mean output per contributor is greater for non-restrictive licenses, whereas restrictive licenses attract more idealistic developers whose goal is to get on the contribution list (Fershtman & Gandal, 2007). Swift is licensed under Apache 2.0 license, which is a permissive OSS license (Swift.org, n.d.). Also the two other Apple's

projects introduced in this this paper are licensed under permissive licenses, WebKit is licensed under BSD license and Darwin is under Apple Public Source License ("Darwin (operating system)," n.d., "WebKit," n.d.). This can be considered a way for Apple to hold the control over its IPR, and unwillingness to get any piece of their software in the scopyleft domain. As stated before this approach is not the most popular among the some of the open source developers (West, 2003).

4.4 Early OSS projects

Linus Torvalds and his associated created GNU/Linux, the first entirely free operating system. The development model for this operating system was revolutionary, as the Linux phenomenon drew vast number of users, debuggers, and programmers in to the development process (Raymond, 1999). This power of numbers is the essential success factor in Open Source Software development; as long as the members of the community are committed, the single developer may progress the development of the software significantly. But here lays also the importance of the licensing, in order to allow large number of developers and users to see the development in the early phases they have to be granted the right to use and modify the software (St. Laurent, 2004).

GNU/Linux was not a new idea in sense of free knowledge sharing and OSS movement can be traced back to the 1960s academic circles, where the attitude was to oppose the restrictive nature of exclusive rights under intellectual rights law. The UNIX operating system was created in the 1970s and 1980s by AT&T employees at Bell Laboratories. In 1985, Richard Stallman established the Free Software Foundation, which would oversee the GNU Project, the foundation would hold the copyright in the software created for it and enforce the licenses. The GNU Project was announced in 1983, and it was planned to be a full operating system and replace UNIX. This project adopted the GNU General Public License (GPL). In 1992, GNU software was combined with a new kernel called Linux to create a complete operating system. The combination was known as GNU/Linux and it was licensed under the GPL (Henley & Kemp, 2008b).

By the late 1990s, some members of the OSS community considered that the anti-IP sentiments of Stallman and others were inhibiting the widespread take up of OSS. In 1998 Bruce Perens and Eric Raymond established the Open Source Initiative (OSI), to promote more wide spread adaptation of OSS. This was done with pragmatic approach, the ethical and philosophical reasons for OSS usage were left to background. The OSI took upon to review

and approve licenses that conformed to Open Source Definition (OSD) (Henley & Kemp, 2008b).

Some of Apple's earlier OSS projects include Darwin and WebKit. Darwin is an open-source operating system released by Apple in 2000. Darwin was released under the Apple Public Source License (APSL), which was accepted by both OSI and FSF, even though FSF do not recommend APSL as it is not entirely compatible with GPL ("Darwin (operating system)," n.d.). WebKit is used to power Apple's Safari web browser and is licensed under BSD-form license, and has been forked from HTML layout engine KHTML. Interestingly enough even though the WebKit is under an open-source license, Apple has decided to trademark this web engine's name, which took effect in 2013 ("WebKit," n.d.).

Apple has had its part in the evolution of copyright law related to software licensing. For example, the Third District U.S. Court held in case Apple Computer, Inc. v Franklin Computer Corp, that "a computer program, whether in object code or source code is a "literature work" and is protected from unauthorized copying, whether from its object or source code version". Proposing that written code is protected as a "literature work" as long it meets other requirements of originality and fixation (Kierkegaard & Adrian, 2010). Another instance, which also showcases Apple's pursuit towards control over its products, was the company's aim to silence a discussion around reverse-engineering Apple's checksum hash encryption. The third-party company, Bulkwiki, is a technology discussion forum and the dispute emerged when Apple suggested that already the talk about reverse-engineering is a violation of U.S. Digital Millenium Copyright Act (DMCA). The operator of the discussion form sued Apple, stating that the company uses copyright law in-order to silence a legitimate discussion (McMillian, 2009).

5 Main reasons for choosing OSS

If the company has the ability and resources to modify the source code the open source solutions will give a company better self-reliance and ability to align the software better with the enterprise specific goals (Koenig, 2006).

One major part of this self-reliance is the knowledge that the software will not become obsolete with the original hardware platform. This can be viable concern with proprietary program that may be discontinued if the publisher no longer considers the project commercially attractive, leaving the software without further maintenance and development. In addition, for the self-reliance of the software and the platforms it can be carried on, the OSS is argued to take time out of the product cycle. The usage of OSS components, particularly in routine tasks, shortens the development phase (Henley & Kemp, 2008b). Decrease in product cycle phase is most likely true for solutions that have to be created in-house, as the ready-made proprietary solutions can be used to perform the routine tasks. The benefit of OSS raises when these routines require customizable solutions, one solution to improve the ability to customization is modularity.

OSS software is considered to be one of the most established examples of open innovation and commons based peer production, having the capability to be a cost reducer or a business value creator. Open innovation is favorable approach in creating business value as the market requires shorter innovation cycles and when the research and development cost grow higher. Peer production is characterized by the decentralized accumulation and exchange of information, and is considered to be superior model for accumulating human skills and knowledge to the creation of information resources (Morgan & Finnegan, 2014).

The innovation generated through OSS project is argued to benefit the innovator and the act of sharing shouldn't reduce that benefit. This is especially true in cases where the communication between the developer and the customer works well: customer feedback leads to improvements in the software benefiting both the customer and the developer. Openness is argued to be beneficial also in cases where the competition can exploit the software, as long as 'co-opetition' is present, if the innovation grows the entire market all companies that are able to sustain their market share will benefit (West & Gallagher, 2006).

In addition of the research that discusses the reasons why companies use OSS, the move from proprietary to opens source software, or open sourcing, is a widely researched topic. Caulkins et al. (2013) state some of the strategic reasons to open source. Firstly, the company

should open source their software if the initial quality is low enough, with this scenario the company could deploy only little resources in research and development. In the the other scenario where the software quality is high, Caulkins et al. (2013) suggest that the company should release the software under proprietary license, and consider the open sourcing only after some optimally determined time or never. This optimal timing depends on several factors like: initial quality, cost of R&D and the costs for adapting the business model (Caulkins et al., 2013).

Apple does not publicly reveal any information about its research activities, but from the company's 10-K can be seen that in 2015 Apple spend 3.5% of its revenue to R&D, which is relatively low when compared to other major technology companies, for example Google's spending was at 15% and Facebooks at 21%. Moreover, Apple has reputation of being deliberate with its R&D expenditures. It is reasonable state that the company advantages from its strong hold over its suppliers, which allows the company to push some of the R&D costs to the suppliers (Satarino, 2015). In somewhat similar manner as Apple has been able to move the R&D to its suppliers, the company notes how its heavily reliable on third-part software development. The company states that developer perception of the company plays a vital role in third-party software attraction (Apple Inc., 2015). Besides developer attraction, the model proposed by Caulkins et al. (2013) does not consider the effects of competition in the open sourcing decision, I consider these factors to be important in the aim of answering my research question, developer attraction is included in the community part of the paper, the competition's affect on open sourcing decision will be discussed next.

Kort and Zaccour (2011) have created a framework for duopoly situation where the companies have to choose whether or not to open source code or not. The start-point in this study is that the two competing companies have proprietary software, which is tied with a complementary product, and the results show that the incentive to open the source code raises if software-sided gets more competitive, and complementary-side is less competitive. Moreover, the results showed that the incentive to open source is high when the competitor has opened their source code. In the other hand, in a monopoly situation the major driver for the open sourcing decision should be the incremental quality that may be achieved by opening the source code (Kort & Zaccour, 2011).

