MASTER'S THESIS | LUND UNIVERSITY 2015

Reaching across – Managing variants of one application on multiple platforms

Alexander Haraldsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-44

Reaching across - managing variants of one application on multiple platforms

Alexander Haraldsson
ada09aha@student.lu.se

September 2015

Master Thesis at the department of Computer Science, Lund University

Supervisor: Lars Bendix, bendix@cs.lth.se
Examiner: Ulf Asklund, ulf.asklund@cs.lth.se
Contact person at Kubicom: Per Lundgren, per.lundgren@kubicom.se

Abstract

The number of platforms to support in today's software projects are many and there are a wide range of differences to consider. There are tons of programming languages on the market and each platform, both mobile and desktop, have different preferences on how to develop applications. This do often result in multiple applications, similar to the end user but different to the developers. The same functionality has to be developed and maintained in multiple versions of the application in different ways.
To solve these issues there is a need to think of the applications and platform in a new way. They have to be unified and commonalities has to be found or made. New application structures and tools are also needed to keep the platforms in sync. When developing those concepts each platform's flexibility must be protected. Each platform has it's own advantages and features like a GPS and camera and they have to be available to build a competitive product.
This report concludes that web technology such as HTML5, JavaScript and CSS is a promising way to introduce common parts in the application. This helps to manage the platforms in one way and reduces the differences. To retain the platform specifics, language bridges are used to directly communicate with the native platform from the web parts. This enables the full strength of each platform and makes the solution a fully featured competitor to native applications.

Keywords:
Cross platform development, Variant management, Double maintenance, Platform features, Web technology, Cross compilation, Configuration Management, Variation point, Product derivation, Reusability, Software Product Line, Product Family

Table of contents

1. Introduction
2. Background
    2.1. Current setup
    2.2. Tools and methods
    2.3. Needs and current issues
    2.4. Method
    2.5. Use cases
    2.6. Variant management theory
3. Analysis
    3.1. Variant perspectives and scope
    3.2. Interviews and code analysis
        3.2.1. Method
        3.2.2. Result
    3.3. Managing the problems
        3.3.1. Finding commonalities
        3.3.2. Working with differences
        3.3.3. Conclusion
    3.4. Summary of requirements
4. Design
    4.1. Finding commonalities
        4.1.1. Cross compiled
        4.1.2. Web application
        4.1.3. Hybrid
        4.1.4. Conclusion
    4.2. Keeping commonalities in sync
        4.2.1. Branching
        4.2.2. Library organization
        4.2.3. Build time instantiation
        4.2.4. Conclusion
    4.3. Working with differences
        4.3.1. Organization and identification
        4.3.2. Low level variants
        4.3.3. High level variants
    4.4. Proposed setup
5. Prototype

- 5.1. Scope
- 5.2. Prototype implementation
  - 5.2.1. Commonalities
  - 5.2.2. Low level variants
  - 5.2.3. High level variants
  - 5.2.4. Library organization
- 5.3. Conclusion
  - 5.3.1. Reflections
  - 5.3.2. Guidelines

6. Discussion
- 6.1. Related work
- 6.2. Limitations
- 6.3. Future work
- 6.4. Reflections

7. Conclusions

1. Introduction

Previously most applications were developed for traditional PCs but as the mobile market grow the demand to support mobile devices increases [16]. Thereby managers and new projects has to consider the support for many platforms. Today there is a wide range of different devices and underlying operating systems on the market. This is nice for the consumers as they can choose the most suitable platform for their needs and personality. Many platforms offer their own specialities with large screens, cameras, location tracking or energy efficiency among others.
But this comes with a negative side as well. The diversity of the platforms make it much harder for developers of the applications. Many parts of the application has to be implemented several times in different programming languages and for different implementations of a specification. This will give the developers a hard time managing the double work and maintenance problem.

This report was developed in collaboration with Kubicom AB as a step to optimize the development process and modernize the code base of their business application. Kubicom is a machine and transportation conveyer in the construction industry. They help customers and suppliers to communicate and cooperate and at the same time widen the market and make new connections.
Kubicom is involved through the whole process starting from when the customer plan his project and orders the resources until it’s time for the supplier to report the work completed and send the order for billing. On top of that, Kubicom’s software offers a wide range of tools to organize and optimize the workflow.
For more information see kubicom.se.

Problem description

The core problem lies in the number of code bases that has to be developed and maintained to develop and deploy an application on multiple target platforms. The traditional way of developing cross platform applications is to create one application for each platform. A platform in this report is defined by a device and its operating system. For example a Samsung device with Android or a Dell desktop computer with Windows. Each platform usually has one or a few programming languages that are supported for application development. This means that if the goal is that the application should look and behave in the same way there has to exist several different implementations of it.
This introduce a double work and maintenance problem which will grow for each new platform to support. Code would not only have to be written several times but if it updates in one place the same change should propagate to all other implementations of the application. In contrast to the traditional double maintenance problem described by Babich [2] this problem has another dimension as the platforms nature are often very different even in terms of programming language.
To avoid large extra expenses and hard to maintain applications for each and every platform there is a need to optimize and share components of applications across multiple platforms and find parts and processes in common.
If this could be accomplished IT departments could put more of the resources into developing new functionality and maintaining the old. On the other hand there is also a need to differentiate the platforms and make use of each platform's strengths and

features. The sharable components still shouldn't imply restrictions on what the application can achieve regarding platform specific features like GPS, camera and file system.

Goals and scope

The overall goal of this report is to find a suitable method to work with cross platform application development and use Kubicom as a real world reference. This will cause the report to move in the direction of Kubicom's needs but will still take a step back and watch it from a wider perspective as well. General solutions will be proposed and discussed but the chosen one will be motivated by Kubicom's needs. Kubicom's platform applications are in a great part very similar to the end user; this is especially true in each of the platform groups: mobile and desktop devices. The similarities is what can make the new process powerful, by making them not only similar to the end user but also to the developers. Therefore this is the main focus, although differences should also be taken care of to preserve the strengths of each platform.
The solution will be discussed with an open mind and no predetermined or preferable way of accomplishing it. One of the most important goals is to reduce the double work and maintenance involved with the variations of the platforms. In turn, this will lead to a more efficient development and maintenance process.
An important requirement is that the developed process should not limit the functionality or the strengths of the platforms in any considerable way. It is common that platforms offer similar functionality, like location tracking and camera access, with only differing interfaces or in some cases with extra features.

The main questions to answer are:

- What methods and tools can be used to ease cross platform application development and reduce the double work and maintenance in comparison to building independent applications?

- What components can be shared and what needs to be kept separated among the platforms?

- How does the proposed methods and tools affect/limit the functionality offered by the application?

Audience

The target group of this report is anyone with an interest in configuration management in general and variant management and cross platform development in particular. The report assumes basic knowledge of configuration management but will explain relevant, more specific parts.

Disposition

The structure of this report is divided into 7 chapters each with its own goal.
The first chapter gives a short introduction to the report by presenting Kubicom, the problem and the focus of the report.

The second chapter will describe the background covering Kubicom’s current situation with application suite and seen problems. It will also introduce related variant management topics and describe their relation to the seen problems.
The third chapter analyses Kubicom’s current situation to find their problems and to discuss the root causes of general cross platform application development issues.
The fourth chapter uses the results from the third analysis chapter to express the problems and discuss possible solutions.
The fifth chapter will put the solution found in the design chapter at test. This will be done by a small example from Kubicom.
The sixth chapter will discuss the contributions of this report and compare it to other work performed.
At last the seventh chapter will get back to the root problem and summarize the results and conclusions from the report.

2. Background

The purpose of this chapter is to give an understanding of the case at hand and the core problems targeted for further analysis. It will also work as a reference for later chapters. The focus will be on Kubicom's perspective but will in most cases be general situations easily applied to other contexts.
This chapter will introduce Kubicom and their current application suite. It will discuss why all platforms are needed and when and analyze what problems this introduces in the development phase. The problems will focus on the cross platform application development and analyze what issues Kubicom's current workflow means.
The first step will be to analyse current applications, tools and working processes. This will give an insight into what Kubicom's situation look like and what needs to be considered in further analysis.
After the current setup has been presented, Kubicom's perceived problems will be introduced; this will be followed by a quick discussion of what is causing them and how they may be solved. Next, the method used by this report and their purpose will be presented.
To create a true feeling for the problems and give real world examples of when problems occur and why there will also be a section giving some use cases. Those are especially important to find where the platforms differ and need special care. The result will be used as reference for later chapters in both the analysis and discussion.
The last section will give some background theory of the root problem and how it is currently viewed in the configuration management field. The reason for this is to get an understanding of how similar problems look today and what the solutions are.

2.1. Current setup

The goal here is to give an overview of a situation of cross platform development that becomes more and more common. The content will be used to understand the extent of the problem. Kubicom's current setup consist of a number of separate applications for different platforms. This section will give a description of which they are and how they are used.

Kubicom currently make their system available as three separate applications for different platforms. The applications are built for:

- IOS
- Android
- Desktop browsers

There are also variants of those platforms as the IOS app support both iPhones and iPads for example and they are different as well; mainly in user interface. In figure 1, the level of devices can be seen. At the top level there are two groups of devices, desktop and mobile. Some differences between them are wanted, such as user interface and some features. The next level are the platforms supported and the bottom level contains the applications for those.

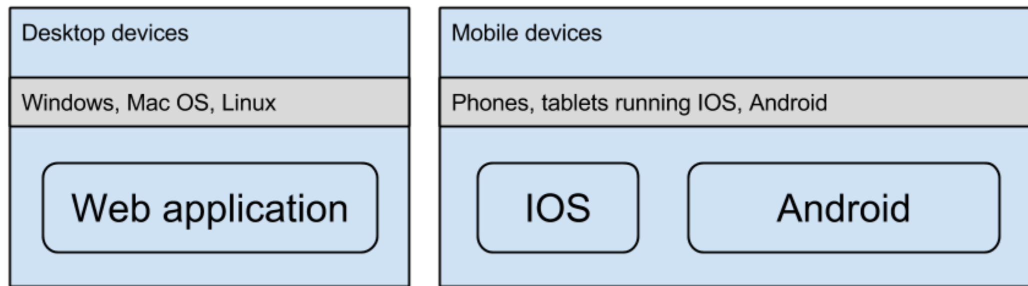

Figure 1. Current application suite

Because the customer base is very wide there is a need to support alternative devices with the same purpose. Therefore it cannot be a choice between IOS and Android for example; both are needed. There is also a demand to increase the number of supported platforms to make the application more compatible and to offer the user to work with their own platform of choice. In many cases companies already have an overall company strategy for which devices to use. Therefore it is important to have support for at least the most common platforms. For example there is a demand for Windows Phone support. If adding new devices would be easier and not cause persistent extra development cost there would be many more platforms on the supported list.

The applications have several different types of variants. Some of them already got good solutions while others would need better support. We'll discuss the variants here and will then focus on the ones that most require new solutions.
Much of the functionality presented to the end user is aimed to be similar between the applications but parts of it differs because of the suitability and underlying OS support. The applications for the two main groups, mobile and desktop platforms, are especially aimed to share functionality and feeling in the group. The reason is that each group are supposed to be used in the same way. It is also preferable to easily be able to move between devices. Especially because it's common for suppliers to work with both smartphones and tablets.

The desktop application is designed for larger screens and can fit more information in one view. It is also customized for the availability of keyboard and mouse which makes it easier to input longer texts and more information.
The desktop application lacks some features that both Android and IOS has. Those features is mostly lacking due to the nature of the desktop and include geolocation tracking through GPS and WIFI, taking photos and push notification support. Push notifications are getting more and more popular though and at the moment both Chrome and Safari supports it. The notifications are a feature that Kubicom would prefer in browsers too but would like a uniform way to manage it across browsers/platforms.
Today users can be notified about new available orders, when information updates and when an order has been assigned and completed. Users are also notified when they get new messages or get project invitations.

