MASTER'S THESIS | LUND UNIVERSITY 2015

Reducing Double Maintenance for Web-based Application on a code- and logical level

Henrik Gyllensvärd, Niklas Welander

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2015-03

Reducing Double Maintenance for Web-based Application on a code- and logical level

Henrik Gyllensvärd
Niklas Welander
Faculty of Engineering
Lund University

A thesis submitted for the degree of
Master in Computer Science

2014

Abstract

A problem within the mobile instustry today is creating third-party applications. Generally developer teams have to develop the same application for several platforms and each platform uses a specific native language. A result of this is that many applications both look and act similar; but have different code bases and work very differently beneath the surface. This is a double maintenance problem for developers that is hard to solve. If a new feature is to be added, each team has to work on making the same functionality work on each device.

Web technology, which refers to HTML5, CSS and JavaScript, have become another possible solution in writing native code on some platforms. If several platforms adopts the same technology, the development team does not need the same in depth knowledge about each platform and thus it will ease the developmet overall.

But the big problem with double maintenance still remains. Even if each application is written with the same technology, if the teams still develop each application separately from each other, very little is gained. But there might be a way to improve this. When applications are written with the same technology, we think that there is a high chance that a big propotion of the code is either the same, or could be the same.

We want to investigate if it is possible to combine the code base into a combined one for platforms supporting native web applications. This should reduce or possibly eliminate the double maintenance problem. But by combining the code bases we believe that it will introduce another problem, namely variants. Provided that we are able to combine the code bases and that it creates variants. We will also investigate how to work with the created variants. We also want to investigate if, and how we can minimize the amount of platform specific code for communicating with each platform's hardware.

After analyzing the code, making iterative experiments along the way and by researching different approaches on the code structure; we managed to create an environment controlled by a python script that simulates one combined code base, while it actually is separated into several different ones. To support variants, a specific markup was introduced that allows each platform to have its own sections of code.

Each platform has a native JavaScript API to access hardware and platform specific features; these differ a lot in structure, usage and function. By building an API that overlaps all platforms native API, we are able to reduce the developers needs to interact with multiple APIs. After studying, and experimenting with JavaScript API design conventions, we produced an API, which through user testing, was found powerful and easy to use. With this single API the development is easier to maintain, contain fewer lines of code and reduces the workload on the developers.

Foreword

The foreword will contain information about how the work was divided, who the stakeholders are, the audience we are targeting and lastly acknowledgements.

Who Did What?

Henrik's focus in the thesis is on the configuration management part. Most of the work relates to the creation, functionality and structure of the Python-script Gizmo and all of its relevant parts.

Details of Henrik's work

The file structure – How to represent the structure of the applications that Gizmo works with.

Investigated the double maintenance problem at code level – The double maintenance is a central part of the thesis and understanding the problems that double maintenance creates has been crucial for building a solution.

The python script – The script is used to keep as much of the code as possible shared between all platforms and at the same time allow variants within the code.

The markup separating the code levels – One of our results is that we could divide the code in three levels, high, general and low. The markup was introduced into the code for us to separate these from each other.

Niklas researched and structured the API, Quack. As an Interaction design specialist he also had a saying in the look and feel of the overall 'product'. To assert the functionality and viability of the API he conducted user testing to evaluate the user design in the workflow and the API.

Details of Niklas' work:

Researching apps for different platforms – Looked into how different operating systems use HTML, JavaScript and CSS when creating apps. I also looked into the native produced API per OS for interacting with the hardware.

JavaScript API Theory – Researched how a JavaScript API should be designed. What are the most common conventions and helpful features available in renowned APIs today?

Developing unison platform API - Quack – Developed under several iterations an API (Quack) connecting the functionality existing in the separate platform APIs for interacting with the hardware.

Developed an app for testing purposes – Built a simple application using Gizmo-Quack. The app could manage contacts and add them to the OS' contacts. Large chunks of the app's code were then removed and later recreated during the user testing.

Results and evaluation of User Testing – The conducting of user results: Planning, setting them up, getting the results and evaluating the Gizmo-Quack based on the answers.

Common interests

Researched app-development with HTML5 – We learned the basics of app development using HTML5, JavaScript and CSS. The platforms each require a specific development environment. We also researched how apps should be developed for each different platform.

Planning and product concept – Discussions behind the concept of the product. What is the problem an how can we solve it? Laying out the work plan and dividing the workload.

Merging Gizmo and Quack – The merging of Gizmo and Quack into a single product. Removing enough bugs for user testing and making a prototype.

Technical adaptation of interaction solutions - Any interaction orfrontend feature deemed necessary, by Niklas, in the overall workflow was instantly implemented by Henrik.

Stakeholders

The Thesis Authors

Our personal interest in this thesis has two parts. We are approaching the end of our education in Master of Science in Computer Science at The Faculty of Engineering, Lund University. To graduate we have to perform a master thesis

spanning over 20 weeks that will deepen our knowledge in a field that we have studied. The subject has to be related to a course we studied at advanced level. The second part for our involvement is that the topic itself is something that interests both of us and we strongly believe that the industry will move in this direction.

The Faculty of Engineering

The Computer Science Department interest is mainly in the academic parts of the thesis. The work performed can lead to new knowledge or praxises on how to perform configuration management in mobile development.

The Design Department will be able to pick up our research and extend their knowledge on the design patterns and conventions on mobile development. Our workflow might also set a good standard for developing applications with multiple designs.

ÅF

ÅF is a consulting firm and part of their expertise is to develop applications for various customers; thus they want to stay updated on the latest trends in the mobile industry. Apart from the interest in learning more about the new mobile platforms they are also interested in the possibilities of using a single code base for applications.

Unknown Interested Parties

Any company or organisation that wants to follow the trends of the mobile industry. It is very hard to anticipate how the market will be in a few years. An important factor for survival in the industry is to be updated on the latest systems and how to work with them.

Target Audience To gain full understanding of all parts of the thesis, a general understanding of both configuration management and interaction design is highly recommended. People with knowledge in computer science should understand the general concepts of the thesis and use it for their own purpose.

Without prior knowledge in the computer science field the thesis will probably not prove very helpful.

Acknowledgements

Lars Bendix

Lars Bendix has shown a great interest in the thesis and helped mainly with the configuration management parts of the thesis, but also been a great help in guiding and giving helpful advices to both of us. He's been a valuable source when writing our report and preparing our presentation.

Mikael Blomé

Mikael has been the supervisor from the Design Department. He has been able to provide guidance with the principals of user testing and how to extract useful data from it.

Thomas Hermansson

Thomas is the supervisor from ÅF and has watched over our work from the start.

ÅF

ÅF has our thanks for both taking in the thesis and for supplying location and workstations throughout the whole project.

Contents

Chapter 1

Introduction

Smartphones are spreading across the world. During 2013 smartphone sales reached approximately one billion units and it is expected to increase even more over the coming years. One important factor in the success of a smartphone is the support and the availability of third-party applications. When releasing an application it is desirable to reach as broad audience as possible; so an application should preferably be released on all available platforms. This is rarely the case though. The process of doing that is costly and requires both commitment and time. Even if the application will be very similar to the user on each platform, it works very different behind the scenes. One of the problems that developers face when developing for several platforms is called the double maintenance problem.

1.1 Problem Description

The double maintenance problem [8] is a recurring problem when developing applications. In general an application should both act and perform similar on all platforms. But many platforms use different coding languages to build native applications; a consequence of this is that a separate code base is required for each platform. So each new feature to be added has to be created in each code base, thus the double maintenance. This process is expensive and time consuming. There is no simple solution to this when the platforms use different native languages.

A recent trend in mobile development is the support for web technologies within some of the platforms. With web technologies we mean HTML, JavaScript and CSS. When several platforms use the same way of building applications the developers does not need to have the same level of understanding of different programming languages. But there are still some problems that are unsolved when building applications. Today there are no specifications on how, for example to use the structure elements of

HTML, the hardware interaction of JavaScript and the design of the app using CSS. The platforms implement their own standards of HTML, create their own APIs in JavaScript and their own design conventions for the operating system. When developing applications this will still lead to the double maintenance problem; since the developers have to write unique code for each platform to perform the same task.

1.2 Objective

Can the web technologies be used to the developers advantage to reduce the double maintenance? There will probably still exist parts that differ between the platforms; but it might still reduce the overall work load. If we know which parts that are unique, maybe it is possible to handle the variants [21] that is produced in each platform? This would reduce the double maintenance problem for developers and hopefully both ease and make the development of applications cheaper.

Before we started the actual thesis we spent two weeks analysing the structure of native web applications for the three web platforms, Tizen, Firefox OS and Ubuntu Touch that we looked at. The results from this pre-study helped us to categorise which parts of the code base that can be shared and which cannot.

Some functionality is defined in the HTML5 standard, like geographic location. A problem here though is that the HTML5 standard was initially written with the web-browsers in mind; thus several vital hardware functions for smartphones are not included.

What we were hoping for was a smooth solution for combining the code bases created from creating the same app for different platforms and unify the interaction with the hardware. The structure and the work flow would be easy to "jump into" and familiar to experienced web developers.

1.3 Methodology

Going into this project we knew we wanted a product that a developer really would want to use. So we tried to find the needs of a web-application developer and see if we could find out the difficulties a developer would face when developing apps for different platforms.

When we started working we decided on three things we knew we would have to do:

Research the platforms and application implementation deeper We knew we had to fully understand the requirements for an application to work if we were to tamper with the files.

Work iteratively with our prototype. If we want users to use the prototype their feedback is crucial. This would allow us to make any adjustments if necessary to make the prototype more user friendly.

Conduct user tests on our proof-of-concept product. Since we would be far from a retail product, and the environment would be undefined and free for the testers to explore, we chose to conduct small and dirty user testing [29], followed by a discussion where we talked about the overall product and specific parts of it [30]. The actual tests in the environment would be documented by one of us writing notes while the other one led the tests by giving instructions and guidance [29].

1.4 Thesis Report Disposition

In the chapters to come we will discuss the problem which is the base of our thesis. It will start with the analysis, after the analysis the design will come where we present the different solutions that we came up with for the problem in the analysis.

After we have presented the problems and the solutions to them comes the discussion where we will present our reflections on the choices we have done and what consequences comes from them.

Lastly, we have a conclusion containing a summary of what we managed to produce during the thesis work, how we tohught we did overall and what future prospects there might be with it.

Chapter 2

Background

This chapter will go through the fundamental parts about mobile application development. Knowledge about developing applications will both help in understanding the work done in the thesis and in general when developing applications. Android, iOS and Windows Phone are the most common platforms on the market today. But Firefox OS, Tizen and Ubuntu Touch are new platforms which all support native web applications that can lead up to an interesting shift in how we build applications.

First we will go through different kinds of applications that exist today and also how they differ from each other. After that there will be a short discussion about mobile platforms and some background around those. Lastly, an introduction to how you develop native web applications will come and some general thoughts.

2.1 Application types

Knowing what kind of different applications there are will help in understanding how they differ and what each platform's applications are capable of. Knowing the restrictions, costs and benefits of each platform can help reducing both time and effort in developing an application.

This section will describe the different types of applications that exist. The most common one developed today is the native application. The complexity and number of platforms has increased in the last years; so research has been done in the area to try to build hybrid and web applications.

Native Applications These applications have been built with a specific platform in mind. They are most often fast, reliable, and powerful but at the cost of being locked down to the current platform. This is the most common kind of application that is developed today [17].

Hybrid Applications Hybrid applications use a third-party framework to allow a developer to write parts of the application in another language than the one native to the platform. What the framework does is to convert some or all into the native language to allow it to execute on the framework's supported platforms [6]. Most of these frameworks has been developed to allow a wider cross-platform compability, today it is usually for Android and iOS [3].

Web Applications Web applications are written with the web technologies HTML, CSS and JavaScript. Not too long ago this was only available as websites through a browser. As the smartphone has become more common some sites now offer a website adapted to the smaller screens. These websites can look very much like an application for a platform.

A problem with these is that they do not allow the application to access the hardware functions of the phone; unless the phone itself has support for native applications to be written with web technologies [10].

2.2 Mobile platforms

This section will talk about the different mobile platforms that exist today. Obviously a mobile device or platform is more than just the language that the application is written in. There are differences in the controls, praxis of use and design conventions. All of these adds an element to take into consideration while developing for a device. Every device has at least buttons for increasing and decreasing the volume and a power button to power on and off the screen. Apart from these, every platform has additional buttons to control the phone; this is another important part to know when developing for the platform.

This section will start with a presentation of platforms that does not support native web applications. It will be followed by the platforms that does support these applications.

2.2.1 OS native languages

The three platforms, Android, iOS and Windows Phone are to date the three dominating operating systems on the market. Combined, they stand for over 97 percent of the market share [16]. Most of applications release today is for primarily Android and iOS while an increasing amount is also being released to Windows Phone.

Android Android has in a few years gone from the new kid in the neighbourhood to becoming the dominant player in the market. The system is developed by Google as an open source project.

Android has changed the way their hardware buttons work over the years. Right now Google has software buttons, this means they are part of the screen and uses a back, home and last active applications buttons.

Native applications to the platform are written with Java and XML. See more at homepage [14].

iOS iOS was released by Apple in 2007 and started the current era of smartphones. The whole operating system is closed source code. Apart from the buttons mentioned above, volume up and down and a power button; the phone only has one more button, the home button.

The system uses Objective-c and recently Swift to build native applications to the platform. See more at Apple's website [7].

Windows Phone 8 Windows Phone 8 is Microsoft's contribution to the smartphone world. This is another closed source project; so only Microsoft has control over the software. Apart from the volume and power buttons Windows Phone has an additional three buttons, the back, start and search buttons.

The operating system uses C Sharp for creating native applications on the platform. See more at their website [23].

2.2.2 Web supported platforms

The platforms Firefox OS, Tizen, Ubuntu Touch and Sailfish are platforms either recently released or soon to be. They come with or intend to support native web applications; this allows them to run applications written with HTML5 and can also access the phone's hardware. This is a relative new way of building applications and could lead to making the process much easier; since the platforms uses the same technology for building them.

Firefox OS Firefox OS is developed by the Mozilla Foundation. Their goal is to have a platform built with and for the web. The system has been released for the lower end of the markets in South America and some countries in Europe with more to come around the world.

Unlike many of the other platforms, Firefox OS can only run web applications. The controls of the smartphone are similar to iOS, they have a one button on the front of the phone that works as a home button. Since the device use the home button design, the app navigation of the phone has to be done entirely in the app. See more at Firefox OS' website [13].

Tizen Tizen has a long history, not as Tizen, but as various other systems before. Currently Samsung is the biggest contributor to the system. Tizen has so far only been released on some of Samsung's smartwatches. No official release date for any smartphone is currently set. One of the rumoured reasons is the lack of applications for the platform. Tizen has a back, home and menu button.

Tizen supports two ways of building applications, apart from native web applications, the platform also supports applications written in C++ [25].

Ubuntu Touch Ubuntu Touch is a mobile version of the desktop operating system Ubuntu. The responsible company for the development is Canonical. Ubuntu Touch has been release in an initial version for developers and has its consumer launch set in certain markets during 2014.

The Ubuntu Touch system has two alternatives to build native applications. One is through the use of Qt Meta-Object Language (QML) and the other through web technologies.

Ubuntu Touch stands out a bit from the other platforms as it is controlled more through gestures than either hardware or software buttons. [11]

Sailfish Sailfish is another competitor in the mobile market. The platform was released at the end of November 2013.

Like Ubuntu Touch, the Sailfish OS relies on gestures and software buttons within the phone to navigate around the phone.

As of today the platform only support native applications created with Qt. They have said they have plans on adding native web applications to the platform, but no date has been set for a release [18].

2.3 Building native web applications

When developing for the web mainly three languages are used, HTML, CSS and JavaScript. To put it in a simple metaphor, you can compare web development to

the human body. The HTML code is the skeleton that keeps everything in place. CSS will control the visible parts and decide the looks of the body. JavaScript allows the body to function and come to life.

HTML has been used in web development a really long time. With every iteration some aspects have been improved, but one thing that has always been the same is tags which are the core way of writing HTML. These tags are containers with different function which are used depending on the information you want to display or how you want to display it. Some of the basic tags are tables, paragraphs, headlines and images [27].

With the emergence of HTML5, the developers wanted to specify and create tags specialised to the best practise for developing to the web today. For instance there is usually a header, some sections, a side bar, a navigation bar and a footer; so with HTML5 we get the tags <header>, <section>, <aside>, <nav> and <footer> respectively [27].

With these detailed elements you might think that the development would get unified, right? Well, not necessarily. The double maintenance will still be a problem due to the differences when developing for the platforms. The difference of knowledge needed for developing for the web based platforms are still significantly high. Even if there are tools available to simplify the development; and in theory the implementation style can be identical. There are no restrictions enforced on how to write the code, so each developer can use different elements for the same purpose. There is always the choice to do what you think is best.