The programming language market is far more fragmented than a duopoly, but never the less, it is interesting see that some of the Apple's competitors have open sourced their programming languages at recent years. It is safe to say that there is a increased focus on the

competition on the software-side can be seen as well. Apple depicts the third-party software attraction as on of the company's major challenges in the future, and the company's service side revenues are increasingly coming from Internet Services, which include App Store (Apple Inc., 2015).

There are some instances when the argument for OSS usage seems to be the gained market position; Netscape open sourced Mozilla browser in order to compete against Microsoft Internet Explorer, leading to situation where Mozilla family of browsers were only slightly behind Microsoft Internet Explorer at a point of time. IBM was even more successful competing against Sun's Netbeans with their Eclipse open source project, shadowing the competitor's project. In other words the software vendor uses OSS to create a market position, which can be utilized later with a proprietary extension for example, or one other option for the vendor is the aim to position the competition out of the market (Gary et al., 2009b). One problem with this approach is that the argument that Gary et al. (2009) made for gaining the market position is simply based on the fact that the OSS product will be superior to proprietary software, with the community made enhancements and marketing name. This can not be true in all instances, for example the marketing name for a company that has been exploiting OSS will not be redeemed with a project that has limited usage for the community. The project that will be able to out shine the competition are likely to be the ones that are fully committed to OSS in their business or at least come out with projects that are helpful for wide range of users, for example.

For Swifts case it is too early to see whether the open sourcing will help the programming language and the company to gain better market position, but the implications to any evidence of this happening are intriguing and therefore the topic is further discussed in the platform strategies section of this paper. Also later in this paper I will show the immediate rise in the popularity of Swift, which does not reveal the gained market position, but will have possible indications of the future of the programming language's position in the market.

One positive aspect usually associated with OSS use is the cost saving related to it. In the development phase the cost reductions come from code reuse and with fewer company employees allocated towards software development. The cost savings carry on to the time after the software is released as the maintaining a widely used program typically keeps accumulating costs, which may typically reach to 40% or over to the deployment costs (Raymond, 1999). Apple is also able to generate cost saving through these OSS concepts. The company states that it has been able to use its Objective-C in the Swift development, and as the *table 1* indicates

some of administrative positions will be appointed to non-Apple community members as the project matures (Swift.org, n.d.). But in the other hand it has to be stated that Swift development process started already in 2010, four years later the project was released to the audience in WWDC and in December 2015 the project was released under an OSS license (Swift.org, n.d.). This means that the project had over 5 years of development dedicated towards it before it was open sourced. Apple does not release detailed information about their R&D costs, but it seems reasonable to say, just by looking the initial development time that the decision, to create new programming language to replace a working programming language, can not be cost saving decision, not at least in the short-term. As the project matures the company is able to appoint community members to all, except project lead positions, inside the community, but this will hardly outweigh the initial development costs dedicated towards the project. In some historical cases it has been clear that the cost savings were not the main driver for open sourcing, as previously noted, improved market position can outweigh the costs related to software creation, IBM spend 40 million dollars on the development of Eclipse before releasing its source code (Lindman et al., 2010).

In addition, the OSS usage is, in some instances, perceived to increase the trustworthiness of the software. The ability to have the opportunity to review the existing code in a software, allows the user to see that there are no hidden features and understand how the application works (Lindman et al., 2009). As stated before Apple has accused of having backdoors in their products, and being very closed system (“Apple ’ s Operating Systems Are Malware,” n.d.). The open sourcing may have effect on publics view of the company, but a piece of software created with Swift will be in most cases proprietary, and thereby the open sourcing of Swift will not by itself change the openness of the actual software created.

6 OSS Community

In this paper OSS community refers to the developer community, which may include both outside developers and company employees who participate in the development process as a part of the community. A community is in the core of all OSS projects, software maintenance is in most cases ongoing process so therefore projects that are unable to attract developers may gradually fade away (West, 2005).

6.1 Developer motivations

OSS community is arguably one of the most important parts of the open sourcing process as a whole, and has direct and indirect implications for OSS business models and reasons why companies use open source software. This part of text is dedicated for the investigation of the reasons why individuals and companies participate into OSS projects, and how companies can improve the community involvement in their OSS projects.

The existing literature covers altruistic, intrinsic, and extrinsic motivational factors for developer participation in OSS projects. Von Hippel and von Krogh note the movement in the area of research from private investment model to the collective action model. The private investment model assumes that the organization grants some rights to the innovation for the innovator, allowing the innovator to pursue private returns through the rights. This model assumes that the investor is better of holding the information regarding the innovation. Where this type of proprietary model aims to hold the knowledge, as any spillover reduces the innovators profits, the collective action model considers that the knowledge can not be feasibly withheld from different consumer groups after the product is launched into the market, therefore the contributors are better off relinquishing control over the product and supplying it to the common pool. The benefit to the society seems to be apparent with collective action projects, but the contributor motivation imposes a challenge, as the the private gains to the contributors are not apparent. (Hippel & Krogh, 2003). The lack of incentive for projects aimed to the common pool begs the question, will they actually benefit the society, this could be the situation in a case where insufficient incentives hamper the level of innovation, due to decreased number of motivated contributors. The more recent research tackles the contributor motivational factors in more detail, and give insights why projects that are aimed to the common pool, or OSS projects, have been so successful.

Hertel, Niedner and Herrmann showed in their research that at least some portion of the people working in OSS communities receive financial compensation for their efforts, the compensation had correlation with the hours spend in an OSS project, but the main motivation participation followed similar path as a voluntary action within a more commonly used social movement (Hertel, Niedner, & Herrmann, 2003). And it is maybe therefore that the early OSS research describes developers who aim for collective gains through volunteerism. A less altruistic view to the phenomena can be also considered from a programmer's view point; code reuse is a way to decrease the steps in the program creation process. Linux, for example, used code and ideas from Minix in the early phases of the development. Eric Raymond states that the aim of coding is not the get an A from effort but from the result, and a trait for a good programmer is constructive laziness (Raymond, 1999). This view makes sense for the project owner, but what motivates the programmers who contribute just few lines of code? Lerner and Tirole argued that career concern incentives, future job opportunities, and ego gratification are the major drives in volunteer participation (Lerner & Tirole, 2003). Hertel et al. include other intrinsic motivational factors like hedonic motivation of accomplishment and pragmatic motivation of improved user experience (Hertel et al., 2003).

Okoli and Oh note the importance of status and respect in the community, and recognize recognition as an incentive for developers to participate in OSS projects. Recognition can be seen as a more tangible motivational factor than mere satisfaction of altruistic contribution. Okoli and Oh suggest that in communities where there is possible to "promote" high number of administrators, this should be done, due to the high correlation between the administrative status and number of contributions. Other factors that have a positive impact on the members' contributions are the interactions between the community members and appropriate level of hierarchy inside the community (Okoli & Oh, 2007). One way to reduce the hierarchy in favorable in the community is democratic recognition and promotion process, something that Apple utilizes in the Swift community on some level. As stated in the *Table 1,* community members have ability to be promoted in to administrator roles, furthermore the code owners are appointed by the community.

According to Bonaccorsi and Rossi, developers value very highly the learning opportunities associated with OSS project participation, noting yet another intrinsic reason to community participation (Bonaccorsi & Rossi, 2006). Even without further research this could be considered to be one highly motivating factor for Swift community, as the nature of project may allow the developers who use the programming language to monetize their skill set with

participating software creation which is aimed for Apple environment. This partly covers also the pragmatic approach to the motivation, the developers who are involved in software development with Swift are improving their own development tools by participating to this OSS project.