Another feature that is missing in the browser app is the possibility to sign an order. When all work is done the customer is supposed to sign the work performed on screen to accept it.
The mobile devices main focus is for working with active orders. Due to the lacking mobility and missing GPS in most desktop computers they are rarely used; then the mobile devices are better choices. Users start the order when they begin their work and then the app tracks both where they are and for how long they work. During that time the user is required to bring the device with him. When the task is done, the user fills in what has been performed and saves it to make it available to the customer.
The desktop application is focused on managing information about orders and projects and plan new work to be done. It is also used to follow up on the work that has been done.
This adds a few modules with statistics and extended information management possibilities that is not available in the mobile app.

2.2. Tools and methods

Kubicom does currently perform much of the configuration management manually without tool support. The variants have been introduced gradually and there hasn't been a focus on optimizing the workflow.
Kubicom uses GIT as revision control system and has three different repositories. One for each platform variant: desktop web app, IOS app and Android app. There is no connection between the repositories and the synchronization is all done manually. This is especially important for the mobile apps where the major parts of the applications are the same.
The strategy when releasing new updates has been to finish the web application first and then implement the same features on first IOS and at last Android. Because of the unclear relation between the platforms this has been hard work and required much care.
The number of developers working with the software varies; there is a small team working continuously with the software but as soon as there are major new features to be rolled out more developers are added from a consultant company.

2.3. Needs and current issues

To understand the problems of working with cross platform development we will now look into what Kubicom perceive as problems today and what the causes for the problems are. The core problem from their point of view is that they feel like they have to do similar work many times. Much of the features and design is the same on several platforms but still code has to be written in different programming languages for each platform. For example, when an application for desktop, Android, IOS and Windows phone is to be developed, there are four different applications and the time it takes to develop them is linear and will be approximately four times the time it takes to build it for only one platform. The goal would be to make it more or less constant independent of the number of platforms, maybe in exchange for some extra setup cost. With the previous example it could be C times the time it takes to develop the application for one platform, where C is a constant between 1 and 4 but preferably as low as possible. See Formula 1. Currently there is a big work penalty when adding a new

platform and it will always stay as long as the platform is to be supported. If this could be removed, there would be a potential to support many more platforms.

Below are the current and wanted formulas for the development time a feature or bug fix would take to implement across the application suite.

Formula 1: Current

Total development time = N * T

Formula 2: Goal

Total development time = T * C

Where N = the number of platforms supported, C = a constant between 1 and N, T = the constant time required to develop a feature for one platform.

Another problem that is closely related to the core problem is the error prone process of keeping all the platforms in sync. The manual process of propagating updates from one platform to another takes time and easily fails and creates mysterious bugs if something is missed out.
The reason the core problem occurs is due to the double work and maintenance problem; that is when errors occur due to the existence of several copies of the same core application and all should be developed and maintained simultaneously. One of those issues occur when an update is made in one copy; then the same update should be made in all other copies as well, exactly the same. Sometime someone may miss out on propagating the update to all copies or make an error. With time the copies will diverge and become inconsistent. In the case of cross platform applications, the updates may not always be exactly the same. This is even though the result is supposed to be the same, the way of achieving the task might be different. This makes the problem even greater and harder to manage. At the moment an update can shift even in the implementation programming language and how the task is performed. Therefore there is a need to unite the way of working with the platforms and make the updates more similar; in short we can call this finding commonalities.
In the best of worlds there would be a common source code that could be ran on all platforms. This could be done already by building a web application, a web application could run on all of the platforms targeted. But this is not completely true as there are some major drawbacks that would reduce the strength of the platform. Although the web application can run, it cannot access and control all the features of the platform like a camera or the file system. If this is the path to be chosen then there must be a way of accessing those features and managing differences between the platforms. We call this process of accessing the platforms features and strengths - working with differences.

The conclusion is that there are two major problems in the way of removing the double work and maintenance problem. Those will be analyzed further in the upcoming chapters and are:

- Finding commonalities
- Working with differences

There seems to be a few core options which in turn has several variations. Examples are to use web technologies or a cross compiler. When using web technologies, standards as HTML and JavaScript could be used for building application parts. To employ a cross compiler one programming language is chosen and it is then translated into the language of each platform.
It seems like a cross compiler could easily get out of scope for this report as the development of of one would be a too great task and the use of an existing could be hard to fit in. Because of this, the main track to follow will be the web technology approach but the cross compiler will not be excluded as a theoretical alternative.

Hypothesis: Use web technology such as HTML, JavaScript and CSS to unite the platforms and introduce shared application parts.

2.4. Method

The method for analyzing the problems this report presents is based on real world examples from Kubicom. This example could be skipped and a more general solution could be discussed. But to find something that really works, if not in all but at least for cases similar to Kubicom, the example is still a good option. It creates a scope that is not overwhelming and a proof of concept. As an extension the goal is also to analyze the result for a more general application.
The method used to achieve the goals in the previous section are split into a number of different phases, each with its own span and focus.

The first phase included the study of Kubicom’s setup. This part was performed to find out what the requirements were and what current issues existed. It also gave the opportunity to find out new potential improvements to consider.
This phase were conducted in two parts. One by studying Kubicom’s current applications to find out the structure, relations and what kind of coding languages etc. used. Second by semi-structured interviews with both management and developers to get their whole view of the picture. The interviews were also used to find Kubicom’s level of expertise regarding configuration management practices.
Other options for achieving the same things could be to use a questionnaire or structured interviews. This could also replace the code analysis. This would mean more time communicating with the employees and if choosing the questionnaire more people could be reached. On the other hand the real problems wasn’t known beforehand and therefore all the questions wasn’t clear. If the code analysis hadn’t been done, the background knowledge when entering the interviews would be far less and the possibility to steer the interviews and use the responses would suffer. The code analysis also gave the ability to form an own opinion on the status from another perspective.
Another option would be to gather information from several different companies. But then the scope might have grown dramatically as the current setup for different companies could require completely different approaches.

The second phase were used to study current research and examine the options available in the industry today. This gave a deeper insight into what was already

possible and used today. It did also give inspiration and ground to stand on when comparing to Kubicom's more specific requirements. Another option could be to investigate other companies solutions and use them as a standpoint. A drawback could then be that the latest findings may not have come to the market yet and the solutions may not have gotten thoroughly analysed and especially not well documented.

The third phase were to evaluate the options found in the second phase and consider them in the light from the first phase. The pros and cons of each option were compared to Kubicom's needs and the possibility to further develop the options and make a process were investigated. Instead of comparing the solution alone, there would be a possibility to present the options to the developers instead. In that way their experience could be used. On the other hand it might get hard to give them the whole picture of both the options, requirements from everyone's perspective and the background research. If done, it would require a lot of their time which would be hard to arrange.

The fourth phase was the prototype phase and was used to validate the findings. This gave real world examples to try it on and a hint on how to work with it. As an alternative, the evaluation could be kept theoretical and instead of trying it, discussing the solution at a deeper level and try to foresee possible issues and advantages. This would give more time to research problems and solutions but probably also a more uncertain result and nothing for Kubicom to continue the development on.

2.5. Use cases

To better understand the application of when and where differences among the platforms arise in development we will now look into two examples and discuss them. The examples presented here are from the real world and are taken from Kubicom's current situation. Most of the wanted differences are between the desktop and mobile devices while the implementation differences that aims to accomplish same behavior exist between mobile applications.

Use case 1

One feature that needs special care is the background tracking task. This feature is used on the mobile devices to track the position and time during the period when a supplier works with an order. The supplier starts the order at the beginning and stops it when the work is done. During this period the application needs to log the position of the supplier through the GPS and then continuously send it to a server. In this way clients can immediately see when an order is ongoing and where it is being executed. The location tracking is especially important when it comes to transportations.
There are a couple of parts in this feature that needs to be taken care of in a cross platform application. First of all, this feature should only be available on mobile devices with GPS. The desktop application has no practical use of it. This means that the feature should only be available on selected platforms and on some of them it may be disabled because of the missing GPS. Another part that needs care is the background activity. Most mobile platforms pause the application execution when it is put in background, for example by locking the device or opening another application. For the

supplier to be able to have the logging on during the day he must be able to leave the app and still have it running. The platforms have different methods of allowing this background activity. Some use polling mechanisms and others use dedicated threads that the application can use.

Use case 2

Another use case is varying design and available features. The purpose of the desktop application is different from the mobile devices. The desktop is focused on planning, editing information and follow up on completed work. On the other hand the mobile devices are meant to be used when the work is in progress and they are more commonly used in the field. Those cases implies that the desktop should display functions targeted to what it is used for, it could also display more information because of the larger screen. The mobile device on the other hand needs large and visible controls and a clean and easy to use interface. These differences could preferably be taken care of per platform but also at runtime by screen size.
To sum it up, there are mainly two different versions of the interface needed; one to reflect the mobile and another for the desktop version. These differences could be taken care of at build time. Except for this there are also a need to manage varying screen dimensions and to make the user interface scale properly. This would preferable be taken care of at runtime.

2.6. Variant management theory

This section will provide some background theory on the problem and the root causes. Already from the use cases we can see that we may need variants in several dimensions. We got variations which impose visual and functional differences that we want between mobile and desktop devices. We also got implementation differences between an application that is supposed to look and work in the same way, like on mobile platforms. It seems like some of those features are best handled at build time while others would be better taken care of at runtime. This is mostly a gut feel though and not certainly the way it should to be.

The problem symptoms searching a remedy for is very similar to what is described by Babich and called the double maintenance problem. The double maintenance problem is when there are multiple copies of the same artifact and all of them needs to be kept up to date. When something is changed in one copy, it should also be changed in the other copies. Those changes may be hard to keep track of and there is a great risk of something going wrong and thereby the copies get inconsistent. In the situation Babich describe, this is only maintenance problem and not that closely connected to development of new features. In this case though, it is. The same problem arise both when new features are added and when bug fixes are made. Therefore we call it the double work and maintenance problem here. The traditional solution to the double maintenance problem is to use branches and merge the changes. In the cross platform world this is harder. Mainly because the changes aren't exactly the same everywhere, the implementations differ between the platforms. If the implementation, or at least parts of it, could be shared maybe the merge technique could be used here as well. [9]

The core problem is similar to what is usually solved with variant management [2]. Variant management has several applications. One of them is to fit the application to different underlying hardware and software. In this case the customization is commonly a smaller part of the application.
A variant could be viewed as described by Tichy, where two source objects are indistinguishable under a certain abstraction [10]. Then the application for each platform could be viewed as a variant. Variants are often smaller building blocks and the classic tools may not fit perfectly in this view. Variant management is also a discipline where “one size fits all” is rare. This is something that needs to be considered in the solution and for the general case. Still, many of the concepts and much of the terminology fits well on this problem.

There are two main approaches when dealing with variants. Those are variant segregation and single source variants. [2]
Variant segregation is a simple but often naive way of managing variants. When a variant is to be created a new copy of the application is made. Often some kind of relation is still kept to the original; for example by making a baseline of the application and create branches for new variants.
One big problem with variant segregation is the propagation of updates after the creation; when bug fixes are made and development goes on. This gives the double work and maintenance problem which is much of what this paper aims to fight. Another issue is unclear separation of time based revisions and the persistent variants.
Single source variants on the other hand keeps the source of the variants shared at all times. Instead, a specific variant can be generated on demand when needed, for example to build a product with specific features. The common solution to maintain the shared code base is to use meta tags in the code. The tags are often simple conditionals; when the application is built, the needed parts of the code can be selected. This approach makes bug fixes much easier as they only have to be fixed in one place. On the other hand the code easily gets cluttered with the meta tags and the code can be hard to follow. As opposed to variant segregation this approach makes the division of revisions and variants clearer.

3. Analysis

The purpose of this chapter is to give a better understanding of the problem and requirements of the solution. The result can then be used when looking deeper into the options.
This part will analyse Kubicom to understand their perceived problems dig deeper into what causes them and the symptoms found in chapter 2. It will describe the symptoms and try to follow the chain back to the root cause. Those causes will then be analyzed and described further; they will also be examined to see if they may cause other problems.
This chapter will start off by looking into what topics will be in focus of this analysis and what may be ignored or left for later research. It will discuss the variant management perspectives and what could be used for different contexts.
As a next step, the findings from the investigation of Kubicom's application suite will be presented. The ground is based on a code analysis and interviews. This will be used as basis for discussion of their current methods and to find what could be used when working with our problem.
After this there will be a discussion about the main two problems found, the lack of common parts and how to ease the work with the differing parts.
Finally there will be a summary of what would be needed to fix the problems of cross platform development.