Chapter 3

Analysis

In the introduction we mentioned the double maintenance problem when developing the same application for different platforms, and how it can be expensive for companies to support several platforms. We also mentioned the arrival of web technologies and how some new platforms support native applications written in HTML5, which should simplify development as all platforms then would use the same technology. While this is true, there are still vast differences between the web platforms Firefox, Tizen and Ubuntu Touch in the structures and APIs they use. Due to these differences the code will still never be identical between the platforms! So there is still a need for separate code bases for the same application on different platforms. So it is still the same problem as before, double maintenance.

Developers also have to learn details about every platform they develop applications to. This process is time consuming, and it is not enough to learn it once. New features are added to the platforms over time, so the knowledge about the platforms has to be refreshed.

What we want to know in this thesis is how do we solve or reduce these problems? Can we, and then how, do we reduce the double maintenance problem now when the code bases created have the same language? And, how can we unify the development so that we can reduce the resource costs when developing one or multiple applications for HTML5 supported platforms?

In this chapter we will try to dig deeper into our issues at hand. We will try to explain the main problem and what it leads to by presenting several consequences caused by it. We will also present the findings of our initial work where we spiked and researched about web application development. With those findings we dig even deeper into the core problems. Everything mentioned will be wrapped up in a conclusion at the end of this chapter.

3.1 Application Development for HTML5 supported platforms

With the arrival of HTML5 based platforms, like Firefox OS or Tizen, a lot of the problems associated with application development like multiple development languages, several platforms, design and structures should be in the past. Unfortunately it isn't so. When introducing HTML5 support, the developers for each platform created their own file structures, conventions and support for the operating system. What this does is creating a rift almost as big as it was before. There are still problems with double maintenance and the resource cost of supporting multiple platforms.

First we will discuss the practical issues when developing for the platforms, including the use of HTML5, the double maintenance problem and what happens when you want variations in for example functionality between the platforms. After that we will present the issues the management behind the development will run into; such as the time, cost and the knowledge base needed.

3.1.1 HTML5, A False Pretense of a Unified Platform

With the decision to include web based development and languages one obvious statement appears; we can use the same language between our apps on different platforms. This is a big improvement when compared to having to have to cope with totally different languages. But the only problem it solves is that the amount of knowledge needed for understanding the code is reduced. If a company would like to develop an application for two web supported platforms there would be large enough differences for the need for two development teams. This is also a kind of double maintenance, as the two teams are performing the same work but in different ways. Even if the programmers could decide upon one way of writing the HTML5 code, there are still things that would lead to duplicated code and fragmentations, for example hardware interaction, platform based help files and design conventions on the target platform.

Even if application developers could decide upon a convention and stick with it, it would be very hard to uphold because of the platforms. Just by looking at the two platforms we have mainly researched, Tizen and Firefox OS, we discovered in sample files and official tutorials that the usage of HTML, JavaScript and CSS was very different. Firefox often tried to encourage the usage of the new HTML5 elements such as section, header and footer, while Tizen used mostly divs [27], an undefined container element, which they gave id's and classes, and then altered them programmatically.

When it comes to JavaScript it is easier to use the same code in different projects (assuming you are using identical HTML between the applications). Some lines of code can actually be copied between files in projects and work with no errors, this was never the case with say Android vs iOS; assuming you are writing an application with no more features than a web page, the JavaScript could be identical between platforms. But then again, we are no longer developing web pages. When writing applications for a mobile device we would probably like to be able to access the hardware functions or the platforms core elements; for instance the camera, album features or contacts. As soon as you want to do this the difference in the JavaScript code is inevitable and is so different it can't be compared to each other anymore. In a way this is also double maintenance; the developer teams need to keep themselves up to date with the platform distributions.

Each platform has written its own API for interacting with the platform. These have nothing in common with each other. They use different JavaScript praxis, different approaches on asynchronous calls and they represent objects (contacts, texts, settings) in different ways. It would be very beneficial for all parties if some standardisation would take place but this is probably just wishful thinking and if it happens it will probably take years before it is in place.

In addition to the application code, the platforms have a different setup of files needed for the application to run. These files are used for describing the application to the system so that you can give the application its title and an icon. Firefox OS use a ".webapp"-file which contain JSON to describe the information while Tizen uses an XML-file which resembles HTML in appearance; here is another instance where standardisation would help a lot.

3.1.2 Double Maintenance

The problem when developing several versions of the same applications is that it creates a problem called double maintenance. In short it says that there exists several versions of the same thing that is updated separately. A typical example of this is that a piece of code is copied to another place within the same software. At a later time a bug is detected within the old code, and eventually gets fixed. But no one remembers or knows about the code that was copied before the fix, so the bug will remain there until the same bug is detected again in the copied code. But the problem does not stop there; the bug in the copied code might be fixed in a different way. Over time the differences between the two parts will grow and soon no one will be able to tell that it was the same code in the first place. If the code was not copied in the first

place but the program was built to call the same piece of code from two places, these problems would go away.

As mentioned before, it is a big improvement that the applications can be written with the same technology. It makes the code comparable between the platforms. Even though the applications in big parts often should both perform and behave the same they still require their own code base; there are still differences in the platforms that will not change only because the applications are written with HTML5. These differences in the code bases has to be kept separate and thus the double maintenance problem remains.

3.1.3 Variations in Platform Distributions

Each vendor has grand plans for their platform, both in ways of how it should look and behave. But these visions are seldom compatible with each other and require developers to learn each of the vendor's visions for their respective platform.

One of the differences in the visions is how to interact with the phones, both physically and in the graphical interface. Physically some systems, like Tizen, rely on a physical back button on the phone that has to be implemented, while phones running Firefox OS do not; instead they want a software back button that is implemented in the header of the application within the user interface.

The many variations are a problem when designing the application. Each platform has wishes on how, for example the navigation should work. These could of course be ignored. But users usually appreciate when applications follow the platform's native navigations, they feel familiar with it, and thus safer to use it. So it can be preferable to strive to follow the native navigation with both the look and usage. But taking each of them into consideration makes the design and implementation more complex.

3.1.4 Different Platforms, High Resource Cost

The problems with double maintenance and knowledge of each platform are seen from the developer's point of view. So far it has been about how hard it is to handle double maintenance when working with the code and how the developers have to learn how to develop for each platform.

But there is more than just the developers behind the creation of an application. The management behind it has an important role. The management is involved in structuring the applications' features, look and feel, and is responsible for the resources needed for developing; these resources are mainly developers.

If we for now see developers purely as a resource for a company we can find two important aspects where double maintenance is an issue and the problem of having multiple platforms emerges: time and knowledge; both of which can contribute to high costs for the company! It would greatly cut the cost if a company could re-use as much as possible when it comes to code, API and even how the look and feel of an application is designed.

3.2 Initial Exploration

When we started working we had a general idea about the problems that mobile application developers face every day. But due to a lack of experience and a little confusion on exactly where to start; we spent some time researching and reading about the platforms. After that we more or less started building a prototype. The prototype would place the developer in a workspace and they had to compile a build to get the applications for each platform assembled.

When discussing what we had done with our supervisors and the developers at ÅF we realised that we were on to something which was working, it was just not good enough. We managed to structure up our work a lot more and get a better focus on the core problems. So the time was not completely wasted; it gave us knowledge and material to discuss how to continue on the thesis. So in the end the time spent taught us some useful lessons.

The first basic prototype was not very good and got scrapped more or less straight away after a meeting with our supervisor. It was not very user friendly and took too much time to use for it to be software that anyone would use.

But the time and discussions made us realise that we had to find the common parts of the code. That is, the code that could be shared between platforms. This would also give the parts that have to be specific to a platform. If we managed with this it would hopefully greatly reduce the double maintenance problem. We did not believe that we could remove the double maintenance completely; and to only reduce it would instead introduce variants. But variants are easier and better to work with than having the whole double maintenance problem.

Another thing we discussed was how developers like to work and create applications. Our product has to be something they would want to work with. So we decided that we would try and focus on solving the problems from a developer's perspective.

The third thing we discussed was about learning the interaction between each platform. Can we find a way to remove the necessity for the developers to learn all

of them? This would eliminate another kind of double maintenance. The developers would then only have to learn one API, instead of several.

When trying to overcome the obstacle developers have in learning the details of all platforms, we were inspired by the Gaia Building Blocks in Mozilla's development structure for Firefox OS. The point of it is that you declare a simple HTML element and then you can add classes and ids to it to change its appearance.

We thought that by building our own building blocks we could acquire two things. First, make building blocks with a generic design so that an application would look the same on every platform; the other thing we thought about was making building blocks that was adopted to the design-conventions on the platform. These blocks would include both CSS (styling) and JavaScript (functionality).

We researched the design conventions and general application development for Ubuntu Touch, Tizen and Firefox; but in the end we decided that building blocks was not in our scope and we tried to adjust the interaction in another way. Mainly because we realised the amount of work behind it, perhaps even an entire separate master thesis. Even though some time was spent on this research and spiking, the knowledge behind application development would prove useful when we changed focus towards easing the interacting for developers when developing applications.

3.3 Conclusion

Today it is common to develop the same application for different platforms. This often requires one team per platform; because each platform uses a different native language, so the code bases cannot be compared to each other. Some platforms have started to build native web applications using web technologies. Since these platforms now use the same coding language their code bases can be compared to each other. But it is not enough to reduce the double maintenance problem. With more platforms the developers have to learn details about even more platforms to be able to develop for them, a form of logical double maintenance.

The root problems we want to try and reduce are the double maintenance problem and the steep knowledge curve with learning all the platforms.

Through our initial experiments we learned that we will try to combine as much as possible of the code and create variants for the platforms.

The experiments also gave a better understanding of how the HTML5 language works in combination with the platforms. The developers have to learn a lot of details about every platform to write a working application. We want to minimise

the amount of knowledge the developers need to have concerning each platform and if possible create our own standardisation.

Chapter 4

Design

The analysis concluded with the root problem of the double maintenance problem when developing native web applications. It exists both at the code level but also at a higher, administrative level, as developers have to learn details about several platforms to perform the same task. This chapter will include discussions and proposed solutions to these problems. At the code level there will be some design alternatives on how to combine the code that can be shared between the platforms, and thus reducing the double maintenance at a code level. By combining the code at that level it will instead make developers work with variants to assemble the application. This chapter will contain the design choices for how to build support for the variants. The other root problem is the double maintenance at a logical level. The alternatives for how to remove the problem of this double maintenance will also be shown and discussed with the pros and cons of doing so. By designing a solution separately for the logical and code level of the double maintenance we were able to create a joint prototype that we performed user tests on, which it passed with flying colours in our opinion.

Firstly there will be the design of how we structured the code; this will be followed by a section about how to reduce the double maintenance on the code level and work with variants instead. Next is the design for how to reduce the logical double maintenance. After this there is a discussion of what is accomplished when these two solutions are combined and how they work together. Last in the chapter you can find the results from the user testing, what did the developers in the user tests think of the solution we came up with? What can we learn from this?

4.1 The Three Code Levels

To build web applications we know they are built using HTML5, CSS and JavaScript. To understand and try to combine the code bases for an application we had to look

on the programming languages and see how they work.

The languages in question are HTML5, CSS and JavaScript. Each has their respective usage when creating an application. Since we want to reduce the double maintenance problem we wanted to be able to classify certain code as general. With general code we mean code that can be the same between the platforms. We started with the assumption that all code can be in the part that we call General. We already knew this was not true, if it was, there would not be any double maintenance problem; but we think it was easier to work from that standpoint and then show that certain parts are not general.

We both examined and discussed back and forth with other developers about what parts of an application that will be specific to a platform. What we came up with was that the calls to a platform's hardware and core functionality will always be specific. The HTML5 standard has defined certain functions to make development more consistent, but as soon as a platform vendor wants to do anything outside the standard, they need to create their own version to handle it. Even though the platform vendors want to achieve the same functionality they have their own preferences or knowledge so their solutions will differ from each other. These calls were the only thing we found in the platforms that has to be specific. As these parts are very close to the platform, and something that a user never will see or notice, we decided to call this area the Low-level.

But development of mobile applications is not that easy. The vendors have their own visions on how their platform should look and behave. A developer does have the option to ignore these ideas and create a mobile application that looks the same on all platforms. Though there is good reason to at least take respective platform's wishes into account and create a Graphical User Interface (GUI) more native to the platform; doing so can give the users a better experience when using the application. But doing this would separate that code from the General code. The difference here though is that this part is not mandatory; because of this we do not want to get it mixed with the parts that are required. This code will mostly contain code that the user either sees or interacts with and we decided to call this the High-Level.

This gave us in total three layers that code can exist in, the High-level, General-level and Low-level or short HGL, see figure 4.1. With these three layers set, we asked around and discussed if we could find any additional layers to use. We concluded that there might be, but they would then be a subcategory of either the High- or Low-layer. If the segregation is of platform specific code it will be under the Low-layer, and if it is a voluntary variant it will fall under the High-layer. If a more fine-grained

control is required of the layers they can certainly be divided up more, but for our thesis this generalisation serves us enough.

There are some advantages to the model that we can see; there are few layers and code within each area has a specific purpose so there is no grey zone to classify any code to. We also see that the model might work for more areas than the development for mobile web applications; both for other languages and platforms. It might be just as applicable on for example software built for Windows, Linux and Mac. The model then becomes very general and it has made us believe that something similar to our model has already been defined before. We see two reasons for that we have not found anything similar. First of all, and what we think is the most probable, is that we did not spend a lot of time searching for one. The second alternative is that there has not really been any use to define something similar before.

On the contrary one might say that a disadvantage of the model is that it is a very general model without any details. But as mentioned, we decided to not divide it up in any more parts as this model will fit our needs for the thesis.

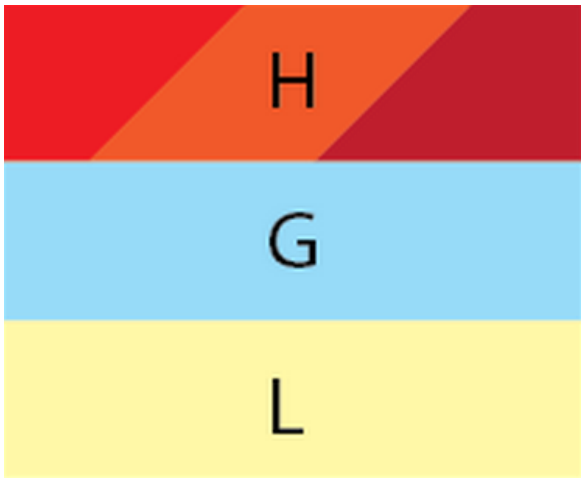

Figure 4.1: Demonstration of the HGL model

4.1.1 High-Level

The High-level is the optional part of an application that is separate from the General-level. It is something that will vary depending on what kind of application that is created. It is possible that the High-level is completely empty if the application has the exact same appearance on all platforms since the styling with CSS in HTML is unaffected.

4.1.2 General-Level

This is the part to maximise, to minimise the double maintenance problem. The General part is for all the code that can be shared among all platforms. In a perfect world this would stand for 100 per cent of the code, alas we know that it is not possible to achieve since the platforms always have slight but significant differences.

4.1.3 Low-Level

The Low-level represents a platform's APIs to access the hardware functions. As long as these functions are used, there will be a variant for that part on each platform that needs to be taken care of. The layer's size will depend on the amount of calls that the application needs to perform to the platform's specific functions.

4.2 Working With the Double Maintenance

This section will address the problem about the double maintenance problem. We have the three levels of code. It is the General-level we want to combine into one, to reduce the double maintenance. What about the High- and Low-level? Having one combined code base and handling the High- and Low-level separate will create one variant for each platform. The price is a low one to pay, but the developer has to get more support from a tool to handle the variants. So while the variants reduce the complexity for the developer, it shifts the complexity of handling the variants to a tool that the developers will use. There are many aspects to take into consideration when working with variants and hopefully most of the important ones are covered in this section.

First there will be a short introduction to different variants. Next a discussion about how it is possible to work and structure the the applications' files within the operating system's file system comes. Then a section about how to support the HGL model within files in the file system comes. After that the structure of the different possibilities for workspaces comes. That in turn is followed by the different structures for the assembled applications, which will be ready to run when in this structure. Lastly a section about how to work with the applications within different repositories comes.

4.2.1 Variants

This section contains descriptions of different variants. Each variant works in different ways and all have specific pros and cons. The variants will play a key role in how to reduce the double maintenance on the code-level. The models are described by Babich [8]. Mahler has also done extensive work on how to work with variants that has helped us in the thesis work [21].

