NTNU
Det skapende universitet

Et kryssplatform API for å kontrollere en ekstern enhet

Arnfinn Refshal Gjørvad

Master i informatikk
Innlevert: desember 2014
Hovedveileder: Maria Letizia Jaccheri, IDI
Medveileder: Rune Alstad, aalberg audio

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Preface

This is a Masters thesis in Software engineering at NTNU as part of the Informatics study. The thesis was done in 2014. The project was made in cooperation with Aalberg Audio. Aalberg Audio has helped with code written for the Aero controller, as well as with technical guidance for the project.

Trondheim, 2014-12-01

Arnfinn Gjørvad

Acknowledgment

I would like to thank the following persons for their great help during this thesis: My supervisors Maria Letizia Jaccheri and Konstantinos Chorianopoulos at IDI. I would also like to thank Rune Aalberg, Aleksander Torstensen and Torkild Indstøy at Aalberg Audio for their help and contributions. They have provided both access to hardware, and help with my code and firmware for the devices.

Abstract

More and more smart devices, such as smart glasses, smart watches or even smart smoke detectors, are being sold all the time. Most of those devices need some form of remote control or configuration. This is often done with a dedicated device or an app on a tablet or smartphone. Unfortunately this can lead to a problem where users are overwhelmed with having to learn new apps and devices all the time. There are options for making bigger control-apps that control multiple devices. However those apps are often limited by small, closed APIs that are hard to use. This thesis looks at a way of extending small hardware centric APIs into bigger APIs, making it easier both to develop GUIs for them and to integrate them with existing solutions.

The goal of this thesis is to find the best cross-platform framework for working with low level hardware, and figure out how suited it is to such a system. It will also look at which design decisions you can make to increase the amount of platform independent code this framework provides. Lastly it will look at how you can use the cognitive dimensions framework to analyze a finished API.

In the prestudy it was found that few cross-platform frameworks come with support for USB. Plugins can be made for some of them, providing the functionality required. However, at the time of writing no such plugins were found, that mature enough to use. Xamarin was therefore chosen as the best framework for such a system.

It is clear that Xamarin isn't a good fit for such a project. While it provides access to the methods required to create USB communication (which the other frameworks evaluated does not), it is just a simple wrapper around the native functions. This leads to a number of negative consequences. Couple this with lacking documentation, and the time saved is probably minimal.

However, the end product worked well, and the code saving was still around 50% (and as the complexity of the core logic of the application increase, so will the amount of cross-platform code). Focusing on design tactics increasing modifiability decreased the problems with using Xamarin. Using the cognitive dimensions framework to analyze the system proved helpful. It allowed the subjects of the user study a way of thinking about their experience in a more structured way. Comparing the results obtained when the subjects scored the system using the cognitive dimensions showed a clear correlation with the results obtained when analyzing the system based on the behavior during the user study.

This system is over USB and only implemented in Android. The system should be easy to change to other communication channels and platforms, but this should be looked into. The cognitive dimensions framework is interesting, and it would be interesting to see if the correlations between users own perceived feedback and the one observed in the study holds in other projects as well.

Sammendrag

Mengden smarte enheter, slik som smarte klokker, briller eller til og med røykvarslere, som blir solgt vokser og vokser. De fleste slike enheter trenger en form for fjernkontroll for å styres eller konfigureres. Det gjøres ofte med en egen kontroll eller app. Dette fører fort til problemer hvor brukerne mister oversikten og sliter med å lære seg alle kontrollene. Det finnes kontrollsenter-apper, men disse er ofte begrenset av små, lukkede APIer som gjør det vanskelig å utvikle gode løsninger. I denne oppgaven ser jeg på en måte for å lage et service lag i mellom en simpel, hardware-nær API og en større, brukervennlig API.

Målet med denne oppgaven er å finne det beste kryssplattform rammeverket for hardware nære applikasjoner, og undersøke hvor egnet dette er til en slik app. I tillegg vil oppgaven se på forskjellige teknikker for å øke mengden kryssplattform kode man kan oppnå. Til slutt vil den undersøke hvordan man ved hjelp av kognitive dimensjoner rammeverket (cognitive dimensions framework) kan teste det ferdige APIet.

I forstudien fant man ut få av kryssplattform rammeverkene har støtte for USB. Ved hjelp av plugins kan man gjøre det i et par av dem, men ingen av dem har plugins som er ferdig utviklet eller offisielt støttet. Xamarin ble derfor valgt som rammeverket for resten av oppgaven.

Det er klart at Xamarin ikke er godt egnet til denne typen prosjekter. Selv om den gir tilgang til alle metoder som kreves for å lage USB-kommunikasjon (noe de andre rammeverkene ikke gjør), så er det bare et enkelt skall rundt de de platform spesifike metodene. Dette fører til flere negative konsekvenser. I tillegg er dokumentasjonen manglede, og dette fører til mye tapt tid.

Allikevel, sluttproduktet fungerte bra, og man endte opp med ca. 50% kryssplattform kode (og når programlogikken blir mer kompleks, vil mengden kryssplattform kode øke). Ved å fokusere på designtaktikker for modifiserbarhet kan man redusere problemene ved Xamarin. Å bruke kognitive dimensjoner rammeverket for å analysere APIet var hjelpsomt, da det lot deltakerne i brukerundersøkelsen reflektere rundt opplevelsen sin på en mer strukturert måte. Når man sammenligner resultatene man fikk når man lot deltakerne selv analysere systemet i følge dimensjonene med resultatet man fikk når man analysere det på bakgrunn av hvordan folk brukte systemet, ser man at det er en klar sammenheng.

Systemer som ble produsert er kun implementert på Android. Det skal være enkelt å bytte ut både plattform og kommunikasjonskanal, og resultatene burde holde på andre plattformer, men det kan allikevel være interessant å sjekke. I tillegg vil det være spennende å se om sammenhengen mellom brukernes analyse og analysen basert på brukertesten også eksisterer i andre produkter.

Contents

Chapter 1

Introduction

This chapter will introduce the relevant research areas for this thesis, and present what this thesis seeks to answer. In 1.5 the structure of the rapport will be presented.

1.1 Background

The Internet of Things is quickly becoming the next big things. Watches, glasses, kitchen appliances, smoke detectors and more are all going to become "smart" and start talking to each other. Every sensor and appliance is getting its own chip, allowing it to talk to everything else. An estimated 26 billion devices will be online by 2020 [18].

The Internet of Things (IoT) is simply the interconnected network of things with access to the internet. This included everything from small RFID sensors to more complex computer-like objects such as the Samsung Gear watches or the NEST smart thermostat. It still lack clear boundaries, and some have compared it to the wild west [22].

As more and more devices require advanced setup and manual controls, some experts are worried about users ending up with a "basket of remotes" problem. Lots of apps, controls and devices, and no easy way to learn them all. Jean-Louis Gassée [13] argues that if we can't solve the "basket of remotes" problem, the internet of things will never succeed (at a consumer level). One way of solving this problem is by having good open APIs, allow everyone to make custom apps for controlling devices. One app can control multiple devices or you can have apps that function in a similar manner (or just choose apps that you feel are intuitive, lowering the learning curve).

Making this APIs are however not always easy. API design is a hard problem, and often you are limited by the hardware. In this thesis I look at a way of designing a cross-platform service, which acts as a layer between the hardware and an open API. By making a service that handles communication with the hardware you can extend the API with new functions and logic, without changing the hardware API. At the same time you can change the hardware, com-

munication methods and hardware API without it affecting any of the graphical user-interfaces made using it. This will make it easier to integrate your hardware into a universal control center, as well as for users to create custom interfaces.

API design is an important subject, and much research has been done on it. There are two main approaches to API design guidelines. Books and articles has been written based on experience, such as [6] and [10]. By using their expertise from Oracle (SUN Microsystems at the time of publication) and Microsoft respectively they have distilled years of working with the Java Development Kit API and the .Net base libraries into a set of guidelines. Those books look at what they have done wrong and what they have got right, both as people and as companies. Other smaller case studies of good and bad APIs exist such as [16].

The other approach taken is trying to find testable hypothesis and then testing them. Microsoft pioneered this approach [8], but other have followed in their footsteps, looking at things such as usage of the Factory Pattern [11] or constructor parameters [20]. This gives a more thorough understanding of not just what works, but how and why. However, the scope is a lot smaller, which limits it's usability until more work is done.

[19] tries to define the full scope of API Design, showing what choices you must consider. They group them by looking at the design guidelines and ideas presented by other works, as well as by reviewing two big libraries (.NET base libraries and The Java Development Kit). A few areas that they consider important for further study are also identified.

Cross-platform tools are also an important area of study, but in many ways it's not as well developed as the API field. [17] and [15] look at a subsection of cross-platform tools and compare them. They do however lack a clear reason for choosing the criteria they use. [21] implements a smart-meter tool to control a smart-meter using an app, in multiple cross-platform languages and look at the differences. However, the main communication pathway between the app and the device goes through a web-service, which differentiates it from the product developed as part of this thesis.

1.2 Approach

First a pre-study was done, analyzing different cross-platform frameworks. This analyze extended and buildt upon the ones found in [17] and [15]. Based upon this analysis a framework was selected for the case study.