3.1. Variant perspectives and scope

To find the most important topics for this report and to give them enough space this section will discuss the scope of the analysis. The most important parts from the needs and current issues section in the last chapter will be lifted.
There are many different aspects of variant management in the applications already. The aspects stretch in multiple dimensions, feature updates between OS versions, implementation differences, functionality and UI differences across platforms. Except for the variants there are also the more basic revisions, which are the evolution over time and are currently unique per platform as each platform has its own repository.

Already from the background chapter it was found that the greatest issues that are most cumbersome to work with and produces most problems with double work and maintenance is the platform specific implementations of the same functionality. This could be how to integrate the camera for image uploads on different platforms. This will be in focus because if those differences could be kept to a minimum, the development time and the errors from maintaining the same features in several places would go down.
Another cumbersome topic is the functionality that we would like to differ between platforms. The application is ran on different types of devices and they are in turn used in different contexts and are therefore required to have different features. Another variation point is in the user interface which should shift depending on for example the just mentioned features and screen sizes. Those variations are connected and often supposed to vary on the same conditions but the purpose can be a bit different. Both

of those differences needs to be managed and kept together. To optimize the coordination between the platforms this topic will be analysed for solutions.
Changes in the same OS over time may also cause the need for special handling as old functions get deprecated and new ones replace them. Those consequences will be considered but is not in focus as they are quite few and could be handled by platform conditionals already.
Some of the platforms concerned, especially the mobile devices, require the platforms corresponding app store to deploy the applications to the public. The different processes for deployment will be ignored in this paper as the scope would be too wide and is seen as a minor part of the whole development process.
Also the testability of the application suite will be very limited in this discussion. It will be commented in later chapters but won't be a part of this analysis. The reason is once again the scope which will instead focus on the core double work and maintenance problem and just leave the testability part as either an advantage or disadvantage in the end result.

Summary of scope
To keep in focus:

- features - *intended differences that is supposed to be noticeable to the end user.*
- implementations - *required variations that will not be noticed by the end user but needs to be there for a feature to work on the platform.*

To consider:

- OS variations - variations in operating systems or the platform over time

To ignore:

- Deployment
- Testability

3.2. Interviews and code analysis

Here we will describe and discuss the findings from the code analysis and the interviews made with Kubicom. This will be done to find all the symptoms of the problems and track them back to the root causes. The result will then be used in the later parts of the analysis for further investigation of the causes.
This section is divided in two parts. The first one explain the methods used to analyse Kubicom and the second describes the result.

3.2.1. Method

The focus was on finding out how Kubicom is working with their application development today; how their application is structured, what tools and methods they use and what opportunities and problems they see.
The analysis was divided in two parts and began with a code analysis to create an overview of the situation. The overview could help understand Kubicom's application setup and the limits and possibilities it had to offer.
Findings from the code analysis were then added as topics in the interviews which were carried out next. The interviews were held with two employees working for Kubicom. Because of the limited number of people working at Kubicom as regular full time employees only the two most experienced with Kubicom were selected. To get

the whole picture there was one manager and one developer. The structure of the interviews were laid out as semi-structured interviews. This gave the opportunity to have an open mind and find the most critical points for Kubicom and dig deeper into upcoming subjects while still keeping the focus on the target topic.
As a final step we went back to the code analysis. The goal was to use it as a complement to the interviews and look deeper into the structure and relations of the applications. It was also supposed to give an insight into what parts of the applications that could be reused and how.

3.2.2. Result

This section will present the result of the interviews and the code analysis and discuss the symptoms and its causes.

Application setup
The current application setup was analysed to find the possibilities and limits with the setup and see what could cause problems. It was also used to get inspiration to possible solutions.
Some of the results identified the platforms that was supported and how they were related to each other. This setup is already described in chapter 2.1. It is clear that the platform variants has little in common and are perceived as different applications by the developers.
It was found that the main shared part of the application suite was the backend server which all the client applications communicated with. By doing so, some of the logic could be moved away from the differing platforms. This can be seen in figure 2 where the applications in the bottom all communicate with the common server backend in the top.
The common backend help relieve some of the double work and maintenance issues. It cannot remove the problems even close to as much as the methods mentioned in the background; but the positive part is that it could be used as a complement without interfering.

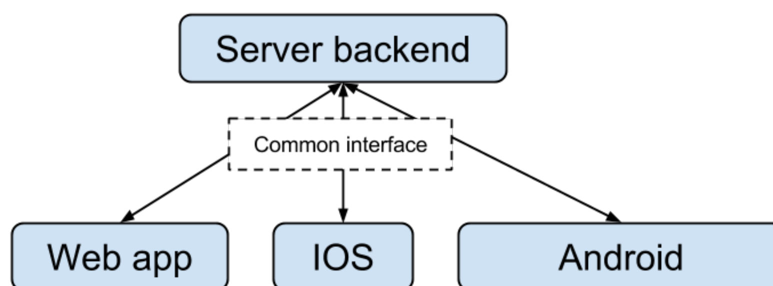

Figure 2. Shared backend interface

Except for the server there were also some web technology parts that were reused among several platforms. The desktop application was a standard web application and the mobile devices used the same code in some of their views. For the mobile applications to support offline mode, those views weren't hosted on the server but in the installed application on the device. Those parts were clearly exposed to the double maintenance problem as although they were kept in different contexts, the parts alone were exact copies. With that said there were variants handled in that code as well.

User interaction events like mouse clicks and touch events that had to be handled differently, especially on desktop and mobile devices but also among the mobile devices as Android and IOS. Possibly those differences could be extracted and managed at another level. The common web parts aren't used extensively but only in smaller parts spread across the applications. They could have potential if used more and had a process for managing them. But currently there are not much gain as there are much manual work involved.
The view of the web parts also have variants. Those variants are managed through CSS and its media queries. This gives a responsive layout that changes with the screen resolution. In this way, custom controls and more information can be shown on large screens while the layout could be simplified for smaller screens. For the limited use of targeting screen sizes this works good and have good support in the web standard. The selection criterias does not seem to be enough to handle all user interface differences though.
Next off would be the platform specific implementations. Currently they are isolated from the other platforms except for the conceptual point of view. Each platform has its own way of working and uses only the backend server as a common interface. The desktop application uses web technology while Android uses Java and IOS uses Objective-C. With those parts each of the platforms needs to be developed independently and there are no gains developing for all the platforms compared to just one; except that if one platform is completed first, that could be used as the model and specification for the rest. Not even the code structure among the platforms are the same. This could be a problem because the uncertainty where a related feature is implemented on another platform. This would lead to more time spent on finding the implementation of features and checking their implementation compatibility.
Because of the different nature of each platform, there could also be a need to have multiple teams working with the platforms, each with its own specialty platform.

Development process

One topic of the interviews was the development process. This is important to find out how the applications are developed in relation to each other and how they are kept in sync. It was found that there hadn't been a clear strategy on how to develop the applications in the first place but it had rather been introduced and polished by time when new platforms came in place. It was found that when it came to development of new features the platforms were developed in different phases where the desktop application often was first of with new features. The reason was the ease of testing and debugging the desktop application which is an ordinary web application. New features could then also be tried on test groups without the hassle of involving an app store. If the features were good they were then implemented in the IOS application. After that the parts that could be reused were moved to the Android version and then the platform specific parts were implemented with the IOS version as a specification. The exact procedure could vary though, especially for features only available to mobile devices the first desktop application step was just skipped.
Kubicom found this process to work out fine as long as it was well organised and each step was completed in turn. Otherwise changes easily got lost and missed out in some place.

Another requirement for this to work smoothly is that there has to be one team working with all the platforms. If not, there could be delays when one team is waiting for another to finish.
When it comes to maintenance and bug fixes, there are different processes depending on what is to be changed. If it is platform specific code that is being changed, it is first fixed on the platform where the bug is found and then the failed test cases are ran on the other platforms as well. If the bug is found on more platforms, it is fixed on each consequently.
If the bug is found in shared code it is fixed for the platform it is found on first. It is then very important to propagate and test the changes on all other platforms. Otherwise errors might be introduced or the fix might get overwritten later on when new fixes are made on other platforms.

The biggest issue is to to update all platforms accordingly. It takes time to do a fix and test it, especially when the developer needs to switch between the platform applications to do so. The switch often includes starting a new development environment and try the fix on a corresponding device. The time it takes to switch makes the developers want to work with one platform at the time and do a complete batch of fixes. If doing so the demand to track the changes fixed increases because the risk of missing any of all the fixes is high.

Time and cost
The management had another perspective of the problems than the developers. They didn't focus on the same symptoms but the root causes was the same. In the interviews, they expressed their concern of the time it took to develop the other platforms after one was done. Especially the mobile platforms which looked very similar. The management saw a big potential gain if the C constant in the formula from chapter 2.3 could get close to 1. If the total time could be reduced and made more or less constant instead of linear with the number of platforms. This would mean that several more platforms could be supported and the selling points would be stronger towards customers.
Of course, if the development time could be decreased, new features would be delivered quicker and the development cost could be reduced or used for more features.
The developers view on the topic wasn't that strong. This was more of a result from the problems they already identified in the development process and they were more interested in discussing the root cause.

Configuration management experience
Kubicom's experience with configuration management is limited. They use revision control tools but with the only focus of tracking different versions of the software and ease team development. Currently there are three separate repositories each containing a platform variant. One for the desktop web app, one for the IOS app and one for the Android app.
The tools used, with GIT as the underlying revision control software, could possibly be used more extensively to support variants but the experience is limited and it would require education and new practices. On the other hand does the limited number of

tools used today open up for new tools to be introduced without incompatibility issues with the current.

3.3. Managing the problems

This section will be used to summarize and understand the issues found and what new possibilities there would be if they were solved.
It is found that one of the core problems of working with cross platform applications is the lack of common parts across the platform variants. That is why this will be the first topic of this section.
When commonalities have been found the same source code could be used in all of the platforms. For the common parts the problem would then be reduced to the double work and maintenance problem.
The next topic will then be to analyse the true differences among the platforms that for some reason still can't be handled by common code. This is important to reach the full flexibility and strength in the platform.

The double work and maintenance problem is about having multiple exact copies and all of them needs to stay in sync. If something is changed in one copy the change must be done in all other copies as well, exactly the same. The changes would come up naturally as bug fixes are made and new features are implemented. The change process is hard work, if managed manually there is a big risk of something being missed out or not fixed the same way. This is especially true if the change isn't made all over immediately or well documented.
The traditional solution to the double work and maintenance problem would be to use branches and after a change is made in one copy/branch it would be merged to the others. The reason this is not possible in the first place is because of the different nature of the platforms and therefore the branches. Even the programming languages are often different between platforms. For those approaches to work the common parts would be a precondition. And even with the common parts, they would not make up the entire part of the application. Depending on the spread of those parts across the application, the merge strategy might still not be the best solution.

3.3.1. Finding commonalities

From the end users point of view many of the platform applications are already the same. For the developers this is not true though, each platform has its own application. What differs the problem of this application from the traditional double maintenance problem is the lack of truly common parts on the code level. This problem has another dimension and is more complex. The code for each feature differ to a great extent and this causes the usual approaches to fail. Usually with the double maintenance problem the parts are exact copies and the problem could be solved by branching and then merging changes between the branches. Another differences is that the double maintenance problem focus on changes after making a copy of an existing piece of the application. In the case with cross platform development, except for bug fixes and maintenance, a great part consist of developing new features. Those new features are often larger parts and could therefore need special consideration as well.
If the complexity of the problem could be reduced to maintaining the same code in several places there would be many more options for finding remedies to the problems

from chapter 3.2. The solutions would specifically take much less time and could get closer to formula 2. That would mean that most of the management's requirements would be fulfilled; the development time would decrease and the cost could be lowered.
Many of the developers problems would also disappear as the propagation of changes would be much easier and the different expertise needed for each platform would be greatly reduced as only minor parts would require other knowledge than for web technologies.