Separate Files Separate files (SF) works much as it sounds. Each variant has its own file that it works in. These files have no superior one, all is of equal value. To clarify, we have platforms X and Y. File A is shared between both platforms and contains what we call General code. For separate files to work no High- or Low-level code can be placed within file A. So both platforms have a file B that will contain all High- and Low-level code. So when executing the application on platform X file A and platform X's file B would be used.

A positive thing about it is that each file can run on its own and the system only needs to pick the specific file without any additional compilation in between.

A problem with SF is that even though file A does not have any double maintenance. Each file specific to a platform will, in this case file B.

Delta A delta configuration does as SF keep separate files for each variant. The difference lies in that it has a master file that contains a complete, fully working version; and the other files contain parts that will modify the master file on certain places temporarily when needed to create the variant needed.

Delta's advantage comes from that when a bug fix is done in the main file it will also be fixed in all the variants, since they will use the same file.

Delta's has two big disadvantages, first is that if the main file is lost, apart from losing the main file and its variant, all the other variants will also stop working as it was using the main file as well and modified it. The other disadvantage comes when representing transformations. How and where should the deltas change the main file?

Conditional Compilation Conditional Compilation (CC) is another method to create variants. But CC only uses one file. All the varieties in the file will have a condition that has will be fulfilled when assembling the code. Say again that we have platform X and Y. The file contains a CC that will depend on if it is platform X or Y that is assembled. So if platform X is built, all the code related to platform Y is removed when X is built.

There is no double maintenance problem with conditional compilation but it comes with other drawbacks. It will much faster become more complex to handle the different compilations. Say you have four platforms. All the code for those four will be in the same place and it can become hard to maintain.

4.2.2 Structure of Files

With the variants Separate File, Conditional Compilation, Delta and the HGL model in the back of our minds we had to work out different ways that this could work together for developing mobile applications.

This gave us two different models to structure. One is the workspace that the developers will work in and other represents the assembled, ready-to-run, mobile application. Each representation we came up with have its pros and cons and support for different variants that is discussed in respective sections below.

The section starts with a short introduction to platform specific files. Next a deeper analysis on how to work with the files that the application is built up of comes.

4.2.2.1 Platform Specific Files

To run an application on a platform, certain settings and information has to be set. These will be in files located within a platform. An example is Firefox OS' manifest file; it for example defines where the application's start file is and what hardware permissions the application requires. The other platforms have similar files, but they work in a different way. We know each platform's individual files beforehand, so they can be handled if necessary. What is important is to have the files in consideration when deciding the structure for how the developers should work when developing the application.

4.2.2.2 Working With the File System

Before defining on how to work with the developer's workspace and the applications we had to figure out how the developers should work with the files. The restrictions on how these are handled will have an effect on how the developers' workspaces can and should function. As mentioned earlier, the platform specific files have an important role. They must be stored somewhere that is logical for the developers or the software will be less user friendly.

The work we did to come up with how to work with the file system has been done in iterations. We had a rough idea at first on how the system should work because of our initial experiments and discussions. From the discussions with the developers we had a rough idea that each platform probably should have their own folder that they are stored in. The folders will then synchronise certain data between themselves; that is, a connection will exist between the platform's folders. For example if a file is modified we want the possibility for those changes to get mirrored to the other platforms' identical file. An example of why this is desirable is that the General code should be the same across all platforms.

We discussed other possibilities, like having everything in one folder, but we always came back to having one folder per platform. It would work in theory, but we do not think it would not be a very user friendly structure. They also in some way ended up being very similar to having one folder per platform, but more complex.

With an unknown number of platforms the solution has to support a varying number of platforms. We started iterative work on trying to find different methods for the developers to work with. We felt the necessity to have some software that allowed us to test our approaches. When discussing how to achieve this, based on our limited time and resources, we did not find any other way than to create our own software. An alternative was to use a specific editor that already exists. But we did not want to lock ourselves to a specific editor for two reasons. The first reason was that we did not want the prototype to lock down a developer to a certain tool. The second reason was that we did not know exactly what we wanted beforehand; so we could not look up in advance if the editor would support what we wanted to do.

We decided that the software would run in the background so that it would not be in the way for the developer. All that would be required would be to start it.

It might have been possible to do the thesis in theory without a prototype, but we think we would get better results this way; it helped us to show the current state of our thoughts and get better feedback. In the end we also wanted to continue with the software to build a proper demonstration. The software will be a big part of our result and instead of only calling it the software we gave it the working name Gizmo.

Holding it Together The software we decided to create is built with the programming language Python. It was chosen because of our prior knowledge and the support for the major desktop operating systems Windows, Macintosh and Linux. The language has been around a while so we feel confident that the script should be able to work for a long time ahead. It might not be the best choice for us, but we knew

Python would suffice for our needs right now. With the time constraint on our work we decided on Python as it would allow us to get started on the prototype faster.

Controlling the Synchronisation and Keeping it Together When building an application we have to store the files somewhere. This will be on the developer's computer's file system. Gizmo works by listening to changes occurring within that file system. It is able to detect changes when creating, moving, deleting and modifying files in a certain file path within the computer's file system. This allows it to act correspondingly based on the action that was performed by the developer. The creating, deleting and modifying events is self-explanatory. A note of interest is that when renaming a file it will count as a move event. Apart from that the move event will trigger when a file or folder is moved to another place.

With Gizmo running in the background, listening to the changes in the file system, it should in theory not matter what kind of editor the developer uses. One developer can use software X and another one software Y to work in the same files. Gizmo will not care until the file is saved, that is when the operating system will post a modify event, that the script will catch.

From a configuration point of view this could be a problem, as the changes will occur whenever the file is saved. A developer might want to restrict when the synchronisation occurs to have better control of the application. One problem with implementing this into Gizmo is the lack of a GUI; it would probably be possible to do it. But the solution would probably not be very user friendly. A better alternative would probably be to have a GUI within an editor where it could have an on/off flag for synchronisation. But again, we had a limited time to execute the work and as there was no requirement to support such a feature we did not dwell upon the problem much.

Another disadvantage of the solution with a script in the background is that is it hard to give good feedback to the developer. It might be possible to give feedback to the developer through popups. But we think it would probably be very annoying and hard to use in a user friendly way. A GUI would, maybe not solve the problem, but make it a lot easier to give that feedback. The lack of integration and GUI creates limitations for Gizmo as it will be hard to communicate if some actions are acceptable to perform or give warnings. With the limited time we did not look too much into it as we think it would take a lot of time to make it work properly. So for our solution now there can never be any actions performed that might need the developers approval before performing it.

Supporting an Unknown Amount of Platforms As mentioned earlier, Gizmo will have to handle many platforms, but the number of them is unknown. There are two things to have in mind here. First is how to decide how many platforms there are and secondly, how it should be structured in the file system.

When it comes to the structure of the file system there are different ways of solving it. One could for example have a file where the paths to the platforms are given, and the script will read them when it starts. Another alternative is that the paths will be given when starting the script as parameters. But we felt these approaches were just in the way for the developer. We skipped the option of having one file with the paths. We believe it will only be a source of annoying errors. Say that the project it moved on the file system, then the developer has to go in an update the path to that platform or the project will crash. Another problem is if several projects exists, how to decide upon which to start? The second option requires the developer to keep track of the path for each workspace whenever starting the script, which also is error prone; it is easy to remembering wrong or giving the wrong path.

What we decided upon was that there is one folder per project, and within that folder, any direct sibling folder will be counted as a platform. So when the script starts with the workspace path, it will count the folders within it and count all of them as a platform. This should remove work for the developers and make it more user friendly. The drawback of this is that there cannot be any other folders within that hierarchy level and that we do not know exactly what platforms the script is building for. But we felt that the ease to use is more valuable here. For the current prototype there is no reason to know what platforms it is working on. This would be easy to change if desirable. This does also solve the problem with the paths. The developer does only have to know the path to the workspace that is currently being worked on and the script will solve the rest.

Separating the Files and Telling Them Apart One big problem arose when working out how to combine support for platform specific and shared files so that they both work at the same time. The problems lies partly in identifying to what category the file belong to, and then react accordingly. Let's say there are three platforms, X, Y and Z. The developer creates a shared file in platform X. Gizmo will recognise that a file has been created and will create identical files in platform Y and Z, so far so good. But what happens when a platform specific file is created in platform X? Right now the script has no way of knowing that the file created is specific, so it will mirror

the file to Y and Z like before. Similar problems occurs when modifying, moving or deleting a file, Gizmo has no idea what kind of type the file belongs to.

But say for example that it is possible to create specific files in each platform. This creates other problems when handling the files. Let's say there is a specific file named file A in platform X. There is also a shared file called file B in every platform. Within platform Y, the developer now changes the name of file B to file A. This is fine for platform Y and Z, but not for platform X. In platform X there is already a file A, so how should the script react? The change is done on file system level so Gizmo can in theory identify the problem. But the script is running in the background, and does not have any way to communicate with the developer and ask what to do. It is also impossible for Gizmo to guess what the developer mean, it would end in disaster very quickly.

Similar problems are easy to find when looking at the delete and create events when trying to combine shared and specific files. In fact we never found any solution where these two types of files can coexist within the same folder without any added restrictions; this is in large because we lack a GUI for the developer to work within. We did come up with two ideas for how to identify the files within the folders. The first one is to follow a naming convention and the second one is to use an ignore file.

Through our tests and discussions with the developers we realised that the most common file would be the shared files. To make it as user friendly to the developers as possible, we want those to be the most easy to use ones; so we wanted to put as few restrictions or special rules to follow on the shared files.

Naming Convention One solution we discussed is to follow a naming convention on the files. All file names starting with, for example "s_", where the s standards for specific, would be handled as a specific file by the script. This way all platform specific files can be created with the naming convention. It will be clear when looking on the filename what it belongs to. On the negative side it is not a very user friendly way and it is easy to get it wrong.

Another negative part can be that the platform requires a file to have a specific name, for example "manifest.webapp", and it will not allow it to be called "s_manifest.webapp". If any file has this requirement it would break the whole system.

Ignore File The second alternative we came up with was to have an ignore file. The file contains the names of the files and folders that are to be ignored. The written

names will then be ignored by the script when either of the events occurs on them. This does, just like the naming convention, allow any file to be made into a specific one.

The name of the file also becomes irrelevant, as long as the same name is in the ignore file, the script will work as intended.

On the negative part it is not clear from just looking on the file what kind it belongs to. Something else that can be seen as negative is that another file is introduced into each of the workspaces. It can be a hidden file, but there has to be one.

It does also allow a list to be made before the work starts. So when the file is created it is already on the ignore list.

Our Decision for Individual Support We decided upon implementing the ignore file within the script. Even though both have roughly the same pros and cons we see it as a big negative aspect that the developer has to use a specific tag within the name of the file to make it specific. There is also the aspect of that the file cannot be named anything.

But even if it is possible to decide if a file is specific or shared it does not solve all the problems. The problem when renaming file B to file A still remains, how do you handle the renaming if the file exists in another workspace? Some platform specific files have to lie in the root folder of the application. This further creates problems as this is where we want to put the shared files. The best way we saw to solve this was to separate the specific files into two categories, those required by the platform and the specific files that the developer wants to create. The required files are known beforehand and can be controlled. They will have specific names and will seldom, if at all, be files with HTML, CSS or JavaScript. The files where the developer's wants specific files will not have a strict naming policy and will probably have a much higher rate of change and be moved around.

By separating the specific files into two camps it allows the platform specific files to exist with the shared ones, provided correct usage of the ignore file. This works due to that the files are known before. These file are often not web-files, files containing any HTML, CSS or JavaScript. Because of this there can be one file like Firefox OS' "manifest.webapp" file and a "manifest.html" within the same folder, so they should not collide.

Then we decided upon having a folder within each platform called Specific that we added to the ignore file. This allows the developer to have individual files in each platform. Within that folder the script will ignore all events occurring, so in there

the developers have free roaming on creating any specific file they want. This was not strictly a requirement, but a wish from developers; they want things from both worlds.

The solution we chose is not flawless, the file B to file A problem can still occur that can cause some problems, but they should be very small with the choices we decided upon. There are variations to how to store the files that can solve the problem in better ways. What is important to have in mind is that a lot of these problems can be avoided if the developers worked through a tool and not directly with the files. As long as the files would be shown to the developer in a logical way; the tool could structure the files just as it wanted behind the scenes. The tool could for example allow us to use meta data on the files to decide if a file is for example specific of shared.

4.2.3 Workspace Structures

The iterative work we performed resulted in rules on how a developer has to work with the files to develop applications. We named the part where the work is done a Workspace. The Workspace is part of how to structure and work with the files within the file system. The rules on how a developer can work within the Workspace were decided based on the possible ways for us to structure it in combination with the developer's feedback. To go into detail about every detail of the iterations would take too much time. The workspaces here will be those that we found working, both for developers and that the file system can support.

The workspaces will be discussed one at a time. In each workspace there will also be pictures to demonstrate the model and discussions about what variants the respective workspace will support and how it can succeed with it.

Workspace 1 One way to represent the workspace is to start with one folder for each of the platforms. In an example with platforms X, Y and Z, this will give three folders. If file A is created in platform X, an exact copy of file A will then also be created in platform Y and Z. So each platform's folder will contain the same files at all time. Figure 4.2 shows the hierarchy and that each platform contains the same files.

When working within a file the behaviour will differ a little depending on what kind of code that is written. When General code is written it will be synchronised to the respective file in platform Y and Z, and vice versa, see figure 4.3 and figure 4.4. What happens is that the General code will actually be duplicated code in all

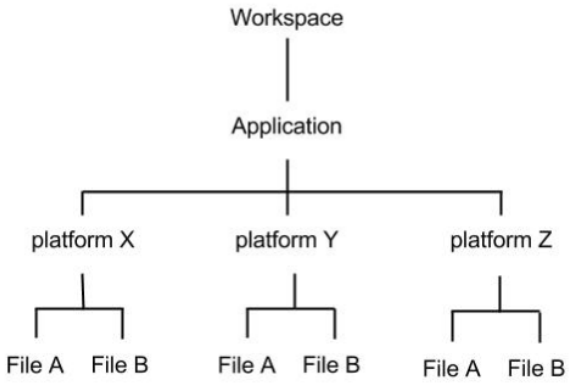

Figure 4.2: File representation of workspace 1

platforms, but it will behave like one synchronised code base. If code has been written in platform X and the developer then decides to continue in platform Y, all the latest General code will already be there. So the developer can continue in platform Y and know that it will also be transferred to both platform X and Z. So even with the duplicated data it does not introduce the double maintenance problem for the developers. We have not taken into consideration when several developers would work in parallel here. If using a versioning tool like Git it would probably give a merge conflict in all of the files at first. But when the conflict is solved within say platform X, it should also be fixed for platform Y and Z. But this is just in theory; we have not tried out the approach.

But we have no way of writing High or Low code to support individual behaviour right now; this is where the variants will help.

One reason for having each platform in a separate folder is that it allows us to have the platform specific files within respective platform's folder. It was an appreciated idea by the developers that we discussed it with. By doing this there is logic behind their location and easy to find when needed.

Another reason for this workspace is that it supports both of the application structures discussed in the section 4.2.4.

Variants for Workspace 1 The separate file (SF) model was described earlier, that each variant has its completely own file. Something similar to that model can easily be applied to the workflow described in Workspace 1. Every platform will already have a synchronised copy of each file. This means that no additional files have to be created for SF model to work. If High or Low code is written in platform X, those differences will not be copied to the other platforms; in this way a variant

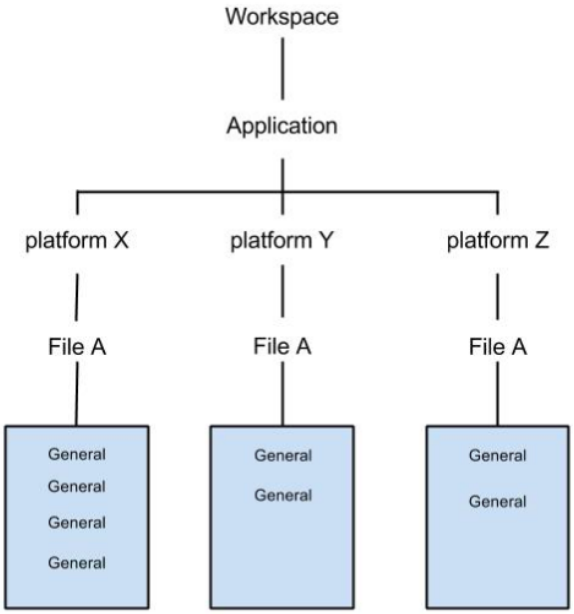

Figure 4.3: Before synchronisation of shared files

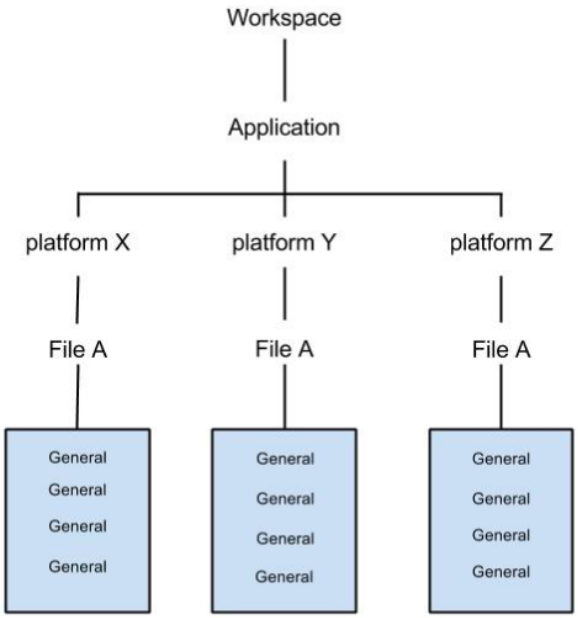

Figure 4.4: After synchronisation of shared files

has been created. An unlimited amount of High and Low code can exist in a single file and the region can be from one to an unlimited amount of lines. Figure 4.5 shows the state before a synchronisation has occurred. The change has so far only been done in platform X. Figure 4.6 shows after the synchronisation has occurred. As can be seen in comparison of figure 4.5 the specific field is nowhere to be seen in platform

Y and Z, just as intended.