In the case study a library was created to communicate with an external unit and provide an API that can be used to create a GUI. For a full description of the system created see chapter 3. The project was then be analyzed for cross-platform code percentage, to see what (if any) gain you can expect from using the cross-platform framework. This, together with more qualitative experiences was used to evaluate the chosen framework.

A test was done to check how easy it is to change the communication channel. The test implemented a simple dummy communication channel. The work was be measured in files affected as well as more qualitatively. Using

this data, the architecture of the system was evaluated. How the framework affected it was an important question, but not the only one considered.

Lastly the finished API was measured by a user-study, using the Cognitive Dimensions framework. The evaluation will be based upon the work done by [9], [8] and [11]. However the evaluation extends those by using the cognitive dimensions framework also as a direct feedback tool, allowing the subjects to rate the system on the different dimensions, and not just as a way of presenting the final analysis.

1.3 Research questions

The main objective of this thesis is to look at how creating a library that acts as a layer between a peripheral device and an app can be achieved, and what problems you might encounter. Specifically the following research questions were considered:

- Which application framework is most suited when you are working with low level hardware, and how mature/suited is it?
- What architectural techniques are important to maximize cross-platform code, for such a system?
- How can the cognitive dimensions framework be used to help with the analysis of a system?

1.4 Limitations

The finished product was only implemented for Android. The reason for this was mainly time constraints. At the same time it was reasoned that since Xamarin makes it easy to see which code can be reused (and which code needs to be custom made for each platform) the additional data obtained by reimplementing the custom parts for other platforms would be unimportant, especially considering the huge increase in workload. Using USB as a communication channel is also not the only (and probably not the most common) channel, so some of the results might have been different using another channel.

The user study performed as part of the evaluation was quite limited, both in number of participants and in time spent with each participant. The participants are also to similar (all are students at NTNU), so this is a clear weakness.

1.5 Structure of the Report

In chapter one a broad overview of the thesis is provided. This includes information about the research questions (in section 1.3), and about the approach, as well as general information about the background and motivation of the project.

Chapter two provides an introduction to some relevant theory, as well as the reasoning behind choosing Xamarin for the case study.

Chapter three starts off with an overview of the whole ecosystem that the case study is a part of. Following this in 3.2 a rundown of the hardware protocol used to communicate with the external device. In 3.3 a full description of the system created is provided.

Chapter four goes into detail about the user study, and the experiences and results from the case study.

In chapter five a conclusion is provided, and some further work is suggested.

Chapter 2

Prestudy

This chapter starts with a review of the available cross-platform technologies considered for the project. Each of them are discussed and compared, before a conclusion as to which one the case study will use is reached. After this a short introduction to some of the most important concepts in system design and architecture, and API design is provided. Following this is a short introduction to the cognitive dimensions framework, which is used in the user study.

2.1 Cross-platform frameworks

Cross-platform frameworks are any technology that allows you to target multiple operating systems at once. It's also often restricted to just mobile operating systems, since this problem has mostly been solved on PCs (a java program can run on almost all computers, and even programs written in other languages, such as C++, can often be cross-compiled)[21]. This thesis will only consider frameworks targeting mobile operating systems.

Cross-platform development gives many advantages over native development [17]:

- Less specialized skill required, since you only have to know one framework and language.
- Faster development time, since you only need to write the code once for all platforms.
- Higher market share, since you can target all platforms

However critics claim that they lead to slower apps, badly constructed apps and that the increase in development speed is not that large [14]. Newer reports do however find that the speed difference between apps are no longer very significant if you use a framework that's built on the heavily optimized webkit engine [21].

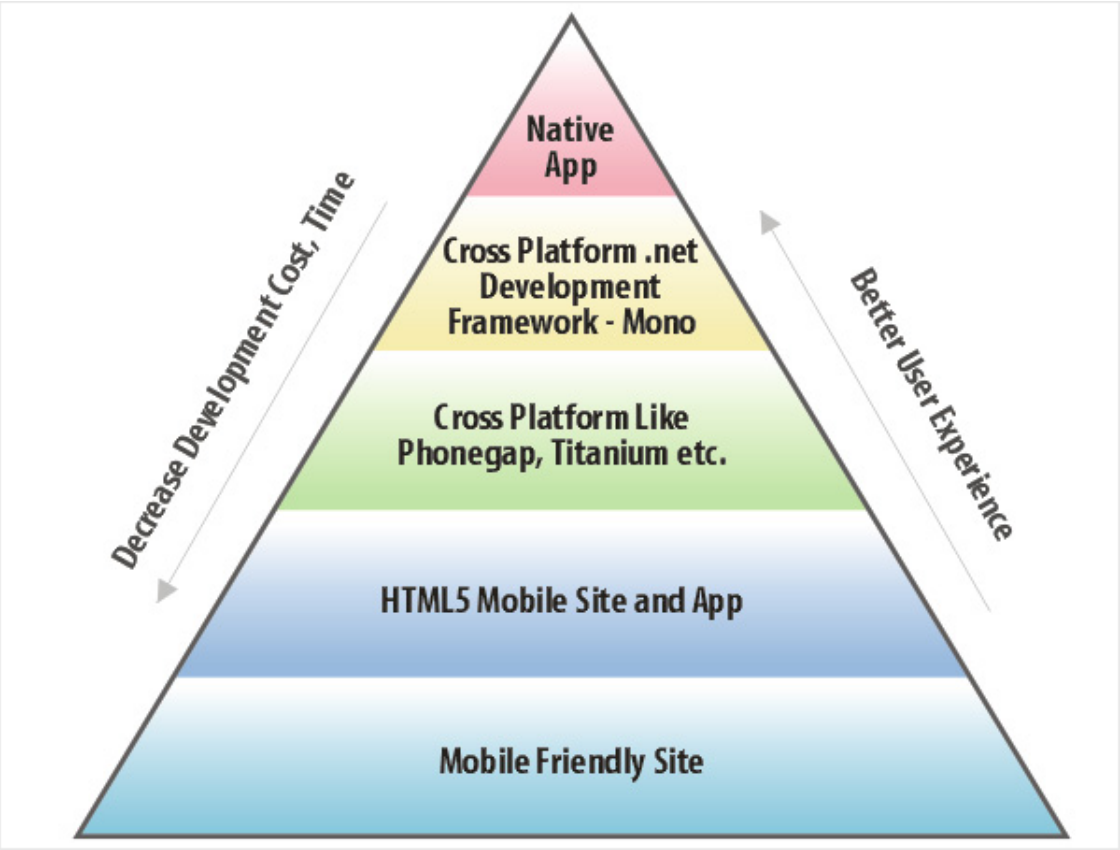

Figure 2.1: Comparison of different development strategies [2]

The frameworks come in many different forms, but they can be divided into 4 main categories [14].

- Web apps
- Hybrid apps
- Interpreted apps
- Cross-compiled apps

When choosing between the main categories there is a tradeoff (see figure 2.1). You trade ease of development for a native end-user experience. There are too many different cross-platform frameworks to look at all. When selecting frameworks for this prestudy three criteria was used. We wanted to consider at least one framework from each main category, we wanted to select mature and popular frameworks, and they needed to support IOS and Android at the least.

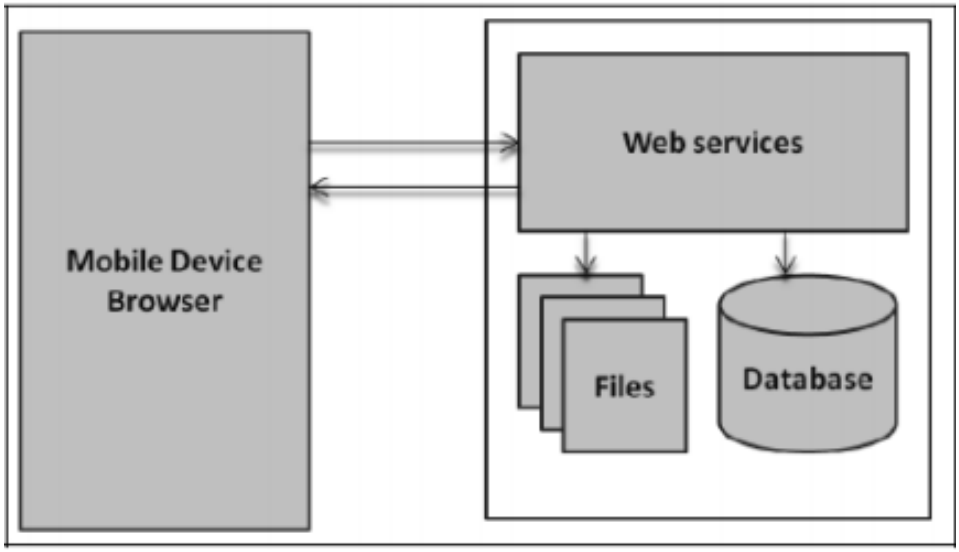

Figure 2.2: How web apps achieve cross-platform compatibility [12]

2.1.1 Web apps

A web app is simply an interactive webpage. Using HTML5, JavaScript and CSS you can create a mobile friendly webpage that feels like an app. With HTML5 you gain access to a few phone functions, such as geolocation and storage. However, it's quite limited both in the functionality you have and in your options for making them feel native. You have some options to change the interface based on platform, but you do not have access to the native GUI elements. You cannot package them to sell on the phone marketplaces which limit the usage. The app relies on the fact that the browsers all implement the ECMA standard to be cross-platform (See figure 2.2).