6.2 Company participation

West and Gallagher notes that a firm that once has been successful in their interface innovations may become blind to external innovations. As a example they give a comment that Apple engineers gave in the 1980s, "not invented here" was reason to reject external ideas like handheld computers at the time (West & Gallagher, 2006)

As already stated before there are multiple different reasons for companies use OSS in their business. For new software projects OSS is suggested to generate cost savings, reduce development cycle time and improve the software quality. One tool to reach these benefits is by code reuse, this is especially true if mature code base can be used, this means that the reused code is already tested and improved by the previous projects' communities. Time reduction and cost savings are then apparent, as process steps can be decreased to the project specific necessities (Raymond, 1999). It is also important to note that the same code reuse practices impose challenges, as the project lead has to manage the development to avoid duplications, irreconcilable technical considerations, and license infringements (Spinellis & Szyperski, 2004).

The fear of revealing competitive advantages to the competition seems to be decreasing with the increasing amount of software considered to be commoditized. This means that the software in the most cases is not the main source of revenue, and thereby companies should not concentrate in creating these commodity components in-house. Even the small number of differentiating software will eventually gets commoditized, requiring companies to find new ways to drive value out of their software solutions. One way that allows a company to concentrate on the development of differentiating components is through open sourcing the commodity components (Lindman et al., 2009).

Instead of using OSS in a new software project company may consider open sourcing their existing software, for instance in a case where the previously differentiating component has been commoditized. This process of open sourcing, meaning that the source code for a previously proprietary software will be opened, withholds the most of the same underlying

reasons for the company to pursuit OSS in their business. Open sourcing is considered to reduce the development costs and improving the software quality (Bonaccorsi & Rossi, 2006).

It has to be stated that the earlier research shows that the primary driver for producing OSS products for firms are economical and technological, rather than social factors (Bonaccorsi & Rossi, 2003). This may lead to a situation where the differences between community attitudes and company goals hamper the volunteer participation in a OSS projects, some of the reasons that caused tension between the OSS community and major technology companies are discussed later in this paper. One of the ways that earlier literature notes as a way to manage this risk is by communication through employees who are part of the OSS community, which is something that Nokia Networks did with its open source project (Lindman et al., 2009).

7 Measuring programming language popularity

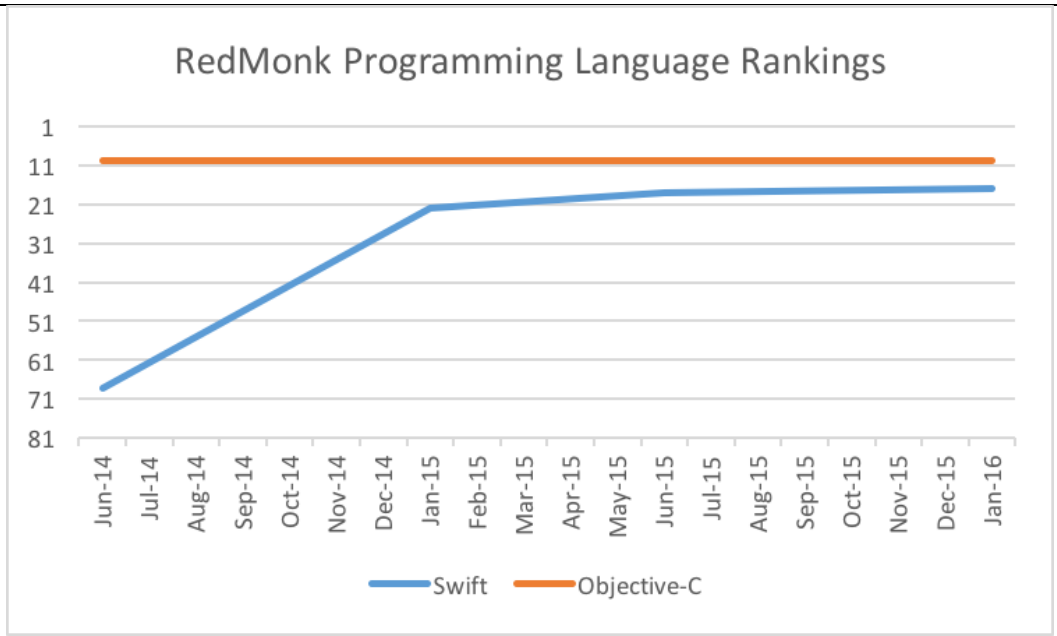
For sake of understanding developer acceptance and adaptation of the language, it is reasonable to have some quantifiable measures of the popularity of Swift. There are number of different ways to evaluate the popularity of a programming language, and here I want to introduce two of these methods, used by two separate companies, in order to showcase the immediate rise of Swift usage during its short lifetime and the change that the open sourcing has done for it. I have to rely on secondary data and evaluate the most appropriate methods, so I can keep the scope of this thesis reasonable.

One of these methods is RedMonk's programming language rankings, the ranking is based on correlation between discussion on Stack Overflow and usage on GiHub, RedMonk claiming that its aim is to extract insights into potential future adoption trends. RedMonk notes that the correlation in their periodical study between GitHub and Stack Overflow rankings have been high, ranging from 0.78 to 0.73. This being said RedMonk does not claim that the rankings are representative of general usage more broadly. GitHub and Stack Overflow have been chosen for their size and ability to draw the needed data for analysis (O'Grady, 2015). GitHub is web based source code hosting facility, or repository, and also the source code of Swift is in this repository ("GitHub," n.d.). Stack Overflow is a question and answer site for programmers, with 4,7 million programmers ("Stack Overflow," n.d.). As stated before the RedMonk programming language rankings do not give statistically viable data for the overall usage of different programming languages. But when I use the results of the same study over the years I will be able to get some insights of how Swift is evolving over its short lifetime and how it compares to the language that is going to replace.

The very first Swift release announcement was done in 2014 at Apple Worldwide Developers Conference(WWDC). RedMonks first programming language rankings after the release was just few weeks later. In June 2014 rankings Swift debuted at 68th position in the rankings. From there the rise on the ranks was stated to be unprecedented. In the next rankings made by RedMonk, in January 2015, Swift had moved up 46 spots to the 22nd position. As RedMonk states the competition gets more saturated going towards the top, but nevertheless the initial jump was something that RedMonk programming language rankings have not seen before. The latest ranking that I am able to include in this paper is from January 2016 and Swift has climbed to the 17th position. The rise has slowed at this point, but not stopped, and RedMonk states that the December's open sourcing was not be felt at this point, but will likely

to show in the near future as developers tend to gravitate towards OSS, and also more third party companies may be interested in investing in a community that they can benefit from (O'Grady, 2015).