3.3.2. Working with differences

During the interviews it was found that there was a number of parts in the platform applications that were implemented using specific methods and workflows. The reason was that the different platforms had chosen different strategies for achieving a task; for example to run a background activity. The need for this kind of implementations could vary depending on the nature of the application.
Because this report focus on the case of Kubicom, their identified features will be used as a basis. It is found that the features would be best divided into two categories; desktop and mobile devices. The reason is that the use cases for them differ. They are not used in the same way and neither with the same purpose. On the other hand, the devices that fit in the desktop or mobile category respectively has the goal of being as similar as possible.
The requirement of platform specific features for the desktop application are few. This is because the usage is focused on simple input and output of data as for example editing text and outputting the result for order data, schedules and statistics. The needs for the desktop devices wouldn't mean any issues but could be all made in a browser application as today.
The mobile devices use a lot more of the features available on the platform. Kubicom relies on automatic logging of the suppliers position and time while working and need to be sure that the data isn't lost somewhere on the way. This is described in use case 2 in chapter 2. To handle this the function rely on the mobiles GPS and the activity is handled specifically to not stop when the device is in sleep mode or the application loses focus. A temporary log with time and position is also kept on the device; in this way the data will only be delayed and not lost if the cellular connection is gone for a while.
If the application lose internet connection on a mobile device it is automatically put in offline mode. This means that only data that is set to be accessible offline will be viewable and all changes made are stored locally on the device to later be propagated to the server and saved.

Some platforms have varying availability of certain features, even among different devices on the same platform. Not all have GPS or WiFi location support and the availability of cameras (front and back for example), gyroscope, NFC etc varies. Another variation is the navigation metaphors, some platforms focus on soft on screen buttons while other platforms, like Android based, have more hardware buttons. For all of those differences there is a need to be able to detect the status and provide fallback methods.

Below follows a list of features found to be needed by Kubicom. Some of them like GPS, camera and push notification support are general features used by many applications while some other are more specific to Kubicom.

Mobile devices

- GPS - *tracking drivers location when working*
- Background activities - *run tasks as tracking in the background*
- Camera - *take photos and access library to upload content to server*
- Push notifications - *push notifications to notify users of any updates and events*
- On screen signing - *customers can sign the order on screen*
- Offline mode - *work with parts of the app without internet connection*

  There are also upcoming features that would need the following:
- Basic UI events and interface from native platform - *to improve the feeling of the native platform*
- Vehicle pluggable - *access vehicle data such as speed, fuel consumption, emissions etc*

Desktop devices

- Customized UI - *for inputs of larger amounts of data and to fit larger screens*
- File uploads and downloads with any file type

3.3.3. Conclusion

There are two major problems that prevent efficient management of cross platform applications. The first one is that there are little or no common parts of the applications and the development can therefore not be optimized when the same features are to be developed on multiple of them.
The other problem is the organization and managing of the same functions that are implemented differently between the platforms.
Those two problems are related because the code that can't be shared among the platforms may instead be organized to simplify the finding of related parts across platforms. There are three main alternatives of how the code is divided between those.

- All different - *all code are different among the platforms just as today. The optimization would then be to organize and relate the different parts of the application. For example according to feature.*

- Mixed - *some parts of the code can be shared across the platforms while others will be different and only organised.*

- All shared - *all code can be the same across the platforms. This means that there would only be one application and the problem would be simple. This alternative seems to exist only in theory as the platforms are so different in nature.*

Preferably as much of the code as possible would be among the common parts as the optimization can then go much further and only be about propagating changes.

If the shared code is to be implemented using web technologies, as described in the hypothesis in chapter 2.3, there are no direct way of integrating the shared web code with the mobile devices platform’s native code. That would require further research which will be discussed in the design chapter.

3.4. Summary of requirements

In this chapter we have found that the root cause of the double work and maintenance problem is the lack of shared code that could be reused across the platforms. This lead to a complex form of the double maintenance problem which need to be simplified. The simplification would be to unite the way of working with the platforms and to create larger parts of commonalities, preferably as much as possible. This would mean that the propagation of new features and bug fixes will be much easier and both the errors that might occur and the development time would be reduced.
There will always be some parts for each platform that needs special care. The ground of the platforms are different and the meta data like application title, icon, identifiers and more is tightly connected to the platform.
The common parts found already was related to web technology and the option to use it as common ground according to the hypothesis still stands as a viable option. It seems to be an opportunity to shift more of the code to web and reuse code from the desktop application. This will also make the transition to the new model easier.
To avoid switching between development environments and thereby waiting for the new platform to be prepared it would be preferable if the application could be managed as one independent of platform. Otherwise the developers might not switch when they would need to and thereby losing implementations on some platforms and introduce a divergence.

The following bullets are found to be required by the solution:

- Have access to platform specific features to support the bullet list in 3.2.2
- Share code to be able to reduce the complexity of the double work and maintenance problem
- Organize code to help relate similar features across platforms
- Offer different features on different platforms
- Ability to manage fallbacks for unavailable functions

4. Design

The goal of this chapter is to discuss the possible solutions to the problems found during the analysis in the previous chapter. By using the requirements from 3.4 and the hypothesis from chapter 2.3 as a base point the result will be developed to work as a model for a prototype. The goal is to find a design that would work for the case described for Kubicom and be an improvement to the problems found in the analysis.

In the previous chapter it was concluded there was two main topics in need for consideration.

- The double work and maintenance problem - *analyse the alternatives for reusing code for the same features across platforms.*
- Platform differences - *allow differences among the platforms and the access the platforms special features.*

We will look for a solution that works across all platforms but to avoid the need of researching all available mobile and desktop platforms for specifics, we will keep the deeper research and testing to Windows, Mac OS, IOS and Android. The reason is that those are Kubicom's platforms of focus and otherwise the scope would be too great.

This chapter will start off by looking at the solutions to the first problem of the differing nature of the platforms. This will result in a unified way of managing features across platforms. Not all of the application can be managed in this way though. The next section will discuss the remaining parts. For them other approaches will be employed to better organize the differences. A method for communicating with the common parts will also be developed.

The final section will be a deeper discussion and sum up of the advantages and weaknesses of the proposed setup and why it was chosen.

4.1. Finding commonalities

Next up will be the discussion of available methods and tools to introduce commonalities in multiple platforms. This will in turn help reduce the complexity of our problem to become more similar to the traditional double maintenance problem. With the reduced problem, changes among platforms could be dealt with by simple merge strategies as discussed by Appleton [1].

Three approaches seems to be three potential alternatives for further discussion; those are web application, cross compiled and a hybrid. Those methods are found either in the industry or in research for similar applications today. In the remaining of this chapter we will describe the alternatives and discuss them in relation to each other and compare them to the requirements from chapter 3.4.

First off will be the analysis of the cross compiled alternative. This is a more complex option, code is written in one general language and is then compiled into the languages of all supported platforms. Thereafter the web application alternative will be discussed. This alternative would be a regular web application that could be ran in a web browser. Finally a hybrid of a web application and a regular native application will

be examined. In this case some parts of the application is written as a regular application for the platform while some parts use web technology. The last alternative is part of the hypothesis for a working solution for this report.
As a final result there will be a reflection of the conclusions that could be drawn.

4.1.1. Cross compiled

With a cross compiled solution the whole app is written in a specified language and is then compiled into code for the targeted platform. Depending on the platform, different methods can be used. There are two major ways of compilation, either ahead-of-time (AOT) or just-in-time (JIT). AOT compiles the code to the final language for the platform immediately while JIT does the compilation at runtime possibly with a compilation to an intermediate language (IL) between.
This approach solves the main part of the double work and maintenance problem as the application code for the platforms share the same code and there is only one copy of it. Instead of keeping multiple copies of the source code the application is built for each platform on deployment.
To use platform specific libraries and some certain features there is a need to call native code. This has to be done through some kind of bridge and then there would be differences again. Those still has to be managed, if few they may be dealt with manually. Otherwise one option would be to work with the organization of the library to keep track of differing pieces. This would help when implementing new features and doing bug fixes on multiple platforms.
One disadvantage of this approach is the need for a cross compiler that suits the targeted platforms. The cross compiler is complex enough to not develop one for a project; that would be way too much work and only take longer. There are some existing alternatives though and their platform support would at least be good enough for Kubicom. One of them is Xamarin [27]; currently Xamarin support native applications for IOS, Android, Mac and Windows. But existing cross compilers doesn't come for free either. They are often quite pricey and Xamarin currently charges $999/year per developer and device platform. This might be a hard cost to cover especially for smaller projects.
Xamarin makes use of the native user interface for their supported platforms. In many cases this is good as the user more often feel safer and gets to know the app quicker if the interface is similar to other parts of the OS. On the negative, it may be harder to switch between platforms as the application will differ more between them. For example as opposed to the web where the greater part of the interface would be the same across platforms.

Positive

- Seamless integration with native features
- Separation of platforms are taken care of with the tool/compiler
- Access to platform specific design elements
- Good performance

Negative

- Can not be combined with browser apps
- Complex build process, requires a cross compiler
- Expensive to use

4.1.2. Web application

A web app is not installed on the device but loaded just as a normal web site in a browser. This is how Kubicom's desktop application is designed today.
A web application is meant to be run on a wide difference of platforms already. By having a web standard and letting browsers implement it, the job of making it run on a platform is up to the browser developer. The developer of the application just have to follow the web standard and may extend the functionality with browser specific features. With the introduction of the term Web 2.0 it was expressed that the web would go towards being a development platform on its own [25]. Those intentions make it easy to develop an application to work on multiple platforms.
The web standard include specifications to introduce variants in layout through CSS and its media queries. Those specifications work on screen dimensions rather than platforms and may need extra configuration if the layout are to differ between platforms with same screen sizes. One solution would be to detect the platform with JavaScript and managing the variants at runtime. Another option would be to have several versions of the app deployed to different places at build time. The prefered method would depend on the size and complexity of the difference. If it is large and complex a separate version may be a better choice. The code could then be kept cleaner and not cluttered with conditionals. If the difference is smaller, a runtime conditional would be easier. Then it wouldn't have to be concerned with the build process and the ways of managing variants would be less.
With concern of the time and cost from the analysis, this web approach would be good. The only expertise needed would be web development. The division of teams in terms of platform or feature wouldn't matter and everyone could easily start working on any part.
With this approach, there would only be the traditional double maintenance problem and changes among platforms could be dealt with by simple merge strategies as discussed by Appleton [1].
There are a few meta tags introduced that can be used to make the app feel more like a native app. The ability to zoom in the app and the scaling to make the app fit on screen can be controlled by meta tags. Those could also be used to customize appearance like title and icon when the user add the web app to their home screen on a mobile device.
Except for the previous features, there are very limited access to the platform and its features. This is the greatest weakness of the web application; almost all of the features mentioned in the bullet list in chapter 3.3.2 fails. Some parts are available with limited options and would not suffice to the requirements of Kubicom. For example the location of the device could be retrieved by using GPS or IP address depending on what is available. But it can't be tracked when moving without polling and if the application goes into background mode the tracking process will be killed immediately. The camera could be used partly, but only to upload already existing photos.
Another issue could be the lack of performance of running JavaScript in the browser [26]. Running a native app has performance advantages and if the application is computationally heavy like a game, web technology may not be enough. Although some view transitions may be slow on older devices, this will not be an issue for Kubicom's application.

Positive

- very easy to develop, both cheap and fast
- can be accessed from any browser
- easy deployment, updates doesn't have to go through the process of an app store

Negative

- limited set of functionality, can only be used for very simple apps
- doesn't have the same feeling and ease of use as a native app
- lacking performance, JavaScript and HTML does not always provide enough performance

4.1.3. Hybrid

A hybrid app is a web application wrapped in a native app frame. Just as with the web application approach the solution to reduce the double work and maintenance problem is to use web technologies. But to make this choice stronger in features a native part is mixed in as well. The native part makes the double work and maintenance problem harder again but helps to fulfill the requirements of platform specific features. The native part of the app is preferably very limited but comes with new benefits. One of them is that it can be deployed to app stores and thereby be installed. This will not make the lives of the developers easier but will ease useage for the end user and this is what's expected on the market today. For the developers, this will be extra work and new rules of the app store to follow.