Even with the downsides web apps are quite popular, since they are truely "write once, run everywhere". They can be run on mobiles and computers, and there is no installation required.

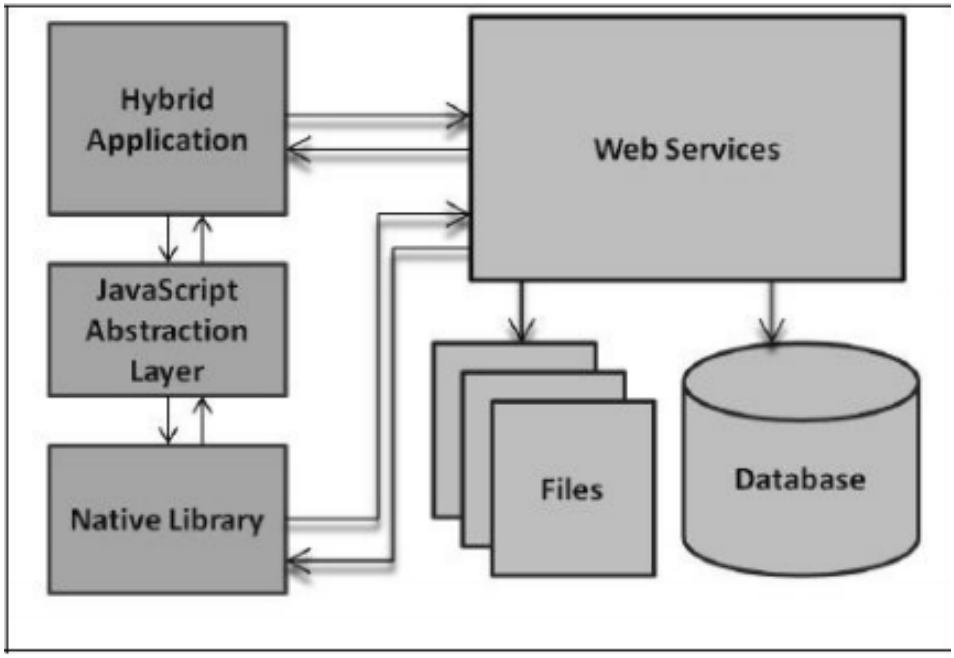

Figure 2.3: How hybrid frameworks achieve cross-platform compatibility [12]

2.1.2 Hybrid apps

Hybrid apps are designed and made in HTML5, JavaScript and CSS, just as pure web apps. But in addition they employ a wrapper layer, allowing you to call functions from the native operating system, which helps to bridge the gap between web and native apps (see figure 2.3. You can also package them and sell them on app marketplaces (such as the Appstore for the iPhone or the Google play store for Android). The webpages are installed on the device, and will therefore load faster and even without internet access. On most platforms the hybrid apps use the webkit engine to render, and are therefore heavily optimized.

Phonegap Phonegap is "an open source solution for building cross-platform mobile apps with standards-based Web technologies like HTML, JavaScript, CSS." [3]. Phonegap is an distrubution of the Cordova engine. The apps are written in HTML5, JavaScript and CSS, which allows it to run on all platforms. You also gain access to a wrapper layer, which gives access to most phone functions. You can write plug-ins in native code to give you access to functionality that isn't in the core Phonegap framework. Phonegap can be used with other JavaScript frameworks such as Jquery or Sencha Touch, which are specialized in providing tools for building GUIs[7].

Phonegap allows you to create one codebase that can run on all platforms, with a similar GUI on all platforms. You can write part of the app platform specific, to allow you to make small corrections to the GUI for each platform, but this is not required. If you use a frontend framework (such as Sencha Touch) you gain access to the native GUI building blocks, which can make the GUI look and feel native.

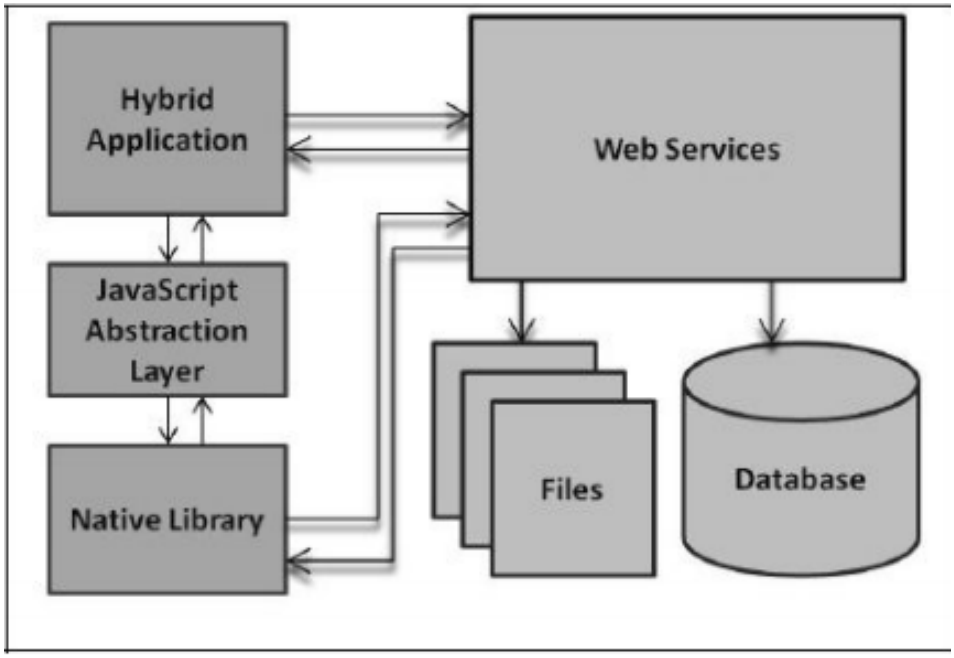

Figure 2.4: How interpreted frameworks achieve cross-platform compatibility [12]

2.1.3 Interpreted

Interpreted frameworks employ an interpreted language such as JavaScript or Ruby. It will then interpret that code at runtime, after it's been deploy to the different systems (see figure 2.4). This allows good access to hardware functions, but can suffer in speed (since there is no precompiling, and you don't get to use the highly optimized webkit).

Appcelerator Titanium Appcelerator Titanium is an open source framework that uses open web standards and architecture. The Titanium Mobile SDK uses JavaScript, HTML and CSS to enable cross-platform mobile development. According to their website [4] more than 70,000 apps has been made using Titanium, including apps for such big brands as Reuters, eBay and Cisco. The SDK comes with its own IDE, Titanium Studio.

Appcelerator Titanium allows you to write one UI code that is then translated into a native GUI using native components. Some platform dependent code might be necessary to make it conform to the platforms GUI guidelines.

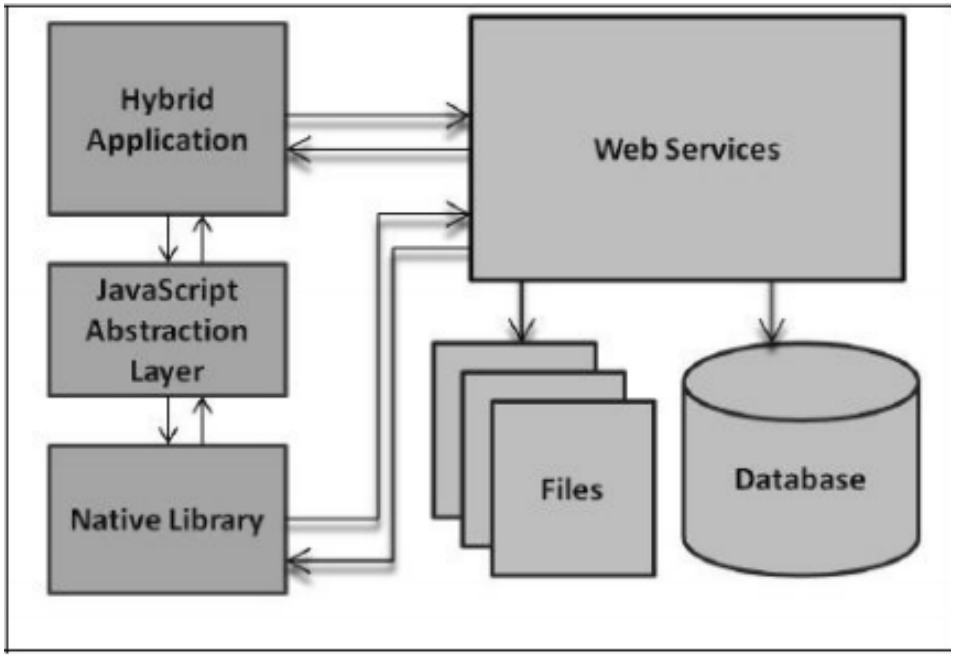

Figure 2.5: How Cross-compiled frameworks achieve cross-platform compatibility [12]

2.1.4 Cross-compiled

A cross-complied framework has a custom made compiler for each target platform. Code is written once, and then it's complied down to native code. This allows high speed and access to all native features.