Graph 3: Programming Language Rankings



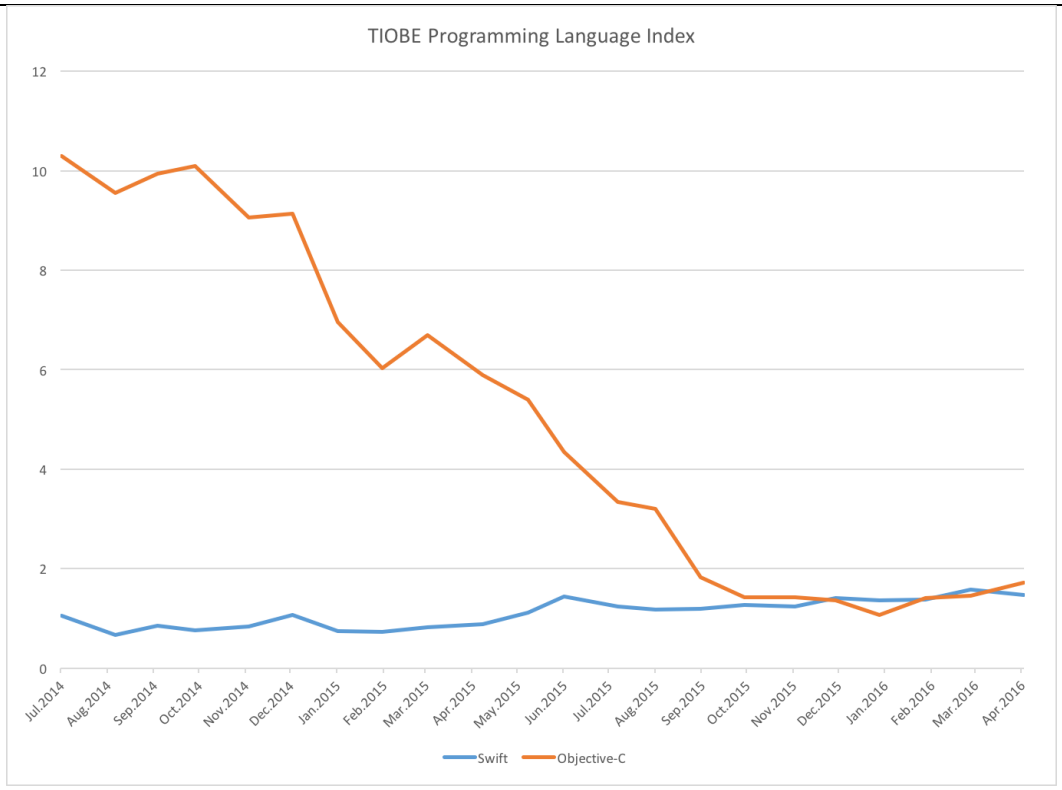
(O'Grady, 2015)

Swift was introduced at Apple's Worldwide Developers Conference in June of 2014, the first ranking recognition made by RedMonk is very shortly after this. The open sourcing of Swift took place in December 3, 2015 ("Swift (programming language)," n.d.). The rise from the initial introduction has been truly unprecedented as RedMonk states, and the language rises from the initial 68th position to the 17th position in under two years (O'Grady, 2015). For the sake of the research the problem comes in with the fact that the open sourcing of the language is hardly visible in this graph as it has so little information after the open sourcing date, and also the rise is naturally slower when getting closer towards the top of the rankings. RedMonk states that this kind of rise on the chart is unusual as the programming languages have usually high switching costs (O'Grady, 2015). The switching cost may be assumed to be lower in this case due to the fact that the language is closely related to the Objective C, but at the same time it is interesting to see that this chart does not show any changes in the predecessor's rankings. This is interesting as it would be safe to assume that the developers drawn to this Apple language would be the same ones that have been working previously with the company's software.

The second view of popularity TIOBE programming community index, which is calculated based on the number of search engine results for queries containing the name of the language. The community index is created and maintained by the TIOBE Company. And in the similar manner as RedMonk's programming language rankings, the index does not calculate the actual usage of a language, but rather the popularity and the conversation surrounding the language. One of the problems with this type of index calculation is the creation of "unused" web-pages that no one reads but still effect the results. TIOBE fights this problem by including 25 different search engines in order to create the index ("TIOBE index," n.d.). Like the RedMonk rankings, TIOBE index helps me to showcase that acceptance of the new language by the developers. Naturally the same problem of the close open sourcing date is present with TIOBE index as well, although the index reaches 3 months further than RedMonk's rankings. The idea to include two separate popularity indicators is mostly done in order to verify the results. TIOBE claims that the number search results may give indications of number of skilled engineers, courses and job worldwide, for the languages included in th index ("TIOBE index," n.d.).

TIOBE index is based on search enginge hits whereas RedMonk's rankings are based on two different sites that can be considered to be devoted to the developers, so the I would put more emphasis on RedMonk's results. The developer community is after all the main concern for a programming language, and sake of this research. In the other hand I assume that the niche nature of programming language should include mostly developers or people who are interested in programming. Some of the differences between these two may result from the fact that the instead of showing the actual rankings TIOBE index, indicates the "market share" among the programming languages included in the index. Some of the requirements that TIOBE sets for the programming languages are: the language has to have its own Wikipedia entry, the language should be Turing complete, and it has to have over over 5,000 hits with for the index appointed search on Google. This further means that there may been some differences between the languages included to these two popularity indicators, but as the main concern here is the change in popularity during Swifts measured lifetime, this fact should have only minor differences in results ("TIOBE index," n.d.).

Graph 4: Programming Language Index by TIOBE



("TIOBE index," n.d.)

In TIOBE's programming language index the Objective-C's fall seems to be more evident while the rise in Swift seems to be subtler. The TIOBE index does not either show any evidence of a clear rise after the open sourcing date of Swift. Both of these indicators are based on some level to the conversation surrounding a programming language, this is something that has to be considered to looking these numbers, as Apple's size and reputation is likely to attract also people who are not actually involved in the software development, in to these channels where the popularity indicators have been collected.

Dahlander and Magnusson state that if there are very similar OSS projects on the market, the new product will have difficult time coming successful if there are no clear differentiating factors. The problems arise especially in developer attraction (Dahlander & Magnusson, 2008). Both RedMonk and TIOBE index have similar results on the fact that Swift is catching up its predecessor at quick pace. These results have to considered cautiously as there are number of factors that may skew the results. Even though there is not yet undisputed evidence that Apple has been able to attract developers to its project. It is clear that the company

has been able to generate conversation around the project and furthermore sustain it to this point. In addition for the reasons previously mention in the community attraction section in this paper, that Apple has been able to utilize in its Swift introduction, one major motivating factor that has to contribute the possible adaptation of Swift is the simple announcement that the programming language will eventually replace Objective-C (Swift.org, n.d.).

8 Open Source Business Models

In traditional business models for proprietary software the company charges customers for the right to use the software. In order for using the software the customer has to accept the software as it is. The company may exclude the customer from the right to use, if the customer is not willing to pay for the piece of software and accept the end-user license agreement. Furthermore the customer does not have ability to access the source code and therefore lacks the ability to modify the software according to personal preferences (Weber, 2004).

OSS, in the other hand grants the right to distribute the software, and thereby eliminates at some level the traditional 'money for right to use' revenue model (Weber, 2004). OSS software can still be sold as customized solutions, for example, but the user is granted with the right to distribute it for free. Customized solutions may be one reason behind OSS sales, but asking money for OSS can have negative affect on the software adaptation, drive away community members, and therefore hamper the underlying benefits of OSS usage (de Laat, 2005). For these reasons companies drive to find other ways in order to generate revenue from their OSS.

Earlier research depicts multiple different OSS business models, which some intervene in their practical adoption. Complimentary product approach seems to include number of OSS business models creating a depiction that gets in a suitable level of detail for sake of this research.

