Master's Thesis

Using MVVM for enhanced cross platform development of mobile and desktop applications

Written by Valentin Rock

Institute for Software Technology (IST)
Graz University of Technology
Inffeldgasse 16B/II,
8010 Graz, Austria

Assessor and supervisor: Univ.-Prof. Dipl.-Ing. Dr.techn. Wolfgang Slany

Graz, February 2015

Abstract

Over the last view years mobile devices started to replace the conventional desktops in many different areas of application. This led to plenty of different mobile platforms that are well established. For most companies it is important to cover a large number of platforms and devices to serve their existing customers as well as potential new customers. Since developing good applications for multiple platforms can be resource intensive and expensive, developers often have to decide if they share parts of their code between the implementations for different platforms. Today there are plenty of ways to develop cross platform applications.

In this work we will discuss two of the many possible solutions of how a cross platform implementation could look like. The first solution uses web technologies to develop an application for desktop and mobile devices. The second solution uses .NET to create a native looking application for the different mobile and desktop platforms Android, Windows Store and Windows Desktop. The main focus of this work is the architectural design pattern MVVM (Model View ViewModel) and how it can be used to enhance cross platform development.

We will see that cross platform implementations can share up to 90% of its code between platforms and that MVVM can helps to reach these high numbers. In addition to that MVVM can provide other benefits like reduced developing costs as well as increased user experience, maintainability, testability and performance. This work will help to understand MVVM as well as many related patterns that can help with cross platform development. The lessons we will learn in this work should help to decide which developing technologies are best suited for individual cases of application and how they can be combined with MVVM to create successful cross platform applications.

Index

1 Introduction

Over the last view years mobile devices started to replace the conventional desktops in many different areas of application. This led to plenty of different mobile platforms that are well established. For most companies it is important to cover a large number of platforms and devices to serve their existing customers as well as potential new customers. Developing good applications for one platform, takes a lot of time and resources. Developing individual applications for multiple platforms can often not be afforded. Therefore, cross platform development can be the right solution to minimize the developing costs and provide applications for a great number of users on their favorite device and operating system.

Most of the well-established applications support at least two mobile platforms. Some applications even support five or more completely different platforms for devices with screen sizes from one to hundred inch. These applications often support lots of different input methods like the conventional keyboard/mouse combination, touch or even camera based input.

There are many different solutions for cross platform development, but most of them do not provide good results for all of the relevant properties like user experience, developing costs, maintainability, testability and performance. It is important to know which development technologies fit the needs of a specific case of application best.

That is why this document will shortly explain some of the well-known technologies that can be used for cross platform development, to provide an overview of today's development tools and how they compare to each other.

Even if the decision for the right developing tools and technologies has been made, there are lots of open questions about how to use these technologies to get the best result. Most platforms are flexible enough to support different application architectures. A good application architecture is important for single platform development and even more important for cross platform development. The **MV*** family of patterns are well known by many developers and will get our main focus in this document. One derivation of those patterns is called **Model View ViewModel (MVVM)** which can be used as the architectural design pattern to build great cross platform applications. This pattern has its roots at Microsoft but can nowadays be successfully used on many different devices and platforms.

Later on we will see how **MVVM,** when used in cross platform applications, can greatly decrease the amount of duplicated code, increase the testability and performance and provide well-structured code that is easy to maintain.

There will be lots of examples from sample applications to support these claims.

Not many years ago very few smartphones and tablets have been shipped compared to today's numbers. The following numbers show that the support of multiple platforms can be relevant to the success of services and applications.

The following list shows the numbers of different users using the W3Scool's website. Since this numbers only reflect a special group of users it is not relevant for each service or application. Nevertheless it shows the trend to a higher operating systems (OS) fragmentation.

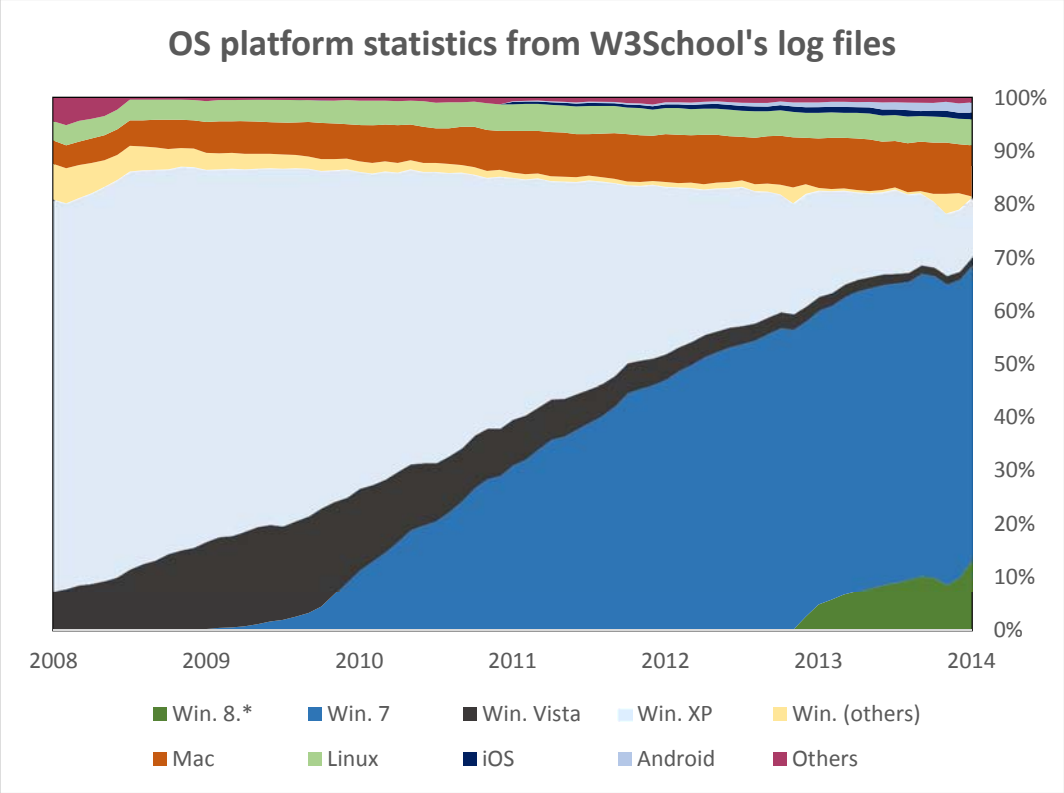

***Figure 1**: OS platform statistics from W3School's log files [1]*

The following statistic from the information technology research and advisory company Gartner, shows the shipment numbers of computing devices in 2013. The high number of mobile phones shipments shows the importance of supporting mobile platforms in addition to desktops, notebooks and tablets.

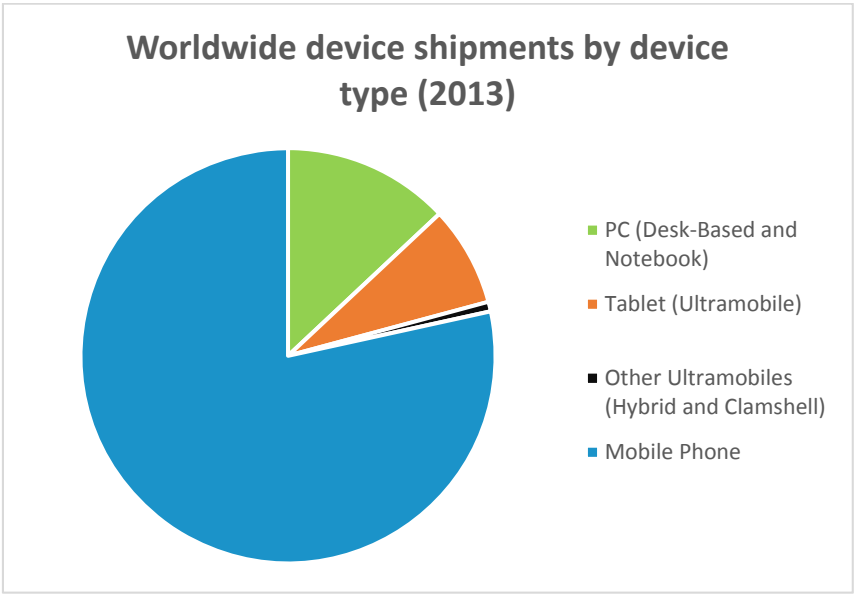

***Figure 2**: Worldwide device shipments by segment [2]*

Since we know now that there is the need to support mobile platforms let us take a look at smartphone sales in respect of mobile operating system's market share in the diagram below. Android and iOS are dominating the mobile OS sales in 2013.