Xamarin Xamarin is a cross-platform framework, based on C# and the .NET framework. Xamarin code is written in two parts, one platform independent application core, containing all program logic, and one application specific part, containing any GUI code as well as any platform specific features (also written in C#). According to Xamarin [1] mobile applications developed with Mono for Android tools have 70% smaller system footprints, 40% faster build times as well as faster installation than java due to the smaller size of the application. Xamarin is widely adopted, with big companies such as Microsoft, Cisco, HP and AT&T as its users.

Xamarin is based on the open source Mono compiler, but Xamarin is closed source. Xamarin has a free version (which has quite a few limitations) and a few different licensed ones that cost money. Xamarin provides an eclipse based IDE and a Visual Studio plug-in.

Framework	Price	Language	Type	HW	IDE
HTML5	Free	JavaScript	Web-app	Partial[1]	Any JavaScript IDE
Phonegap	Free	JavaScript	Hybrid	Partial[2]	Can be used with any JavaScript IDE, no official
Titanium Appclerator	Free	JavaScript	Interpreted	Partial[3]	Titanium Studio
Xamarin	From free to $1899 a year	C#	Cross-compiled	Full	Xamarin Studio and a Visual Studio plug-in

Table 2.1: Overview of available cross-platform frameworks

2.1.5 Comparison of the different frameworks

Since access to the hardware is so important for this application Xamarin was chosen as the development framework. It was the only one that gave access to Bluetooth and USB communication straight from the box (see table 2.1. No mature and well-functioning plug-ins were found for the frameworks requiring them, and writing them would negate many of the pros of using a cross-platform framework as well as take a lot of time.

2.2 System design

2.2.1 Design Tactics

[5] describe a design tactic as

> A tactic is a design decision that influences the achievement of a quality attribute response — tactics directly affect the system's response to some stimulus. Tactics impart portability to one design, high performance to another, and integrability to a third.

Tactics can help you make informed choices about what quality attributes you focus on (and which tradeoffs you are willing to make). It is important to identify the relevant tactics for each project, based on the important quality attributes and on the project scope.

Modifiability The most important quality attribute for this project was modifiability, i.e. how easy the app is to change. This is important for two reasons. For one thing we want it to be easy to change the underlying communication platform, without having to change the outward facing API. Since we want it to be cross-platform we also want the core to be largely unaffected when changing something, since any change there could have unintended consequences on other platforms.

- Reduce the Size of a Module

  The cost of modifying large modules is larger than the cost of modifying small modules. This means that

[1] some access to storage and a few functions.
[2] Full access can be obtained with plug-ins, but you need to write those in native code for each operating system, or find premade libraries
[3] Full access can be obtained by writing a module

splitting modules into smaller ones is an effective way of reducing the cost of making changes. This also makes it easier to place whole modules in the core of the application.

- Increase Cohesion

  Cohesion is how strongly the responsibilities of a module are related. If cohesion is low, a change to one responsibility can often affect another responsibility. We therefore want to keep cohesion as low as possible. The way of increasing cohesion is simply to find modules with low cohesion and move some unrelated responsibility to another module.

- Reduce Coupling

  Coupling is the chance that a change to one module will affect another module. Coupling reduces modifiability, since you need to change more than one module when you want to make a change. Many tactics exists to reduce coupling. All of them work by placing intermediaries between the modules.

  We can *encapsulate* the functionality, by introducing an explicit interface. The interface can hide details from users, allowing them a static entry point to the functionality, even if the implementation changes. This is also important for cross-platform code, since we need the code to change between platforms.

  We can *use an intermediary* to break a dependency.

  Limiting visibility can be a good way of *restricting dependencies*. This once again allows us to change the internals of a module, without being afraid that someone depends on it. This is often seen in layered architectures.

  If more than one module has almost the same functionality we can *Abstract the common service*. * If they share the exact same functionality we can *refactor* them, by moving common responsibility out of the module. This would allow us to move more functionality into the core of the application.

- Defer Binding

  The later you can bind values the better for modifiability. However this (more so than the other tactics) comes at a performance cost. Binding later allows more flexibility, and easier extensions and changing of code. Polymorphism is an example of a pattern based on this tactic.

2.2.2 API Design

An Application Programming Interface (API) is an external interface for a class. Almost all languages, libraries and frameworks contain their own API. When designing an API there are many stakeholders, all the designers and the users. The users are often unknown, since they might be external (or even exist far in the future). An API that works well for one programmer does not have to work well for another one. [19] This makes designing APIs quite hard, and therefore much research has been done on the right way to do it. Traditionally the research has been based around expert opinions and user studies on full APIs, and then trying to distill that into a set of guidelines. In 2003 Microsoft first started doing a more scientific approach, setting small hypothesis and testing them [9].

When creating an API there are a number of different decisions one most consider. [19] tries to map all those decisions (see 2.2.2). However, while they provide a framework for structuring the decisions, they do not claim that they list all of them. Not all of the decisions they have found will be relevant for all projects and languages, and not all decisions are of equal importance. However it gives a good overview of the full breadth of the decisions you need to consider. There are some overlaps between the sort of decisions you make when choosing between design tactics, but it also include some other decisions (such as choice of framework, what fields and methods to provide, etc.). They suggest that API design decisions are prioritized based on four dimensions:

- Design frequency: How often the decisions is relevant when designing an API.
- Design difficulty: How often the designer will make the wrong decision.
- Use frequency: How often the users are affected by the decisions.
- Use difficulty: How costly a sub-optimal decision is to the users of the API.

They suggest that both designers and researchers focus on decisions that are both used frequently and that are hard to make right (or where there are no clear answer to what the best way is).

2.3 Cognitive Dimensions Framework

In 2003 researchers at Microsoft published a paper detailing the work they've done using the cognitive dimensions framework to describe and test APIs[9]. While the framework was originally intended to be used to evaluate and describe the usability of a programming language, Clarke et al. showed how it could with some minor adjustments be useful for API design as well.

The modified framework describes a list of dimensions (see table 2.2). When doing user studies the answers are then analyzed based on this dimensions. This is useful for two reasons. First of it provides a common language to talk about APIs, you can describe trade-offs in terms of which dimensions are affected; you can describe problems, and so on. Secondly it makes it easier to find the true cause of problems, since you can relate them to a more abstract concept. Instead of saying "the user doesn't find what he wants in the references" you can say: "The API has the wrong abstraction level for the user, and he is therefore looking at the wrong place"[9].

Since all users have different requirements for the APIs they use they will also rank the importance of different dimensions differently. Microsoft has identified three main categories of programmer personalities. The groups have different focuses on what's important, and therefore what dimensions are most important. [11] The three personalities are systematic, pragmatic and opportunistic.

- Systematic programmers tend to have a defensive coding style. They like to tinker with components, and want to be able to see and understand the inner workings of the APIs they use. They therefore value a low level of abstraction.

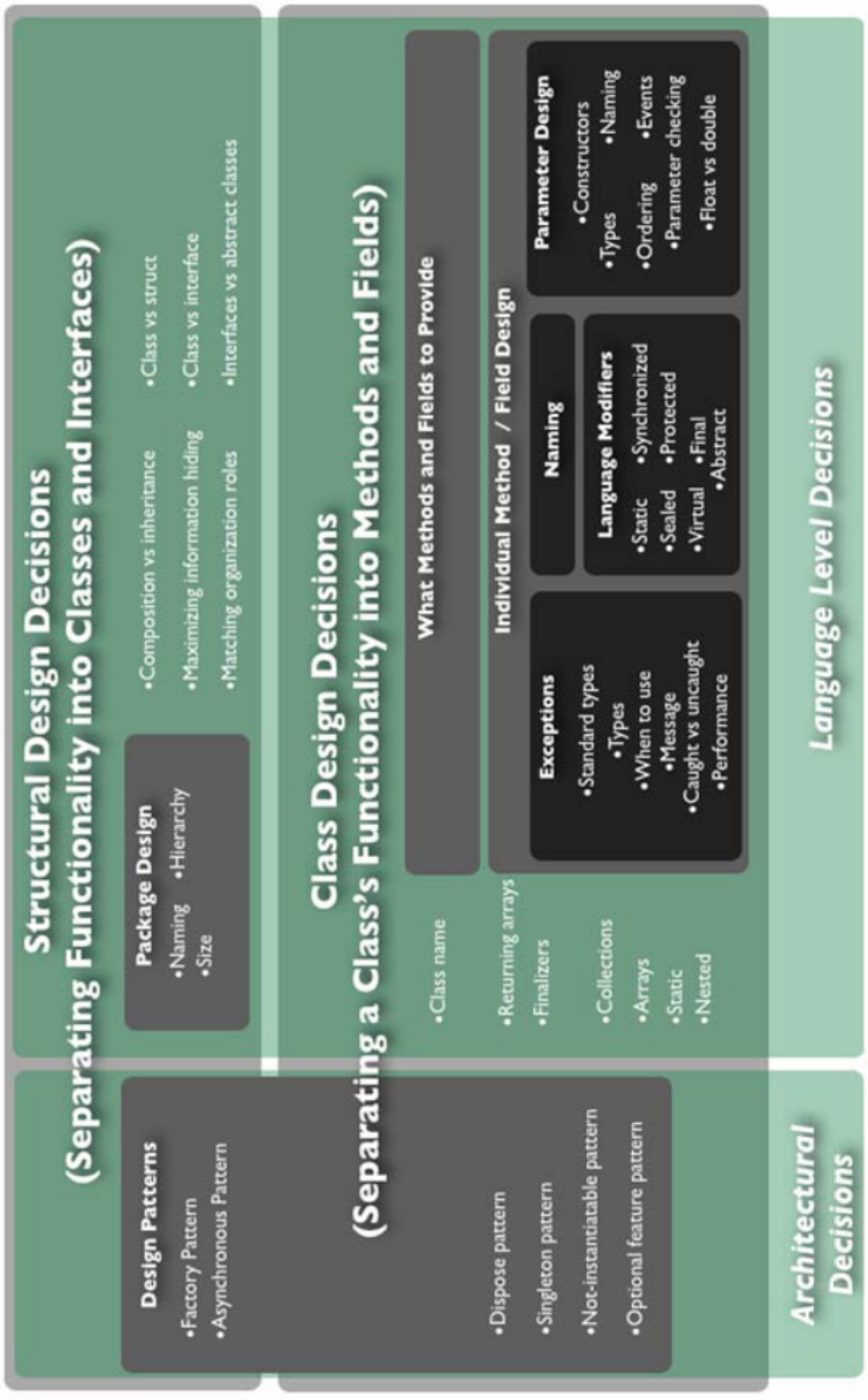

Figure 2.6: The space of API decisions [19]