For comparison, I will present a few more detailed OSS business models recognized by earlier literature. With dual-licensing, a company offers software with limitations or requirements or alternatively one with a fee. The premium software option benefits from the existence of the free software, as the possible improvements in the OSS side are introduced in the premium side. Another benefit from the dual licensing is the broad audience that the free software project may enjoy (Koenig, 2006). Optimization can be utilized in a modular environment where some of the software layers can be considered to be commodities and these layers are unprofitable or generate only marginal profit. The key is to utilize the adjacent, interdependent layers of the software, to be optimized in way it generates value (Koenig, 2006). Some other commonly noticed revenue models are: services on top of the OSS, hardware connected to the OSS and proprietary applications that will have to be used with OSS (Rajala, Nissilä, & Westerlund, 2007).

8.1 Complimentary Products

When source code is freely available the firm loses software sale revenue, requiring other paths to be taken in consideration. One way to utilize OSS as a viable business model is with a sale of complementary products or services whose demand would increase if the software were adopted more widely (Caulkins et al., 2013). The complementary products that are attached to the OSS can be hardware, software, consulting services, maintenance contracts, training, certifications, etc. There are varying views on how the OSS business models should be divided for sake of my research the level of detail is sufficient at complimentary product level. The complimentary product view is essentially perceived as increased value for the customer as a product is tied-in with the open source software (Kort & Zaccour, 2011).

Apple utilizes this complementary approach in their business with introduction of Swift. The programming language it self does not generate direct revenue for the company, but the software developed with the programming language, and sold in App Store generates. It can be argued that, the improvement in the programming language has even more indirect value creation impacts, if one may assume that the company is able to generate better quality programming language with OSS, which creates a situation where the developers are able to generate better quality software quicker, increasing not only the sales on App Store but also making the Apple's products more appeling to the end-customers. In the App Store it self the company has decided not to offer "traditional" OSS, the company's view is that the users lack the technical knowledge of complying and installing these software themselves (Abdul-Rahman, 2014). This yet again indicates the level of the control that the company imposes. Furthermore, it would be possible that the introduction of copyleft software could jeopardize the control over some of the existing software infrastructure.

Apple has utilized OSS business models before, the company's Safari browser is an example of firm selling services for 'free' software. Safari isn't open source, but its browser engine WebKit is, this type of partial open sourcing was also used in Darwin project and will be discussed in more detail later on this paper (West, 2005). In 2002, Apple Computer decided to build its own web browser to guarantee that their customers would have one. The result was Safari web browser, which was built upon libraries from the Konqueror web browser developed for the KDE open-source desktop. The decision to built on top of open-source project went in line with the company's OS X strategy, as the Darwin was created to share its modifications of the BSD Unix code. Both projects used open source and contributed changes back to the

community, but Apple decided not to release the remainder of the proprietary code for neither browser or OS (West & Gallagher, 2006). Apple does not only combine the proprietary code when it utilizes OSS but it usually pools the projects with other OSS projects in order to increase the community effect. For example, for Darwin Apple pooled such projects as FreeBSD, NetBSD, and OpenBSD, both receiving and contributing intellectual property among these pooled products (West & Gallagher, 2006).

9 Challenges with OSS

OSS use imposes number of challenges, especially in a situation when it is utilized in a commercial environment. Majority of these challenges are associated with community attraction and steering. Firstly, OSS communities may be wary about commercial companies participating in OSS projects. One way to improve the trust is by deploying the company employees to participate to the projects (Lindman et al., 2009). The tension between the developers create a challenge that the companies utilizing OSS have started tackling. In addition, the OSS use requires ongoing community management efforts in order of being successful. Community management can be associated with some of the more technical development challenges that OSS projects faces, which will be discussed later in this part of the paper.

9.1 Tension between OSS and established tech developers

Many conventional technology companies are encouraging OSS vendors to enter into patent co-operation. Open Invention Network (OIN) is established to defend the OSS movement from patent infringement actions. OIN members include major tech companies, which all agree not to use their Linux-related patents against each other. The OIN has also begun purchasing relevant patents and provides free licenses to all its members. For example, one of the OIN members, IBM, employed in 2007 more than 300 Linux kernel developers and had one of the largest technology patent portfolios. IBM pledged, in 2004, that it will not use its patents against Linux (Henley & Kemp, 2008b).

The community has been wary towards the trend of major players promoting OSS, with the concern that the companies' selfish interests my hamper the principles of open source. As an example Gary et al. (2009) point out the confusion related to Java as Sun has failed to relinquish enough control over it, and on the other spectrum, the fact that Microsoft has showed little enthusiasm towards supporting OSS in the media, but yet supports the CodePlex open source repository (Gary et al., 2009a).

For the technology vendors who can leverage OSS in their hardware sales, the rise in OSS software can arguably seen more favorable, than for the software-only developers. But still some of the companies that rely heavily on their software sales, acknowledge and give room for OSS development. For example, in November 2006 Microsoft entered into an agreement with Novell under which it promised not to assert its patent rights against customers

who have purchased SUSE Linux from Novell, and 2007, the tech-giant entered into two more agreements with Linux distributors, Xandros and Linspire. (Henley & Kemp, 2008b).

The community's wariness is arguably justifiable, because the change from major technology companies considering OSS being a direct competition, to this aim to attract OSS community and utilize the OSS in their businesses has been happening in relatively short time period. As I will later show the open source movement has created increased competitive pressures for some of these traditional technology firms. Microsoft, probably being one of the most software dependent of these, even publicly attacked OSS movement, stating that the GPL license is "viral" by nature and can contaminate existing software solutions (West & Dedrick, 2001).

9.2 OSS development challenges

OSS can be considered to have multiple positive aspects associated with it, like a generally higher demand due to the lack of licensing fees and the ability of customization (Caulkins et al., 2013). But introduction of OSS also creates a unique project leading responsibilities for these these companies, majority of these problems like community attraction, have been touched already in this paper, here I present shortly some of the more technical issues.

A company utilizing OSS has to take several administrative and managerial factors account, in order for creating successful OSS project that is aligned with the enterprise needs. One of administrative tasks that the company has to consider is related to one of the major advantages of OSS projects, the code reuse. The reused code may result in undesired coupling and duplicated code. The coupling may occur when the programmers tie parts of code together without using designing modular interfaces. This may further result in unneeded dead code, but even in when using the components in strictly defined interfaces problems may occur. The dependencies between the components may have effect on the long term sustainability and maintainability of the software, for instance. Some other code reuse issues include the possibility for the lack of backward compatibility, and yet again license infringements (Spinellis & Szyperski, 2004).

Some aspects that can be considered without need for participating to the development of Swift, as Baldwin et al. (2005) suggest as a learning tool for system modularity and option value (Baldwin et al., 2005), are for Apple: the long in-house development period("Swift (programming language)," n.d.), administrative roles ("Community Guidelines," n.d.), and incremental development approach("Swift - Contributing," n.d.).

10 Computing platforms

This part of the text takes a look for the evolution of computing platforms, further explaining the reasons behind the open sourcing decisions for major technology companies. The major focus is on Apple's previous open source projects, and how the company has been able to retain its control over its software while utilizing some of the benefits open source software. The conversation is then turned towards some of the concepts of adaptation and appropriabiliry that are related to the open sourcing of Swift.

The early computer systems on the market were proprietary platforms, where the system manufacturer controlled both the hardware and the software of a system. Two operating systems, Unix and Windows, reduced the control of the traditional manufacturer and shifted the control to the operating system vendors. Later on Unix was tightly related to yet another software revolution, as it worked as a bases for open source operating system Linux. This move from proprietary platforms to open source operating systems indicate the two extremes of appropriability and adaptation. In order to outweigh the cost of development, a proprietary platform appropriates the economic benefits of that platform to it self. But in order of gaining economic benefits the platform has to adopted in adequate level. While a company can pursuit a wider adaptation with open source strategy it also allows the economic returns to be shared with other members in the value chain (West, 2003).