The native part of the application is in its simplest form just one view that launches the app and presents the user to a WebView, that is like a browser without any controls running inside the application. This WebView contains the web app part and is all the user sees.

Depending on the use case more of the logic could be transferred to the native part. To access any of the platform specific features more native parts will be needed. One option would be to implement them in the native part independently of the web part. The advantage would be the ease, as there is already a basic native part and more could just be hooked on to it. To improve the interaction between the native and web parts another solution would be to use a language bridge to make the web part and native parts communicate with each other.

This would allow for the web part to be make up a greater part as only just the platform specifics needs to be handled in native code. Otherwise there may be a need for more views, buttons and initializers in the native part and this would decrease the effect of reducing the double work and maintenance problem as the code would no longer be shared.

For the native parts that differ there would be a gain in organizing them for better control and more natural places to implement

Except for the already mentioned advantages, this solution can also make use of most of the positive effects of the simpler web application approach.

One example of a current solution that works in a similar way is Apache Cordova [13].

Positive

- Shared code can be separated
- Platform specific features can be implemented in platform specific code
- can be made available on app stores

Negative

- could be hard to separate native code and HTML and in the same way make them communicate
- The parts written in the native programming language of the platform are still victims of the double maintenance problem

4.1.4. Conclusion

We have now discussed three different alternative for dealing with the core solution of working with the double work and maintenance problem: finding commonalities. The alternatives discussed were cross compilation, web application or a hybrid of web and native technology. Now it is time to compare them closer and find the most suiting approach for Kubicom.

The web application can seem like a good choice by being very simple and the current web application for the browser could make up a good ground to start from. But the solution fails on one major requirement; the need to access platform specific features. In the analysis it was concluded that the desktop application didn't need much of the platform features and was therefore working as is. But the mobile application had many of the core features related to the platform and couldn't be without them. Because of this the web application is no longer an option for Kubicom.

The cross compilation approach also seems like a good candidate and doesn't have any large weaknesses compared to the other alternatives. One advantage of the cross compiled alternative is the performance, a cross compiled application will be transformed into native application at build time and will then have the full performance. This will make the app run faster and especially on older devices smoother.

Both the cross compiled and the hybrid alternatives can have a large part of the application in common but also needs their own specific parts to manage platform differences.

Depending on the preferences of the application developer and their application users it could be either a pro or a con that the user interface have a native look and feel per platform, as in the cross compiled alternative, as opposed to having a unified design across all devices as is the more probable option with the hybrid alternative. In Kubicom's case it is probably better for the users to have a unified design, at least for the suppliers as they will often shift between platforms and will then easier learn a new platform when they have already worked with a similar one.

Another thing to consider is the experience of web technology versus the language for the cross compiler. If starting a new company from scratch this might not be a concern but otherwise both development time and education costs could be high enough for a new technology to choose the familiar one instead. For Kubicom they already work with a wide difference of languages and platforms and this would not be a big issue.

As both approaches seems like good options we will look into Kubicom's case and consider the evolution versus revolution. Kubicom already have web application running in a browser as a desktop application. With both the web application and the hybrid approach the current web application could be used as a basis for the new approach. That would make the transition easier by making it both faster and cheaper. There are also smaller parts of WebViews in the current mobile applications which makes the concept familiar already.
For deployment reasons the hybrid could also be better. Because the major parts are based on web technology, the application could easily be ran in a browser as well but with less features.

For those reasons the hybrid alternative will be used for further development and research.

4.2. Keeping commonalities in sync

We have now found an approach for introducing commonalities by using a hybrid application. But there are still concerns regarding how to keep the platform variants in sync. Babich propose the use of branches and merging updates between them [2]. Appleton then does a deeper analysis of what branching strategies are best fit for different problems [1].
Here we will discuss the branching strategy and compare it to other solutions like library organisation and build time options.
Finally in a comparison we will find the most suitable solution for this case.

4.2.1. Branching

As mentioned before, the traditional way of dealing with multiple exact copies is to to make a branch per copy. In this case each platform would have its own branch.
The branches can then be named according to their platform. It is important to keep track of the branches and keep them separated from the usual temporary branches used when developers work in parallel. As opposed to the temporary parallelization branches the platform branches are permanent and needs clear conventions to avoid conflicts. There are no ways of ensuring consistent naming of branches. This is one of the reasons not to use permanent branches.

When making updates to common code the change can be made in one branch and then propagated to the others by merging. Appleton suggests a couple of suitable approaches to branching this kind of model [1]. Either there could be one common master branch containgen the frame and the common code. When committing to it, changes are automatically propagated to all platform branches as well. This would impose a given way of where to do changes and all the slave branches would stay in sync. On the other hand the master branch would be unused with its only purpose of being editable. This setup is described by Figure 3 where the development branch act as a master and propagates changes to the slave platform branches to the right.

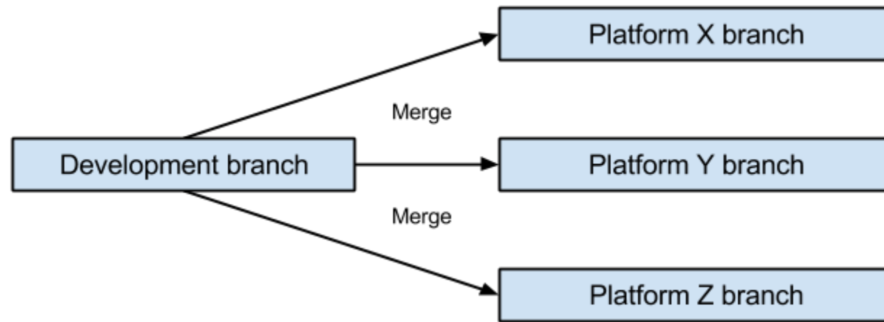

Figure 3. Branching strategies - master with platform branches.

Another option would be to only have platform branches which are all equal. The changes could then be made in any branch and the changes could then be merged into the others. If automatic propagation is not used, conflicts could easily be introduced and the developer has to remember to merge to all other branches. But the good part of branching is the ease of solving conflicts. The propagation process is described by Figure 4, each of the platform branches are equal and changes should propagate between all platform branches.

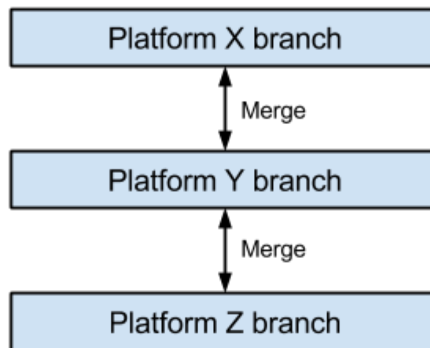

Figure 4. Branching strategies - equal platform branches.

4.2.2. Library organization

Another option is to rely on the library and the file system. Because the common parts are exactly the same across all platforms and should always remain so, the copies could be maintained by file system features like symlinking. This is a way of making symbolic links of files or directories to other places. Instead of making a copy of the shared code it could be linked. If the source is changed, all the links will update accordingly.
Because one of the requirements from chapter 3.4 was a tight integration of the platforms to ease switching development between them it is preferable to keep the structure between them related. If all the platform share a library structure the linking could easily be setup between them. Each platform could then have its own directory with the parts for building in the library. This solution would completely remove the redundant parts of the common code. Although space is rarely an issue today, it would also save disk space.

4.2.3. Build time instantiation

To propagate the common parts the build time process could also be used. The propagation of changes would then be centered along the instantiation of the platform. The common parts could be kept in a single separate directory and when a platform is to be instantiated the relevant files are copied to the platform build, overwriting possible existing files. This process would allow for additional processing on the way and making changes on the fly. If there is already a build process, this addition would be easy to add.

4.2.4. Conclusion

When looking at the presented alternatives they have different advantages. Although the branching strategies are well tested and have clear methodologies for working with them, they also have several disadvantages. Because Kubicom use revision control software as GIT, there wouldn't be any issues introducing platform branches and managing changes through them. This is also the only approach that allows a clear separation that doesn't need the copies to have relative paths to each other in the library.

Kubicom doesn't currently employ any specific build time process except for compiling. This makes the build time instantiation require a new tool only for the purpose of syncing commonalities. With the alternative of both the branching and build time process the double work and maintenance problem still exist as there are real copies of the code.

If using the library organization on the other hand the copies are only virtual. The library organization is definitely the alternative with least configuration options but that is also why it is the easiest option. At the same time it fulfills the requirements and its purpose.

That is why the library organization based on file system features are the chosen strategy for syncing commonalities.

4.3. Working with differences

When having the shared parts in place and a safe way of keeping them in sync we now need to manage the remaining part; the differences and their relation to the common parts. They have to be managed both in terms of platform implementations and platform specific features. They need to be related to help the configuration identification and to ease in the development and find related parts. The common parts does also have to be kept clearly separated from the platform specific parts to not mix them up.
The first part of this chapter will discuss how to organize the library. The organization will focus on how to identify and instantiate a variant. It will also discuss how to separate different parts and keeping them separated with the help of components and build scripts.
The next two parts will focus on the management of differences. From the analysis it was found that the main variants are platforms but that there are also smaller differences in the platforms. Because of that components will be discussed on two

levels; the low level platform part which constitutes the component and the higher level smaller variants in the platform.

4.3.1. Organization and identification

It is important to keep a well organized library when working with variants. If not, the variants and the common and shared part can easily get mixed up.
There are two main options for structuring the the platform in the library. Either by the platform as a whole or by components per feature. If structuring per platform the code for each platform will be contained in its own directory hierarchy. This can help keeping the variants apart and implies quite few requirements on how the platform specific code is structured.
The component per feature approach would instead keep the platform specific code for all the platforms together based on feature. For example the camera feature could have its own component and both Android and IOS implementations would be contained in the component. This approach makes the related code for each platform easily identified and the component could also specify an interface that should be used when accessing it. The interface could then be a JavaScript file which could be called by the common code. It would then in turn call the platform specific implementation through a language bridge. The component feature structure make them reusable as building blocks for other applications as well. Because everything is packed together they can act as independent parts.
The selection of the correct platform implementation have to be made as well. To minimize the code deployed to a platform this is best done at build time. Because the platform specific code for the majority of platforms need to be compiled there is a need to do this after the correct pieces for the platform have been put together.
Because the common parts and the platform specific parts are not compiled together there is a need to keep track of the components required for the build. This could be done by interpreting the JavaScript calls but that would be hard. Instead, because the required components wouldn't shift to often and in most cases they won't be too many, a bit of manual work could be sacrificed. By doing so a configuration file telling which components are needed could be used.
All platforms except the desktop browser application has a specified skeleton that is the same independent of the application. The skeleton can then be configured with meta data in configuration files to set basic properties such as application title, icons, allowed orientations and splash screens.

With this background the following structure is proposed. The common code is placed in its own directory to keep it isolated, this is the common code part in Figure 5.
The components with a JavaScript interface and the implementations for each platform are placed in a separate directory and each component gets its own isolation by its directory. The wrapper directory for components is called components in Figure 5.
Each component then create its own directory containing the parts that branches below the components part. There is also a configuration file to specify the needed components. The configuration file then acts similar to what Daniels describe as bill of materials (BOM) [4].
Each platform gets its own directory where the skeleton for the platform is placed. At build time the components platform implementations are copied to its corresponding

platform and the JavaScript interface is concatenated with the rest of the common web parts.
The common parts are always kept in place and up to date in the platform directory by linking.