The SF model as described above was said to create the double maintenance problem. While this is true it should not cause a problem here. The General code is synchronised and then only has to be written once. When a developer decides to write code for either High- or Low-level it does create double maintenance. But this is double maintenance from the start and has to be without further tools. What the structure of the workspace does is to help with keeping the problem minimal.

An interesting note is that the developer only has to know which platform is currently being worked in when writing either High or Low code.

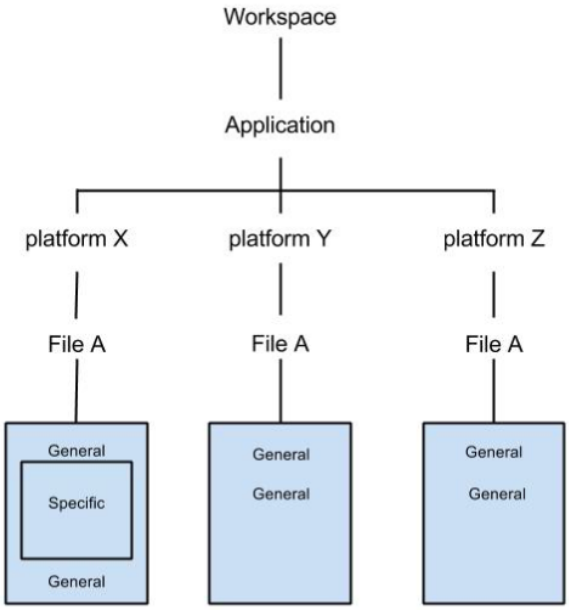

Figure 4.5: Before synchronisation

In this structure we do not see any purpose to use neither Delta nor CC as methods to help the developers create variants. Both of these variants need a master file that the compilations can work from. In this model there are no files like that since all files exist everywhere.

Workspace 2 The Second alternative is built upon the structure of Workspace 1. The difference is that another folder, called Common, has been added in the same hierarchy as the platform folders. The folder has a specific purpose and will only contain the General code that has been produced, see 4.7.

So when writing code within the either of the platforms, the code is not only synchronised to the other platforms, but the Common folder as well. If code is

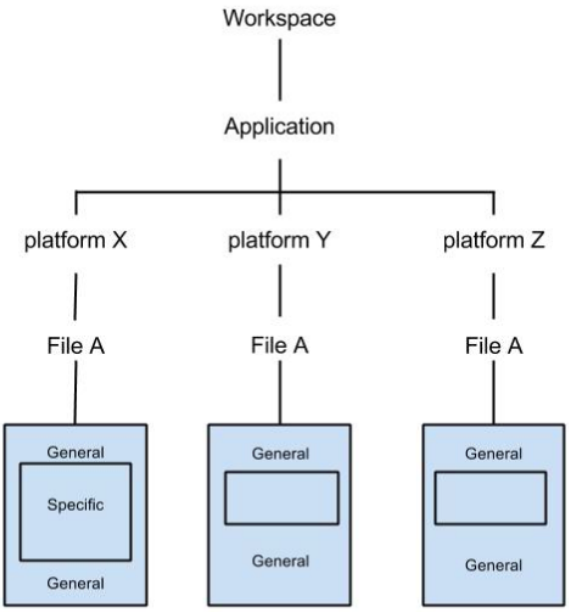

Figure 4.6: After synchronisation

written in the Common folder it will also be transfered to all platforms. The common folder behaves like it was another platform; any General code written in the common folder will be synchronised to the other platforms and vice versa.

Neither High nor Low code shall exist in any file in the common folder. This is a truth that can change with the support from variants discussed below.

The advantage of this file structure will become clear when discussing the possibilities for which variants that are supported.

The additional added folder Common will not make any difference in how to handle respective platforms individual files. They will only exist in the platform folders and not in Common.

This model will also support both of the application types discussed in section 4.2.4.

Workspace 2's modifications on Workspace 1 will not interfere with the support for the SF model to function. But it requires a developer to be more observant on what file that is currently being worked on. If either High or Low code is written within a common file it will not work properly. From the tool's perspective this is working as intended as it should not contain any High or Low code. But the developer will probably be a bit frustrated during development because of this. It is a problem that could be avoided with a better graphical tool that could inform the developer of the wrong doing. But in our prototype the developer has to be observant to what

file they are editing.

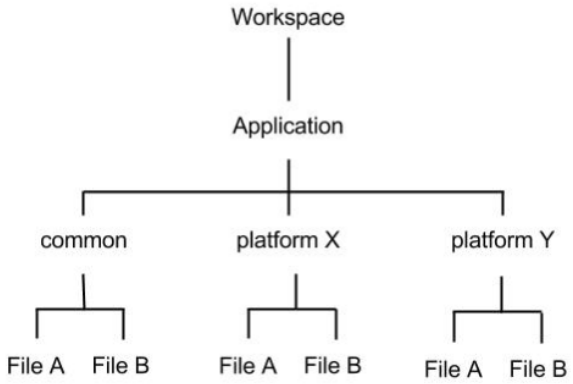

Figure 4.7: Figure describing Workspace 2

Variants for Workspace 2 The shared files in the platform folders still contain the support for the separate file model as it is based on Workspace 1. No additional changes have to be done for it to work the same. But it will not work when writing within the files in the Common folder. There is no reason for it to work in there.

One addition can be done here and that is the support for a model similar to CC. Within the files in the common folder it is possible to create something we have decided to call a Conditional Compilation tag. Let's continue with the example of having platform X, Y and Z. When the CC tag is created within for example file A in the common folder, a dedicated area for each existing platform is created, in this case three; one for X, Y and Z respectively within the shared file A. At the same time platform X will get one dedicated field within its file A and the same will happen for platforms Y and Z. See figure 4.8 on how the structure will look. This will create a bond between these areas. Even though they are areas for High or Low code, it will be synchronised within this tag. So High or Low code can be written in file A from the Common folder, and these changes will propagate out to respective area in the platforms. For example, if a change occurs in the common folder's file A meant for platform X, those changes will be synchronised over to platform's X file A as well, but not to the others.

Now this does not exactly work like CC, the normal case for CC is that is has one file and when compiled, it will remove the unnecessary parts. To make an example, if we would compile like CC; a file in the common folder would then be used and the code for the other platforms would be removed. What happens here instead is that if changes are done in a file in the common folder, that specific part of code is moved

into the platform's file. Now the advantage of doing it this way is that it will support both of the application structures discussed in the next section. If the common file was as with normal CC, only application 2 would be supported.

An advantage of working like this is that most of the applications can be developed from the common folder. All the general code will be synchronised as before. And the High- and Low-level code can also be written here if used in a CC tag and it will still work as if it was written in the respective platform's files.

The disadvantages of the approach are the added complexity for the support of the variant.

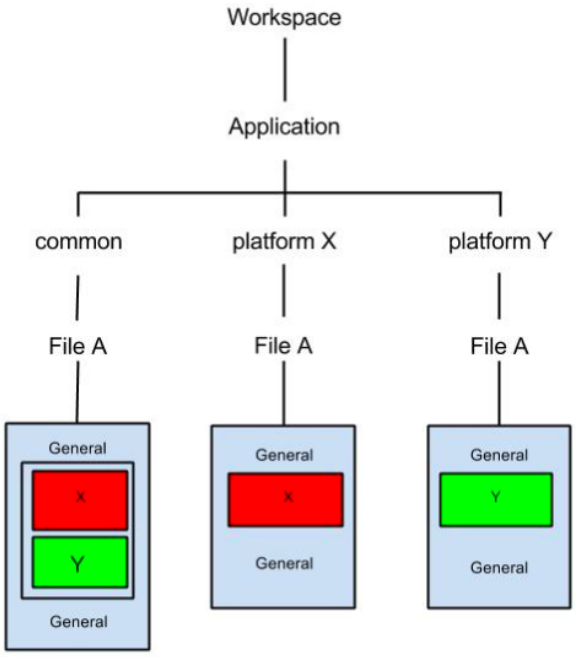

Figure 4.8: Conditional compilation in Workspace 2

Workspace 3 The third option for a workspace is a mutation of Workspace 2, all the folders, one for each platform and the common folder still exists. The difference in the workspace lies in the platforms' folders. They will no longer contain a complete copy of the General code; the General code will only exist in the common folder. So all work on the General code for all platforms has to be done in the common files. Work on a platform's specific parts, that is High and Low code, will still occur inside its respective folder.

The solution for how to work on the High or Low code in the platform can differ. One possibility is that there still exist a copy of each file, but they will now only

contain High and Low code. Another possibility is that each platform folder will only contain one big file that will contain all the High and Low code, see figure 4.8. It would require some system to link together the sections to the right file in the common folder, but it should be possible.

This will strip down the code and the developer only has one option for writing on the General part. When working, this could be less confusing as the developer will instantaneously know in what file he is editing; by seeing if the file contains General code. To write High- or Low-level code for a platform the developer would go into that platform's folder as before.

One direct drawback from this approach is that now only Application 2 will work. Application 1 might still work, but we think it would be extremely confusing for the developers to have the working application within the same files as the developer's files when they are not the same. Another problem would be how the naming would work; it could cause problems with files having the same name but contain different information.

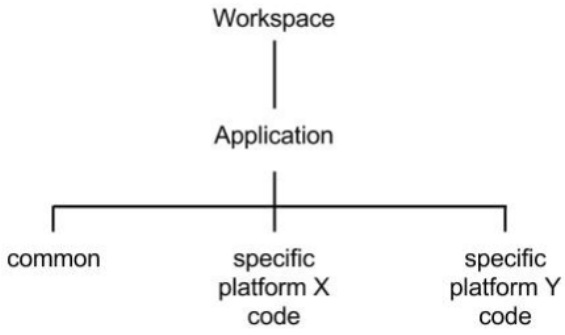

Figure 4.9: Showing the structure of Workspace 3

Variants for Workspace 3 Another approach similar to the CC tag described in the variants for workspace 2 can be used here as well. The difference is that when building the application it has to pull out all the code not associated with the platform currently being built. Depending on the decision on how to structure the files, the work flow will naturally be a bit different; overall it will work quite the same. But instead of having the common folder as a complement to get an overview, the files within it will now be the main one that the script works with.

The advantage of working in the common folder's files is that it is clear where the development is occurring; since there is only one set of General code. The platform's

files will have a very different look, as they do not contain any General code, so they are easy to distinguish from the common folder's files.

With the support for the CC tag in the common folder's files there is no explicit need for the files containing High and Low code within the platforms anymore. There is no harm in having them and they can work as a complement to the developers to get an overview of the High and Low code written for a specific platform. It would not introduce more double maintenance as the data will be synchronised at all time. But it will be hard to put it into context of how the code will work when run in the application.

Neither SF nor Delta model is useful here. The SF model because even if there would be exact copies of each file they would still not be of equal value anymore. The main files exist in the common folder and the development will happen there, and the CC tag should give enough help to the developer. The Delta on the other hand is not applicable here at all; there are no files in the structure for it to work on.

4.2.4 Application Structures

So the different approaches of how to allow the developer to work was done in the previous section about the workspaces. But there is an additional aspect to have in mind during development. To be able to run the applications on a device, there are certain rules that the platform forces upon the structure. If these rules are ignored the application will not execute properly on the device. We decided to call the structure that we will use, based on the platform's rules, the Application structure. Web applications do not require to be compiled into an executable file for it to run. Compiled software has to be stored within a compiled file and will look very different on the inside compared to the code that has been developed. But since web applications does not need this, it opens up interesting alternatives for how to store the code and at the same time follow the platform's rules. The work to come up with the structure of the applications has also been part of the iterations done when designing the work with the file system and workspaces.

This section will discuss the two structures the application can have. The applications will contain discussions of their respective pros and cons and how they fit in with the workspaces.

Application 1 The structure of Application 1 is the same as Workspace 1; this means that each platform will have its own specific folder containing all of its own HGL code and platform specific files. With the application like this there are two

options on where to store the structure; either the files coexists within the same folders as Workspace 1 and 2 use or the files can have its own folder outside.

If the files would be on the outside, the workspace has to be moved into the application folder, either as soon as a change is detected in the workspace or on certain occasions decided by the developer. If the second option is chosen, there is a risk that the application is not up to date when executed and can cause unnecessary annoyance as the developer forgets to make the synchronisation to the application.

Another negative aspect is that it might confuse the developer. Since the application will consist of the normal files that exist in the workspace they could in theory be changed within the application's files as well. The change will work in the specific platform that the change have been done in, but firstly it will not work in the other platforms and most importantly the changes will be overridden and disappear as soon as the workspace is synchronised over to the application folder again. It could possibly be avoided by having synchronisation between the application and workspace but the complexity and confusion becomes higher, so it was not something we would recommend. Another possibility is to make the files read-only on a file level. So the developer has to make a conscious choice to change any file.

One positive thing about it can be that there has to be some synchronisation to transfer between the Workspace to the Application. While this process occurs it can also have the possibility to change the code, something similar to compiling the code. For example it could be to remove white spaces or comments to trim the size. It would also be easier to store older versions of the application. When a change occurs it is easy to, for example, use a new name for the folder, and the old folder then becomes a backup of the previous version. Storing the previous version could also be solved with using for example git.

The other alternative is that it would coexist within the workspace, for it to work either Workspace 1 or 2 have to be used. Since web applications do not need to be compiled before they can run, the code within the workspace files are already ready to be executed by the platform. With both the workspace and application in the same folder the developer do not have to think about which version currently working on. It will always be the latest version in the workspace. This can be both an advantage and a disadvantage. It will remove some of the control for the developer to make it easier to use.

Workspace 3 is not compatible if we were to try and store them in the same folder, since the platform folders in the workspace do not contain the General code to make the application work.

Application 2 Application 2 is a further development of Application 1 and not very different. The difference is in the addition of a folder containing the common code, much like in Workspace 2. The folder will not be able to work on any of the platforms. The reason for having it will simply be for debugging purposes. It makes the most sense to combine with Workspace 1 as it does not have the common folder itself. There are not many advantages over Application 1, but it can serve a purpose.

4.2.5 Individual Code Within Files

With the Workspace and Application structures a developer can now build and deploy applications to the platforms. But so far there is no way of supporting the separation of High and Low code from the General code. Without this what we have done so far is of little help, but it will be addressed in this section.

We did not see any other way than to have some kind of markup within the files that will define different sections to separate the parts from each other. We still had work to do on the workspaces and make that work before continuing with this, but it was in the back of our minds during the iterations. This came to a resolution quite abruptly after that. On a meeting with several people working with configuration management we discussed the thesis work and the question about the markup came up, and the answer was the same; the only way we know is to do it through having a markup within the files. There are other ways of working with markup, for example using invisible markup where the area is indicated by a colour. But with the limited time and limitations of only having a script to work with; we could only implemented a markup through text within the files.

So we continued with the premise that markup was the only way forward. So what are the consequences of the markup? We have to use lines within the files to organise it. We also have to figure out how to use the markup to identify the differences of the code. We also asked ourselves if there is anything in common between the coding languages we had to support, HTML5, CSS and JavaScript; the answer to this is comments. Comments are a part of each of the languages and they will not disrupt the execution of the application when running. The advantage of that is that the markup would not have to be removed before assembling the application. So having the Workspace and Application within the same folders remained a working option.