10.1 Apple's Darwin operating system

As stated before proprietary platform consists from the architecture of related standards, which may be controlled by one or multiple sponsoring firms. For computer systems key architectural components normally are a processor and operating system, together allowing the user to run programs on this platform (West, 2003). Here I go through some of the historical events that have created the dynamics of competition in computing platform market. I also include the partly evolution of Apple's operating system creation, which shows how the company has utilized the OSS without loosing the control over their differentiating components.

Proprietary platform firm's ability to generate profits from technological innovations is depended on the firm's ability to control its intellectual property rights (IPR). This protection can be done through legal enforcement, like license requirements, or through practical protection, including trade secrets and implementation strategies. Furthermore for these proprietary platforms the inability to protect the IPR leads to marginal cost pricing and drives

profit margins to zero, requiring the company to aim for other ways to outperform their competition (West, 2003). Some of these ways to compete with similar products are timely market entries and complementary assets that may reinforce the value of the initial product (Teece, 1986). Especially early computing platforms had high switching cost, due to specialized application platform interfaces (APIs), making the market entry difficult for latecomers (West, 2003).

In addition, for Unix with Windows, moving the power more towards the software vendors, Unix evolved into a portable operating system hiding the differences in both hardware and software applications. This evolution was further accelerated with C programming language, which worked as a substitute for hardware-dependent assembly languages. These changes lowered the switching costs, as the operating systems started to share APIs across different hardware vendors. The paths of Unix and Windows differentiated as Windows retained its proprietary APIs under control of a single firm, while Unix moved towards "open systems movement" publishing vendor independent standards. Some of these implementations evolved into open source projects (West, 2003).

In mid-90s, Apple, among with IBM and Sun faced competitive pressures from Microsoft, requiring them to seek for new strategies in order to gain market position. Historically Apple had been the primary competition for Microsoft in 16-bit PC platforms. Where Microsoft benefited technologically advanced hardware vendors, Apple pursued a complete platform strategy, offering tied-in software and hardware. Coming in to the turn of the century the hardware vendors, who had benefited from Microsoft success, wanted to increase their independence and some of these vendors turned towards OSS solutions. For Apple this meant, that the company had to consider for adapting open source strategies, which the company should be able to be align with its core competences, in order to stay competitive in a market where the control over software had switched (West, 2003).

The accusation of NeXT, in 1997, introduced both Unix based operating system and open source code to Apple's long-term platform strategy. The resulting operating system, Darwin, was the central core of Apple's Mac OS X Server and Mac OS X. Apple's decision to utilize OSS components in their core competences were combined with some layers that stayed entirely proprietary. As a result, the Darwin was a partly open operating system, and in fact some of the Apple-controlled technologies prevented the users from using Darwin in other than Apple hardware. In addition the company chose to utilize BSD-style licensing, which would

allow the company to use publicly provided modifications to be placed under a proprietary license (West, 2003).

West states that Apple was able to grasp the best of all possible worlds, with its open source strategy. The company was able to leverage BSD communities in the development, the low-level documentation freed company's efforts on application software support, and the company was still able to retain the differentiation in the traditional areas of its business. At the same time the decision to open the technology just partly could have made the company at some levels less attractive for the developer community, decreasing the underlying benefits of OSS use (West, 2003).

11 Discussion

As the historical view on the evolution of Darwin operating system points out, the competitive pressure seemed to be one of the forces pushing Apple to utilize open source in its primary operating system. Even with the introduction of OSS components in the core of the company's software solutions Apple managed to hold the control over its operating system by opening just parts of the project. This seems to be, on some levels, similar case as it is now with Swift. The company open sourced a programming language which main function will be third-party's ability to create proprietary software. This proprietary software will mostly be functional on the company's interfaces. Also, for the control, the actual software creation will likely face the already established screening processes from Apple, in which it does not except, for example, OSS software for its application delivery platforms (Abdul-Rahman, 2014). This leaves the end-user again unable to see the underlying functionalities of the software, and does not give the ability to make modifications to the software. Furthermore, a level of competitive pressure seems to be present as some of Apple's competition have lately released their own open source programming language projects (Derballa, 2015).

Even while open source programming languages have been around in general use for a long time, the creation of these new languages create intriguing opportunities for further research, especially when the current projects, like Swift, mature. Further research could be expanded to include also multi-sided platform effects that are associated with platform-mediated networks, this could showcase the possible revenue generation that a company may attain through open sourcing (Tuunainen & Tuunanen, 2011). This would be especially intriguing when some of the programming language arguments are considered: if the open sourcing results in improved developer experience and thereby generates more, better quality, software to the platform, it would result in increased revenue generation for a company from multiple different channels.

12 Conclusions

The aim of this paper was to answer the question: why Apple opened the source code of their programming language Swift. Open Sourcing has received increasing attention from major technology companies in the very recent past (Derballa, 2015). Many of these companies have been leaning towards proprietary solutions and therefore this phenomenon creates an interesting research topic. The aim of the research is to reveal some the underlying reasons for open sourcing. The case company, Apple is arguably one of the more interesting ones among these technology companies, because even though the company has utilized OSS before Swift it has also been accused for abusing OSS projects (West, 2005), and further more as a company it has been considered to "own its customers" (Montgomerie & Roscoe, 2013).

A case study was chosen as the method to answer my research question about the phenomena. The aim was to create a fluent conversation with the earlier literature and publicly available data in order to create a holistic picture of open sourcing process in general and also reveal some of the underlying reasons for the case company's open sourcing decisions. The earlier literature included a vast amount of history of OSS in order of pointing out some of the similarities between the case company's previous and current OSS projects, as well the some of the steps that the OSS movement has taken so far.

According to Montgomerie and Roscoe (2013) Apple's business model is to drive consumers into its ecosystem and hold them there with high switching costs. Moreover, the company maintains this multi-channel platform integration with legal and technological means reaching the control from customers to all the way to suppliers and manufacturers (Montgomerie & Roscoe, 2013). Looking at the history of the company, it has been able to maintain similar kind of strong control over its software solutions as well, these include its previous open sourcing projects, which the company has utilized partly in its operating system and web browser (West, 2003). The current open source project of the company, Swift, is created in order to replace programming language Objective C, and thereby planned to take over the duties of primary language for software creation for all Apple's interfaces.

Apple states that now when Swift is open source, all software written in this language will be easier maintained and kept up to date (Timmer, 2016). An other company statement reveals the perceived importance of third-party software attraction is one of its major

challenges going in to the future, and that increasing amount of company's revenues are coming from "Internet services" (Apple Inc., 2015). Some outsiders consider the open sourcing decision to be a way for the company to lower the running and maintenance cost with utilization of Linux servers (Tofel, 2015). Earlier research depicts instances where the open source software use has resulted in improved market position, and that there is a pressure to follow the competitions' lead to move in open source solutions in order to stay competitive (Gary et al., 2009b).

The cost savings related to OSS use is a widely argued and researched topic, but for Swift, a programming language that has been in development inside the company for over five years, and is set to replace a still functioning programming language, it seems to be almost safe to say that the cost savings is hardly the main concern for the company, at least not in the short-term. Another perceived positive impact from OSS use is the trustworthiness gained through ability to let the end-users know that the software does not include hidden features, and how the application works (Lindman et al., 2009). Apple's programming language is now in the open source environment, but does not change the fact that the software created with the programming language stays proprietary, leaving the gained trustworthiness at very best in a conceptual level, which may be gained through the media coverage over the open sourcing decision.