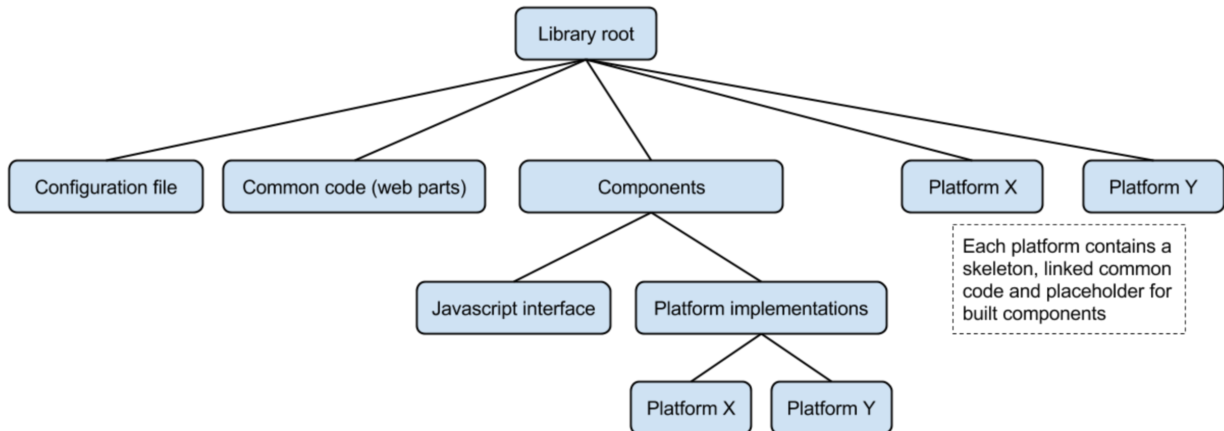

Figure 5. Library organization

4.3.2. Low level variants

The Low level variants are the large variants that make up the building blocks of the application. Winkler calls this program variation in the large [11]. In this case those variants are the components of platform specific implementations. Those variants are the parts that could not be part of the common code and are therefore specific to each platform. The features that they provide and in some cases the logic should be the same but except for that, they are very different in nature. The lack of similarities make the variant segregation approach described by Mahler a good option [9]. Because there are no similarities there will be no true double work and maintenance.
The most important feature of the low level variants is the access to the platforms specific features. The access is implemented through a communication bridge between JavaScript and the native programming language. At the same time this communication is isolated in the component which makes it more modular and its responsibility more clear.
The internal structure of a component is depicted in Figure 6. Here the JavaScript interface is the entrance to the component and the rest is divided into platform specific implementations selected at build time.

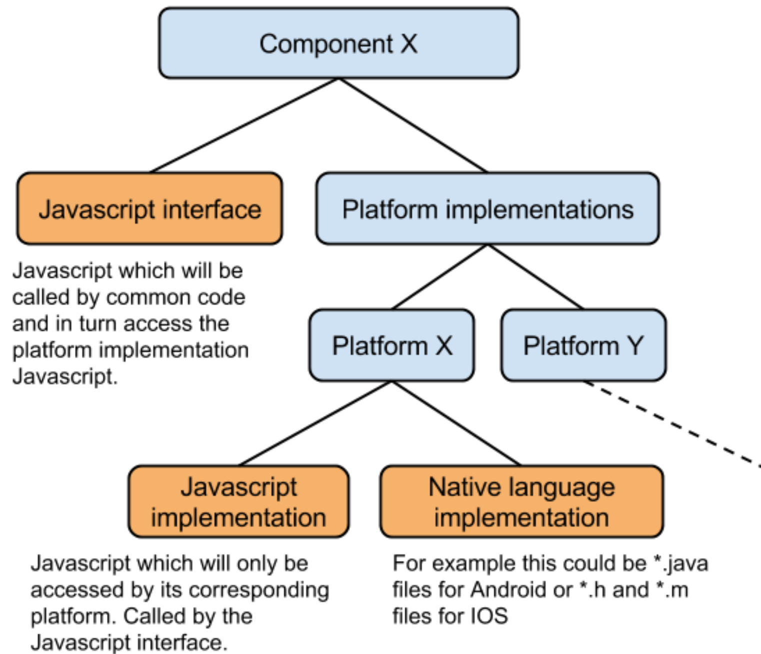

Figure 6. Internal component structure

4.3.3. High level variants

The high level variants are the small variants that are variations in the low level components. Winkler calls this program variation in the small [11]. When the large differences that make the platform specific access possible are dealt with the smaller changes are considered. From the requirement summary in the analysis a number of smaller variations are found. Examples of those are differences between versions of the same OS, fallback for lacking or differing features. Those variations can be managed as internal variations in the low level components.

One way of managing those variants would be the same as the low level components by structure and at build time. The advantage would be that there is a common way of managing all variants. The negative side is that they do not fit as well and the number of variants would be very many. Because all the combinations would create many variants they would be very hard to separate while still avoiding the double maintenance problem. In difference to the Low level variants, the high level variants could often be implemented using similar code and there would be reusable parts. When working with similar variants Winkler propose a method called single source variation. This method offer a nice way of mixing common and variant code. Winkler suggest to use meta constructs to select the right parts for the variation and managing the selection at build time. [11]

Instead the variants can be managed at runtime with conditionals. The positive effect of runtime management is that all the variants doesn’t have to be known at build time and may even change during runtime if the platform settings are changed. The complete specification of the platform with all available features is seldom known beforehand.

4.4. Proposed setup

There is now a complete proposed solution to the problems found in the analysis. This section will summarize how the found problems are dealt with and what possibilities and limits this means.
The found solution is centered around web technologies to introduce common parts. With this the double work and maintenance problem is reduced to keeping copies in sync.
It was a tight decision when choosing between the web technology and cross compilation. Both approaches seem to be good choices but differ on a few key points as user interface and deployment possibilities. The final decision to go with a hybrid is much based on Kubicom's current setup and possibilities to evolve the current application and reuse existing parts. The generally applicable parts of the solution are a later topic in chapter 6.2.

To keep the common parts in common research has already found good options, some of which are presented by Appleton [1]. In this setup it is found that they are not the perfect fit. They would work but are unnecessary complex. The simpler way of keeping the common parts organized and using file system features keeps the process all automatic and removes the double work and maintenance problem for those parts as there are no real physical copies.

When it comes to accessing the platforms specific features and working with the differences, two levels of variants are chosen. The first one is the low level, implementation differences which are to be managed with components. The second one are smaller, high level variations managed internally in the low level components. This split introduces two different ways of working with variants which might be bad because of complexity reasons. But the gain of having the variants separated supersede the bad parts. The logic seems clearer and the complexity of the code is reduced by separating the platform components. At the same time the minor more similar differences can be managed tighter together and thereby avoid redundancy.

For the complete setup to work a tight relation between the platform applications are needed. Independent of the platform, the whole application are managed as one. This could definitely be a good thing as changes can be easier applied to all platforms and updates are not as easily missed out in any platform.

5. Prototype

The prototype was developed to try the found solution on real world examples and give a proof of concept. To make it more relevant to Kubicom and create a base for further development one of Kubicom's features were picked for implementation. This decision also made it possible to reuse parts of the existing code in the transformation to the new solution.
This chapter will first describe the scope of the prototype, what was implemented and why those parts were chosen. The next section will discuss the implementation phase; the approaches taken in the realisation of the solution concept, what tools were used and considerations made on the way. The last section will give some guidelines when setting up this solution and what is important to think of to make it efficient.

5.1. Scope

Because the limited time and focus on other parts a whole application couldn't be tried with this solution. Therefore some key parts of an application that surfaces the major parts of the problems found earlier and the solution concepts found will be tested.
After the experience from the code analysis made in the beginning of the project it was found that the supplier tracking mechanism made up several key problems and was also one of Kubicom's most important features in regard to platform specific access. The prototype will cover both Use case 1 and 2 from chapter 2.5.
The parts that will be tested are the commonalities and their synchronization across platforms, access to platform specific features through components with native code and finally feature variants and fallbacks on different platforms.
The prototyping and development were run on Mac, some of the tools and methods might differ on other operating systems but the concept should remain the same.

5.2. Prototype implementation

The solution found in the design chapter was general in terms of tools and development platform. This section will now make the solution more concise and give a suite of tools and a real example of a small sample application.
The structure will be connected to the design chapter and the problems solutions and what they contain in this example will be described in turn. First off are the commonalities, then the Low level variants with platform specific access and after that the high level variants for fallbacks and feature variations. At last an overview is given of the structure and where each part belong.

5.2.1. Commonalities

The commonalities are made up of web technologies like HTML, JavaScript and CSS. In this prototype the common parts are used to create a simple user interface to start and stop the tracking and to log the results. The controls presented to the user are hooked up with JavaScript and calls the functions in the corresponding component's JavaScript interface. How those are connected will be explained later. The common code is also responsible for communicating with the server and save the results such as positions with timestamps and to read and present earlier results. In this prototype the server communication is made up of a REST API [22] communicating with JSON [20].

According to the proposed solution in the design chapter, the synchronization of the common code to the platform variants should be dealt with by library organization and file system features. The common code is kept separate from other parts in its own directory. This is placed in the root and is called “app” in Figure 7. This directory is then mirrored to each platform directory and thereby placed in the right spot with the platform specific code. The mirroring is achieved using symbolic links or as they sometimes are called, symlinks [23]. This does completely remove the double work and maintenance problem for the common parts as the synchronization is handed over to the file system.

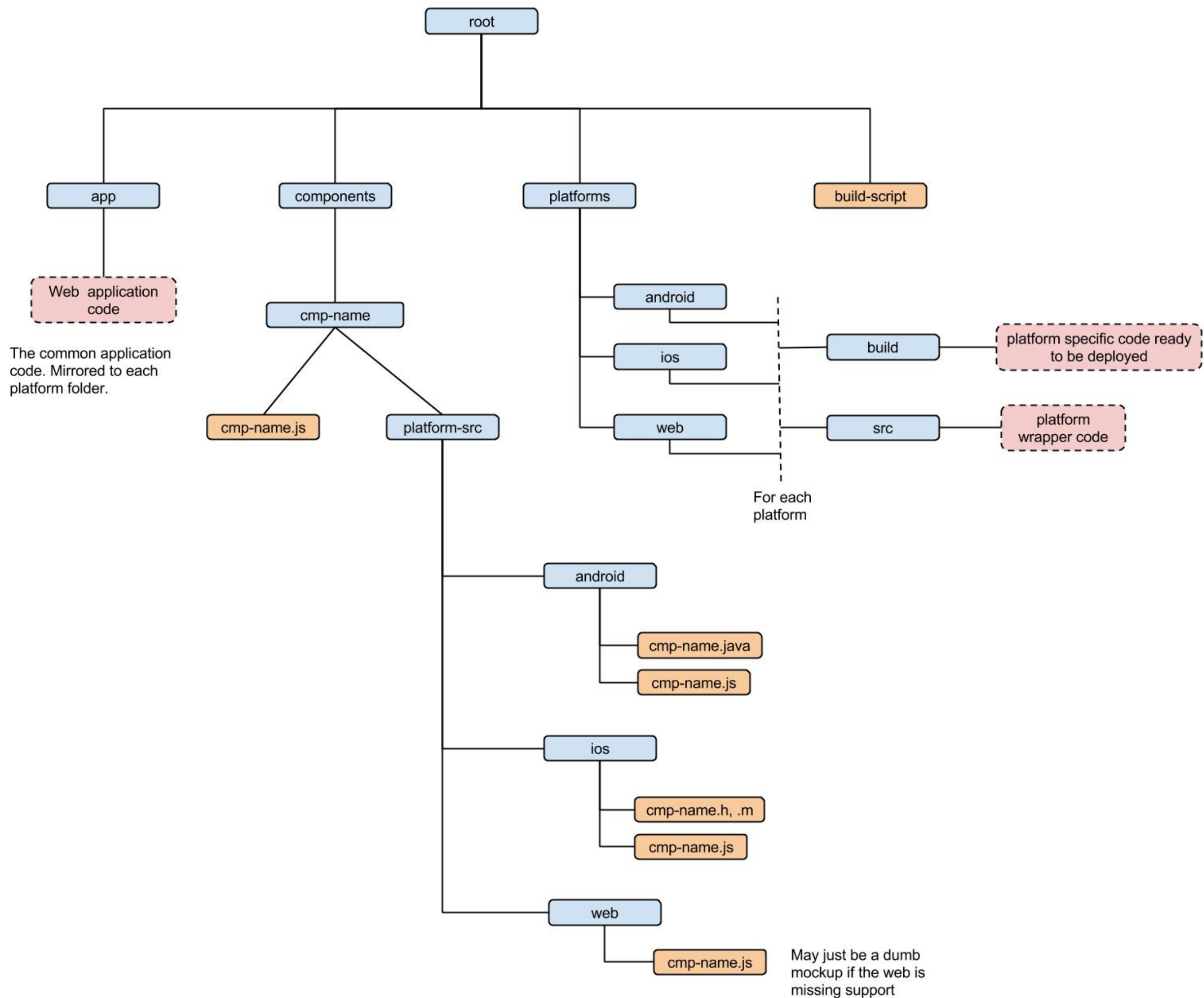

Figure 7. Prototype library. The light blue boxes are directories and the orange ones are files.