Next we thought about how to identify these areas. For this we came up with two options. Either to have a comment that is analysed as a start tag for the markup and then have a number on how many rows below the start tag that belongs to the specific area. The other solution is to have a pair of tags, a start and end tag. Between these

two tags everything will be counted as part of the markup tag. We went will the second option. Having a number to say how many rows that belongs to the tag is asking for trouble. Developers would forget to update it and it is not a very user friendly choice.

So now we had a start and end tag that should be written as comments within the files. For our needs we had no reason to make a difference if it is High or Low code, so the solution can look the same for both, the tags can even contain a combination of both if the developer wants it. The three languages do not use the same kind of comments, so we had to build three versions of the start and end tag. Our initial proper proposal for HTML5 looked like this:

```
<!-- specific start -->
Anything in here will count as
specific code, and thus not be
synchronised to other platforms.
<!-- specific end -->
```

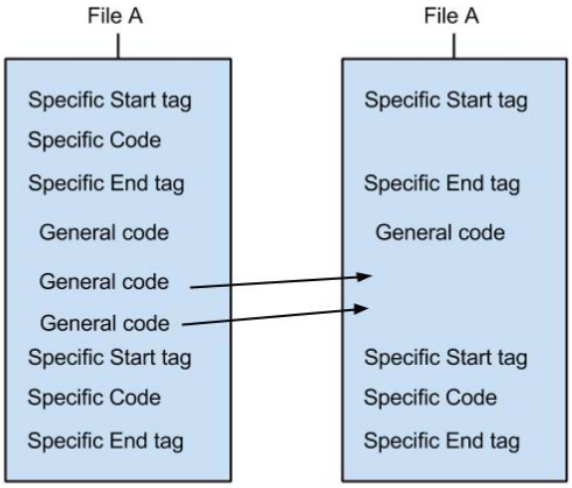

Figure 4.10: Only general parts are copied

Figure 4.10 shows a demonstration of how the General parts will be mirrored to the other files while the specific areas remain in the file works.

We tried the mirroring on the developers and discussed it afterwards; the functionality was received positively. They felt it was easy enough to use but it could be improved some. Figure 4.11 shows that it is not always very clear what specific tag that is the corresponding one in the other files. We felt that we had to find some way of creating a logical connection between the markup sections to make it clearer.

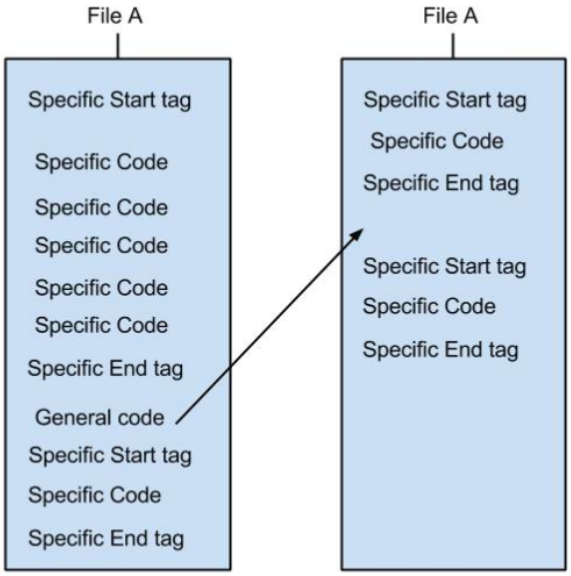

Figure 4.11: One problem with the tags

It was a question about organising the tags and that it was much to write. To solve that we added a unique ID to each of the sections so it could be identified easier. So the new version of the tags will look like this:

```
<!-- (302) specific start -->
Anything in here will count as
specific code, and thus not be
synchronised to other platforms.
<!-- specific end -->
```

To solve that it was much easier to write we made a shorthand command, this can be anything, but we chose "!sp". Sp is just short for Specific and the exclamation mark was added to make the combination uncommon. When the tag is detected by the script it will create a new tag with both the start and end tags combined with an id. The id's gets mapped to each other to give the developer a logical part in the code to keep the specific tags together.

Let's yet again use the example of the three platforms, X, Y and Z. One thing we decided upon doing is that if a specific tag is created within file C in platform X, tags will also be created within file C in platform Y and Z with the same ids. So when working in platform X the developer can easily switch to platform Y and see exactly where there is specific code in platform X. This will give a good hint about that there should be platform specific code here as well. If no specific code is needed

for platform Y the field can just be left empty; since it is only comments it will not disrupt the execution. One thing to remember is that since our solution is dependent on the file system to recognise the modification of the file, so the synchronisation will only occur when file C is saved in platform X.

An advantage of the markup solution is that as it is made it does not stop the execution of the application. Since it is only comments it will just be ignored when executed. At the same time it gives a pretty good overview of what parts are specific in the files.

The disadvantage we see is that it is quite fragile. Right now the script has no error detection and will only accept a 100 per cent correct tag. If either the start or end tag cannot be identified properly the code will then be counted as General code. It is easy to fix, but can create annoyance for the developers.

Another problem that we have not addressed is the coordination of how the ids will work when there are several people working in the same project. We did not look into this during the thesis. A way forward may be that because of that the ids only map to each other. They can be swapped to another id if needed, as long as all instances gets the same id. So if a collision is detected one of the logical instances could be assigned a new id that is not already taken. Another possibility could be to give the tags a unique identifier from each commit if a version tool is used. So the first commit would be 1-101. And if a second commit is added with the same id, it would have 2-101. And it would be easy to distinguish and solve.

A thing worth mentioning about this is that since the solution only relies on comments to work and not something more specific within any of the programming languages the solution of the markup should in theory work for any programming language that has comments built into it. The script does not care what kind of code that exists either inside or outside of the tags; it will work as long as it can identify the tags.

4.2.6 Resulting Design

The section has gone through a lot of different things. In some areas we only saw one possible way forward, like with the markup. But others we have had more alternatives, like with the workspaces. What we in the end decided to continue and implement was Workspace 1 combined with Application 1. So we had both of them in the same folder. With the platforms example of X, Y and Z, this would require three folders.

Both we and the developers saw that Workspace 2 would be a better option to develop in with the additional functionality. But as it is a further development of

Workspace 1 we decided to start with that, and in the end we did not have the time to build a prototype for Workspace 2 and test how it would work in practice.

That the developer does not work through a controlled GUI does create complications, as the developer will be working directly with the files and we did not find a good way to communicate with them. But we also realise that it is a dream scenario, as software like that would take a lot of time to build up. But it would remove some of the problems, like the importance of how the files are structured. The files could then be hidden from the developer. That would allow the handling of the files to be more complex in the background. Then it might be more appropriate to have another workspace structure and store the application outside the workspace.

From what we have to work with we feel that the solutions give a good way forward and we see the possibility to continue the work to continue on the prototype and implement workspace 2 and see the results from that. It should be possible to further develop the markup to implement more features to make it more advanced and helpful to the developers.

From a configuration's point of view we managed to build something that reduces the double maintenance problem and instead works with variants in the files. We are also open to the idea that there might be other variations of workspaces and applications that could work. The solution does have some drawbacks, such as we are not fully confident in the solution about sharing the specific and shared files as there is still the chance for it to break. At the same time we only built an experimental build to show a proof-of-concept. But so far the solution we have done has seen positive responses from the developers, so we would dare to say that the double maintenance was reduces and call the work successful.

4.3 Interacting With Multiple Platforms Simultaneously

To find the impact the two different APIs (Firefox OS and Tizen) had on a developer we developed a test application using some platform interaction. After reviewing the documentations we found one category which had custom representation of objects and some core functions for handling; Contacts. So we constructed a small app capable of showing, adding and deleting contacts existing on the device.

When we had one application for each of the two platforms we could find the practical differences and even compare the code written. Which was easier to read, fewest lines of codes and made the most sense?

During the time we worked with the Low part of the development, we wanted a working title which could match Gizmo; since we were researching and working with interaction or even speech in some sense, we chose to call it Quack. First we will make an evaluation of the two APIs (Firefox OS and Tizen) and then compare them to each other when it comes to hardware availability, similarities, their look and feel, and how we could combine them.

4.3.1 Evaluating the Platform APIs

If we were to in some way bypass, connect or in any way solve the problem with two (perhaps more later on) APIs we needed to get to know both APIs. To understand the problems that could arrise we made a comparrison of the two.

Overall Tizen seems to be more structured and based on the heuristics on C++/Java developers. You can tell from the structure that the developers wanted a more object oriented approach than what is uaually found in JavaScript. It also becomes quite apparent that the applications in Firefox is nested in an even bigger web application, since the entire OS is a giant web application. To access the hardware and OS you accessed the window and navigator properties which is typical for standard web development. Tizen instead includes an external JavaScript file containing their entire API and hardware interaction. This is then accessed using appropriete namespaces such as: tizen.contact.getDefaultAddressBook().

When trying to access the contacts of the device, which is the purpose of the experiment at hand, we wanted to know how this was used in regard to execution-time and order. Not for benchmarking purposes but to understand how the APIs was built around the concept of Asynchronous programming. JavaScript and web does not really have support for asynchronous usage since a web page will only ever have one thread to run on. Using events and tricks JavaScript makes it feel like things happen simultaniously when in fact it is a single thread jumping around doing everything.

When keeping this in mind we found out that firefox took advantage of JavaScript and let you offer callback methods at logically bizarre moments in the code. In Firefox you would create a "DOM Request object" from every hardware call. To this object you would then pass the methods you want to call when the operation was successfull or found an error. This passing of methods could be done anytime you want after the request object has been created. If the operation is not called untill the methods have been filled in or if the methods is fired up when it appeals to the execution is unknown

to us. But what we know is that Firefox uses some interpetation of "asyncronous" programming.

Tizen does not use this kind of asyncronous programming. Instead Tizen wants you to supply all the information available when using a hardware call. Tizen does not use any request object nor any methods for executing when success or error has been hit. Instead Tizen wants you to use try/catch statements and through these catch the success or error.

The result will be the same apart from the timing. Tizen's code will be launched instantly while Firefox's will wait untill the method has been passed and then called. This is something to take into consideration when trying to figure out a possible solution for the dual API problem.

After seeing the similarities and differences we started discussing possible solutions. While no final solution came to mind we found it possible to create our own API, and in the end this evolved into what we now call Quack. With some experimentation we knew we could somehow redirect calls between the platforms with this API.

4.3.2 Varying Availabillity of the Hardware?

When discussing if we could create one API for all platforms we started looking into what features were available for the different platforms. We were afraid that if the support were to inconsistent between the systems it would be hard to create a generic API. If it would have been the case then we could probably catch misuse in some fashion and sending the developer messages with the logging system, but it would not be favourable. We thought that the options available should be transparent and not hit you in the face mid-development.

As it turns out, the platforms have almost identical coverage of features. The usage of the features is also consistent within the platform, methods and structures are almost always called in the same way. The only thing really standing in our way for easily handling every type of feature is the way the platforms handle permissions. To be allowed access to the hardware one need to request permission from the configuration file (.webapp for Firefox and config.xml on Tizen). This will always have to be done manually. We had some discussions in creating a guide or wizard were the developer could create a project for multiple platforms but even though it's probably possible we deceided it was outside our scope.

With this we saw no problems with the availability of the hardware features and could focus on how we would create a API ourselves.

4.3.3 Combining Two APIs

We had now established that we wanted to create an API that could call both (all) APIs; Quack. But our overall solution and thesis is about reducing duplicated code which rules out having one big API included in every work enironment. We could probably create one big API using some statements checking the active platform and direct what part of the code in the API should be called. But we wanted to avoid having to do system checks runtime. And we also had an underlying reqiurement to have the environment ready in a out-of-the-box-fashion i.e. always available to extract from the rest of the workspace and be fully working. By having a large API file we would then have large chunks of code that wasn't used for all platforms. Instead we would like to try to achieve a requirement of having a slim one-purpose API file for each platform.

We agreed upon a requirement: we wanted a specialised API per platform but with the same conventions and usage. Our first solution to this was simply to have an "interface" in each platform API. A structure of methods which we could then fill in depending on platform and place this JavaScript file in the correct workspace. We did some experimenting and found out this works really well. When we had a working solution which gave us no apparent errors and was intuitive we prioritised not spending further time investingating other solutions.

When developing Quack we relised we got some double maintenance ourselves. By having empty function-shells which we filled in the different Quack files we successfully divided and unified the Firefox OS and Tizen hardware calls. But, we found a part of our API which gave us duplicated code in the separate API files; the custom classes and representation. In context to our example application this was our Contact-object and a filter used for searching after contacts. We had written the exact same code defining a Contact in both files, and we thought this was a poor example of solving the double maintenance problem.

After a brainstorm we came up with a solution. By dividing Quack in two parts we could reduce the duplicated code to a minimum, the split can be seen in figure 4.12. The first part was a shared JavaScript-file. It defined all the namespaces, linked the functions and held all the object representations. This file would be used by all platforms. Further down the line we could possibly use this file for importing the second part of Quack. The second part of Quack is the one we've been discussing. An interface-API containing all the hardware function calls. The only duplication now was the line containing the function name, but this was essential and totally acceptable.

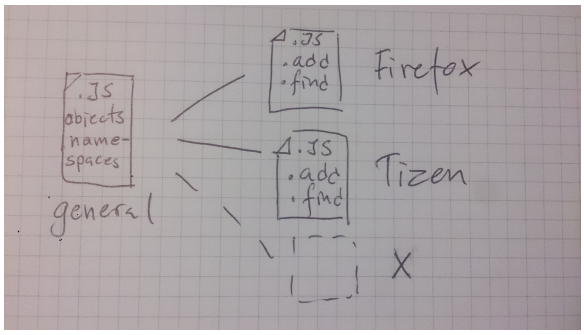

Figure 4.12: A figure showing the split of Quack, a general part and the platform parts

One issue we have with Quack built up this way is that we do introduce double maintenance in the management of Quack. It is dependant on the platforms and even when developing we noticed that Firefox OS changed their representation of an attribute from a string to a list of strings. This is something the management behind Quack needs to be observant with.

4.3.3.1 Look and Feel, the Right Conventions

Creating an API that works is not really hard. Creating an API that's intuituve and easy to use, now that's a challenge. There are a lot of APIs regarded as easy to use and an inspiration when making your custom JavaScript API; one of them is jQuery.

There are a lot of aims and designs to achieve when designing an API, like using command query separation[24], making it fluid[24] or Restful[15], and adjusting the naming conventions in a certain way. When wanting to develop an API for an end product these designs are all things to strive for but we are not making a full fledged API. The aim of this API was to make it work with the hardware and bridge between the two platforms when using Gizmo. We just needed it to work to evaluate the product as a whole.

With this proof-of-concept-thinking no exact principles were followed when designing. We simply looked at a lot of APIs, including the ones from Firefox and Tizen, and took what we thought was the best parts. This is probably not the way it would happen when creating an API as a commercial product, but we had the attitude that if we could understand it as developers then other could too. If during the user tests it would turn out not to be working, these parts would simply be rewritten with more rules and guidelines in mind.

4.3.3.2 Unifying the Representation

As stated before Firefox and Tizen both have their way of representing objects and methods. Some examples of these can be found when creating a Contact object. Tizen uses attributes such as 'firstname' and 'lastname' while Firefox uses 'givenName' and 'familyName'. The differences between these are actually substantial when you read the dynamics behind them[28]. Unlike western naming traditions, the lastname is not necessarily the family's name in eastern part of asia- it's usually the reverse so instead of Huckelberry Finn you would get Uchiha Sasuke, where Uchiha is the family's name.

We could see Firefox'S logic of using the family name and the given name as attributes and agree with it, but instead we chose to use first name and last name from Tizens API. Even though using family name when communicating would avoid misunderstandings, lastname just seemed more familiar to us. When we wanted to implement a new contact using firefox we had to look up the attribute givenName, even though we knew familyName. But when developing with Tizen we knew instinctively there was a value called firstname when we had used lastname.

When a contact has been created in Tizen and Firefox, the platforms uses different function names to store these in the hardware, or the addressbook which they both call it. Firefox uses the word 'save' while Tizen instead calls it 'add'. These two words chosen by the platforms does not mean the same thing and when choosing our own word to use we can't think of any better word. In our oppinion 'add' is more accurate to action performed; 'save' should be used when a file or state has been altered and needs to be saved, if something have been created and should be appended to an existing set 'add' is just more appropriate.

When creating Quack we needed to decide on what to call functions and attributes; the namespaces and object names is just the ones we thoght made the most sense at the time and have been the same since. There might have been some ideas behind the naming process we didn't research but we decided to take the best parts of the existing APIs would be good enough. When naming the functions we usually went with Tizens approach. It felt like Tizen was more appealing to the western culture and typical programatic thinking, and basically that was our target audience at the moment since our testers are programmers in a western culture.

Keeping Availability to the Native Platform

We discussed for quite some time if we would keep the possibility to access the native representation. With native in this sense we mean the Firefox' or Tizen's version of the object, and in this explicit example the contact object. We guessed that if some

third party script or plugin wanted to use native classes this would be favorable. Still we saw no use of including access to the native element as an inital feature, it would mean work beyond our proof-of-convept scope.