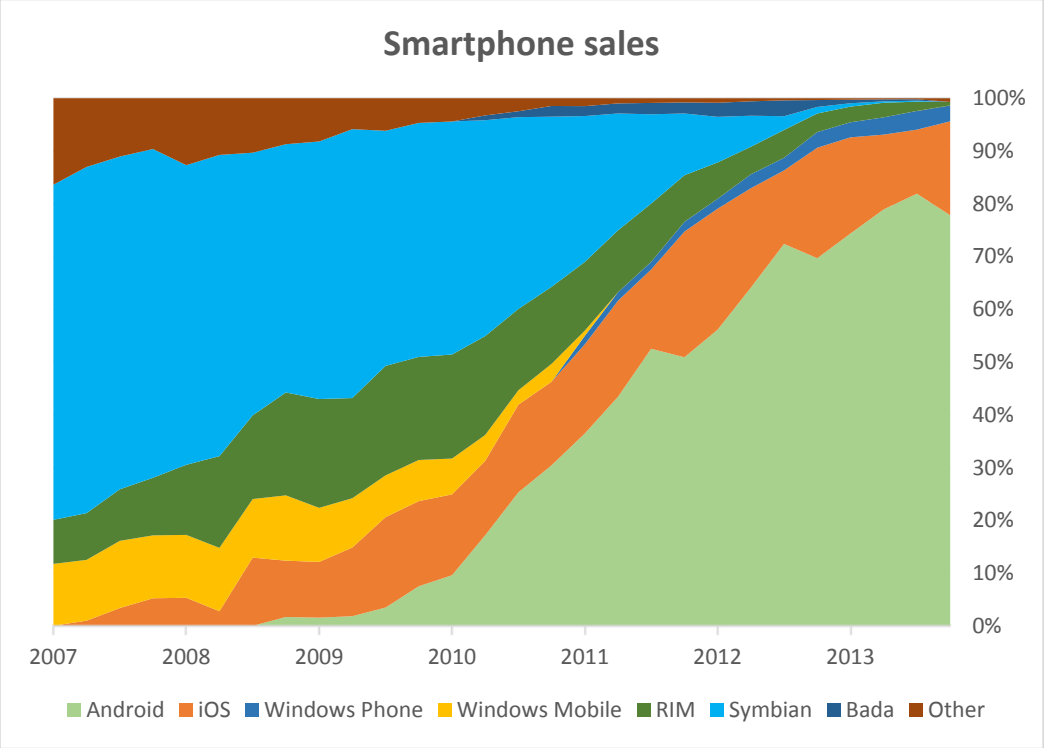

***Figure 3**: Smartphone sales numbers [3]*

The diagram below shows the total app store revenue of the biggest four app stores. In contrast to the smartphone shipments, where Android was the clear winner, the play store from Google delivers not the highest app revenue. This is likely the result of third-party app stores, payment platforms and the large number of free apps.

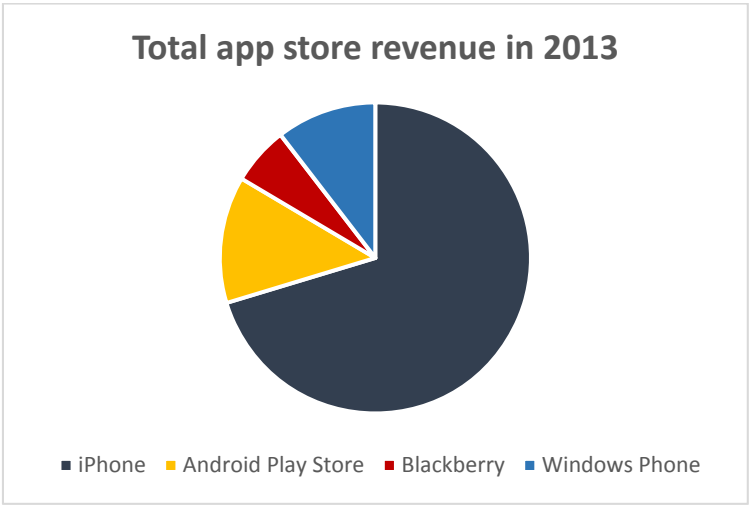

***Figure 4**: Total app store revenue in 2013 [4]*

When looking at device sales of all device, the operating platforms are well distributed with the tendency of an increased Android market share. The numbers from 2012 and 2013 are based on collected data. The numbers for 2014 and 2015 have been predicted by Gartner.

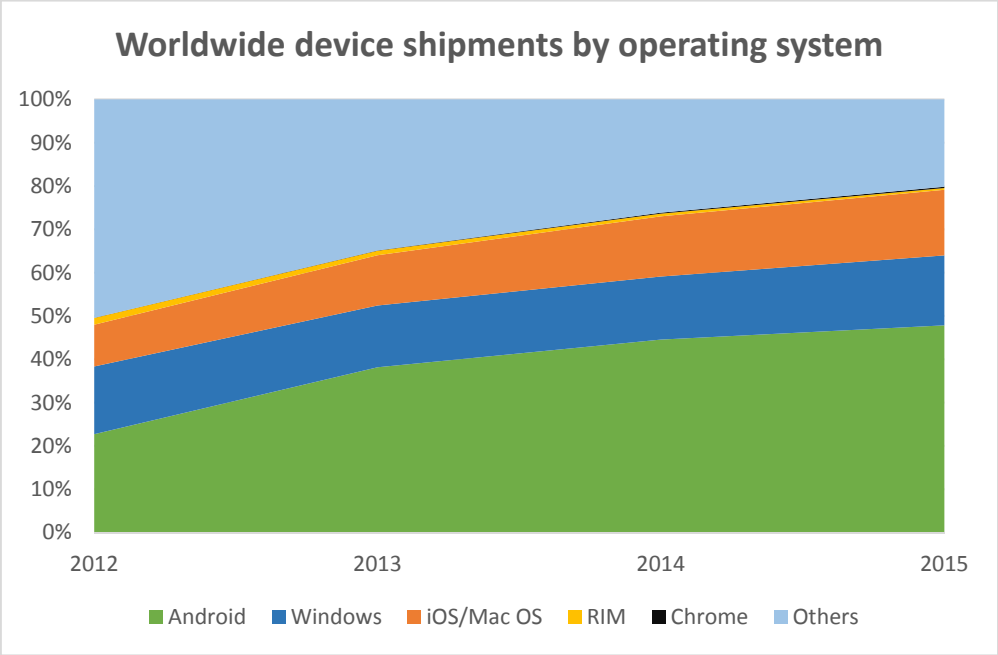

***Figure 5**: Worldwide device shipments by operating system [2]*

2 Understanding MVVM

MVVM, when used correctly and to its fullest extent, is more than a single pattern that consists of **Model**, **View** and **ViewModel**. In addition to these three essential parts that define the fundamental structure of the application, there are a large number of other patterns that can be used to enhance **MVVM** and improve the written code.

To completely understand how the individual patterns work in theory and how they can be used on each platform, this chapter will provide a detailed description of **MVVM** and some other patterns that can be used in combination with **MVVM**.

2.1 Related patterns (MVC and MVP)

To better understand **MVVM,** the patterns **Model View Controller (MVC**) and **Model View Presenter (MVP)** will be described as well, since they all are user interface architectural patterns and related to each other. This will show the similarities and differences of those three patterns and will point out the advantages and disadvantages for individual applications. All three patterns can help to improve code quality with regard to cross platform development.

2.1.1 Model View Controller

The **Model View Controller (MVC)** is the predecessor of **MVP** and **MVVM,** as well as other structural patterns associated to graphical user interfaces. Variations of **MVC** are widely used in web based applications and applications on other platforms that provide massive graphical user interfaces.

Model The **Model** is the data representation of the application. It can be used for any kind of data source like the local file system, a relational database or a web service. It notifies its **View(s)** about changes of its data.

View The view fetches the needed information from the **Model** and visualizes this information for the user. It notifies the **Controller** about the user's interaction with the user interface (UI) and receives requests for changes of its data representation.

Controller The controller receives user interaction notifications from the **View.** Based on the user's interactions the Controller will manipulate the **Models** data. The **View** will be notified from the **Model** about changes of the **Model's** data, to be able to update its UI. The **Controller** can also request the **View** to change its representation (visualization) of the **Model's** data.