5.2.2. Low level variants

These differences are related to variants across platforms and thereby operating system, software and hardware capabilities on the device. In the solution they are proposed to be structured as components and and put together at build time. In this prototype there will be two components, one simpler and one a bit more complex. The simpler is the location tracking using GPS and Wifi to target the device’s current location. This task is simple as both Android and IOS has quite similar methods for

tracking the users position. The harder feature is the background activity, the reason for making it a bit harder is because the methods and the behavior offered by each platform differs more. Neither of the platform specific methods are to be accessed by the desktop device and are not taken care of here but the variation in the form of a missing feature will be dealt with later with high level variants.
The low level variants was found to be best solved at build time. There are several options to build the components and put them together for the platform. From the beginning, several build tools were investigated to find a good candidate for building the application for each platform. There are a large number of build tools available like Grunt [18], Broccoli [15], Gulp [19], Apache ANT, Maven, Gradle etc. Each tools has its own advantages and focus. For example Grunt, Broccoli and Gulp are focused on web technology and JavaScript builds. Therefore those are good for the common web parts of the application and for merging the components JavaScript interfaces with the common code. They do also have capabilities for minifying and concatenating JavaScript which could be handy in a real project.
But in the end it was found that for this simple proof of concept prototype, Make which is also a build tool [17], was the easiest to setup and could manage the requirements. If the build process are to be extended further, other more powerful tools might be considered.
For the two components, each has its directory and JavaScript interface file which are could be called from common code and is responsible for calling the platform implementation JavaScript file. The platform implementation JavaScript file is file unique to the platform and is responsible for managing the communication with the native code. All the required JavaScript files for the components are merged into one and included in the common code. In this way the JavaScript code can be called from anywhere in the application.
The platform specific files are chosen at build time depending on which platform is built.
When all files are in place there is still the problem of the JavaScript and the native code not being able to communicate. To solve this bridges are used. In the case of Android this is very easy as it is natively supported and described by the Android WebView class description [12].
On IOS there is no direct way of communicating and here a separate library called WebViewJavaScriptBridge is employed [24]. Because the desktop application is used as a browser application, no bridge or native code is needed in this case.
The organization of the components are depicted in Figure 7.

5.2.3. High level variants

The high level features are variations with closer relation to each other and do often consist of smaller variations. The variations are implemented using the same programming language and can more easily share parts of the code between variants compared to low level variants.
These variants can be used to implement various feature and styling variations. In contrast to low level variants these can be handled at runtime and are made to be more flexible.
In the prototype these variants was mainly used to handle user interface variations across platforms and to make a fallback for the desktop application which shouldn’t support tracking.

The high level variants was taken care of internally in the low level components and do therefore not need any special organization. They are implemented similar to the single source variants described by Winkler [11]. But instead of making the selection at build time the functions available in the platform and through web technologies like JavaScript and CSS at runtime are used to determine which parts to use. CSS can be used to shift interface appearance for different screen dimensions and JavaScript can also detect platform specifications and act accordingly. The CSS methods works very nice in this prototype but can't access as many parameters as JavaScript. JavaScript has great power but is not as good for managing variants with conditionals. On the other hand there is also a possibility to use ignore the platform specific code in the components and implement variants in the JavaScript platform file instead. This would do a first filtering on what JavaScript should be used on each platform and it can then further narrow the variants at runtime. The downside of this is that code can't be shared in the same way across the variants and there may be a new double maintenance problem.

5.2.4. Library organization

The library structure used for the prototype is described by Figure 7. It contains three directories in the root. The first one is the app directory which should contain all the common code and constitute the major part of the application. The next is the components directory which contains all the components with low and high level variants. Each component directory contain a general JavaScript interface file. In the figure below it is called cmp-name.js and is contained in cmp-name. In the components platform-src directory each supported platform has its own directory. This directory contains a JavaScript platform file and implementation files written in the native language of the platform.

5.3. Conclusion

The purpose of this section is to discuss the findings and results from working with the prototype. It will result in some points to consider if choosing this approach but also some guidelines to access its full strength.
The first part will be some reflections on the success and weaknesses found. This will be followed by the guidelines for how to work with the solution.

5.3.1. Reflections

Overall the found solution works great and from the author's point of view it's a success although there are many improvements that could be made.
The reason to interpret it as a success is because all the use cases made it through and could be managed while still keeping the root cause for the earlier double work and maintenance problem away.
The most obvious possibilities for improvement is in the build. That constitute the most complex part of the solution today as it requires quite a lot of configuration. There are great possibilities of improvement if a better build tool more fit for the problems was found because it could then be extended. Examples of extensions are dependency management between components, a one place configuration of the platforms that should exist/be supported and which components are needed. This would not only

give an easier way to set it up but also give a better overview of the character of the application.
During this trial there was no problems with the organization. This is something that would need more testing and especially with a complete application and during a time of development. If not careful new implementation might be misplaced and the separation might get more fuzzy.

5.3.2. Guidelines

There are two major things to think of when working with this approach, one targets the common parts while the other consider the platform specific parts.
One of the most important things to remember when working with this solution is that the application is in the common code. The major part of the application are to be developed here. The reason is that there are no double work and maintenance problem and the configuration and work had to be done to manage it are nonexistent.
When there actually are platform specific features that need the native code a component could be introduced. One of the key points with components are their ability to be reused. But for this to be true they need some care. The responsibility for a component should be clear and kept to a minimum. With a good dependency management for components they can then depend on other components to fulfill their full purpose instead of doing all the work themselves. This makes the components much easier to reuse both in multiple places in the same application but also in other applications.

6. Discussion

This chapter will discuss what was accomplished by this report and analyze how it stands against other current solutions.
This chapter will start off by relating the topics of this paper to other research and the state of the industry. The perspective of this paper will be compared to the others and the similarities and differences will be lifted.
As a next step there will be some reflections on decisions taken during the research and explain why it was done and what else could have been done. It will also discuss what this report does not cover and what it does not consider.
This will be followed by a discussion of what could be done as a next step to further investigate the topics of this paper.
Finally there will be some reflection of the report in general, what the contributions are and what could have been done different.

6.1. Related work

This section will discuss similar work found in the same area and compare it to the topic of this paper. The purpose is to give an understanding of what is happening in the area see what is new in this report.

A paper with a similar problem description is written by Gyllensvärd and Welander [6]. The scope does differ though and the focus is on web technologies and platforms that support it running as a native programming language.
Charland and Leroux made a comparison of web versus native applications but does not use them for cross platform applications but rather as a comparison of performance and user experience [3].
Another approach discussed by Lakshman and Thuijs is to build thin native clients and letting a common backend do the major work [8]. Their approach is a good option and does not exclude the solution found by this paper. The work of keeping the clients thin are often much easier than introducing and managing common parts across the applications. The thin clients can’t solve the double maintenance problem to the same degree though.

The research of working with variants in general is not as specific but do provide concepts that can be used for dealing with the problems found with cross platform development.
Mahler discuss solutions of having to maintain variants of application components which have to be maintained and kept in sync [9]. The synchronization issue is found to be the main cause of problems in cross platform development although it doesn’t contain the whole problem.
Appleton does also propose several strategies to keeping parts in sync using branching patterns [1].

In some parts, especially when taking a deeper dive into the techniques to use in cross platform development, it seems like the research can’t keep up with cutting edge

technology being introduced in industry. Some research does discuss the advantages and weaknesses in existing existing tools and frameworks but does not propose any alternatives. One of those papers are the one written by Heitkötter et al. [7]. Another paper by Dickson discuss only Xamarin as an alternative to a cross compilation approach in cross platform development [21].
This paper differ from those papers because it dives into the root causes and construct solutions for each part with its base in the causes.
In the industry there are several tools and frameworks with varying success that have popped up over the last years. Most of them are constructed using some of the alternatives presented in this paper. Some of the popular alternatives to cross platform development tools are listed below:

- Apache Cordova - *Native and web hybrid used for mobile applications* [13]
- Xamarin - *Cross compilation solution which can be used for both desktop and mobile applications* [27]
- Appcelerator - *Native and web hybrid used for mobile applications. As opposed to Apache Cordova, it can utilize native UI interface [14]*

A conclusion that can be drawn from related work is that there are very few options for cross platform development that support both mobile and desktop devices. The main focus is on mobile applications, probably because the market is more diverse in terms of platforms and because desktop platforms often have cross platform solutions like Java already. Another reason could be that the intended differences among mobile devices are much less than the differences between the groups of mobile and desktop platforms.

6.2. Limitations

The purpose of this section is to inform the reader of certain limitations of this paper and discuss the effects of them. It will also mention what could be done to remove those limitations, although some solutions may introduce other.
One limitation in this report in terms of generalization is the focus on Kubicom. Kubicom forms the use case and is the only company evaluated to find the root causes. Although the problems are discussed in a general way and are analyzed for future possible extensions and thereby covers more applications there are a risk of missing requirements of many other companies. The focus on a solution for Kubicom is intended as the report is produced in cooperation with them and the analysis would otherwise not fit in the scope. One of the decisions most coloured by Kubicom is the solution of finding commonalities. The cross compiled alternative stood strong and should be understood before choosing a strategy for another application. If changing the common parts there would also be a need to consider other solutions to the problems in the remaining part of the design chapter. For example, if the applications and variants wouldn't be as tightly coupled in organization the traditional solutions to keeping copies in sync by branching would probably be needed.
As mentioned there are things to consider before choosing the solution but the applicability of the solution can in many cases be generalized. If there is already an application that should be evolved into a cross platform application the proposed solution in this report would fit best for an application which is already intense in the use of web technology. For example if a web site is to be developed into applications

the found solution is a great choice as much could be reused. This is one of the important reasons for Kubicom to choose the solution as this is more of an evolution rather than an revolution. The found solution is also good if a similar user interface is wanted across platforms, because the web technology look the same independent of platform this is much easier accomplished using web technologies than a native user interface.
If starting from scratch the solution should be compared to the cross compilation option. The selection of solution could depend on the performance or the native user interface for the application; if any of those are required the cross compilation would be better off. Another case for when a cross compiled application is best could be if the application is very large. This is because the web languages seldom got strong structures and aren't strong typed. Therefore a cross compiled solution could be easier to refactor and maintain in the long run.

Another decision that could have had an effect on the end result is the lack of the traditional native apps as an alternative. The alternative is always in the background as a reference and Kubicom's current application suite is centered around native apps. But if the native alternative would have been analyzed as an alternative, some perspectives might have been weighted different and others might have been found.
In some cases very different techniques are discussed as options for solutions. This can make the understanding of the techniques more shallow than if similar techniques with only smaller tweaks was compared. But at the same time the current approach could be defended by the wide perspective the result gives although more shallow.
A topic that got very little attention in this paper is the current tools and frameworks on the market today. Instead of reinventing a solution from research, existing solutions could have been analyzed for pros and cons. Then there would be real world solutions that has been extensively tested already and many weaknesses would have been found and solved in the process.
The testing and proof of concept of the solution offered by this report is limited. The solution has only been tested on a small piece of common code and only on a small portion of platform specific features on Android and IOS. Especially the high level variations in the components have been little tested and those constitute a wide variation of variant combinations.

6.3. Future work

The continued work in this area could be carried out in a number of ways. There are both topics closely related to the found solution and more general investigations in terms of cross platform development.
Although the found solution worked fine on the tried prototype the solution could be polished and the proof of concept strengthened by developing a complete application and deploying it.
In the current prototype, especially the build scripts are a bit dumb. The specification and inclusion of components could be made easier and automated to another level if the component directory were automatically searched and interpreted and only needed components were specified in a single build configuration file.
To help enforce the library structure skeleton generators could be built. Those could be command line scripts that are run to create the folders and files for a new component

for example. The reason to do this is because the structure is vital for the build process to work and that developers might forget what to put where. If errors are introduced they might be hard to track back to the source.
Another thing that could possibly reduce the double maintenance problem and modularize the application further is if the components would support dependencies. Then they could be split into even smaller parts and be reused by several other components.
The new library structure for each platform introduced by the proposed solution may differ from the structure of the corresponding native platform. Because of this IDE:s and other development tools may not function as usual or at all. This should be tried out to not lose important and well functioning tools used for native platforms today.
To widen the perspective and deepen the evaluation the found solution could be compared to existing frameworks. The comparison could give new ideas for improvement to the solution found here but also to the existing frameworks on the market.