When we continued implementing further functions to Quack we were faced with the process of removing contacts. When removing a contact on both systems you needed to fetch a contact and send in that native element into the function trying to remove that contact. The reason behind this is that the contact object held, in both cases, a read only value- an id attribute. This id was generated when the contact was created and could not be reproduced in any way. First we tried implementing around this. But the workaround would be substansiveand since this is probably not the only example where the native element is needed it would lead to a huge amount of work down the line. As a direct result of this we chose to use our representation as a shell containing the native element with the option of fetching it would the need arise. As you can see in figure 4.13 there's the Quack object containing the native object and methods for extracting it.

As it turned out we guessed correctly in the beginning and we were satisfied when we decided to use the shell-approach. This reminds us a lot of jQuery which is regarded as a very well implemented API so we thought that we must be doing something right!

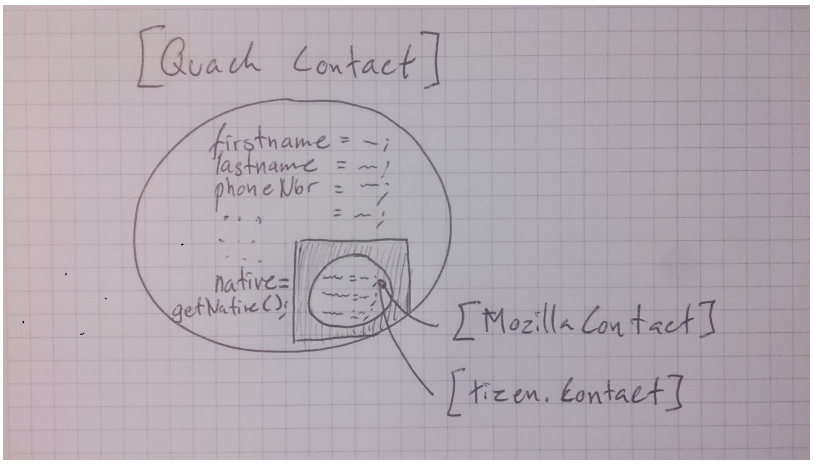

Figure 4.13: A figure of how the native object is kept within the Quack-object

Handling the Different Implementation of Events

As mentioned earlier Firefox and Tizen use different ways of handling the timing of interaction with the hardware and core features. Firefox use a kind of asyncronous

handling and Tizen lets you do everything straight up sequentially. This was a problem we needed to solve.

When trying to solve this different approach problem we first thought about which approach did we like the best. The Firefox way was much cleaner, but not very intuitive compared to what we were used to- it actually took a long time before we understood how it worked. On the other hand we understood directly how to use the Tizen approach and so we decided to copy it. Let's take the adding of a contact to the OS as an example. The way the Tizen function, and our Quack function, worked was by taking three arguments: The contact to be added, the on-success-function and the on-error-function. The contact to add would have to be in a Quack-Contact format; the functions passed were optional and used if you want to catch the result of the function. In a typical JavaScript-fashion the last or two last arguments could be dropped and the function would still work [12].

We then implemented the functionallity for the function in for the different platforms respectively and we got it to work. When having a notation we liked and chose from the beginning and a function that worked on both platforms we figured it would be hard finding a solution that would work better and kept our intial approach.

One note to this is that both systems use their own code execution management as they did before. Tizen still executes right away and Firefox uses its asyncronous handling. We realised that we could if we wanted to change our handling of the method so that Tizen also got asyncronous in the same way Firefox is by using some listeners or similar actions; and use some kind of request object similar to Firefox'. But it seemed unessesary to change something that worked when there were no measurable improvements, especially since we found the firefox approach less intuitive.

4.4 Gizmo-Quack, two solutions brought together

After we had gone separate ways trying to find a solution to the double maintenance problem, not only on the code level but also the knowledge based logical level, we had the mission of trying to merge the two products. The merge never made an impression as hard to perform since it's was actually just the insertion of the JavaScript files in a Gizmo controlled environment. But the effect of the merge was noticable isntantly.

When we placed Quack in a Gizmo workspace we inserted the whole test project of creating an application for managing contacts. At the time of the merge Quack was not in its final state. The act of separating Quack into two parts had not yet

been done and not all functions were implemented. We inserted Quack earlier to test Gizmo-Quack as a product early on internally and as a result we ended up using Gizmo-Quack when continuing developing Quack.

When researching, testing and further developing Quack it was incredibly convenient to use Gizmo. We were working with multiple files with similar content and later on needed to extract parts of Quack into the common file it was easy to manage using Gizmo since we didn't have to copy lines of codes back and forth instead Gizmo would handle it for us.

As we have discussed earlier Gizmo solves parts of the double mananagement on the code level, and Quack solves double management on the logical level. When merging the two we should now be able to solve both at the same time, and that's exactly what we do. Not only that but the effiency when developing gets higher, since we can cut down the resources needed for developing for multiple platforms, which solves another of the problems proposed in the analysis- the resource cost for the company. As we have mentioned many times before Qizmo-Quack is not a finished product. If resources were spent on developing this product maybe we would achieve even higher effiency and reduce the double manintenance problems even further. This is something that could be targeted as a feature worth developing for someone in the future.

While Gizmo-Quack has been merged and is in our opinion an amazing tool it's nocessary for us to use the two products together. Gizmo is a very special new way of thinking within configuration management and Quack is a convienient layer to place between developers and the platforms they're developing for. Even though they work so well together they can still perform well as stand alone products. Gizmo can be used with probably all languages with some configuration and the variation handling does not have to be used to acknowledge the need for multiple platforms, maybe we just want simliar but different distributions of a product. Quack have similarities with other products on the market and can, if developed properly, surely be equivalent or better than what's available, the competion will be discussed more closely in the next chapter. While we don't need Quack more than any other cross-plattform-API we can see that the use of such an API exists and therefore Quack is a useful product.

During the merge of Gizmo and Quack we also discovered a feature which struck us as something we must implement further. When wanting to make the interaction and workflow in Gizmo-Quack more intuitive and effective we agreed that we wanted some form of auto-completion feature. When arguing the content we would have in

this feature we came up with a solution that made quite an impression. We invented our own standard of using snippets.

4.4.1 Snippets in Gizmo-Quack

When Quack was ready to merge with Gizmo we realised another oppurtunity with the python script listening for changes and certain keywords in the files. We realised we lacked the auto-complete feature often found when developing and we saw this as a minor problem. So in addition to listening to the "!sp" we came up with the idea to create our own auto-complete action which is more extensive then the ones we are used to. These "improved snippets" might be a step in the right direction even as a stand alone feature.

The auto-complete feature we developed is based on the save-and-modify-files feature in Gizmo and we realised it would be hard to compete with the fast paced auto-complete found in certain editors such as Eclipse or Sublime. In these editors one can use tab, enter or other button combinations to quickly fill in a proposed word, variable or function. Since we didn't have that convienient quick way of helping the developer use Quack we realised we could compensate in some other way. What if we could introduce a large amount of useful information to the developer right there in the code?

The more we thought about it, improved snippets seemed like a good idea. Just importing the function in a minimal fashion has its advantages but importing more information than that has not been used before to our recollection. When developing one often search the web for the documentation and sometimes for tutorials or tutorials showing a working example of code one can mimic. What if we could give the developer the option to get that help right away, already working in the code? This was absolutely something we would like research further!

Since we always search through the entire file searching for an exact match (instead of using algorithms for finding a probable word) we have to the possible matches in a detailed fashion. We wanted to have an exact string which would be recognized and replaced with a snippet when writing functions. An example of this is the function add, which would be found in Quack in the contacts namespace. So when writing the phrase "quack.contacts.add" and saved the file we wanted some snippet take its place.

When we wanted to specify the content of the auto-complete we found it hard to decide upon how detailed the snippet that would be pasted in place would be. We had some different ideas on the content of these snippets and in the end we discovered

three possible apperances. We could either follow a more typical autocomplete and paste in a minimal row with proposed variable names filled in (typical in Eclipse)

```
quack.contacts.add(quackContact, successCallback, errorCallback)
```

or we could give some help with the implementation in a fashion a developer would usually write these functions.

```
quack.contacts.add(
    new quack.Contact({
        //TODO: Enter input for [quack.Contact]
    }),
    function(){
        //TODO: Successfull callback function
    },
    function(err){
        //TODO: Error callback function
    })
```

The third option is the special one mentioned above. It first struck us as strange and very uncommon but the more we looked at it and used it we realised it's potential. Simply put, it would contain a lot of information. It would give you documentation, the basics given in snippet number two and also an example implementation.

```
quack.contacts.add(
    // param1 [quack.Contact] The contact to be added to the platform
    // param2 (optional) [function] A function that will be called when the conta
    // param3 (optional) [function] A function that will be called if the contact
    new quack.Contact({
        firstname:"Some Firstname",
        lastname:"Some Lastname"
    }),
    function(){
        console.log("[quack.contacts.add] Contact was added successfully");
    },
    function(err){
        console.log("[quack.contacts.add] An error has occured: " + err);
    })
```

One can argue the use and usage of the third option. How often will it be used? And does it strike as a natural option to the developer? When discussing the three levels of information in the snippets, this one in particular, we discovered that we could divide them in three stages of developing: When you're starting out, when you're getting the hang of it and when you are an experienced user of Quack. In the beginning it's reasurring to have the code examples and guidance, when it becomes a little clearer it's ok with just some guidance and when you know what's going on all you need is the minimal snippet to save time. We found these options intruiging and were very excited about the result from the user testing.

In the end we decided that all these snippets are of use to the developer so we included them all in Gizmo-Quack and the user testings. We did this by using the second snippets as the standard, getting fetched when using the standard string. The first option as 'minimal' and fetched using "quack.contacts.add-m", and the last one as a 'full implementation' and fetched using "quack.contacts.add-f". If there would have been more graphical options available to us, as we would get using our own editor, we could probably make the selection of what snippet to use much more user friendly, and powerful.

4.5 User Testing

User testings can usually pinpoint problems and solutions in an final product, but since the nature of our product was a proof of concept we knew early on that this would not be the case. The tester, or developer, would have access to his or her entire knowledge of the languages, or lack thereof, and their preferences in working with them. This makes the testing environment very large and hard to observe small and distinct results. When conducting the testing adn extracting the results from them we had to keep in mind that an immediate transgression when using our procuct does not mean it's badly designed, since the developer might have commited the error due to old experiences unfamiliar to us. Instead what was sign of an error in the product was if the developer had to struggle with using some parts of Gizmo-Quack and if errors was repeated.

Based on this very open test environment we wanted to observe how the developer thought he would use the product, and how easily the developer could adapt (if they at all needed it) to how we think the product should be used. In addition to this we also wanted to get opinions and more thought through ideas on how the product can be improved.

With these criterias we decided we'd set up "dirty" user tests[29] with just three test subjects and then afterwards have an open discussion[?] where we could ask questions freely. Even with this approach we were still fairly confident that the behaviour of the small group of developers would probably represent the overall developer and that with a discussion we could get opinions from both user and fellow developer alike.

The results below are observations and statements of the users when trying to implement a simple contact manager application for Firefox and Tizen using Gizmo-Quack. After presenting the actual behaviour and comments from the user we will try to evaluate the product with the test result in mind in some different categories. We will try to evalute Gizmo-Quack as a tool and how it's experienced from the developers. We will then see what the developers thought about having to use the structure we've chosen, the improved snippets, and using a underlying script when using a familiar editor.

4.5.1 Using Gizmo-Quack as a Tool

When placed infront of the computer and Sublime (which we used as our editor during the tests) all three developers were confused the first minute. But we noticed quickly that this was because they had never done these kind of free tests and didn't really know where to start. After checking out our cheat sheet (a register over functions and actions for using Gizmo-Quack) and started working with Gizmo-Quack the developers grew more confident.

You could tell this wasn't something they've done before, and when faced with the unique-sections they had to think twice before figuring out what to do. But as they filled out the fields and saw it copying code (and not copying from the unique section) they all made a remark in the line of: "well, that's nifty". The second time they were supposed to use a unique-section they created it promptly and used it correctly.

When seeing this we knew right away we had done something right. If the developers could learn a foreign workflow like this as fast as they did, it must in some way make sense to them and be intuitive, one of the goals we were searching for!

The test subjects bumped into some errors, such as having trouble inserting a "!sp-section" (will be covered below) but for the most part, the thought process behind their work was spot on with our core concept. What surprised us what the faith the developers put in our product as early as they did. We had set up two windows side by side on a computer. Both holding a Sublime instance with three tabs in them: one for the HTML, one for JavaScript and the last one for CSS. As soon as one of the

developers finished with one of the tasks in the set, and saw that the information was copied between the editors, she minimised the Tizen/right window; she said: "Well, I know I will develop for that platoform too. Might as well get it out of the way.". This was the best confirmation we could ask for! For someone to put so much faith in the product and showing us how the actual end users will probably use the product.

When discussing the product after the tests we spoke about when will Gizmo-Quack really be needed, if ever. With this we mean at what size will the tool be: unbearable, redundant, convinient etc. - when developing for one platform, two- or many? What we all agreed on was that with many target platforms Gizmo-Quack is a must-have. If there is no other option available, the workflow of Gizmo-Quack is just too good to discard, even when working with just two platforms! When it comes to one platform it might not be necessary, you only have to learn one language and have one codebase, but then we all thought why just not learn Gizmo-Quack. Then if you want to add a platform it can be done in seconds instead of weeks. It's still just one language and codebase!

4.5.2 Synchronizing Files and Structures With Gizmo

The most unfamiliar feature when using Gizmo-Quack was to use 'save', especially when trying to import snippets. It's just striaght up not natural for the developers to expect actions to come from saving the file. After writing a matching string for our auto-complete the developers often tried to get it to work by using TAB or CTRL+ENTER etc. Basically all the other keyboard shortcuts they have been introduced too. In Sublime (which was the testing environment) the snippets and auto-complete is activated with TAB so this was almost frustrating for the user- not getting the desired result in a editor they knew it should work with.

When developing Gizmo-Quack we learned to use save to sync quickly and we know exactly what's going on when you save. The developers during the test did not. They hadn't seen the underlying code, they just wanted to work like they normally do. Even if they got the save to work after a while it was often accompanied by a comment such as: "oh yeah, you had to use save...". At the end of the session they used it without any problems but it took too long for them to get it right, the saving feature is obviously not the correct way to go since it violates the developers natural heuristics!

There also emerged a problem during the user test we hadn't seen before. When we have used Gizmo-Quack we use save all the time. We know that no harm can come from it and we know that we're jumping between files that need to be synchronized.

The developers during the tests sometimes forgot to save when moving between the "same" file on different platforms. This would lead to the developer being prompted: "The files have changed, do you want to update?". If they chose to update the changes they had written in that file and not saved afterwards would be discarded. If they aborted the update the synchronization would not take place, which could lead to some problematics copies and workarounds which were undesireable to say the least. This is another example of where having a more regular interaction with the code written and the developer, instead of analysing and modifying files, could help us merge the changes in a simple fashion.

How would you then solve such a problem as using the save feature? Well, the saving feature is used since we are managing the code on a file structure level. If we could instead make changes in the code itself we could probably change it to some other more familiar key shortcut. But what do we need to able to do that? Well, we would probably need our own editor. This was brought up both as a result from the tests and during the discussion afterwards. With our own editor we could link the files and Quack much stronger into the Gizmo workflow. The editor would not need anymore features than interacting with syncronization and snippets interaction so it would not have to be advanced, maybe a modified distribution from an existing editor such as Sublime (even if this too would create further double maintenance when having to keep that distribution up to date!). If there were to be additional features in a stand alone editor they are just not apparent at this time. If we wanted a quick-fix for the Sublime, and perhaps other editors too, we could create custom snippets to the editor and then you could at least use tab for the import of snippets. The syncronization and import of unique-section will still have to be inserted by using save.

We have mentioned the "!sp-section" or the unique-section before; this is the variant section whose content wont be copied between platforms. When the developers used save correctly and inserted a !sp-section in both respective files they quickly realised its usage and what to do. In the earlier parts of the test where the user was told: "Put 'something' in the unique-section, then save and then..." they realised the use of the unique-section. But when they where asked to "do 'something' specific to each platform" they didn't really know what to do. But when we explained that "!sp" was our way to represent specific, they instantly knew what to do, insert a !sp-section and fill it. What we first thought was a flaw in the product turned out to be a flaw in the instructions and a miscommunication of what we call things. Once

they understood what we meant with specific they grasped the concept and how to work with it much better than we had anticipated.