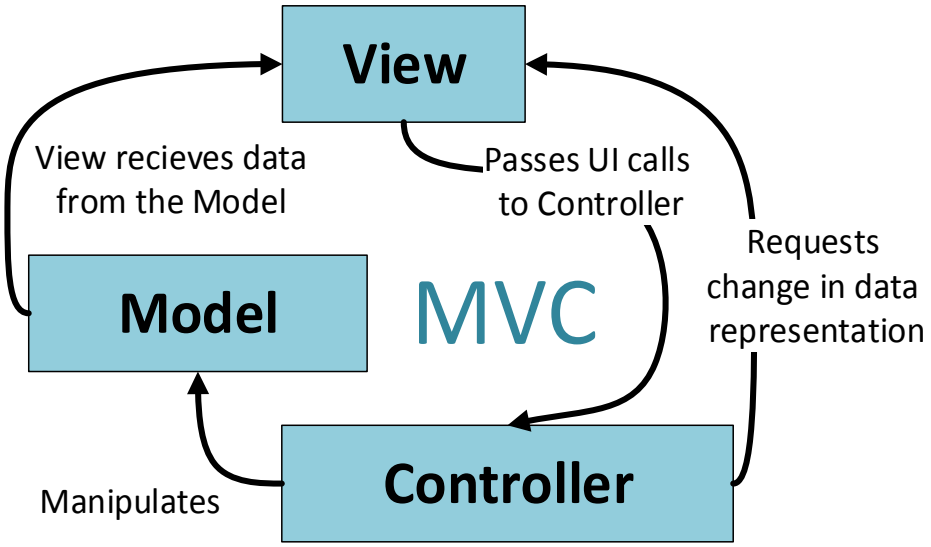

Figure 6: Simple dataflow diagram of MVC

The following sequence diagram shows how the **View**, **Model** and **Controller** work together to update the UI after the user clicks on a button.

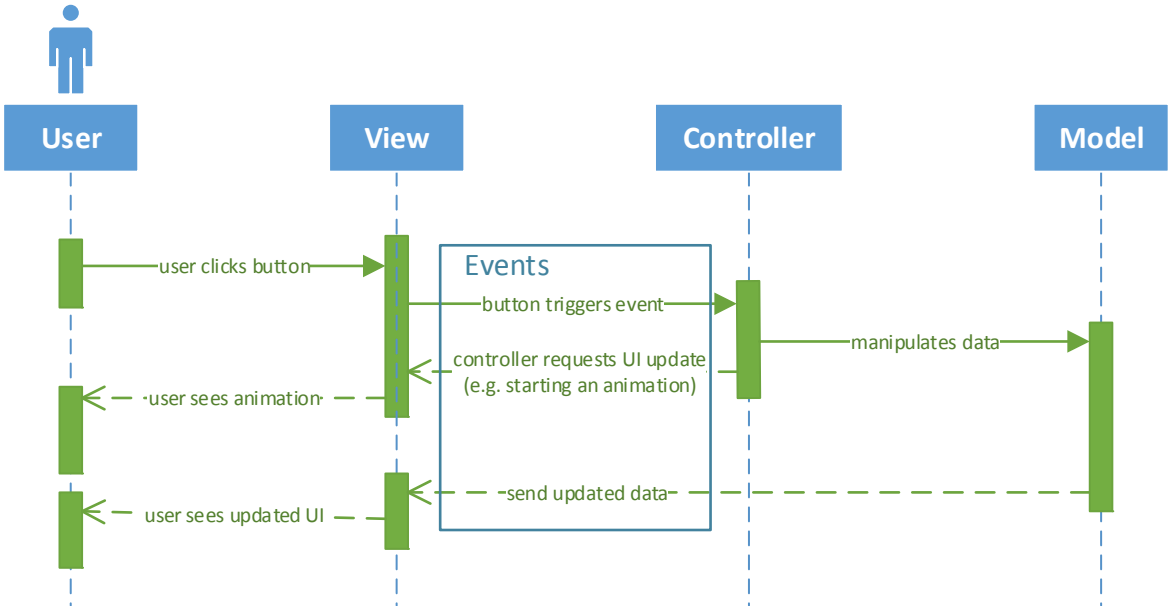

Figure 7: Sequence diagram of a simple user interaction using MVC

2.1.2 Model View Presenter

The **Model View Presenter (MVP)** is a derivative of the original **MVC**. The main difference to **MVC** is that all dataflow passes the **Presenter** in both directions. This can be a great improvement to **MVC** since the **Model** does not need any information about the **View** and the other way round. This results in better decoupled software design which is essential for cross platform development.

Model Similar to the **Model** used with **MVC,** the **Model** of **MVP** represents all data that is used by the application.

View Similar to **MVC** the **View** is responsible for visualizing data and handling user interactions. Communication between **View** and **Presenter** is event based. The **View** passes events triggered from user interaction back to the **Presenter**, since it has no direct connection to the **Model** and communicates only with the **Presenter**.

Presenter In contrast to **MVC** the **View** does not get the information about updated data directly from the **Model**. Instead all dataflow passes the **Presenter** in both directions and can be manipulated before it is passed to the **View**.

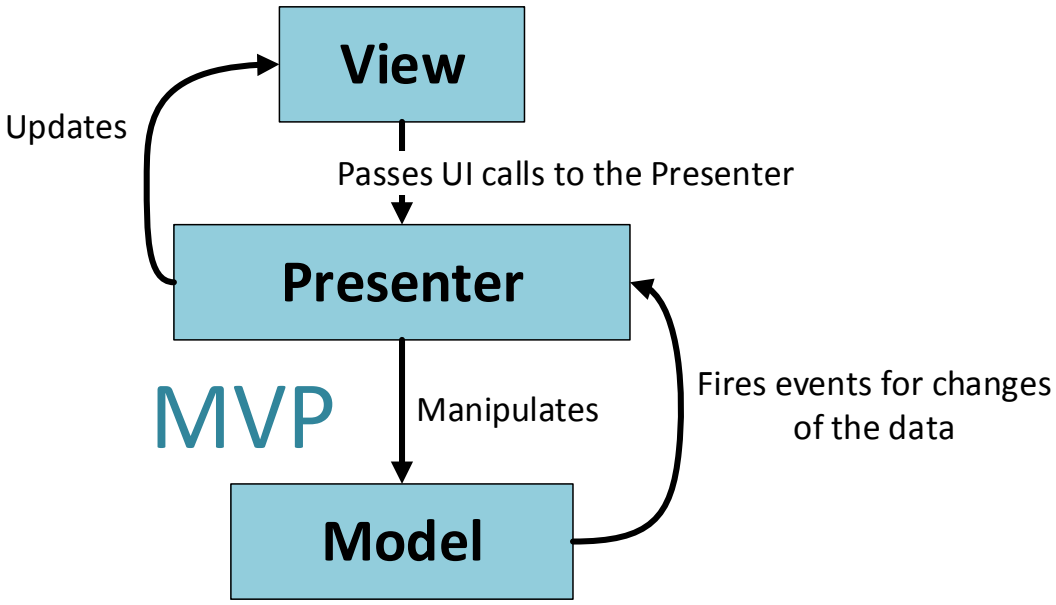

***Figure 8**: Simplified Dataflow diagram of MVP*

The following sequence diagram shows how the **View**, **Model** and **Presenter** work together to update the UI after the user clicks on a button.

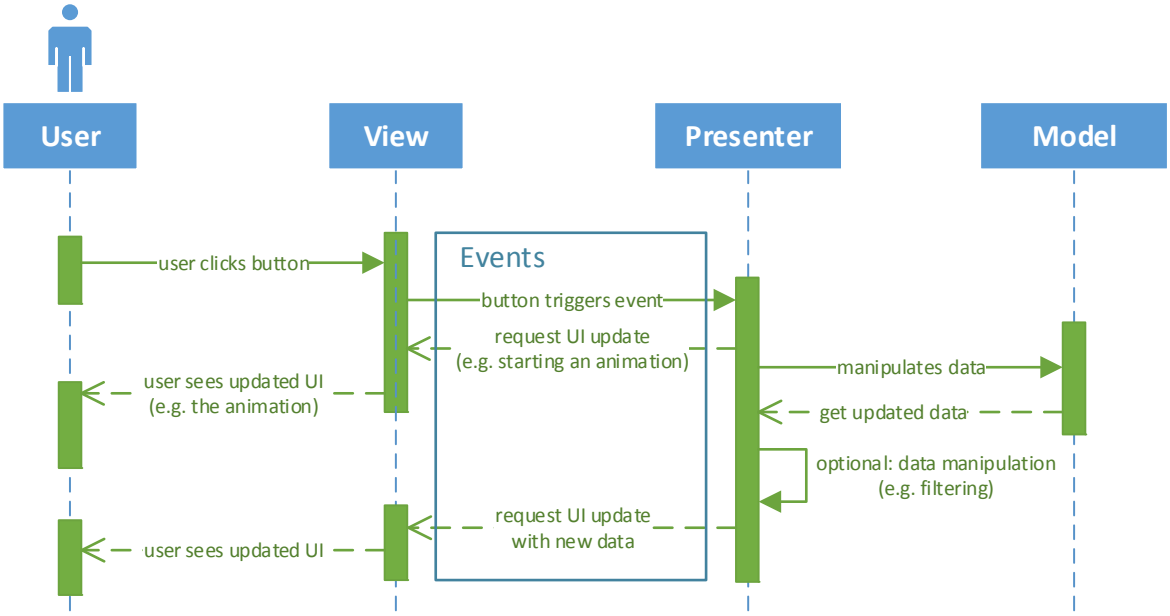

***Figure 9**: Sequence diagram of a simple user interaction using MVP*

2.1.3 Model View ViewModel

The **Model View ViewModel (MVVM)** is derivative of the **MVP** and acts in most aspects similar. The main difference is the introduction of a binding mechanism that is responsible for the communication between the **View** and the **ViewModel**. This binding mechanism can add additional flexibility and reduces the amount of written code. How this mechanism works will be described in detail later.