- Pragmatic programmers want productivity, and they are willing to trade some control to achieve this. They like to learn as they go, and therefore place a high value on progressive evaluation.
- The opportunistic programmers want productivity above all. They place less value on having access to the inner workings of the APIs they use, and tend not to want too many controls. If something isn't working for their problem, they can find some other framework that does. They therefore want a high level of abstraction.

Of course finding developers that fit neatly into one archetype is often not possible, but knowing them you can find which category your users lean most toward. This is helpful both in troubleshooting problems (my users are mostly systematic developers, so I should decrease the abstraction level) and in testing (finding developers that lean toward the different archetypes can make sure different types of users can use the system).

Abstraction Level	The minimum and maximum levels of abstraction exposed by the API, and usable by a targeted developer
Learning Style	The learning requirements posed by the API, and the learning styles available to a targeted developer
Working Framework	The size of the conceptual chunk needed to work effectively
Work-Step Unit	The amount of a programming task that must/can be completed in a single step
Progressive Evaluation	To what extent partially completed code can be executed to obtain feedback on code behavior
Premature Commitment	To what extent a developer have to make decisions before all the needed information is available
Penetrability	How the API facilitates exploration, analysis and understanding of its components, and how a targeted developer can go about retrieving what is needed
API Elaboration	To what extent the API must be adapted to meet the needs of a targeted developer
API Viscosity	The barriers to change inherent in the API, and how much effort a targeted developer needs to expend to make a change
Consistency	How much of an API that can be inferred once part of the API is learned
Role Expressiveness	How clear the relationship between each component and the program as a whole is
Domain Correspondence	How clearly the API components map to the domain

Table 2.2: The dimensions considered in the cognitive dimensions framework

Chapter 3

The System

This chapter will give a short description of the system created as part of this thesis. First there will be a short description of the other relevant systems that already exist. Then a short look at the designed API, as well as the system architecture will follow.

3.1 Other related systems

There are two main parts to the Aalberg Audio ecosystem. First there is the hardware that Aalberg Audio has developed. This consists of an Ecco Pedal (which is a stombox that you connect to a guitar, which gives a delay effect when activated) and an Aero controller. The Ecco Pedal and Aero Controller talks with each other through a proprietary wireless technology. The pedals can be controlled from the Aero controller, changing all settings, or directly on the device. When the app was started the Aero Controller only allowed USB connections (in addition to the proprietary wireless technology) and this was therefore chosen as the communication medium. While it's not the best fit for such a system (Bluetooth or Wi-Fi are more traditional choices, which probably work better), it is nevertheless a valid option. In the future there are plans to remove the USB and instead focus on Bluetooth communication.

The Aero controllers also provide an API that anyone can use to change the settings of the pedals. However, this is not well documented, likely to change with later revisions and quite basic. In this thesis an extension layer on top of this protocol was created. The layer provides all the code required to talk to the device through USB, and it provides an extended open API that anyone can use. The service layer is created so that multiple GUIs can easily be created to fit different use cases. A simple GUI will be created as part of this thesis, for testing purposes.

3.2 Hardware protocol

When this system was first started Aalberg Audio had yet to design and implement a hardware protocol. Together we designed a protocol that people at Aalberg Audio then implemented. Because of hardware and time limitations it was decided to keep it minimal, with the possibility of extending it at a later time.

The protocol uses two USB endpoints. One in-endpoint (sending information from the device to the tablet), for receiving information about the connected devices (giving a list of all devices and their options). Then you have one out-endpoint (sending information from the tablet to the device), that allow you to send a new value for one of the settings belonging to one of the connected devices.

3.3 The API

The API is divided into two main public classes. The GuitarControllerService (which is a platform specific implementation of the API class, implemented as an Android service) and the pedal class. The GuitarControllerService is the main entry point of the API.

The GuitarControllerService is responsible for providing access to the pedal objects (which represents the physical stomp boxes being controlled). In the first iteration of the system this was simply done by calling "GetConnectedDevices". However, in later revisions a few convenience methods were added, to get a connected device either by ID or by name. This would make it easier if you wanted present just one device, e.g. when a device in a list is clicked. Two methods were also made to return either all device names or all device ids, since this could be useful for presentation purposes.

A save, get saves and a load save method is also provided, allowing you to save settings for all connected pedals, and quickly applying them. However, since there could be multiple ways of applying those settings (if you have multiple similar devices) how the settings are applied is left up to caller, which would often be the GUI. So you can call getSaves() to get a list of all saves, and getSave to get a specific save. The save is a list of pedal objects, that you can then apply to your own pedal objects to overwrite the current settings.

The pedal class contains accessors for all data, such as name, id and current settings (given as Option objects, which are just a custom data wrapper class around an options name and max, min and current value). In addition to this it contained a few methods. For changing the value of an option (either set it directly or increase or decrease it by one) and saving and loading the preset values. GetAllPresets returns a list of presets (another data wrapper class) and ApplyPreset will change all values to the ones saved in the preset. SavePreset takes in a name for the preset and adds it to the preset list. The preset list is stored on disk.

3.4 Overview of the System

The system is made up of three layers. See figure 3.1. The bottommost layer is the hardware layer, and it is responsible for talking with the external unit. By making it implement an abstract class it is easy to switch out. If you were to implement it again on iPhone or another platform all you would need to do is reimplement four methods (the actual sending of data) as well as the constructor. There is no logic needed. This makes it easy to extend, or you could even switch the communication form from USB to something else. By making the abstract class implement an interface you could implement a communication form that utilizes another protocol, without having to change any of the already implemented ones.

The uppermost layer is a platform specific entry point for the app. In Android it's implemented as a service. This is done for two reasons. First it allows you to easily implement entry points for other APIs (such as opensoundcontrol). Secondly it's required since the implementations of services are quite different in the different systems. If you just wanted to create a GUI for a platform, you could directly interact with the API class, as this is also marked as public.

The core of the system is in the middle and handles all logic. This part of the system is fully cross-platform. It handles everything from saving, loading, to parsing things received from the API. As much of the code as possible should be in this part. The controller factory must be able to create controllers for the different platforms, so it will not be truly cross-platform.