We had one issue with the !sp-section we hadn't really thought about, its name. The reason behind "!sp" is that we wanted a unique string for the script to recognize and "sp" represents specific. Once again this was a problem for the developers but not for us since we knew the context and background. We had never had any problems with writing "!sp" when trying out Gizmo-Quack but it seemed like the developers tried all permutaions available: "sp", "!s", "sp!", "!" etc.. Combined with trying to import the section by hitting TAB this also lead to them chaning the string since "Well, the TAB can't be wrong- I'm using Sublime, and Sublime uses TAB". We have no suggestions to what the !sp should be replaced with, if at all, since whatever string we choose it has to be learned and used a couple of times before used naturally.

The use of the !sp-section itself was easy to understand to the developers. The section had a start line and an end line and this is commonly used in some programming languages, especially HTML where you use a start- and end tag in most instances. Since these developers knew web, and the target users too, it was easy for them to see that the !sp was a tag and between the star- and end tag is there the magic happens. It was good to see that the !sp-sections purpose was both intuitive and effective since the import of it was not as easy.

The one main confusion about the !sp-sections was when there was something specific to one platform where the other didn't do anyhting. When a !sp-section is created it's copied between the platforms on the same line in the code. If you want you can fill them all with code but you don't have to. This was sometimes confusing to the developers, especially when there was an empty section. An example of this is when importing JavaScript files in the HTML, Tizen needs to import its API while Firefox has its bundled in the OS. This means that the !sp-section on Tizen holds an import line while the Firfox file has an empty section: "Why have something there when it's not used".

The developers were right to some extent, why do use something that's not used. Well, at this stage in development the section is used. Even if it's not used by the developer the script needs it to keep track of the sections. Without it the script would get confused and corrupt the code when trying to fix the "imports" when there are none. Maybe further down the line the sectons management can be handled elsewhere. By giving the developer some information on the benefits of the empty sections maybe the confusion could be avoided, for instance: an empty section shows you where something specific happens on another platform etc..

We discussed the availability of the !sp-sections after the tests and we came up with a concept with the developers that might be worth researching. When you want to remove a !sp-section you can write "-remove" in the start-tag and save. The script will automatically remove the section across platforms (along with the content in them). Why not use this functionallity further? The developers proposed more functionallity such as 'remove from this file only', and 'send this content to some other platform'. This sounds like a powerful next step and definitely worth looking into, the question is how good is the idea outside of the drawing board? Will it be easy to use and will it be used?

4.5.3 The Usability and Structure of the Quack-API

One thing we would have wanted from the user tests is to have some feedback about Quack. We got some results, but mostly from the lack of feedback with the reasoning- if they used it without hesitation and went about implementing the methods without errors then we must have done something right. This could also be becasue if you get presented with a sheet of functions and how to use something you don't really question, you just assume this is how it's suppose to be and you use it. But we did find some things that can be improved and we established the praxis the developers liked to work with.

Since Quack is not complete it's hard to say how all functions will look, but they will probably have a common look and usage. Our choice of having the success- and error functions as arguments seemed to be the way to go. There was no hesitation when reading the documentation or implementing the functions and when later asked during the discussion the developers said that this was the way theu usually develop JavaScript. There was one instance where one developer tried to use the find function (a function for finding contacts) as a returning method instead of sending functions as parameters but she quickly realised that she didn't know what to do with the returned object so she looked in the documentation and changed the format of her find function to the appropriete implementation. Clearly our way of using functions was easy to understand and use.

When creating objects we took a chance and made the constructor recieve a custom JavaScript object that the developers could fill with how much or little information they wanted too. An example of this can be found in 4.4.1 in the final example snippet; the "full" version creates a Quack-Contact using a "" notation, this is the JavaScript way to create a generic object. We then translate the values in the object to the right attributes. This way of working with JavaScript was new to one of the

developers but she admitted she had not been working with JavaScript for very long. The other two developers used without any hesitation and during the discussion we agreed that it's the simplest and most efficient way to create object, it's apparantly used in a lot of situation according to the other two developers. With this we knew that we could keep this implementation.

Our documentation at the moment is the same as our snippets. The snippets holding the largest amount of information is equivalent to our documentation. During the user tests we noticed that the developers, even the experienced ones, tried to enter information in places you can't really write code in JavaScript. We have some comments describing all the parameters but apparantly these are placed in such a way that it disturbs the developers and makes them commit strange errors. We discussed this together with the developers and came up with a solution. Instead of having them where we have them now, between the function call and the first parameter, some other placement would help with this problem. We could move the comments outside the function. This way the developers would see that it's not a place for inserting code but it's some lines of comments outside the scope and probably easier see them as overall guidelines. There's also the option of dividing the comments and placing them above their respective parameter. This way the developers would get a clean instruction to what this exact element they were working with was. The later is probably the way we would choose to implement were we to further develop Gizmo-Quack.

Chapter 5

Discussion

During our thesis we have defined a problem statement, researched the possibilities of reducing double maintenance at both a logical- and code-level. We created a product that can live up to this aspiration. We have created a real-time file based script and complemented it with an API that can replace native calls to the hardware in the platform. But what about other related work within the area? Are there other solutions that work in different ways? If these exist, what are their pros and cons compared to our work? What do we think about our product and what do we think about our thesis.

We will first discuss any work we have found that is related to ours, and how these solutions compare to our product. We will then make a reflection on Gizmo and Quack respectively, followed by a reflection on the product as a single unit, pros and cons, and what the future holds. We will then make a reflection on the thesis itself; the result, how we have worked and the problems we have run into, and what we could have done better.

5.1 Related Work

We have not found any work that has closely resembled ours that has been documented. We believe that a big reason for this is that the problem domain is new. The possibility to run native web applications for mobile systems has not existed for very long. Another reason for this is that the platforms that have been released have not gained a very big market share yet. The work about variants has related work that has helped define the space but so far it seems like what we have done is a rather unique way to handle files. But there are some similarities which we will discuss. When it comes to the hardware interaction this is nowhere near a unique research

attempt. The idea behind it is simple even if the execution might be unique from other products.

In this section we will discuss how variants can be handled and if there are similar projects from a configuration management point of view. After that we will discuss a renowned product which links multiple hardware-calling APIs together, Cordova.

5.1.1 Related Configuration Management work

Double Maintenance is a problem that exists in many places and in different forms. The problem was identified by Wayne Babich [8] and has been a big guideline for the work that has been performed in the thesis. Even though the paper was released in 1986 the problem still exists and is still a current problem faced in the industry.

The other big part of the configuration management of the thesis is the discussions of variants. There are several different ways of structuring variants to help reduce the double maintenance problem. This leads to more papers on variants. One big help is the paper written by Mahler [21] that has given guidelines on both how to keep things together, and how to tell them apart; which is more or less the name of his paper. What he means is how do we identify and know what parts of the code that should be General and what should be High or Low? That is to tell them apart. In our solution we also have to synchronise and make sure that they are not mistaken for one another, that is keeping them together.

When it comes to the implementation of variants there is a lot of work. The work of Critina Gacek and Michalis Anastasopoules [22] has helped in the understanding of how some companies work with variants in the real world. We also found other articles on how to handle variants. From these articles we learned mostly how we did not want our solution to work and some of the problems that their solutions suffer from; they involved either forking or cloning [19] [20] of the repositories. We do not believe that it would be a suitable approach and we guessed it would not reduce the double maintenance to a satisfying level.

Other ways of supporting variants are discussed in the articles [9] and [22]. They discuss Aspect Oriented Programming (AOP) as an alternative to support variants. We believe it to be interesting because it could reduce the double maintenance in more ways. Say that the application contains a map. The code for the map can be shared between the platforms. The only difference between them is that they should have different markers on the map. With AOP a call to the map can be done in the general code. But before that call is executed, each platform can call their respective method before to set the type of the marker. We believe that the General code would

become more clean and readable. We have not spent a lot of work on it, but it does look like a very promising alternative for additional help for the developers. The biggest problem from the start is that it will not work for HTML and CSS. And we had to build a solution that included all three of the languages. One difference though is that in [9] they refer to the C and C++ coding languages.

As mentioned, we did not spend much time on this. But with Gizmo we think it would be interesting to investigate closer if AOP could work for HTML and CSS as well. JavaScript already supports AOP to be performed as a part of the language. Not to go into too many details about JavaScript, but you can call a method that exists in another file that is imported to the project. So if method saveToDataBase is called in General code. If the method does not exist locally in the file, JavaScript will look for the method in the global scope. So it would be the exact same method? Not necessarily. With Gizmo each platform can import different specific files. So if each platform has imported different files, and each contains the method saveToDataBase, the application will execute properly and call the respective platform's specific method.

Part of the early work we did was on how to structure the workspace. When discussing Workspace 34.2.3 we saw that a possibility could for example be structured though the usage of XML [26]. This solution would work with one or several xml files. Each of these files contains a structure where it will hold something similar to a key-value pair. The key will work as the identifier. So when working in the file that will contain different variants, the developer would place the name of the key there. When building the variant the key would be removed and instead the key's value in the XML file would get inserted. In the end we scrapped it because we did not think that it was a very user friendly way of working.

5.1.2 Cordova API

When it comes to Quack we are dead sure about an existing competing API, Cordova [2]. Cordova is an API designed to convert JavaScript function calls to native on multiple platforms, not only the web based we have discussed (Firefox OS, Tizen and Ubuntu Touch) but also Android, iOS and much more. It's well known within the application development industry. Cordova is mostly known for being the API used in PhoneGap [4], a renowned product for creating web-hybrid apps for Android, iOS and Windows Phone. Even if PhoneGap is sometimes discarded as viable, according

to developers at ÅF, when it comes to having hybrid web-apps with hardware feature support (because of performance issues), it should still be highly viable when developing for the native web platforms.

When it comes to supporting multiple platforms Cordova falls short in regard to some. Cordova supports Ubuntu 100% (because Ubuntu uses Cordova as native API) but when it comes to Tizen and Firefox there are some features (not the same) on the platforms that does not exist [5]. This will make Cordova less attractive when wanting to develop to all web based platforms.

5.2 Reflections about Gizmo

In general we think Gizmo has been a success. It is meant to be a proof-of-concept that de,pmstrates it is possible to combine the code bases for native web applications; which is something it can handle. It also gives ideas on what the problems are and a point for someone to continue to work with the problematic of variants. But there are disadvantages of the solution that we decided upon. When making the design decisions we were aware of most of the problems and restrictions that it would bring. But some problems have come to our attention at a later time; either that we found them ourselves or through the user testing that was performed.

Due to the time limit we had, the only option for a prototype we saw was to run a script that listened to the changes in the computer's file system. Because of this the developers have to have direct access to the files so that they can manipulate them themselves. This is probably the biggest problem. Because of this there has to be a logical structure on the files so it will be usable for the developers. But at the same time the synchronising has to work properly. To support both at the same time does make it more difficult to build. There are many unpredictable errors that can happen when the developers modify the files as they can do it outside of the tool's control.

That the synchronisation of the variants has to work is a natural part. So how do you remove the developer's need for direct access to the files? We say through some kind of GUI, so integration with some IDE. By having that we could give the illusion of a proper structure of the files but store and handle them just as we like in the background. This is not an easy task though; building a GUI that is user friendly and has the functionality that the developers require will be a big commitment.

A GUI would also allow other ways of handling markup. The markup could then be shown in for example different colours instead of having comments surrounding them. We think it would allow for a more robust experience. The comment markup

can by accident be removed and then break the synchronisation of that field. We believe a GUI would make everything more robust.

In the end we are satisfied with Gizmo as a standalone software, it does have its flaws, and some of them can probably be fixed with more time, even though we would rather see a GUI to complement it. But it is definitely something that can be built upon to improve it and in the end help developers further.

5.3 Reflections about Quack

Considering Quack was designed only as a proof-of-concept we were surprised by how well recieved it was during user tests. But as we said it's only in a proof-of-concept-state and it's questionable if it will ever leave that state. But what can be done better, what was good enough, and is it worth completing?

In this section we will discuss a possible optimization of Quack, if these are worth doing and if the API will be used by developers at all!

5.3.1 Quack Optimization

As we have said a million times over by now Quack was developed to support a proof-of-concept idea with the put together oroduct Gizmo-Quack. The effects of this are described in the design of this thesis but the short of it is that the design decisions made when constructing Quack and the choises behind its usage were only made to fit with the Gizmo and workflow model. While we have noticed that the design of Quack, and Gizmo-Quack, was rather successfull we do not base the degree of completeness of the API of this reception in any way. Quack is not finished, it's barely even started.

As a non completeted product, the word optimize seem like poor choice. But the finalization of Quack can be done by varius methods and undertakings Firstly, Quack only supports one hardware feature; Contacts. If Quack were to compete with say Cordova this is something that needs implementing. This is the main reason why Quack can be regarded far from complete.

The structure, work procedures and language usage is chosen by us based on our preferences. These can be switched out if deemed necesarry but it's not something that needs to be done. Our way of implementing Quack could be the most optimal way of implementing an API, or it could be quite the opposite. The only thing we have found out during our thesis is that it works and it's well received by the target audience.

So as far as optimizing Quack completing its implementation is a great first step and where to go from there is probably outside the scope of this thesis.

5.3.2 Is Quack Worth the Effort

Is Quack wort the effort? That is a hard question to answer, but probably the most important question from a commercial point of view. The answer is probably the most bland there is, both yes and no.

Is Quack something that could be used to develop applications effieciently? Yes, most definetely according to our user studies. Would it be worth to put in the manhours necessary for completing it and how much would you get in return? Now that is hitting the nail on the head.

We are unsure about how long it would take to complete the API. But it would probably go a lot faster then it took constructing it. We knew nothing about constructing APIs, the platforms or even JavaScript theory. When, and if, Quack would be completeted we would have constructed a template for how to implement methods and structures. It would be a rather simply assigment for someone with a little know-how in JavaScript to complete Quack following our standards. When Quack is completeted it would need something we want to minimize, maintenance When the plattforms update their hardware or API Quack need to get updated too. But having access to the API code doesnt only mean you have to update it, it also mean you could customize it!

Even though Quack works there are other products that could fill in the gap, Cordova for instance. But the strength of Quack is not merely that it is a API talking to the hardware, that's just the content as of now. An API like Quack reduces the amount of work for a developer. They now only have to learn one API instead of one for each platform. Because of this it will move code from Low to General. So it will be the same for every platform. Every feature that are used repetedly in development within a company could be placed in Quack. Quack is your customizable, personal API. You could place Cordova inside Quack and complement it with some features just like we've done now with Firfox and Tizen APIs. Companies already have their APIs and shared development features between projects. Quack isn't an extra API, it could possibly unite spreads of APIs and simultanously work as a cross platform API. Quack is what you make it.

5.4 Reflections about Gizmo-Quack and it's features

The combination of our work gave a better overall experience, some interesting results and made us feel that it would give developers more support in building applications in an easier and faster way. But the work is by far from done if a complete product is to be made and some of our thoughts and ideas are discussed here.

The section starts with reflections of the pros and cons of combining Gizmo and Quack. The chapter is followed by a discussion about the snippets that we added and end the chapter with future ideas.

5.4.1 1 + 1 = 3?

We were satisfied with the results that our work had shown us in their respective area. But the real strength came when we combined Gizmo and Quack. The script managed the synchronisation of all the General code, so developers did not have to worry about that. With a more developed API it would allow the developers to avoid any Lower part of the code at all. One thing we see as a potential problem is that we feel that the amount of contact from the supervisors at both ÅF and the design centre. It is something that was lacking from both sides; we could have been better at contacting them and vice versa. If that would have affected the direction of the thesis is hard to speculate in. What we know is that it would have been nice with more contact. This combined would both reduce the double maintenance on code level and the developers would only need to learn on API for all platforms. Compared with building the same application separately for each platform with the tool we created, very little effort had to be put into it.

Even though we see a lot of potential in the prototype, it is still a prototype and not a full-fledged product. With more time put into Gizmo-Quack, it could become very useful for developers to work like this, or in a similar manner.

But the product is by far complete. There are probably many things that can be improved, altered and reworked to increase productivity, intuitively and the workflow of Gizmo-Quack.

5.4.2 Future Ideas

By not only having Gizmo as an external script but by incorporating the functionality in a full-fledged editor we could change our product for the better in so many ways. Not only could Gizmo have more control over the files and be able to work more

efficiently but we would also be able to remove the problem we had of using the save feature for executing commands.

Another part that could be worked on is the variant handling. One method was touched before, namely Aspect-Oriented Programming. It is an interesting topic that can be integrated into Gizmo-Quack and would give the developers another way of working with variants, but still keeping the General code clean and only contains that. Compared to our CC-tags it would keep the code much cleaner. There is also the implementation of Workspace 24.2.3 that we did not have time to implement. Both we and the developers showed interest in this and it would be interesting to see how having that additional support for variants would come into play.