Model The **Model** of **MVVM** acts very similar to the **Model** used with **MVP**.

View Like in **MVC** and **MVP** the **View** is responsible for interaction with the user. It displays data from the **ViewModel** by using data bindings and notifies the **ViewModel** about user interactions also with the help of the binding mechanism. In addition to data bindings, there is a **command** mechanism, which replaces event based notifications to achieve better decoupling from **View** and **ViewModel**. With these mechanisms it is especially easy to switch **Views** on different platforms or for different display resolutions.

ViewModel The **ViewModel** is the central part of **MVVM**, as the **presenter** is for **MVP**. All information passes the **ViewModel** and can be manipulated by it. It implements an observable interface, so that its **View** can consume the data and state without the **ViewModel** knowing anything about the **View**.

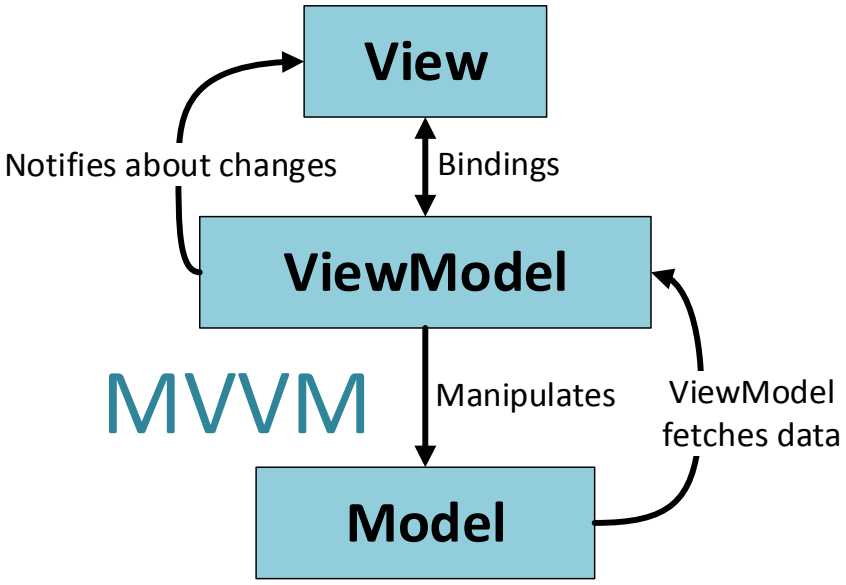

Figure 10: Simple dataflow diagram of MVVM

The following sequence diagram shows how the **View**, **Model** and **ViewModel** work together to update the UI after the user clicks on a button.

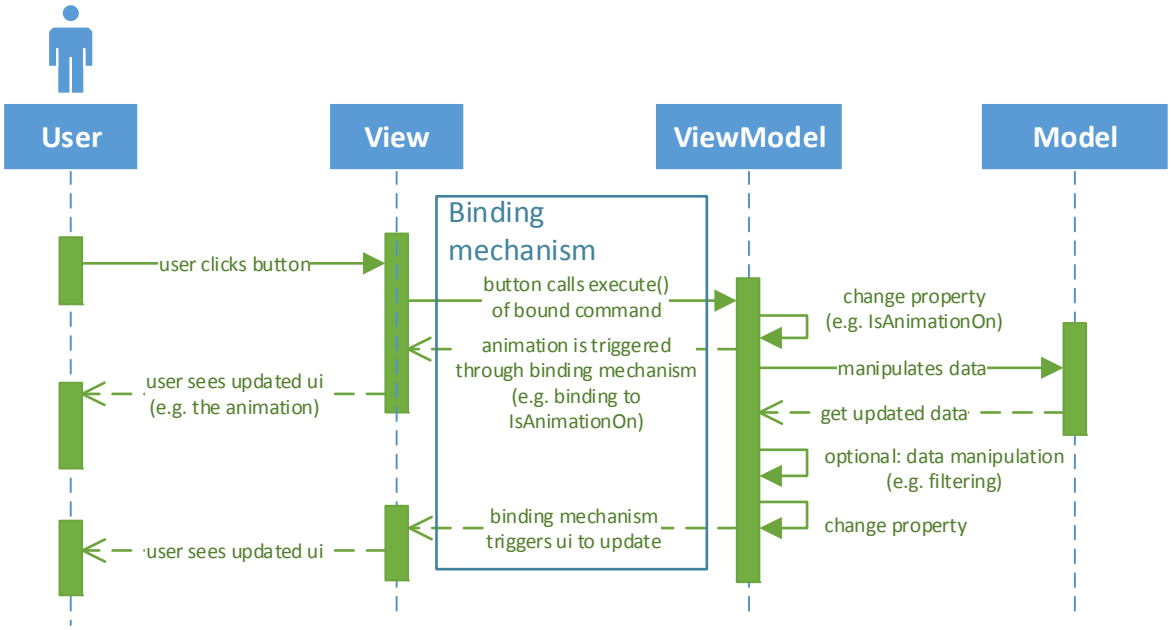

Figure 11: Sequence diagram of a simple user interaction using MVVM

2.2 MVVM theory

In this section we will learn how the **Model**, the **View** and the **ViewModel** work together and what their purposes and responsibilities are. In addition to the descriptions of **Model**, **View** and **ViewModel**, there will be descriptions of other patterns relevant for this work.

2.2.1 Model

The **Model** represents all the relevant data that an **MVVM** application uses. The source of this information can be a domain model or a data access layer that represents content.

The data can be managed with an internal data structure that is stored on the disc, the wrapper of a web service or any other source of data. Often it provides benefits to have different implementations of the **Model**. One common use case for more than one implementation is to create an additional implementation to display design time data. On platforms where developing tools with design time visualization are available or if the used **Model** contains distributed data that is not easily accessible at development time, a design time implementation of the **Model** may provide significant benefits for developers as well as designers. Design time data with the right tools can save designers a lot of time, since the often big applications do not have to get deployed to see visual changes in the UI design. On some platforms there are additional tools especially targeting designers, for these tools a design-time **Model** is necessary to provide content for the visualization inside the design software. Developers can also benefit from a design-time implementation of the **Model**. With its help the developer can find errors in data bindings and some parts of the **ViewModel's** logic without running the application.

2.2.2 View

The **View** is responsible for the visualization of its **ViewModel's** data. It is recommended that each **View** has exactly one **ViewModel**, but not necessarily the other way round. One of the many benefits of **MVVM** is the loose coupling of **View** and **ViewModel**. The **View** can be replaced without any changes in its **ViewModel**. Often there is the need for different implementations of the **View**, since the native components of graphical user interfaces are very different on each platform or not supposed to look the same for different cases of application. Different UI and user experiences (UX) for different screen sizes can easily be created by providing additional **Views**. Depending on the implementation of **MVVM** the different **Views** can be statically selected for each configuration or determined and switched at runtime. When cross developing applications for devices with a strong variation of screen sizes there is often the need to display multiple **Views** at once. To support this functionality **Views** can be nested within other **Views**. Another way of providing advanced functionality to show multiple **Views** in different ways on the screen is to introduce **presenters**. These **presenters** have nothing to do with the **presenter** of the **Model View Presenter (MVP)** pattern. **Presenters**, as described in this document, are responsible for abstracting the way that **Views** are placed on the screen. How **presenters** could look, like when used for real world applications will be described later in this document.

The Type of the View is usually a UI element like Page, Window, UserControl or a HTML div element. In some scenarios it is useful to introduce a derived class of the UI controls to add some common functionality often used in Views.

Sometimes additional manipulation of the data is required for visualization and the manipulation is not supposed to be made by the **ViewModel** for some reason. For example the conversion from a portable UI object to a native UI object, the inversion of a value or any other data manipulation. For these tasks most binding frameworks support **converters** or similar approaches.

2.2.3 ViewModel

The **ViewModel** is the central part of **MVVM** it is the only connection between **View** and **Model**. Each **ViewModel** has one or more **Views.** It is even possible, but not often used, to have multiple **Views** connected to one **ViewModel** and visible on the screen at the

same time. These **Views** will have exactly the same state without an additional synchronization mechanism between the two **Views**. The **Views** do not even know about the other's existence.