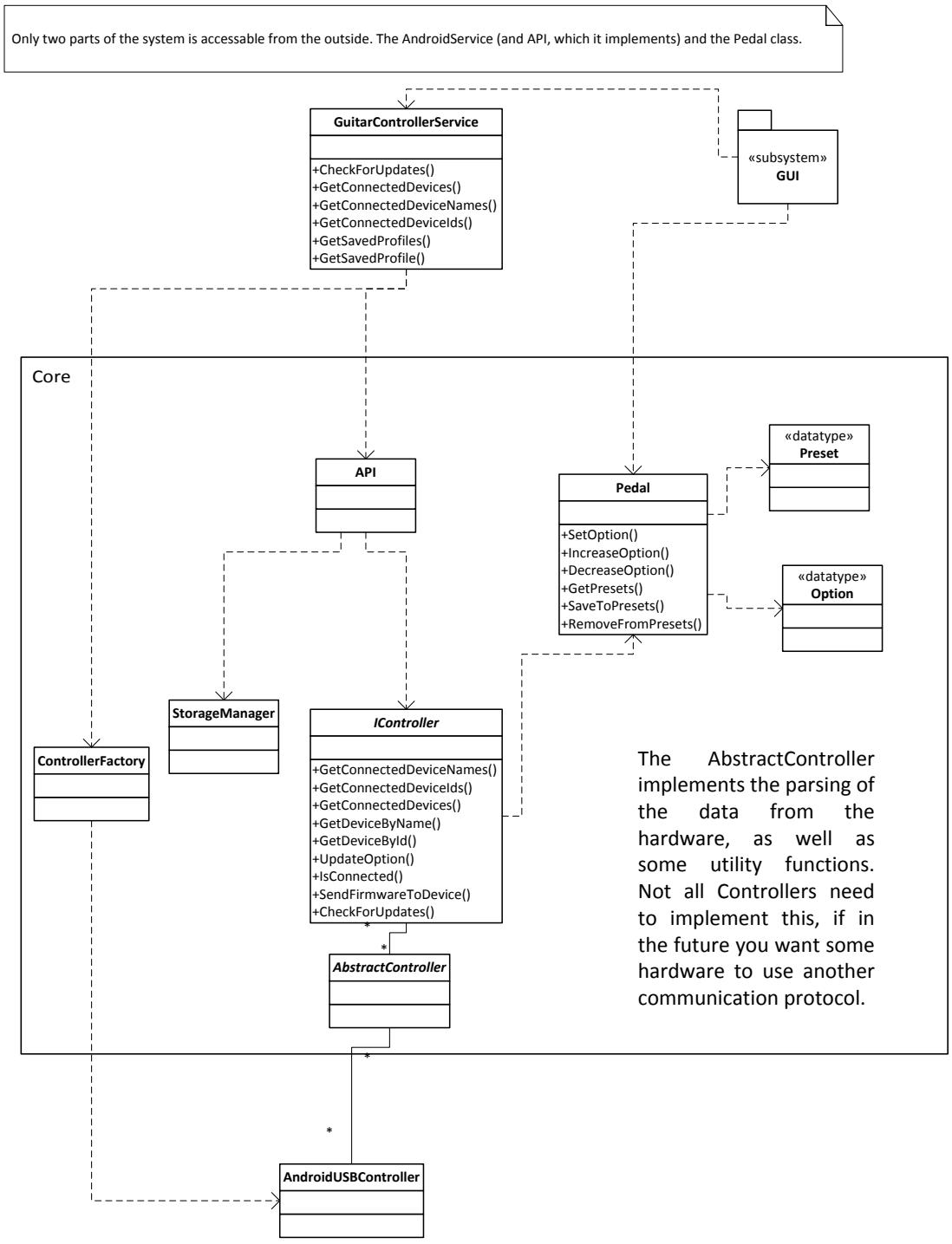

Figure 3.1: An overview of the structure of the system. Some parts in the core are omitted for brevity

Chapter 4

Results and Discussion

This chapter starts with some reflections around the cross-platform framework used and its readiness for these kinds of projects. Then the results from the analyses of the system will be looked at. Lastly the results from a user study on the API will presented and discussed.

4.1 Xamarin

Initially Xamarin looked like the most suitable system for this project (and it might still be). However using it there were multiple problems. The documentation around USB communication was lacking, and the implementation was not optimal. Instead of a true cross-platform solution it just gave access to the native functions, which created problems at other locations of the system, e.g. since the class communicating with the hardware needs access to the application context (an Android specific entity) either you needed multiple distinct ways through the core (one for each platform requiring a platform specific context to be passed) or you needed to allow the topmost layer and the bottommost layer to talk together, bypassing the core. Both of those solutions are bad, since they muddle the architecture of the system. For this system it was decided to use a factory to construct the IControllers. This factory must be updated for each new platform added which adds complexity when changing things (lowering the amount of modifiability) and makes the system less platform independent.

Apart from that Xamarin seemed to be stable and mostly mature, and it would probably be a good fit for most projects. It integrated nicely with Visual Studio and the Android development tools, as well as the .Net library.

4.2 Analysis of the system

According to Xamarin[1] you can expect to share about 75% of code on average. This was not found to be true for this project. There are probably multiple reasons for this. For one the support for USB was lacking, which increased the amount of native code required. To be able to mock USB devices a lot of boilerplate code was required to provide accessible constructors for some USB classes (which were required to be able to test the system without an USB device). Furthermore for a system such as this the application logic is for the most part simple.

However, 45% is still enough shared code that from a code perspective it is hard to find any reason not to use a cross-platform library such as Xamarin. 20% of the code is also GUI-code. If you exclude the GUI-code (since it was only written to test the feasibility of the system, and it's not required for the library) the platform independent code is 57% of the total code base. Since the amount of code required for external communication is relatively constant (for each USB-endpoint), if more logic were added to the code, you would see an even greater shift toward platform independent code.

Changing the communication with the hardware turned out to be quite easy. In the end 3 files needed to change. First IController needed to be implemented with the new protocol/technology. This is where most of the work lies. Then the controller factory must be updated to add the new class. If we wanted to only use the new protocol we could just change those two files (or if we just changed the AndroidUSBController, we wouldn't have to change anything else, but this would lead to bad naming and confusion). However to give the user an option to use both the old and the new one an option must be added to the API. This is done in the GuitarControllerService.

Focusing on modifiability turned out to be the most important factor for making the system easy to modify and also to have a high degree of platform code. The same elements that allowed the underlying protocol to be changed made it logical and easy to keep all platform specific code in one place (since the code required to communicate with hardware is highly platform specific). Encapsulation allowed the core to interact with platform specific parts of the system, with no knowledge of the platform specific implementations.

4.3 User Study

A short user study was done to test the cognitive dimensions framework, and to analyze the finished API. 9 students were given a set of scenarios which they then tried to implement using the framework. They were then asked about their experience, using questions based on the ones in [8]. In the paper the questions are provided as a way of self-analyzing a system, but here they are also used to see if the users' experiences agree with the observed reality.

4.3.1 Scenarios

All subjects were asked to solve a series of scenarios. The scenarios used is presented in table 4.1. The scenarios was created to test most of the functionality in the API, while still being short enough that multiple people could be tested. The users were given some boilerplate code (to create the service) as this is code that Xamarin require, that would have taken too long to create (code related to the creation of the initial GuitarControllerService). They were also not required to output the results on the screen since this required more Xamarin code than most were familiar with. Instead all outputs were recorded to the console, since it's the API that is being tested.

Scenario 1	You want to display a list of connected pedals. How do you get this list?
Scenario 2	Try to increase the amplitude of the Ecco 1, and decrease the delay.
Scenario 3	You want to set the delay of Ecco 2 to 60. How do you do this?
Scenario 4	You want to save the settings on all pedals. How do you do this?
Scenario 5	Create a new preset with delay 40, amplitude 30 for Ecco 2.
Scenario 6	You want to check if any new pedals have been connected, how do you do this?
Scenario 7	You want to apply the settings from your saved profile. How do you do this?

Table 4.1: Scenarios

4.3.2 Subjects

When selecting the people for the test the goal was to select at least one person from each of the programmer categories used by Microsoft [11], Opportunistic, Pragmatic and Systematic. To identify the programmer type of the subjects a short questionnaire were used. Of course none of the programmers cleanly fit in a type (those are after all archetypes), but it will give an indication of what they find most important. There were a majority of opportunistic programmers. This is probably an effect of selection bias. All the subjects were students at NTNU at the time of the study. The subjects were found at P15 October 7th and interviewed the same day. Due to lack of subjects with substantial experience with the Xamarin framework all of them are beginners, but they all had at least a year of programming experience. The subjects are presented in table 4.2.

Sex	Age	Field of study	Year of study	Programmer personality
Male	23	Informatics	4	Opportunistic.
Male	24	Informatics	5	Systematic.
Female	23	Engineering and ICT	3	Opportunistic.
Male	26	Informatics	3	Opportunistic.
Male	21	Informatics	2	Systematic.
Male	25	Computer Science	5	Pragmatic.
Male	20	Computer Science	2	Opportunistic.
Female	23	Informatics	3	Systematic.
Male	23	Informatics	4	Pragmatic.

Table 4.2: Subjects of user study

4.3.3 Results

Most users seemed to handle the scenarios well. However a few parts of the API seemed to cause some confusion. There were some confusion related to the two different forms of saving (presets on a pedal and saving all configurations for all pedals). The differences between the two methods are explained in depth in the documentation, but none of the users went to it before trying. It's unclear how this should be remedied. Maybe it's only a naming issue or it might be a more structural problem.

All the subjects were debriefed using the cognitive dimensions framework. First the subject got a short introduction to the cognitive dimensions. Then for each dimension they were asked a few relevant questions (such as:" Did you feel like you could stop in the middle of the scenario and check the progress of work so far?" and "Did you feel like you could find out how much progress has been made?" for premature commitment). This helped explain the categories for the users, and presented a few questions for them to reflect over (which was used as a more qualitative result). After that they were asked to give it an overall score on a scale from 0 (worst) to 10 (best) in that category. 10 is the score of a good system, even in negative categories (e.g. a score of 10 on premature commitment means that the subject felt that the system require a perfect amount premature commitment, even if that is very little). Looking at the final score (see figure 4.1) they are mostly positive.

There are some weaknesses of the results. They could be influenced by the low experience levels of the participants or it because the some of the categories were hard to understand. However they seem to fit with what we observed. Most of the subjects had few problems with the scenarios presented. Most of the negative comments we received was about the code we provided, to start the service. Some also said that they liked to look through the methods of a class using the autocomplete feature of the IDE, and that the GuitarControllerService had too many irrelevant methods (most of those are methods are Service methods that we gain when we subclass the service class).

Learning style and API viscosity were the two worst categories (5.9), with progressive evaluation (6) and abstraction level (6.2) following close behind. Abstraction level also had the highest standard deviation. The programmers that identified themselves as systematic programmers were the ones that scored this category lowest. This implies that the abstraction level might have been too high, since systematic programmers often wants to "get under the hood" of an API.