6.4. Reflections

This report has analyzed the problems found in cross platform development and discussed their root causes. In the new light of the causes and the requirements of Kubicom, a number of possible solutions are presented and analyzed in a step by step process.
The result is a big solution toolbox with wide differences among the tools. Some of the major new contributions are a proposed solution assembled from the ground up and not just a comparison. The solution is true cross platform, not just between mobile devices but also for desktop devices which offers an extended application and more efficient remedy to the problems.
The solution is tried on a real world existing example to show proof of concept. Because the proposed solution in the report conform to the hypothesis, it is also proved to work for this setup.

The report had three main goals presented in the beginning of the report. Those will each be presented here again together with a short version of the found answers.

What methods and tools can be used to ease cross platform application development and reduce the double work and maintenance in comparison to building independent applications?
The answer found is that commonalities can be introduced by using web technologies and by the use of a tight relation between the applications the commonalities can easily be kept in sync by file system mirroring.

What components can be shared and what needs to be kept separated among the platforms?
The shared parts can constitute the majority of the application. The parts that need to be kept separated and managed differently are those which are specific to the platform and either can’t be accessed by web technology or make up a wanted variation across the platforms.

How does the proposed methods and tools affect/limit the functionality offered by the application?
The proposed solution does not impose any considerable limitations to the application. Depending on the needs for the specific application, parts like a native user interface or lack of performance can be minor limitations. Except for that all features accessible by a native application can also be accessed by this setup.

The problem presented in this report, mainly the double work and maintenance problem, could have been a bit too wide. Not that the double work and maintenance problem is too wide itself but the different solutions to them releases a chain of new problems that have to be taken care of to create a working solution. The reason the chain differs between the solutions are their different nature. For example a cross compiled solution and a hybrid solution both need ways of supporting platform specific features but their solutions introduce different concerns and creates different following problem chains. If the hypothesis would have been more clear and the scope would be closer to the hypothesis, the silver thread in the report could have been clearer. Although the desktop application got a well working solution for this case and the number of requirements from the platform generally are much less than for mobile devices, the desktop platform got noticeably less focus than the mobile platforms. Because of the different focus found for mobile and desktop devices it might have been better to make a clearer separation between them and therefore more clearly show their respective requirements.

7. Conclusions

This report investigates the problems that occur with double work and maintenance when working with cross platform applications for desktop and mobile devices. Because of the natural variations between the platforms, one platform could not just be copied and ran on another. This opens up for new approaches and other ways to optimize the workflow in cross platform development.

It is concluded that common parts of the application has to be created artificially. These parts can then be reused on all the platforms. There are different approaches to create those parts but the basis for most of them are to either wrap an application written in a language that could be interpreted by all platforms, like JavaScript and HTML, or to cross compile the app to the native language of each platform. This makes up the basis to work with the application for all platforms in a common way.

It was found that Kubicom's application, which already had many web parts, benefitted most from the web technology approach. Especially because many parts of the application were already web implementations and the work needed to transform the application wouldn't be that great. Another reason is that the environment needed to setup and develop a web application is much less demanding and easier to get started with than a cross compiler. On the other hand the web alternative won't be the best fit in all cases; if the application needs the native user interface and design elements from each respective platform or if the application is very computationally heavy, a native cross compiled application would probably be better.

Now when there are ways to share common parts between the application platforms the first goal of the report, to improve the work with double maintenance, seems doable. The next step is to decide how the application are to be organized and how the code should be run.

Because it was decided to go with the web option, this is what was further analyzed. There was options to run the code either in application wrappers or as a standalone web application run in the browser. In this case the best option for Kubicom was to go with a mix. On the mobile devices there was a demand to access native features of the platforms. The mobile application heavily relies on tight integration with location tracking, the camera and photos applications and the file system for offline storage. Because neither of those features are required or very fitting for the desktop version this could be ran in a simpler way in an ordinary web browser.

The solution to the organization problem found by this report relies on a well defined application structure where each part has its own place. The structure is a component based solution with focus on sharing as many parts as possible but still offer extensions to utilize the platform's strengths. The structure impose relationships between the code bases of the platform that by the help of the file system can completely remove the double maintenance in the common code. By this the remaining parts of the first goal of the report could be seen as fulfilled.

To fulfill the second goal of the report to allow access to the platforms strengths, another step is needed. For the mobile devices there is a need for a small app wrapper that hosts the web application. This wrapper makes the app installable and with the help of a communication bridge gives the JavaScript part access to all the native

platform features. The bridge works both ways and could thereby be used to call both native methods and listen to native events.
With the prototype for Kubicom completed this solution works well for the tried example. Both the build process and the communication between the application parts work. The build script today leaves more to wish for to be efficient and has several possible automation steps. Although the prototype only constitute a minor part of a whole application and all the possible issues may not have shown, the prototype is successful and seems to be able to offer what Kubicom needs. It makes a great part of the application reusable directly between the platforms and thereby reduces the double maintenance. The new solution may also offer an extra dimension by reusing components between completely different applications.
The third and last goal of the report was to investigate the limits imposed by the solution. As the the platforms features can be fully accessible the limits is rather in terms of preference. For example the native parts of the user interface cannot be accessed but at the same time it is easier to develop a similar looking application for multiple platforms.

References

[1] Appleton, Brad et al. “Streamed Lines: Branching Patterns for Parallel Software Development” 1998

[2] Babich, Wayne A. “Software configuration management: coordination for team productivity.” Addison-Wesley, 1986.

[3] Charland, Andre, and Brian Leroux. "Mobile application development: web vs. native." *Communications of the ACM* 54.5 (2011): 49-53.

[4] Daniels, M. A. “Principles of Configuration Management, Advanced Applications Consultants., 1985”

[5] Dickson, Jared, "Xamarin Mobile Development" (2013). Technical Library. Paper 167.

[6] Gyllensvärd, Henrik and Welander, Niklas. “Reducing Double Maintenance for Web-based Application on a code and logical level” 2014

[7] Heitkötter, Henning, Sebastian Hanschke, and Tim A Majchrzak. "Evaluating cross-platform development approaches for mobile applications." *Web information systems and technologies* (2013): 120-138.

[8] Lakshman, TK, and Xander Thuijs. "Enhancing enterprise field productivity via cross platform mobile cloud apps." *Proceedings of the second international workshop on Mobile cloud computing and services* 28 Jun. 2011: 27-32.

[9] Mahler, Axel. “Variants: Keeping things together and telling them apart” 1994

[10] Tichy, Walter. "Software Configuration Management Overview." 1988.

[11] Winkler, Jürgen FH, and Clemens Stoffel. "Program-Variations-in-the-Small." *SCM* Jan. 1988

[12] Android WebView class description 20 May. 2015 http://developer.android.com/reference/android/webkit/WebView.html

[13] Apache Cordova 2 Aug 2015 https://cordova.apache.org/

[14] Appcelerator 5 Aug. 2015 http://www.appcelerator.com/

[15] Broccoli 8 Aug. 2015 http://broccolijs.com/

[16] “Gartner Says Worldwide PC, Tablet and Mobile Phone ..." 2013. 29 Jan. 2015 http://www.gartner.com/newsroom/id/2408515

[17] GNU Make 13 Sep. 2015 https://www.gnu.org/software/make/

[18] Grunt 8 Aug. 2015 http://gruntjs.com/

[19] Gulp 8 Aug. 2015 http://gulpjs.com/

[20] JSON 8 Aug. 2015 http://json.org/

[21] “RANKED: The Highest-Paying Programming Languages“ 13 Jun. 2015 http://uk.businessinsider.com/best-tech-skills-resume-ranked-salary-2014-11?r=US&IR=T

[22] Representational state transfer 8 Aug. 2015 https://en.wikipedia.org/wiki/Representational_state_transfer

[23] Symbolic link 6 Aug 2015 https://en.wikipedia.org/wiki/Symbolic_link

[24] WebViewJavascriptBridge 20 May. 2015 https://github.com/marcuswestin/WebViewJavascriptBridge

[25] What Is Web 2.0 1 Aug. 2015 http://www.oreilly.com/pub/a/web2/archive/what-is-web-20.html

[26] “Why mobile web apps are slow“ http://sealedabstract.com/rants/why-mobile-web-apps-are-slow/ 25 Aug. 2015

[27] “Xamarin” 2 Aug. 2015 http://xamarin.com/

EXAMENSARBETE Reaching Across – Managing Variants of One Application on Multiple Platforms
STUDENT Alexander Haraldsson
HANDLEDARE Lars Bendix (LTH)
EXAMINATOR Ulf Asklund

Apputvecking – en app för alla smartphones och datorer

POPULÄRVETENSKAPLIG SAMMANFATTNING **Alexander Haraldsson**

För maximal spridning av en app bör den finnas tillgänglig på flera olika enheter. Men skillnaderna mellan enheterna är flera, det är som att de talar olika språk. Kan man få appar att prata samma språk, oavsett enhet?

Inledning
När man bygger en app idag och vill att den ska finnas tillgänglig på flera olika sorters smartphones och datorer stöter man lätt på problem. Om man ger instruktioner till en smartphone eller dator, gemensamt kallade enhet, förstår sällan en annan enhet vad man precis har sagt. Det beror på att enheterna är olika i grunden; de stödjer ofta olika programmeringsspråk och har sina egna styrkor och svagheter. I mångt och mycket är programmeringsspråk lika vanliga språk, varje språk har sina egna definitioner och uttryck. För utvecklare av en app innebär skillnaderna och språkförbistringarna flera problem. Det blir som att utveckla samma app en gång för varje enhet, eller skriva en uppsats på varje språk om man ser till språkanalogin. Det gör att det både tar flera gånger så lång tid att utveckla appen och därefter ska apparna hållas synkroniserade; när det görs en ändring i en app ska den göras för alla andra också. Då är det lätt att missa något och det kan leda till olika appar som är svåra att underhålla.

Hur appar kan prata samma språk
För att komma till rätta med problemen behövs ett sätt för enheter att prata samma språk. Det kan lösas på olika sätt, två goda kandidater är att använda ett gemensamt språk som alla förstår eller använda sig av en översättare. En översättare när det kommer till apputveckling innebär att man skriver kod i ett programmeringsspråk och därefter kompilerar/översätter det för respektive enhet. Översättningen kan då göras automatiskt med ett verktyg. En annan väg att gå är att hitta ett gemensamt språk, motsvarande engelskan. I apputveckling kan det vara webbteknik; alla enheter för konsumenter idag kan tolka en webbapplikation.

Båda metoderna innebär en stor förbättring när det gäller underhåll och minskad utvecklingstid men är samtidigt inte fullständiga. För att utnyttja en enhets fulla potential krävs en integration med enheten och dess språk. Vissa uttryck finns helt enkelt inte i alla språk och då måste de hanteras separat. Anledningen är att komma åt funktioner som gör enheten speciell, det kan vara användningen av GPS, kamera eller filsystemet etc. Genom att ge instruktioner till enheten på sitt eget språk i begränsade områden och därefter bara kommunicera resultatet kan även detta lösas. Tekniskt sett innebär det att utveckla komponenter som består av både webbteknik och enheterns egna språk samt använda srpåkbryggor för att koppla samman dem.

Resultat
I det studerade fallet visade det sig att det gemensamma språket i form av webbteknik var den bästa vägen att gå. Det innebar en stor mängd minskad kod och bättre metoder för att hålla apparna synkroniserade. Därmed blir potentialen i tids- och kostnadsbesparingar god. Lösningen kan användas för majoriteten av projekt som kräver stöd för flera enheter med undantag för prestandaintensiva appar och de som kräver ett gränssnitt likt plattformens egna. I de fallen är översättaren i form av en korskompilator bättre.