Furthermore, the **ViewModel** provides a subset of the **Model's** data to its **View(s)**. It provides **commands** and the associated **actions,** so that they can be used (bound) from the **View**. Based on the executed **command** the **ViewModel** makes changes to the **Model's** data. Each change of the **Model's** data will be immediately reflected back to the **View**, using the binding mechanism. Sometimes it is necessary to synchronize the state of two **ViewModels**. For this purpose a **messenger** can be used. How a **messenger** could look like and what benefits it can offer will be discussed later in this document.

2.2.4 Properties

Properties were introduced long time before **MVVM**, but due to the fact that **MVVM** can benefit a lot from **properties** they will be discussed here as well. **Properties** are the combination between the accessor (`get`) and mutator (`set`) methods. A **property** can be used like a field but internally the get and set methods will be called. This mechanism reduces the amount of code and make it better readable. A **property** can be fully accessible, read only or write only.

For the use with MVVM bind-able **properties** are used. Therefore, the ViewModel and most data objects have to implement an observable interface to listen to changes of **properties**. If the value of a **property** has changed, the implementation of the observable interface notifies all registered observers that one of its **properties** has changed. If the observer is interested in the changed **property** it can read the new value of the **property** by calling the `get` method of the **property**. It is recommended to have strict guidelines when to use a **property** or when to provide an explicit method. For example the `get` and `set` method of a **property** should not be used to run resource intensive or blocking code. The `get` method should also not be used to manipulate the object's state. That means a second call to the `get` method, executed immediately after the first one, should not deliver a different value.

2.2.5 Bindings

The **binding** framework is sometimes provided by the platform itself. It is the mechanism that connects the UI control's properties with the **properties** of the **ViewModel**. It is the implementation of an observable object on the data side combined with an automated mechanism to observe properties and make the corresponding changes on the UI side.

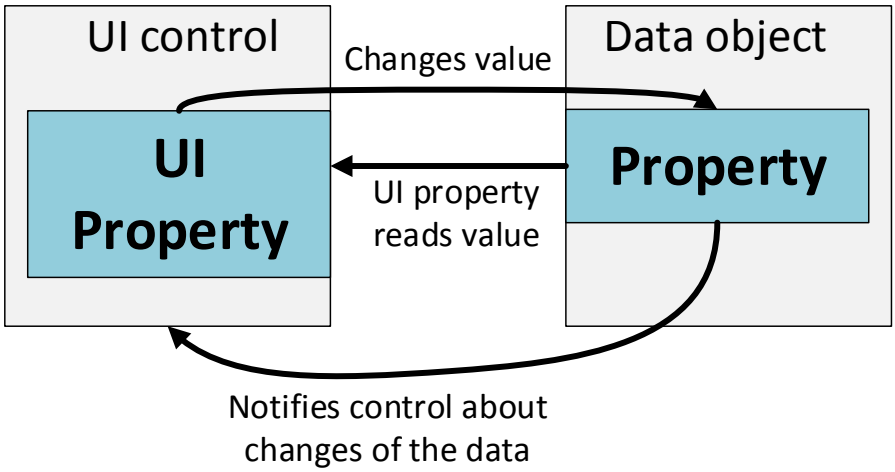

***Figure 12**: Diagram of data binding mechanism*

The above data flow diagram shows the principle of a data binding mechanism. How this mechanism works in detail looks different on most platforms and will be described for each platform individually later in the practical parts of this document.

2.2.6 ValueConverter

The **ValueConverter** can be a specific Interface with two methods `Convert` and `ConvertBack`. It is used among other things to perform platform dependent conversions or conversions that are not supposed to be included in the ViewModel. **Converters** can be reused multiple times. The goal of cross platform development is to share as much code as possible. In real world applications most **converters** can be reused across **Views** and platform.

2.2.7 Commands

Commands (for .NET platforms) are bindable **properties** that implement the `ICommand` interface. This interface has two important Methods:

`Execute(object parameter)`: The *Execute* method also known as **action** gets fired from the UI control that binds to the **command**. This mechanism is already implemented for most .NET based UI frameworks. For frameworks that do not provide this functionality it can often be easily added to existing controls.

The optional parameter can be used to provide additional context for the execution logic (**action**).

`CanExecute(object parameter)`: `CanExecute` can be used to let the UI control know about the state of the **command**. It returns a boolean value that indicates if the command can be executed or not. The UI control can use this information to change its UI's state. For example a Button can use this information to decide if it is enabled or not.

2.2.8 Actions

Actions are the execution logic of **commands**. They manipulate the **Model** and change the **ViewModel's** states.

2.2.9 Messaging

Most **MVVM** frameworks provide a **messaging** mechanism to synchronize **ViewModels** with each other. The **messenger** provides a mechanism to subscribe to a specific kind of message. It also provides functionality to publish new content. The **messenger** will then notify all registered listeners about the changed content. The **messaging** mechanism provides a loose coupled way to communicate within the application.

The following sequence diagram shows how the messaging mechanism can be used to synchronize changes between ViewModelA and ViewModelB.

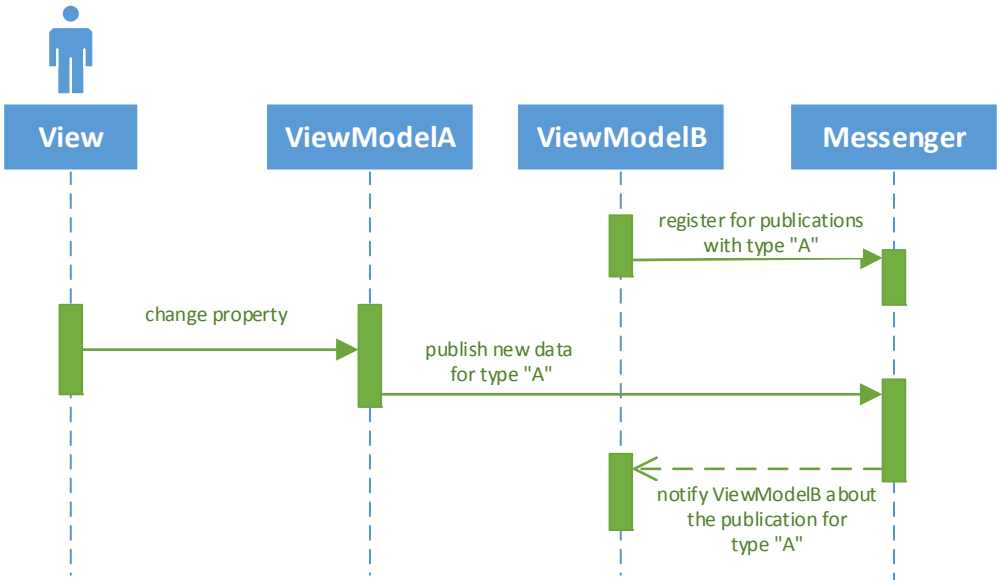

***Figure 13**: Sequence diagram of a simple messaging implementation*

2.2.10 Services

A **Services** is a construct to encapsulate individual parts of the application. They can be used to abstract platform dependent functionality. The platform specific APIs can then be used from code inside an independent (portable) part of the application that has otherwise no access to platform specific APIs.

A **service** consists of one interface and at least one implementation of that interface. Usually an inversion of control (IOC) system is used to register the **services** as singleton instances and let them be accessed from everywhere within the application. In addition

to that, the concept of constructor dependency injection can be used to get an better overview of what interfaces a component uses.

3 Cross platform development with web technologies

When looking for cross platform development technologies, web tools are a good place to start. Nearly any smartphone, tablet or desktop computer supports at least one browser to run web applications. Developing applications with common tools like PHP, Active Server Pages (ASP.NET) or Java Server Pages (JSP), where most of the logic for data manipulation and UI creation runs on the server side, often provides fast ways to create big applications. Server side web developing is still common for different cases of application. However, there are many situations, where running most of the code on the server side is not wanted.

This chapter will describe many aspects of client side web development and tools that help to increase the development experience. Each aspect will only be described briefly to give an overview of the different tools and libraries to choose from. In the following chapters some of the below described languages, tools and libraries will be used to create sample implementations.

3.1 Server side compared to client side code

Server side code

Web applications that use mostly server side code are developed with tools like PHP, ASP.NET, JSP and other technologies. With this technologies HTML code gets generated on the server side and the full HTML page will then be sent to the client's browser. To update the UI the server has to send the full HTML page or parts of it once again to the client. Every time the UI gets updated the full HTML markup is being sent to the client.