The API is not all that configurable, and that might be the reason that it scored so badly at API Viscosity. If users wanted to change anything substantial they would have to subclass parts of the API, which would lead to very messy code.

It is clear that using the cognitive dimensions framework as a debriefing tool is possible, but some of the categories might be slightly unclear. The results were mostly consistent with what was observed. The users were able to identify things they found confusing, and explain it using the dimensions from the framework.

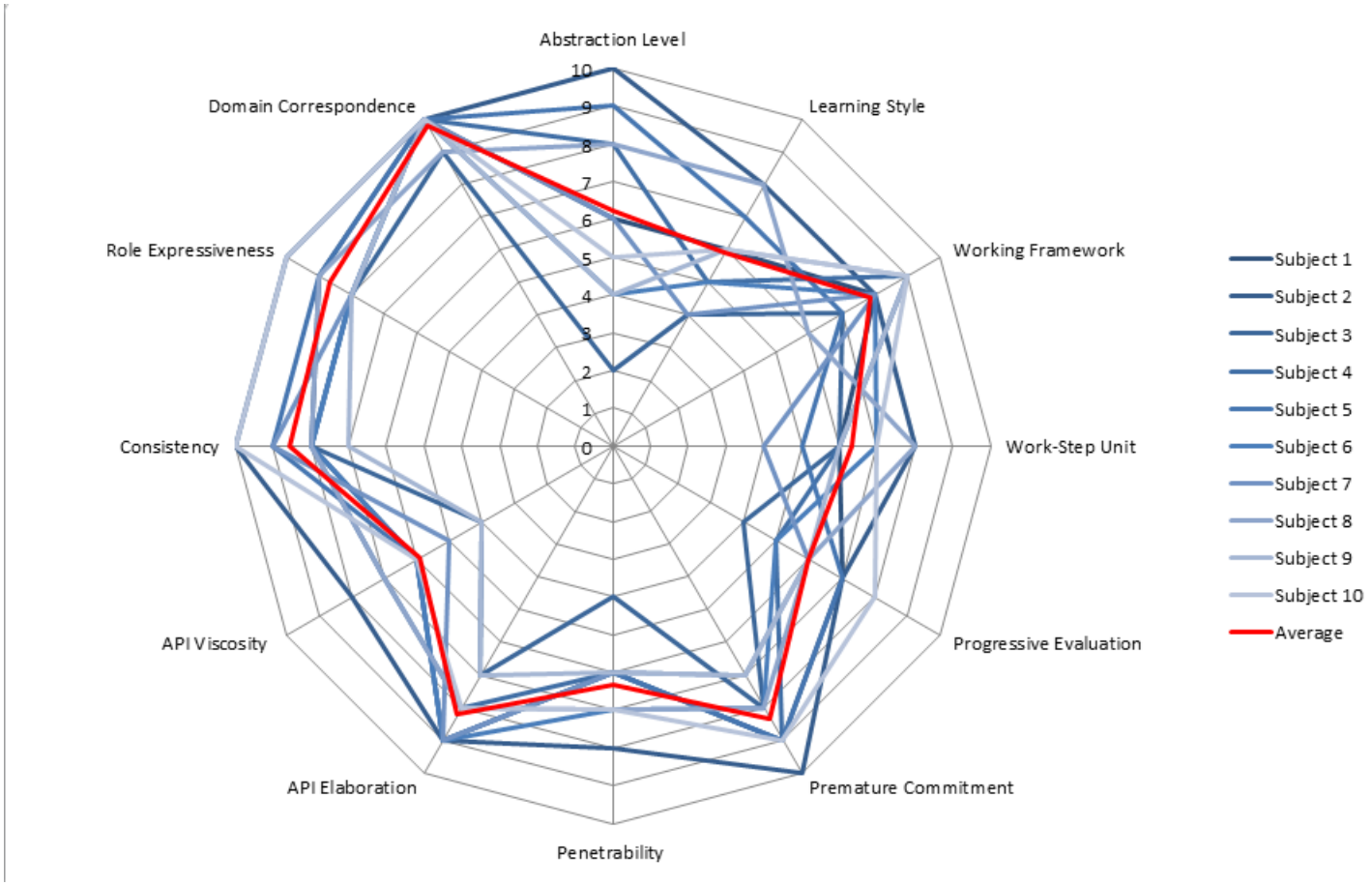

Figure 4.1: The scores from the user study

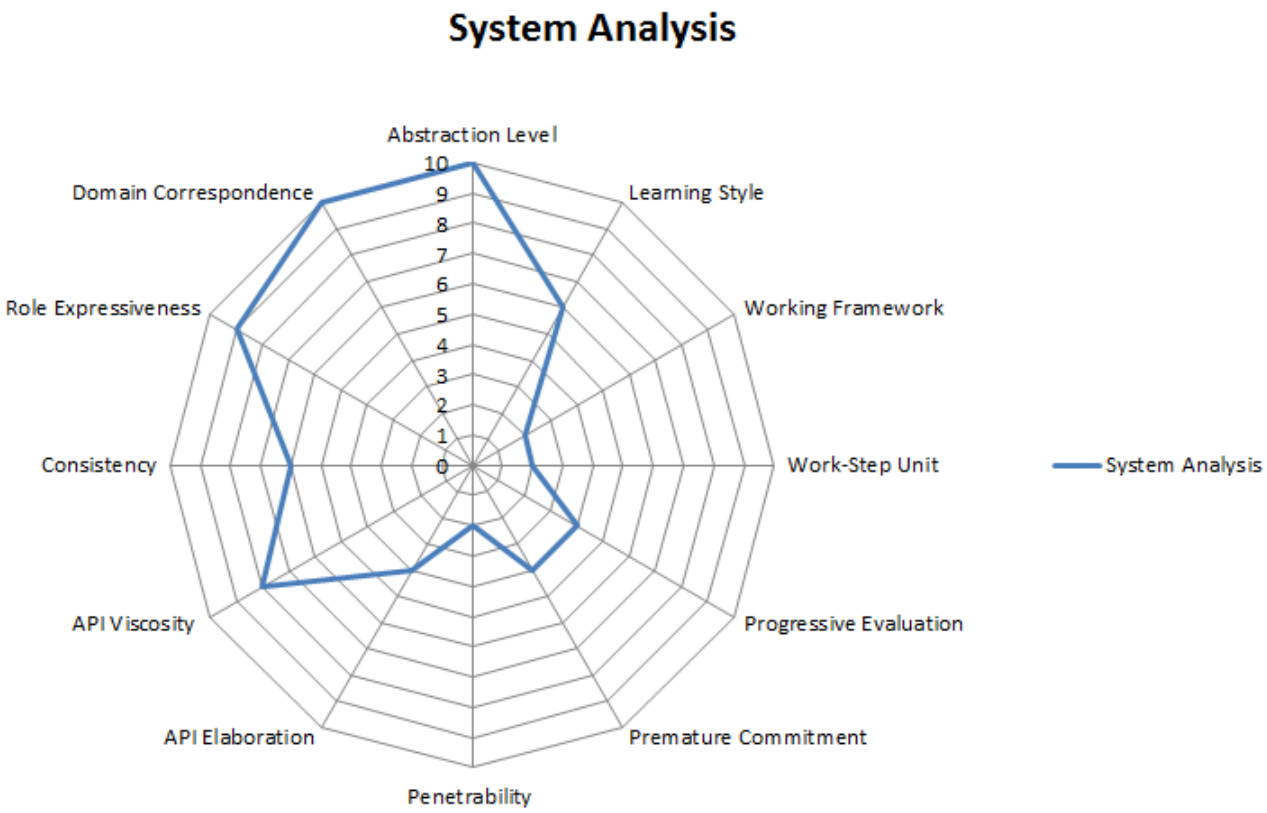

Figure 4.2: Cognitive dimensions analysis

Using the data gathered from the tests, a graph was created (See figure 4.2) that use a more traditional approach. Using the results from the study (both the actual feedback and the information gathered while observing them using the API) the API was ranked on the dimensions. This graph does not go between worst to best, but instead from low to high. Different developers will have different preferences (e.g. one developer would like a high abstraction level while another one might like a low one). Looking at this we can see that the system is made to be easy to use, while providing few opportunities for changing or "getting under the hood". This fit opportunistic programmers well, while systematic programmers might find this limiting. This also fit well the results gathered from users (see figure 4.3 to see the average results just from the opportunistic programmers), where the users that most want an easy system rated the system highest.