Apple showcases some of the preferred development models associated with OSS implementation, which are depicted in the earlier literature, and should increase the company's possibilities to create a successful OSS project. These implementation practices include Apple's preferred development model ("Swift - Contributing," n.d.), the administrative roles inside the community for developer attraction ("Community Guidelines," n.d.; Okoli & Oh, 2007), and the learning opportunities in the development process, as well as pragmatic motivational factors (Bonaccorsi & Rossi, 2006; Hertel et al., 2003).

In addition, this research took steps in order of finding indicators of the acceptance that the language has received. Two programming language popularity indicators were introduced but the close proximity to the open sourcing date didn't accumulate indicators of evidence that the open sourcing would have significant impact on the popularity. Instead the popularity rankings showed that the language has been enjoying rapid growth in popularity since its initial introduction date and it is catching up its predecessor in a quick pace. Possibly indicating that similar OSS project may be successful in attracting developers without clear differentiating factors.

A large portion of this paper was dedicated towards the history of different kind of open sourcing projects. Especially Apple has been able to sustain some of the same characteristics of its OSS usage since first introducing it partly in the company's operating system. The company has been able to retain the control over its software solutions while enjoying at least at some levels of the benefits of OSS use. This project seems to follow similar pattern, as the company open sourced a programming language which is mainly dedicated towards proprietary software creation, yet again allowing the company to maintain a strong hold over its computing infrastructure, but in the process, if successful, Apple may be able to attract more developers, and thereby create multiple different channels for additional complimentary product value generation.

13 References

Abdul-Rahman, S. (2014). The Effects of Open Source License Properties, (April).

Aksulu, A., & Wade, M. (2010). A Comprehensive Review and Synthesis of Open Source Research. *Journal of the Association of Information Systems*, *11*(11/12), 576–656. Retrieved from http://www.scopus.com/inward/record.url?eid=2-s2.0-78650385795&partnerID=40&md5=a63849db3968e4a9fb915e162319d011

Apple ' s Operating Systems Are Malware. (n.d.). Retrieved May 15, 2016, from http://www.gnu.org/proprietary/malware-apple.en.html

Apple Inc. (n.d.). Retrieved June 15, 2016, from https://en.wikipedia.org/wiki/Apple_Inc.

Apple Inc. (2014). The Swift Programming Language. http://doi.org/10.1016/S0022-3913(12)00047-9

Apple Inc. (2015). Form 10-K.

Baldwin, C. Y., Clark, K. B., David, P., Bessen, J., Henkel, J., Aoki, M., … Von, E. (2005). The Architecture of Participation : Does Code Architecture Mitigate Free Riding in the Open Source Development Model ?

Bohon, C. (2016). Apple ' s Swift programming language : The smart person ' s guide. Retrieved April 8, 2016, from http://www.techrepublic.com/article/apples-swift-programming-language-the-smart-persons-guide/

Bonaccorsi, A., & Rossi, C. (2003). Why Open Source software can succeed. *Research Policy*, *32*(7), 1243–1258. http://doi.org/10.1016/S0048-7333(03)00051-9

Bonaccorsi, A., & Rossi, C. (2006). Comparing motivations of individual programmers and firms to take part in the open source movement: From community to business. *Knowledge, Technology & Policy*, *6*(4), 1–6. http://doi.org/10.1007/s12130-006-1003-9

Caulkins, J. P., Feichtinger, G., Grass, D., Hartl, R. F., Kort, P. M., & Seidl, A. (2013). When to make proprietary software open source. *Journal of Economic Dynamics and Control*, *37*(6), 1182–1194. http://doi.org/10.1016/j.jedc.2013.02.009

Chen, L. (2015). The world's largest tech companies: Apple beats Samsung, Microsoft, Google. Retrieved June 5, 2016, from http://www.forbes.com/sites/liyanchen/2015/05/11/the-worlds-largest-tech-companies-apple-beats-samsung-microsoft-google/#3d67ee37415a

Colazo, J., & Fang, Y. (2009). Impact of License Choice on Open Source Software Development Activity. *Journal of the American Society for Information Science and Technology*. http://doi.org/10.1002/asi

Community Guidelines. (n.d.). Retrieved April 24, 2016, from https://swift.org/community/#community-structure

Dahlander, L., & Magnusson, M. (2008). How do Firms Make Use of Open Source Communities? *Long Range Planning*, *41*(6), 629–649. http://doi.org/10.1016/j.lrp.2008.09.003

Darwin (operating system). (n.d.). Retrieved May 18, 2016, from https://en.wikipedia.org/wiki/Darwin_(operating_system)

de Laat, P. B. (2005). Copyright or copyleft? *Research Policy*, *34*(10), 1511–1532. http://doi.org/10.1016/j.respol.2005.07.003

Derballa, B. (2015). Open Sourcing Is No Longer Optional, Not Even For Apple. Retrieved February 5, 2016, from http://www.wired.com/2015/06/open-sourcing-no-longer-optional-not-even-apple/

Edmondson, A. M. Y. C., & McManus, S. E. (2007). METHODOLOGICAL FIT IN MANAGEMENT, *32*(4), 1155–1179.

Eisenhardt, K. M., & Graebner, M. E. (2007). THEORY BUILDING FROM CASES : OPPORTUNITIES AND CHALLENGES, *50*(1), 25–32.

Fershtman, C., & Gandal, N. (2007). Open source software: Motivation and restrictive licensing. *International Economics and Economic Policy*, *4*(2), 209–225. http://doi.org/10.1007/s10368-007-0086-4

Fortt, J. (2007). Will Apple ' s control issues hurt the company ? Retrieved April 18, 2016, from http://fortune.com/2007/11/08/will-apples-control-issues-hurt-the-company/

Gary, K., Koehnemann, H., Blakley, J., Goar, C., Mann, H., & Kagan, A. (2009a). 2009 Sixth International Conference on Information Technology : New Generations A Case Study : Open Source Community and the Commercial Enterprise. http://doi.org/10.1109/ITNG.2009.313

Gary, K., Koehnemann, H., Blakley, J., Goar, C., Mann, H., & Kagan, A. (2009b). A case study: Open source community and the commercial enterprise. *ITNG 2009 - 6th International Conference on Information Technology: New Generations*, 940–945. http://doi.org/10.1109/ITNG.2009.313

GitHub. (n.d.). http://doi.org/10.1515/9781400821334.toc

Go (programming language). (n.d.). Retrieved May 17, 2016, from https://en.wikipedia.org/wiki/Go_(programming_language)

Hammouda, I., Mikkonen, T., Oksanen, V., & Jaaksi, A. (2010). Open source legality patterns: architectural design decisions motivated by legal concerns. *MindTrek 2010, October 6th-8th 2010 Tampere, Finland*, 207–214. http://doi.org/http://doi.acm.org/10.1145/1930488.1930533

Henley, M., & Kemp, R. (2008a). Open Source Software : An introduction, *24*, 77–85. http://doi.org/10.1016/j.clsr.2007.11.003

Henley, M., & Kemp, R. (2008b). Open Source Software: An introduction. *Computer Law & Security Review*, *24*(1), 77–85. http://doi.org/10.1016/j.clsr.2007.11.003

Hertel, G., Niedner, S., & Herrmann, S. (2003). Motivation of software developers in open source projects: An Internet-based survey of contributors to the Linux kernel. *Research Policy*, *32*(7), 1159–1177. http://doi.org/10.1016/S0048-7333(03)00047-7

Hippel, E. von, & Krogh, G. von. (2003). Open Source Software and the "Private-Collective" Innovation Model: Issues for Organization Science. *Organization Science*, *14*(2), 209–223. http://doi.org/10.1287/orsc.14.2.209.14992

Kierkegaard, P., & Adrian, A. (2010). Wikitopia: Balancing intellectual property rights within open source research databases. *Computer Law & Security Review*, *26*(5), 502–519. http://doi.org/10.1016/j.clsr.2010.07.008

Koenig, J. (2006). Seven Open Source Business Strategies for Competitive Advantage. *IT Managers Journal*, 1–6. Retrieved from http://riseforth.com/pdf/seven_open_source_business_strategies.pdf

Kort, P. M., & Zaccour, G. (2011). When Should a Firm Open its Source Code : A Strategic Analysis, *20*(6), 877–888.