Client side code

When using client side code the application is loaded inside the client's browser or installed on the device by using application packages for the target platform. Once the application is loaded and the user interacts with the application, there will not necessarily be a communication with the server. The UI will be updated through client side code and only if remote data is needed, pure data, without HTML markup, will be sent from the server to the client.

The server is not necessarily required for the application to run. Therefore, the application can be used without any internet connection once it is loaded. This enables the application to run completely offline if required.

Comparison of server and client side code

The following lists will highlight the benefits of each solution.

Client side code	Server side code
The application can be used offline	Server side programming languages and libraries can be used.
Network traffic can be significantly reduced by sending only the pure information to the client without having the overhead in the form of HTML markup.	Most parts of the program's code will not be seen by the user.
Reduce server side central processing unit (CPU) load by processing most of the input data on client side.	Servers have lots of processing power in comparison with small mobile devices and can run resource intensive tasks faster. This can save battery life for mobile client devices.
Client side manipulation of the DOM is often required for server side applications as well, since responsive and fluent user interaction cannot be achieved with pure server-side code.	It is often easier to implement security features.
Applications can be deployed using individual app stores like Google's Play Store and others.	
When deployed as app for an app store the application is able to use native application programming interfaces (APIs) with the help of tools like PhoneGap.	
Benefits of great client side libraries like jQuery and others.	
Better scalable software architecture.	

***Table 1**: Comparison of client and server code*

There is no strict boundary between these two architectures and they can be combined to fit the application's needs best. The scope of this work will only cover client side code.

3.2 Development tools

Good developing tools are important and can help the developer to write better code in less time. Especially, for bigger development teams, integrated development environments (IDEs) have many features the team can benefit of. Modern IDEs have features like syntax highlighting, code completion, automated refactoring, integrated source control, and many more.

Three of the best tools to develop web applications will be described and compared in the following sections.

	Aptana Studio [5]	Webstorm [6]	Visual Studio [7]
License	Open source	Commercial	Commercial
Developing Platforms	Windows, Mac OS X, Linux, Eclipse plugin	Windows, Mac OS X, Linux	Windows
Language support (WEB)	HTML5, CSS3, JavaScript, CoffeeScript, Ruby, Rails, PHP, Python	HTML5, CSS3, JavaScript, TypeScript, CoffeeScript, Dart	HTML5, CSS3, JavaScript, TypeScript, CoffeeScript, ASP.NET
Source control integration	Subversion, Mercurial, Git, Perforce, CVS, TFS	Subversion, Mercurial, Git, Perforce, CVS, TFS	Subversion, Mercurial, Git, Perforce, TFS
Debugger	Integrated	Integrated	Integrated

***Table 2**: Comparison of web developing IDEs*

3.3 Programming languages and language extensions

When developing web applications there is currently no way to ignore JavaScript as a developing language since it is the only way to run code in a wide range of browsers. The dynamic typing of JavaScript provides flexibility for software developers to realize their own software designs. JavaScript is an object oriented programming language but has no support for classes or inheritance like the traditional object oriented programming languages C++, Java, C# and others do. Instead it provides a flexible prototype based programming paradigm. ECMAScript is the formal definition of the current version of JavaScript and gets continuously improved to adapt to the changing needs of software developers.

JavaScript is a basic but powerful scripting language and can be used to create great web applications. JavaScript's flexibility and programming paradigms can often lead to problems though. It can be challenging for new programmers especially, when creating larger applications. That is why it is important to introduce coding standards and additional programming paradigms to secure good maintainability. The MVVM pattern can help to further increase maintainability by providing strict rules about the architectural design of the application.

In addition to MVVM there are plenty of other patterns and frameworks that provide solutions to common problems occurring during the development of web applications using JavaScript.

Another solution to increase developing experience is the introduction of a replacement for JavaScript as the main programming language. Since most browsers can only run JavaScript, all replacement languages have to compile to JavaScript.

Apart from writing JavaScript code or any of its alternatives a big part of developing web applications is styling the application using cascading style sheets (CSS). Similar to JavaScript replacements there are libraries that extend or replace CSS and provide numerous improvements.

The following sections will state a short overview of some common JavaScript replacements and CSS extension languages. This comparison should help with the decision on which techniques to use.

3.3.1 CoffeeScript (JavaScript replacement)

CoffeeScript is a compact scripting language that compiles to JavaScript. Its syntax is inspired by Python and Ruby. CoffeeScript implements many features of those two languages. Although, the language is not as comprehensive as Dart or TypeScript it is a very compact language and allows to create powerful applications while writing little code. In addition to that, CoffeeScript has some nice features that make writing code easier and faster.

CoffeeScript is not very different from pure JavaScript despite of its syntax. The language was developed to fix some problems developers see in the current version of JavaScript.

The book "The Little Book on CoffeeScript" [8] delivers a solid introduction to CoffeeScript and is the main source for this section.

Syntax

The syntax of CoffeeScript often looks similar to its JavaScript counterpart, but it is not a superset of JavaScript. This means valid JavaScript code is not necessarily valid CoffeeScript code. One reason for this is that CoffeeScript does not use curly braces for scope definitions. Instead it uses whitespaces like Python does.

Typing and generics

CoffeeScript is a weak and dynamically typed language like pure JavaScript. Therefore, it has the benefits and performance of a dynamically typed language, but lacks the maintainability, as well as some of the possible support that integrated development environments (IDEs) can provide for statically typed languages.

Variables, functions and namespaces

Variables are defined similar to JavaScript. One of the biggest sources of hard to find JavaScript errors are accidentally created global variables by forgetting the `var` keyword. Therefore, CoffeeScript removes the JavaScript way of defining global variables and uses only the "window" object or an "export" object to define global variables.

Functions in CoffeeScript look a little different than their JavaScript counterparts. CoffeeScript uses the single arrow "->" or double arrow "=>" to define functions. The following single arrow function gets the parameters x and y and returns the arithmetic product of both parameters.

```
multiply = (x, y) -> x * y
```

The difference of single and double arrow functions is the context of the function which can be accessed with the `this` pointer. The function defined with the single arrow behaves like a conventional JavaScript function. Therefore, the scope is the executing object. The double arrow behaves like functions created with common object oriented languages like c++, Java and .NET. Here the context is always the local context where the function was defined.

In addition to the basic functionality functions can have default values and accept lists of arguments.

There is no common way do define namespaces in CoffeeScript, but like JavaScript, closures can be used to create individual solutions to support concepts similar to namespaces.

Classes, interfaces and inheritance

CoffeeScript has in contrast to JavaScript a concept for class definitions that supports inheritance as well.

```
class Animal
  constructor: (name) ->
    @name = name

class Cat extends Animal

cat = new Cat("Pinky")
```

Interfaces like the one known from Java and other object oriented languages are not included in the CoffeeScript language definition. Instead, class inheritance can be used to achieve the same functionality.

External libraries

CoffeeScript is able to handle existing JavaScript libraries. So for developers there are not many differences between including JavaScript libraries or native CoffeeScript libraries.

3.3.2 Dart (JavaScript replacement)

Dart is like CoffeeScript a replacement for the current version of JavaScript with the purpose to enhance the developing experience when creating web based applications. It is an open source language, which is manly developed by Google. Unlike CoffeeScript Dart feels more like a completely new language in contrast to a JavaScript improvement. Dart provides several improvements over JavaScript. Furthermore, it has its own APIs for many tasks a developer faces while creating web applications.

In contrast to JavaScript, which was originally designed to provide only simple client side DOM manipulations for existing multi page applications, Dart focuses on single page applications.

For developing MVVM applications there is an interesting Dart library called “AngularDart”. It is a binding framework similar to its JavaScript counterpart “AngularJs”, but written natively with Dart and therefore offers better integration.

Dart already contains many powerful libraries to manipulate the DOM, communicate with web services, as well as other common functionality that is used by most web applications. These libraries are well designed and easy to use. Since Dart is a completely new language, it does not have to deal with the often criticized legacy features of JavaScript.

The improvements of Dart compared to JavaScript are a major benefit, but all that comes for a cost. Like CoffeeScript, Dart compiles to JavaScript to run on a wide range of browsers. Therefore, every function Dart adds to the basic features of JavaScript has to be implemented by the developers of Dart. Furthermore, all Dart libraries have to be transferred to the client’s browser in some way. Dart differs much more from JavaScript than CoffeeScript differs from JavaScript therefore, errors occurring at run-time can be difficult to find.

This is a reason why Google has the idea of running Dart directly inside each browser. Having Dart implemented from the browser itself would make it unnecessary to send the whole Dart framework to the client. It should also result in a mayor improvement in performance due to the efficient design of Dart and the lost overhead from compiling it to JavaScript.

With that Dart could replace JavaScript as the only native programming language supported by browsers. However, this vision from Google is not likely to happen due to the incompatibility with the current version of JavaScript and the objections that other mayor web companies like Microsoft, Apple and Firefox are likely to have.