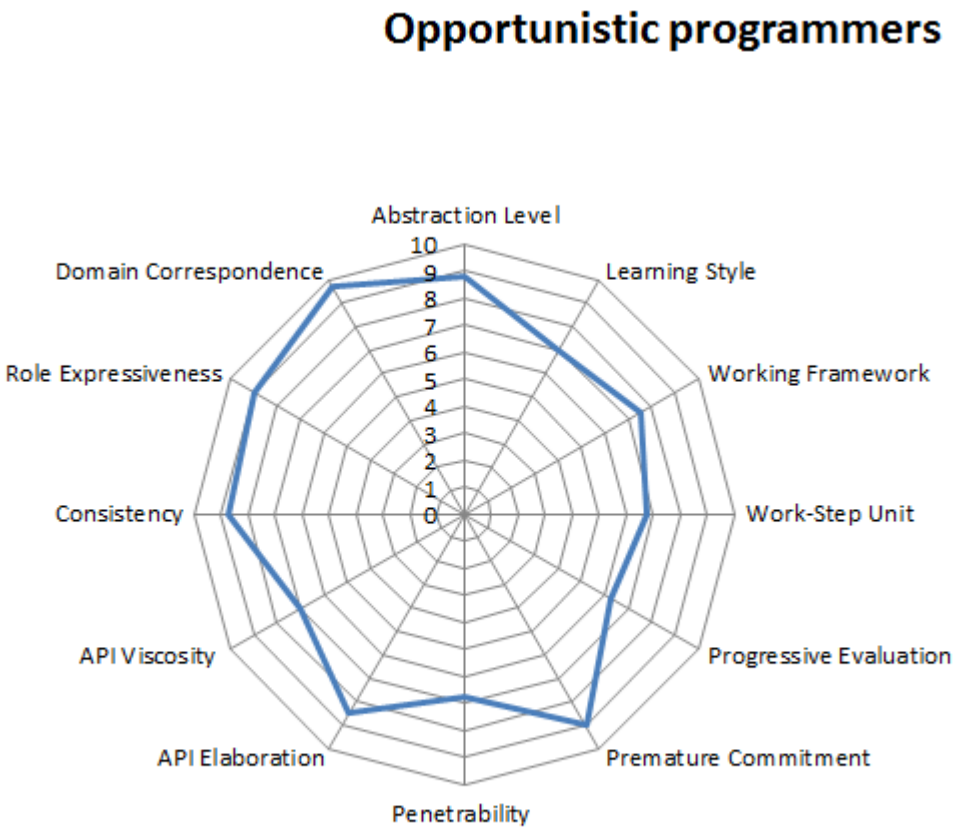

Figure 4.3: Average user study scores for just the opportunistic programmers

Chapter 5

Summary and Recommendations for Further Work

This section starts off with a short summary of the work done, as well as the results obtained. The chapter ends with some different opportunities for future work.

5.1 Summary and Conclusions

The main goal of this thesis was to look at the possibility of creating a cross-platform layer to increase the usability of a hardware communication protocols. To do this a prototype was created and evaluated, based on a few metrics and the cognitive dimensions framework. The prototype was created in Xamarin, since that was found to be the best fit for this project.

The system acted as a layer between a GUI and a guitar stomp-box created by Aalberg audio. Aalberg Audio has created a simple communication protocol; allow their hardware to respond to USB inputs. The layer handled communication with the stomp box, and some business logic around saving and loading, as well as a few other convenience functions.

The advantages of such a system are two-fold. First of since they are cross-platform they are easier and cheaper to produce, reducing development time and cost. Secondly they provide unique opportunities for fixing mistakes in the initial API design, as well as making developing for the device easier. This will in turn make it possible to integrate into other control solutions, to reduce the "basket of remotes" problem.

In the prestudy it was found that few cross-platform frameworks come with support for USB. Plugins can be made for some of them, providing the functionality required, but none of them are official or mature at the moment.

Xamarin was therefore chosen as the best candidate for such a system.

It is clear that Xamarin isn't a good fit for such a project. While it provides access to the methods required to create USB communication (which the other frameworks evaluated does not), it is just a simple wrapper around the native functions. This leads to a number of negative consequences, both on an architectural level and on a programmatic one. Couple this with lacking documentation, and the time saved is probably minimal. However, the end product worked well, and the saving was still around 50% (and as the complexity of the core logic of the application increase, so will the amount of cross-platform code).

By focusing on modifiability it was easy to move the platform specific code out of the core of the application. However, some architectural concessions had to be made because of the low support for USB in Xamarin. The program context (which is an Android specific object) had to be passed down to be used in communicating with the hardware. This led to a more muddled design, and this (and a few other similar issues) makes it hard to recommend Xamarin for any future similar projects. However, for projects that don't closely interact with the hardware (such as any thin client) would probably benefit from using Xamarin, or a similar framework.

The user study identified a few problems with the API, mainly related to saving and loading (since there were two methods to save, with few ways of knowing them apart). Using the cognitive dimensions framework made it easier to identify the cause of the problems, and find the root cause of them. It was also useful as a tool for making the users think about how they felt using the API, when they were asked a few of the questions recommended by Microsoft [8]. The cognitive dimensions framework was used in two different ways (allowing the users to rate the system on the different dimensions and by a more traditional evaluation) and the two results closely matched.

5.2 Recommendations for Further Work

In this thesis a prototype that demonstrates one way to create a cross-platform layer between a hardware device and one or more GUIs was created. The prototype was mostly finished and gave a good picture of how such a layer might look and perform. However a few questions were raised in the evaluation, and more work to look at them might be interesting.

- Saving and loading: This was the functionality that most people had a problem understanding. Looking at different ways of implementing it and evaluating them could provide interesting insight.
- The application was only implemented on Android due to time constraints. Seeing whether or not the finding hold for other platforms might be interesting. The required work to get the app running on other platforms should be fairly low, so it might be possible to do it on multiple platforms to compare.
- Only USB-communication was evaluated in this thesis. There are other (more popular) options for communicating with peripheral units, such as Bluetooth and Wi-Fi. Seeing how Xamarin handles those technologies might be interesting, even if our initial finding lead us to believe that it is done in a similar fashion to USB.

- Our finding on the cognitive dimensions framework point to a new way of using it (allowing the users to evaluate the framework using it), that in this case showed similar results to the more traditional way of doing it. It might be interesting to see if our finding holds in other larger projects and studies, with a wider range of participants.

Bibliography

[1] Xamarin homepage. URL http://xamarin.com/platform.

[2] Comparison Chart for Mobile App Development Methods. URL http://www.alliancetek.com/article-comparison-chart.html.

[3] Phonegap homepage. URL http://phonegap.com/about/faq/.

[4] Appcelerator homepage. URL http://www.appcelerator.com/titanium/.

[5] Len. Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, volume 2nd. 2003. ISBN 0321154959.

[6] Joshua Bloch. *Effective Java*. 2008. ISBN 9780321356680. doi: 10.1016/B978-075067929-9/50038-5.

[7] Jonathan Campbell. A cross platform mobile application for clackmannanshire council. 2013.

[8] Steven Clarke. Measuring API Usability. *Dr. Dobb's Journal Windows/.NET Supplement*, 10:S6–S9, 2004. ISSN 1361-4533. doi: 10.1080/13614530412331296826.

[9] Steven Clarke and Curtis Becker. Using the Cognitive Dimensions Framework to evaluate the usability of a class library. *Proceedings of the First Joint Conference of EASE PPIG PPIG 15*, 61:359–366, 2003.

[10] Krzysztof Cwalina and Brad Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. 2005. ISBN 978-0-321-54561-9; 0-321-54561-3.

[11] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. *Proceedings - International Conference on Software Engineering*, pages 302–311, 2007.

[12] Joy Friberg. Institutionen för datavetenskap Evaluation of cross-platform development for mobile devices av mobile devices. 2014.

[13] Jean-Louis Gassée. Internet of things: The "basket of remotes" problem, 2014. URL http://www.mondaynote.com/2014/01/12/internet-of-things-the-basket-of-remotes-problem/.

[14] Gustavo Hartmann, Geoff Stead, and a DeGani. Cross-platform mobile development. *Tribal, Lincoln House, The ...*, pages 1–18, 2011.

[15] Henning Heitkötter, Sebastian Hanschke, and Tim A. Majchrzak. Evaluating cross-platform development approaches for mobile applications. In *Lecture Notes in Business Information Processing*, volume 140 LNBIP, pages 120–138, 2013. ISBN 9783642366079. doi: 10.1007/978-3-642-36608-6_8.

[16] Michi Hennig. Why changing APIs might become a criminal offense. *ACM QUEUE*, (June), 2007.

[17] Manuel Palmieri, Inderjeet Singh, and Antonio Cicchetti. Comparison of cross-platform mobile development tools. *2012 16th International Conference on Intelligence in Next Generation Networks, ICIN 2012*, pages 179–186, 2012.

[18] Janessa Rivera and Rob van der Meulen. URL http://www.gartner.com/newsroom/id/2636073.

[19] Jeffrey Stylos and Brad Myers. Mapping the space of API design decisions. *2007 IEEE Symposium on Visual Languages and Human-Centric Computing Mapping*, pages 50–57, September 2007. doi: 10.1109/VLHCC.2007.44.

[20] Jeffrey Stylos, Steven Clarke, and Brad Myers. Comparing API Design Choices with Usability Studies : A Case Study and Future Directions. (September 2006):131–139.

[21] Alexander Zibula and Tim A. Majchrzak. Cross-platform development using HTML5, jQuery mobile, and PhoneGap: Realizing a smart meter application. In *Lecture Notes in Business Information Processing*, volume 140 LNBIP, pages 16–33, 2013. ISBN 9783642366079. doi: 10.1007/978-3-642-36608-6_2.

[22] M. Zorzi, A. Gluhak, S. Lange, and A. Bassi. From today's intranet of things to a future internet of things: a wireless- and mobility-related view. *Wireless Communications, IEEE*, 17(6):44–51, December 2010. ISSN 1536-1284. doi: 10.1109/MWC.2010.5675777.