Lerner, J., & Tirole, J. (2003). Some Simple Economics of Open Source. *The Journal of Industrial Economics*, *50*(2), 197–234. http://doi.org/10.1111/1467-6451.00174

Lindman, J., Juutilainen, J.-P., & Rossi, M. (2009). Beyond the Business Model: Incentives for Organizations to Publish Software Source Code, 47–56.

Lindman, J., Paajanen, A., & Rossi, M. (2010). Choosing an Open Source Software License in Commercial Context: A Managerial Perspective. *2010 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, 237–244. http://doi.org/10.1109/SEAA.2010.26

Lokhman, A., Abdul-Rahman, S., Luoto, A., & Hammouda, I. (2011). Managing Open Source Legality Concerns - A sustainability Catalyst.

McMillian, R. (2009). Apple Is Sued After Pressuring Open-source ITunes Project. Retrieved April 18, 2016, from http://www.pcworld.com/article/163909/article.html

Montgomerie, J., & Roscoe, S. (2013). Owning the consumer — Getting to the core of the Apple business model. *Accounting Forum*, *37*(4), 290–299. http://doi.org/10.1016/j.accfor.2013.06.003

Morgan, L., & Finnegan, P. (2014). Beyond free software: An exploration of the business value of strategic open source. *Journal of Strategic Information Systems*, *23*(3), 226–238. http://doi.org/10.1016/j.jsis.2014.07.001

O'Grady, S. (2015). The RedMonk Programming Language Rankings: January 2015 – tecosystems. Retrieved April 25, 2016, from http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/

Okoli, C., & Oh, W. (2007). Investigating recognition-based performance in an open content community: A social capital perspective. *Information and Management*, *44*(3), 240–252. http://doi.org/10.1016/j.im.2006.12.007

Rajala, R., Nissilä, J., & Westerlund, M. (2007). *Revenue Models in the Open Source Software Business*. *Handbook of Research on Open Source Software: Technological, Economic, and Social Perspectives (St. Amant, K. and Still, B., Eds.; 2007) [Book review] IEEE Transactions on Professional Communication* (Vol. 52). http://doi.org/10.1109/TPC.2008.2007877

Raymond, E. S. (1999). The Cathedral and the Bazaar. *[Online]. Available: http://en.wikipedia.org/wiki/The_Cathedral_and_the_Bazaar*, 1–40.

Russolillo, B. S., & Cheng, J. (2012). Apple ' s Stock-Market Sway. Retrieved May 15, 2016, from http://www.wsj.com/articles/SB10001424052702303990604577366332861232436

Satarino, A. (2015). Apple Is Getting More Bang for Its R & D. Retrieved May 9, 2016, from http://www.bloomberg.com/news/articles/2015-11-30/apple-gets-more-bang-for-its-r-d-buck

Shaikh, M. (2015). Negotiating open source software adoption in the UK public sector.

Government Information Quarterly. http://doi.org/10.1016/j.giq.2015.11.001

Siggelkow, N. (2007). PERSUASION WITH CASE STUDIES, *50*(1), 20–24.

Spinellis, D., & Szyperski, C. (2004). How is Open Source Software Affecting Software Development? *IEEE Software*. Retrieved from \url{C:\Dokumente\nund\nEinstellungen\Administrator\Desktop\Promotion\Literaturquellen\nOS\Open\nSource\nin\nEndnote\SpSz04\n-\nHow\nIs\nOSS\naffecting\nsoftware\ndevelopment.pdf}

St. Laurent, A. M. S. (2004). Understanding Open Source and Free Software Licensing. *Ariadne*, 193. http://doi.org/10.1093/toxsci/kft290

Stack Overflow. (n.d.). Retrieved May 18, 2016, from http://stackoverflow.com/about

Swift - Contributing. (n.d.). Retrieved May 4, 2016, from https://swift.org/contributing/#contributing-code

Swift (programming language). (n.d.). Retrieved May 18, 2016, from https://en.wikipedia.org/wiki/Lisp_(programming_language)

Swift.org. (n.d.). About Swift. Retrieved January 18, 2016, from https://swift.org/about/#swiftorg-and-open-source

Teece, D. J. (1986). Profiling from technological innovation: implications for integration, collaboration, licencing and public policy. *Research Policy*, *15*(February), 285–305.

Timmer, J. (2016). A fast look at Swift, Apple's new programming language. Retrieved February 10, 2016, from http://arstechnica.com/apple/2014/06/a-fast-look-at-swift-apples-new-programming-language/

TIOBE index. (n.d.). Retrieved January 1, 2016, from https://en.wikipedia.org/wiki/TIOBE_index

Tofel, K. (2015). After Apple open sources it, IBM puts Swift programming in the cloud. Retrieved April 8, 2016, from http://www.zdnet.com/article/after-apple-open-sources-it-ibm-puts-swift-in-the-cloud/

Tuunainen, V. K., & Tuunanen, T. (2011). IISIn - A model for analyzing ICT intensive service innovations in n-sided markets. *Proceedings of the Annual Hawaii International Conference on System Sciences*. http://doi.org/10.1109/HICSS.2011.234

Välimäki, M. (2005). *The Rise of Open Source Licensing A Challenge to the Use of Intellectual Property in the Software Industry*. Retrieved from http://pub.turre.com/openbook_valimaki.pdf

W. Gibb Dyer, J. A. L. W. (1991). Better Stories , Not Better Constructs , to Generate Better Theory : A Rejoinder to Eisenhardt Author (s): W . Gibb Dyer , Jr . and Alan L . Wilkins Source : The Academy of Management Review , Vol . 16 , No . 3 (Jul ., 1991), pp . 613-619 Published by, *16*(3), 613–619.

Weber, S. (2004). *The Success of Open Source*. http://doi.org/10.1086/510004

WebKit. (n.d.). http://doi.org/10.1515/9781400821334.toc

West, J. (2003). How open is open enough ? Melding proprietary and open source platform strategies, *32*, 1259–1285. http://doi.org/10.1016/S0048-7333(03)00052-0

West, J. (2005). Contrasting Community Building in Sponsored and Community Founded

Open Source Projects, *00*(C), 1–10.

West, J., & Dedrick, J. (2001). Open source standardization: The rise of linux in the network era. *Knowledge, Technology & Policy*, *14*(2), 88–112. http://doi.org/10.1007/s12130-001-1008-3

West, J., & Gallagher, S. (2006). Challenges of Open Innovation : The Paradox of Firm Investment in Open Source Software Challenges of Open Innovation : The Paradox of Firm Investment in Open Source Software, *3*, 319–331. http://doi.org/10.1111/j.1467-9310.2006.00436