Syntax

The syntax of Dart looks similar to JavaScript and offers lots of useful additions.

Typing and generics

Dart is an optionally strong and dynamic typed language and supports most features a strong typed language has to offer. The usage of generics can improve the benefit of strong typed application development further.

Variables, functions and namespaces

Variables in Dart are defined as they are in JavaScript, but with the possible addition of specifying the variables' types.

To define a function Dart offers multiple syntaxes. There is a shortcut for single line functions using the "=>" operator that looks similar to the function definition with CoffeeScript. Multiline functions are defined with a syntax similar to JavaScript. In addition to multiline functions, Dart offers several shortcuts with typed and non-typed parameters and return values. [9]

Multiline function syntax	**Single-line function syntax**
`int power(int n) { return n * n; }`	`int power(int n) => n * n;`
`power(int n) { return n * n; }`	`power(int n) => n * n;`
`power(n) { return n * n; }`	`power(n) => n * n;`
`var power = (n) { return n * n; }`	`var power = (n) => n * n;`
`(n) { return n * n; }`	`(n) => n * n;`

***Table 3**: Syntax of function definitions with Dart*

Dart has no concept for namespaces, instead it has a library concept that can be used to separate code. Each library has its own namespace to prevent naming collisions.

Classes, interfaces and inheritance

Different from the prototype based approach of JavaScript, Dart offers solid concepts to support classes, interfaces and inheritance in a common object oriented way. The following code shows a simple inheritance scenario.

```
class Animal {
  string name;

  Animal (string name) {
    this.name = name;
```

```
}

class Cat extends Animal {
  Cat (string name): super(name);
}

void main() {
  Cat cat = new Cat("Pinky")
}
```

The following code shows the implementation of an interface with Dart using an abstract class.

```
abstract class I {

  f1();
  f2(String x);
}

class A implements I {
  f1() {…}
  f2(String x) {…}
}

void main() {
  A a = new A();
  a.f1();
  a.f2("s");
}
```

In addition to basic class usage, Dart provides support for named constructors that can make code easier to read.

External libraries

Dart offers good support for calling existing JavaScript code from an external library through the Dart library “js.dart”.

3.3.3 TypeScript (JavaScript extension)

TypeScript is an open source script language mainly developed by Microsoft. TypeScript is like CoffeeScript and Dart a language trying to improve the current version of JavaScript by introducing lots of useful concepts. Unlike CoffeeScript and Dart, TypeScript is a superset of JavaScript. This means that every valid line of code written in JavaScript is also valid TypeScript code. This fact makes not only switching from JavaScript to TypeScript very easy, but also lets you use existing JavaScript libraries with ease.

Like the name of the languages indicates, TypeScript is all about introducing static typing to JavaScript. This will improve readability and maintainability while providing similar performance as JavaScript.

One nice fact about TypeScript is that it implements most of the features of the unfinished ECMAScript 6 language definition for the coming version of JavaScript. Technically TypeScript is a snapshot of the possible future of JavaScript, which you can already use today, to develop great applications. TypeScript is implementing the ECMAScript definition carefully and introduces only minor differences. That is the reason why TypeScript is probably more future-prove than CoffeeScript or Dart. While writing your TypeScript application that compiles to the current version of JavaScript, the same application could

be natively supported by all browsers in a few years without the need for major changes inside the application.

Although the compatibility of JavaScript brings many benefits, it is also a weakness. One of the severe problems of writing JavaScript applications is the variety of ways you can write and structure the code. Languages like Dart force the developers to write their code in a more predefined way, so others can understand it more easily. Since TypeScript is compatible to the current version of JavaScript, it offers even more different ways to write an application. When using TypeScript this structure could be artificially enforced by development tools or advanced coding guidelines instead of the compiler. Another problem that results from the nature of TypeScript is the enforced support of legacy APIs of JavaScript that are criticized by many web developers.

An extensive overview of the TypeScript language can be found in the TypeScript language specification [10], which also builds the basis for following description of the TypeScript language.

Typing and generics

As mentioned above TypeScript is all about introducing strong and static typings to JavaScript by maintaining the possibility to use dynamic and weak typings. Therefore, TypeScript is an optionally strong and dynamic typed language. Using types in TypeScript looks a little different than it does in Dart or most other object oriented languages. This is a result of the required compatibility to the current JavaScript version. In addition to basic type features, TypeScript supports the usage of generic types.

Variables, functions and namespaces

Variables, function return values and parameters can be defined with or without type specification. This ensures the compatibility to current JavaScript and the benefits of writing strong typed code. Typed code provides lots of benefits like improved readability and maintainability. When compiled to JavaScript, all additional type information is removed for better performance.

The following code shows how to add type information to a variable definition and how to use generic objects.

```
var x: number;
var y = new Y<string>();
```

TypeScript functions are exactly the same as JavaScript functions. One of the mayor imperfections of JavaScript is the `this` pointer, which always points to the calling object and not like in C++, Java and .NET to the parent object, where the function was defined. To get this functionality with TypeScript, developers have to use some tricks already known from the current version of JavaScript or by using the following lambda expression. Lambda expressions always use the local context for the `this` pointer and not the calling object.

```
var a = () => { return this; };
```

Classes, interfaces and inheritance

As necessary for a full featured statically typed language TypeScript offers a feature rich implementation of classes, interfaces and inheritance.

The following implementation of the animal inheritance example looks like the following with TypeScript.

```
class Animal {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
}

class Cat extends Animal {
  constructor(name: string) {
      super(name);
  }
}

var cat : Cat = new Cat("Pinky");
```

The following code shows how to use interfaces with TypeScript.

```
interface I {
    f1();
    f2(x: string);
}

class A implements I {
    f1(){}
    f2(x: string) {}
}

var a = new A();
a.f1();
a.f2("s");
```

External libraries

Since TypeScript is a superset of JavaScript every existing JavaScript or TypeScript library can be used within a TypeScript application. To benefit from full IDE support additional typing information can be added to existing JavaScript libraries. This additional typing information is available for most common JavaScript libraries in the form of an external file, which is used only at development time and will not be deployed to the client's browser.

3.3.4 SASS and LESS (CSS extension)

Like CoffeeScript, Dart and TypeScript that are trying to replace or extend JavaScript there are also libraries trying to do the same with CSS.

Syntactically Awesome Style-Sheets (SASS) and Leaner CSS (LESS) are two similar languages that compile to CSS. They introduce variables, nestings, mixins, inheritance and many other useful concepts. These concepts can make CSS more maintainable, themeable and extendable.

Since it is not within the scope of this work, we will not describe the differences of these extensions. Detailed documentations of SASS [11] and LESS [12] can be found on the websites.

3.4 UI frameworks

To create good looking and competitive web applications there is often the need for one or more UI frameworks. Teams can decide to write their own frameworks, but with lots of different browser engines to support, this task can be quite resource intensive. There are currently countless good UI frameworks to choose from. Most of them offer similar functionality, but differ in the visualization of the controls. Some libraries like jQuery only provide basic functionality for DOM manipulation where other libraries can build on.

3.4.1 JQuery

The useful tools that jQuery [13] provides are an essential part in many modern web applications. JQuery's powerful methods for DOM traversal and manipulation, event handling, AJAX calls and all of its great extensions save lots of development time by resolving browser incompatibilities and offering advanced features that are easy to use.

DOM traversal and manipulation

JQuery features a rich API for traversing and manipulating the DOM. It provides an easy to use interface for many common use cases and allows great results by writing little code. An implementation of CSS selectors inside jQuery makes the traversal of the DOM powerful and flexible, by using a familiar syntax.

The following line of jQuery code selects the HTML element with the ID "MessageBox" that has an Attribute with the name "`data-field`" and the value "`HeaderContent`". The method sets the inner HTML of the selected element to "The Header".

```
$("#MessageBox [data-field=HeaderContent]").html("The Header");
```

Event handling

Event handling with pure JavaScript is not easy, especially when developing applications that target different browsers including older ones. JQuery provides a solid solution that makes event handling easy and comfortable.

The following code registers a function to the click event of a button. This function changes the visibility of the HTML element with the ID "Box" to "visible".

```
var box = $("#Box");
$("#Button").on("click", function (event) {
  box.show();
});
```

AJAX calls

Many modern web applications use AJAX calls to load data dynamically without forcing the browser to navigate to another page. This procedure is difficult to implement with pure JavaScript but jQuery offers many easy to use functions to help with down- and uploading data using AJAX.

The "`get`" method used in the code below fetches the HTML code contained in the remote file "`sample.data`" and calls a function with the loaded data after the AJAX call has finished.

```
$.get("sample.data", function (data) { … });
```

3.4.2 JQueryUI

JQueryUI [14] offers a wide collection of UI related interactions, effects, widgets, and themes. It is built upon jQuery hence, it is easy to use for developers with jQuery experience.