There is also a topic that we have neither worked nor discussed a lot about. We built a prototype to see if it is possible to work like this. But the work is focused from a single developer's view, what happens when you are several people working on an application? The current solution to handle the IDs on the tags was a quick solution for us to solve our problem for one developer. But what happens when developers work separate from each other and then want to synchronise their code with each other? How to handle this will require a substantial amount of understanding and work about configurations.

Snippets, or better yet our snippets, were something of a bi-product when merging Gizmo and Quack. We quickly realised the possibilities, tried them out and we liked what we saw. When we asked the developers during the tests how the snippets worked for them, they agreed with us and said that it would be something that they would be using themselves. It is not ground-breaking but it is a new way of thinking that reduces the need for searching online when in need of a small example. We really think that these kind of snippets should be used in established products out there today like Eclipse and other IDEs. It could make a big impact on the developer community and it would be exciting to see where this could lead us.

5.5 Reflections of the Thesis Work

We think that we have achieved our goals, which was to see if we could reduce the double maintenance for mobile web applications. During the process we have learned a lot, for example how to better structure code, better teamwork and deeper knowledge of the coding languages we used. In short we have become better software engineers. The work has been mostly painless and the cooperation has been completely so. There has been some trouble with technicalities, such as IDEs and the platforms and some

with our project management. But in the end we feel that the result of the thesis was not affected very much.

In this section, we want to talk about the general thoughts we have about our work. We will discuss general problems, such as time management and the platforms maturity. We also were unsure about the design aspect of the thesis for some time and we will explain this a bit. Lastly we will discuss what we could have done differently during our thesis work.

5.5.1 General Problems

One thing we see as a potential problem was the amount of contact from the supervisors at both ÅF and the Design centre. It is something that was lacking from both sides; we could have been better at contacting them and vice versa. If that would have affected the direction of the thesis is hard to speculate on.

Another thing to look out for is to dig too deep into pure development. This is a thesis work and it has two parts, experimentation and an academic part. Even though we had a workflow to help us in not doing that it was still easy to do it. This was a big thank you to Lars that continued to remind us of this. A big help was a paper we received from Lars that contained directives on how to work when doing a master thesis.

Time Management

Time was probably the biggest problem for us in the thesis. The thesis had a decided time limit and it is something that we failed to follow properly. A simple reason is that we did not plan well enough at the start and set up deadlines to when to stop, for example the coding and experiments and start with the report. Another reason for this is because we were away in the middle of the work, this pushed the deadline back and we did not consider the consequences of this properly. One consequence of this was that we missed the deadline for handing in the report and be allowed to present our work before the summer. When we realised this some of our motivation disappeared as we had to wait approximately three months before we could present it at the school. The idea of having these set deadlines are in general good, but for any thesis that is delayed or started too late it will become a problem. For us it was a combination of not planning ahead enough and that we were away during the work.

In the end this made us work on the thesis longer than we were supposed to do. Now that might not be a problem itself if you plan ahead and are aware of the decision and consequences for it. But yet again we did not do that. If possible we would recommend scheduling the thesis after a specific presentation and being aware of that deadline. As the thesis work is supposed to be 20 weeks long; it is desirable to start at least that many weeks before the presentation. Even better would be to start one or two weeks before that. This will give some room for changes during the work.

Maturity of Developer Platforms

Something that took up quite a bit of our time was the maturity of the platforms. When trying to do the necessary steps to start developing and testing on the platforms we stumbled into a lot of problems. Firefox OS is the only system that has been released in a stable version and was definitely the platform that was easiest to work with. But it is noticeable that they are still in an early phase and that things shift. This was noticeable in their documentation, as the documentation stated different things on different places.

We also managed with Tizen, but the process was not entirely easy. The developers of Tizen have decided on integrating their tool within the open source IDE Eclipse. This gives them a good start and a tool that is widely used. But they are obviously not done in with their integration into Eclipse's platform. When starting the tool it is common to get some kind of exception or warning message that something went wrong when initialising the tool. When we searched for the errors we managed to fix most of them so it worked as intended. But there were some errors that we just had to live with and worked our way around them as much as possible.

Ubuntu Touch's development tools were in an even worse shape. We tried for a while to make them work and they partially worked. But it was not in a good enough state for us to try it out properly so we decided that it was not worth the time with the current state of the development tools.

Unsure Design Focus

We had guessed at the start that there would be some early problems with the platforms, but not that they would be so big; thinking that the platforms are about to go out to the market soon the tools are critical for third-party developers. There

is nothing wrong with trying this out, but when working with new software it should be taken into consideration that the setup might not be very smooth.

When starting the prestudy and sometime during the actual work we had another disposition towards the design element of the thesis. The aspect we thought about developing was not bad. But it was hard to grasp the context in relation to Gizmo. The design part we ended up with was an API, user studies and minimizing logical double maintenance- things that complemented Gizmo in a good way.

Before the logical maintenance idea we thought more about making a theoretic work by documenting the different graphic and user designs of the platforms; where do platform A place their taskbar, does platform B even have one, why does platform A have a colour code for different apps when plattform B does not? etc. We wanted to use this information to then later on (if there was time, which we decided there was not) create our own version of Mozillas Gaia Building Blocks [1]. Our building blocks would allow the user to create an application using either the platforms own design conventions or a shared graphic design that could be customized by the developers. This is not in any way a bad idea. In fact we think it's a great idea! But, it's definately outisde the scope of our thesis. The blocks would benefit greatly from having an already complete Gizmo-Quack when developing them and using them. Also, the workload of developing these blocks is at least a master thesis on it's own!

Perhaps it's something that can be researched by somebody else but as we've said earlier, we quickly found that the size and direction of the building blocks where outside our scope.

5.5.2 Possible Improvments

This thesis could have gone better with more and better planning. That is what we feel would be the main improvement we could have done. Both when it comes to decisions and management. Apart from having a plan for the whole thesis we should have continuously checked that we were following the plan and if not, taken appropriate actions.

The section on Time Management is proof of us needing to have better planning. With some better planning our thesis would probably have been done much sooner. We are rather certain the quality of the thesis has not been compromised by the extra time we spent performing the work, but for all parties involved it would have been better if we were done sooner.

We sometimes rushed decisions because we thought we had researched all other possibilities. We were very lucky during our work because our decisions were mostly

correct or worked very well. But it could also have gone the other way so some research before diving off into a new direction would still be a big improvement. The paragraph above, Unsure Design Focus, is a good example where we should have done some research before pursuing it as a goal. In the end it cost us two to three weeks of work. Luckily enough this could be used to some extent since it taught us things about the Firefox OS platform, but some time was wasted.

Chapter 6

Conclusion

Our problem statement mentions how difficult it is to avoid double maintenance and how easy it is to end up with large quantities duplicated code. We wanted to investigate if and how one could try to combine the code bases using the same language and also support the option of introducing variants between the products without having to deal with double maintenance.

At the start we discovered quickly that a solution is not hard to find; but to find the best solution is very hard. We realised that "the best" solution was so dependant and related to the developer that we based our product a lot of the feedback we got from developers around us and from user testing.

In our work we have shown the possibility of combining the code bases applications written in HTML5. This is achieved using a python script that reads, analyses and modifies the files used in development. The solution in itself does not have to be constrained to just building native web applications, it may be utilized in other areas too.

The script does not solve every problem and it might be a better solution to integrate it into an editor. As an editor the combining of code would no longer have to be done straight into the files but through a GUI. This would also open up the work flow on a whole new level and make the development even more efficient.

As mentioned in the introduction we also want to reduce the need for having to learn how to develop for different platforms. If you're making a web-app you should only have to know how to write it using prior knowledge in web development.

When trying to find a way to bypass the learning curve of using different hardware APIs we developed a new API acting as a layer above all available hardware features. This API has a shallow usage but acts as a proof-of-concept and as guidance on how to develop the API further.

We have successfully constructed a smooth solution for combining code bases for different platform's applications and have reduced the double maintenance both in the code and for the users since the interaction to the hardware has been unified. During the thesis we have created a prototype which during our user tests was found to be intuitive, fast and powerful. Users were surprised on how much they achieved with a small amount of time and no prior experience with the platforms.

Appendix A

Abbrevations

HTML Hyper Text Markup Language

HTML5 Fifth version of the HTML standard

CSS Cascade Style Sheets

OS Operating System

API Application Programming Interface

MVC Model-View-Controller

JSON JavaScript Object Notation

XML EXtensible Markup Language

GUI Graphical User Interface

CC Conditional Compilation

IDE Integrated development environment

Bibliography

[1] Building firefox os. *Homepage for Firefox OS Building Blocks*, 2013.

[2] Cordova. *Homepage for Cordova*, 08 2014.

[3] Mobile application development. *Wikipedia*, 08 2014.

[4] Phonegap. *Homepage for PhoneGap*, 08 2014.

[5] Supported features. *PhoneGap/Cordova homepage*, 2014.

[6] AppBuilder. What is a hybrid mobile app? *Telerik*, 08 2014.

[7] Apple. Homepage for ios. *Apple's own*, 08 2014.

[8] Wayne A. Babich. *Software Configuration Management: Coordination for Team Productivity*. Addison-wesley publishing company, 1986.

[9] Herman Tromp Bram Adams, Wolfgang De Meuter and Ahmed E. Hassan. *Can we Refactor Conditional Compilation into Aspects?* AOSD'09, March 2–6, 2009, Charlottesville, Virginia, USA, 2009.

[10] Raluca Budiu. Mobile: Native apps, web apps, and hybrid apps. *Nilsen Norman Group*, 08 2014.

[11] Canonical. Homepage for ubuntu touch. *Canonical's own*, 08 2014.

[12] David Flanagan. Javascript: The definitive guide. *inkling*, 2011.

[13] Mozilla Foundation. Homepage for firefox os. *Mozilla foundation's own*, 06 2014.

[14] Google. Homepage for android. *Android's own*, 08 2014.

[15] Thomas Hunter II. Principles of good restful api design. *CodePlanet*, 12 2013.

[16] Rob van der Meulen Janessa Rivera. Gartner says annual smartphone sales surpassed sales of feature phones for the first time in 2013. *Gartner*, 06 2014.

[17] Coru Janssen. Native mobile app. *technopedia*, 08 2014.

[18] Jolla. Homepage for sailfish. *Jolla's own*, 06 2014.

[19] Goetz Botterweck Marsha Chechik Julia Rubin, Andrei Kirshin. *Managed Forked Product Variants*. SPLC'12, September 02-07, 2012, Salvador, Brazil, 2012.

[20] Marsha Chechik Julia Rubin, Krzysztof Czarnecki. *Managing cloned variants: a framework and experience*. SPLC '13 Proceedings of the 17th International Software Product Line Conference, 2013.

[21] Axel Mahler. *Variants: Keeping Things Together and Telling Them Apart*. John Wiley Sons Ltd, 1994.

[22] Cristina Gacek Michalis Anastasopoulos. *Implementing Product Line Variabilities*. SSR'01, May 18-20, 2001, Toronto, Ontario, Canada, 2001.

[23] Microsoft. Homepage for windows phone. *Microsoft's own*, 08 2014.

[24] Rodney Rehm. Designing better javascript apis. *Smashing Magazine*, 10 2012.

[25] Samsung. Homepage for tizen. *Samsungs's own*, 08 2014.

[26] Hongyu Zhang Weishan Zhang Stan Jarzabek, Paul Bassett. *XVCL: XML-based variant configuration language*. ICSE '03 Proceedings of the 25th International Conference on Software Engineering Pages 810-811, 2003.

[27] W3Schools. Html tag reference. *W3School's own tutorials*, 08 2014.

[28] Wikipedia. Personal name. *Wikipedia Webpage*, 08 2014.

[29] Jenny Preece Yvonne Rogers, Helen Sharp. *Interaction Design: Beyond Human - Computer Interaction*. 02 2002.

[30] Jenny Preece Yvonne Rogers, Helen Sharp. *Interaction Design: Beyond Human - Computer Interaction*. 02 2002.

Varianthantering och gemensam kodbas för webbaserad apputveckling

POPULÄRVETENSKAPLIG SAMMANFATTNING
HENRIK GYLLENSVÄRD, NIKLAS WELANDER

Handledare: Thomas Hermansson (ÅF)
Examinator: Lars Bendix (LTH)

En inblick i applikationer

När man bygger ihop mobila applikationer idag så är det vanligt att man gör det till flera olika system och telefoner, idag är Android och iPhone de mest kända exemplen. Man vill ofta bygga samma applikation, den ska alltså både se ut och bete sig likadant. Problemet för utvecklarna är att plattformarna har sina helt egna språk och strukturer. Tänk dig att en applikation är som styrdonet på ett fordon och du tillverkar dessa. Problemet är att mobilplattformerna nu är så olika att de kan ses som helt olika fordon: Android - flygplan och iPhone - motorcykel. Uppenbarligen kan vi inte använda samma ”ratt” på alla fordon.

Problem med dagens utveckling

Tillverkningen av styrdon skiljer sig så mycket att tillverkningen tvingas delas upp i helt skilda grupper spritt över världen och dessa grupper har ingen anledning till att kontakta varandra, de vet hur ”applikationen” ska se ut. För ditt företg gäller då att göra investeringar i specialistkunskaper och grupper världen runt och hålla kolla på dessa. Och om en ändring ska göras på ”applikationen” så måste alla grupperna planera och införa den seperat från varandra.

Nytt fordon, flera modeller

Just nu introduceras nya mobilplattformar på mobilmarknaden som stödjer appar skrivna i web-format. I och med att det finns flera plattformar som stödjer detta så kan vi se dessa som olika bilsorter och applikationen blir en bilratt. Man vill dessutom ofta när man utvecklar appar komma åt hårdvaran på mobilen, vi kan se detta som knapparna på ratten (blinkers, torkare, stereo etc.). Även om nu alla har möjlighet att göra bilrattar på exakt samma sätt uppkommer ett snarlikt problem från tidigare. Olika bilmärken gör olika fästanordningar för rattarna så vi har fortfarande problem om vi vill göra en ratt till alla bilmärken.

Exakt samma ratt, med varianter

Det vi vill undersöka är möjligheterna att kombinera ihop byggandet av rattarna. Kan vi lyckas komma runt problemet med att ha olika lag som jobbar på samma sak? Skillnaderna i fästningsanordningen kommer kvarstå, men man kan kanske bygga en lösning för själva ratten och bygga ihop olika fästen på den? Man kommer fortfarande ha flera modeller för fästena, men man har ett gemensamt sätt att bygga övriga delar av ratten på. På så sätt kan man minimera det till att ha en grupp som utvecklar ratten för flera modeller samtidigt.

Kommunicera med hårdvaran

Att slutanvändaren, bilisten, ska ha en behaglig användarupplevelse har varit givet under en längre tid. I ratt-tillverkningen har vi velat underlätta interaktionen, men inte med bilisten. I varje bilmodell finns en fästningsanordning olik den nästa och det finns ett kluster med sladdar och kontakter som skiljer sig i varenda en. Som utvecklare behöver man nu, beroende på ratten man ska tillverka, lära sig vart varje sladd ska gå och hur varje kontakt ska kopplas in. På mobilsidan motsvarar detta ett bibliotek av anrop till hårdvaran. Även om biblioteken är skrivna på samma språk så är de så annorlunda att utvecklarna måste lägga tid på att lära sig alla (om man inte vill dela upp tillverkningen, och det ville vi ju inte). Så vad vi har gjort är att skapa ett bibliotek som länkar ihop de olika plattformarnas bibliotek. På bilen motsvarar detta en adapter som passar alla bilar men har samma utseende för ratt-tillverkaren.

Målsättning

Lösningen kan tyckas banal för exemplet med ratten. Tyvärr är de mobila plattformarna betydligt mer komplexa och det finns många faktorer att ta hänsyn till. Poängen är däremot densamma. Det vi alltså vill undersöka är om det går att kombinera ihop utvecklingen till de olika plattformarna. Detta skulle då lösa problemet med det dubbla underhållet. Dessvärre är problemet fortfarande att man behöver ha olika versioner för varje plattform, fästet skiljer sig ju fortfarande! Det man åstadkommit är att underlätta det dubbla underhållet. Däremot introducerar man olika varianter av ratten. De skillnader som finns behöver man hantera på ett bra sätt för att minimera mängden extra arbete. Så vi kommer även undersöka vilka problem som finns angående hanteringen av dessa varianter.

Slutsats

I slutändan lyckades vi att minimera det dubbla underhållet när man skapar applikationer. Resultatet blev ett verktyg som utvecklare kan använda sig av för att uppnå detta. Det är fortfarande bara en prototyp och det finns fortfarande vissa problem kvar, men vi har visat att konceptet fungerar och det går att jobba vidare med. Sammanfattningsvis kan man säga att man nu bara behöver en grupp som löser konstruktionen av styrdonet till flera olika typer av fordon, långt mycket bättre än det vi började med!