Interactions

Advanced user interactions are not always easy to create without the help of additional libraries. JQueryUI provides solid tools for the most common user interactions like dragging, dropping and resizing of UI elements. In addition to that jQueryUI provides tools for creating collections of HTML elements with the ability to let the user reorder or select individual items.

The code below makes the HTML element with the ID "`Draggable`" draggable. If the draggable element gets moved and dropped inside the element with the ID "`Droppable`" a JavaScript function will be called.

```
$(function () {
    $("#Draggable").draggable();
    $("#Droppable").droppable({
        drop: function (event, ui) { ... }
    });
});
```

Effects

Effects are a big part of jQueryUI. There are plenty of different animations for different use cases like showing, hiding and moving HTML elements.

The following code animates the width of the DOM element with the ID "`EffectContainer`". The animation changes the current value of "`width`" via linear interpolation from the current value to 500 and will take 1000 milliseconds to complete.

```
$("#EffectContainer").animate({width: 500}, 1000);
```

Widgets

The widgets included in jQueryUI are built on top of all other features. JQueryUI offers these controls to show the user a well-designed UI with animations to make user interaction look responsive and seamless.

The following code turns the input element with the id "`Button`" into a jQueryUI themed button.

```
$("#Button").button();
```

Themes

The theming system is a great way to offer the user a choice of different designs with little development effort to create them. JQueryUI offers an online tool with the name "ThemeRoller" [15] that provides an easy and interactive way to create themes for jQueryUI controls.

3.4.3 JQueryMobile

JQueryMobile [16] is a UI framework similar to JQueryUI without the basic functions for interaction and effects. Since jQueryMobile is built like jQueryUI on top of jQuery those

two frameworks can be used together with little complications. The main difference between jQueryUI and jQueryMobile is that jQueryMobile especially, targets mobile platforms with touch input. With jQueryMobile the developers can create good looking applications that target smartphones, tablets and desktop devices.

JQueryMobile provides a wide range of useful widgets to create rich mobile applications. It also offers tools for virtual page navigation, which enables back and forward navigation in applications where all code is nested within one single html document. Most modern applications, especially when used offline, need such a mechanism.

As with jQueryUI the tool "ThemeRoller" can be used to create multiple themes for jQueryMobile controls with little effort.

The following examples show how easy it is to create rich mobile UIs with jQueryMobile without writing any JavaScript code.

The Page control that is used for in app navigation

```
<div data-role="page">
  <div data-role="header">header content</div>
  <div role="main" class="ui-content">main content</div>
  <div data-role="footer">footer content</div>
</div>
```

A Touch friendly themed button

```
<a class="ui-btn ui-corner-all" href="#">Button</a>
```

Slider control to select a numeric value between 0 and 10

```
<label for="slider">Slider</label>
<input name="slider" id="slider" type="range" min="0" max="10" value="3">
```

3.4.4 WinJs

Microsoft introduced native JavaScript development to create fast and fluent Windows Store apps, with the release of Windows 8. The development with JavaScript is similar to its .NET and C++ alternatives. The reason for the introduction of those tools was, among other things, to attract the many web developers to write apps for the new Windows 8 platform. This step of Microsoft was also a big win for cross platform developers that want to share lots of their code between multiple platforms without sacrificing native look and feel. For users it is almost not possible to distinguish weather the application was developed with web tools, .NET or C++.

WinJs is available as an open source library that should work within all browsers and within web based application environments like PhoneGap and others.

Microsoft provides all UI controls known from their Extensible Application Markup Language (XAML) markup language with WinJs to create great applications. The following examples will show how to define Windows Store controls using HTML.

Defining a button themed as the back button commonly used inside modern Windows applications.

```
<button data-win-control="WinJS.UI.BackButton"/>
```

Windows Store applications written with HTML, CSS and JavaScript can use Templates with similar usage as when written in XAML and .NET.

```
<div class="itemtemplate" data-win-control="WinJS.Binding.Template">
    <div class="item">
        <img class="item-image" src="#"
             data-win-bind="src: backgroundImage; alt: title" />
    </div>
</div>
```

This template can then be used to create Lists of items where each item uses the same template.

```
<div class="itemlist win-selectionstylefilled"
     aria-label="List of this group's items"
     data-win-control="WinJS.UI.ListView" data-win-options="{
        itemTemplate: select('.itemtemplate'),
        ...
    }">
</div>
```

Microsoft introduce their own binding framework to WinJs to provide all the benefits developers had when using XAML. The binding framework will be explained in detail later on. MVVM, with all its power, is fully supported without the need to use any additional library. However, if for some reasons alternative binding Frameworks are required, it is easily possible to combine WinJs with other frameworks like KnockoutJs or AngularJs.

WinJs was at its introduction closed source and only used with Microsoft's web engine. During the time this work was composed it was made open source. That is why it does not jet provide the same good experience on all browsers. Microsoft has published a roadmap for WinJs that promises better compatibility for a wide range of web engines. In addition to that the support for theming and other new features was promised [17].

The following list includes all the major controls and features that WinJs has to offer [18]:

- ListView
- Semantic Zoom
- FlipView
- Animations
- AppBar
- NavBar
- Flyout
- Ratings
- Tooltip
- DatePicker
- TimePicker
- Hub
- Pivot
- Repeater
- Searchbox
- Toggle
- Promises
- Scheduler
- XHR
- Binding
- Binding
- Templates
- Fragments
- Page control
- Navigation

3.4.5 Other Platform specific UI frameworks (Android and iOS)

For most major operating systems like Android and iOS there is no out of the box support for the development of native looking applications with web tools. This could be achieved

by using specialized UI libraries that are trying to imitate native look and feel. For example, the company Telerik [19] offers native looking UI controls for multiple platforms like Android, iOS, Windows and others.

3.5 Binding Frameworks

Data bindings are an essential part of MVVM, without them the creation of the MVVM architecture would not be possible. Web based technologies (HTML, CSS and JavaScript) do not have a built-in concept for the support of data bindings.

However, there are lots of different binding frameworks to choose from. Some of the most common binding frameworks are listed below. Three frameworks that seem most suitable for our sample application will be described in detail. For the sample application created with web technologies which will be discussed later, the framework **KnockoutJs** was used.

- **AngularJs** [20]
- **KnockoutJs** [21]
- Durandal, a combination of jQuery and KnockoutJs [22]
- **WinJs** [23]
- BackboneJs [24]
- Derby [25]
- Ember [26]
- JsViews [27]
- jQXB Expression Binder [28]
- Meteor [29]

3.5.1 AngularJs

AngularJs [20] is a full featured open source binding framework but has also some additional tools for the creation of single page web applications. It provides a solid structure for client side web applications with the help of a good software design architecture. It officially calls the used design architecture MVC but due to the fact that the communication between view and controller is handled by the binding framework it could also be called MVVM.

AngularJs is mostly developed by Google and greatly influenced by the experience and vast knowledge of the developers working on Google's successful and modern web applications.

For Dart developers there is a similar project called AngularDart which provides the same binding and structure feature for Dart applications [30].

Bindable data objects

AngularJs does not require to add code to data objects like many other binding frameworks do. Instead all existing data classes can be used without any change. This is especially helpful when receiving AJAX data from the server. For this scenario most other binding frameworks have to dynamically add code to make the received data objects bindable.

Binding syntax

AngularJs provides a simple and yet powerful syntax to specify bindings inside HTML. AngularJs adds several additions to the HTML syntax to make bindings as easy as possible.

The most important addition to the HTML syntax are probably the double curly braces "{{ … }}" that are used to set up the binding.

The following code inside the double curly braces sets the binding of the HTML content for the `span` element. It uses the text inside the "`name`" variable from the controller, which is responsible for this section of the HTML.

```
<span>{{name}}</span>
```

Controllers form the Model View Controller (MCV) pattern are used to specify the context of HTML elements. The "`ng-controller`" property can specify which controller is responsible for a hypertext markup langiuage (HTLM) element and its descendants.

The following code sets the context of the HTML body element and all its descendants (unless otherwise overruled) to the controller with the name "`SomeController`".

```
<body ng-controller='SomeController'>
```

The property "`ng-app`" is used to tell AngularJs which part of the DOM it is responsible for. This can be used to embed an AngularJs application inside an existing non AngularJs web application. The following code tells AngularJs that it should manage the bindings for the whole HTML page.

```
<html ng-app>
```

Lists and templates

The template mechanism from AngularJs can easily be used to visualize lists of objects. To visualize multiple objects inside a collection variable the template is copied for each item of the collection and then added to the DOM.

The following code shows how to use lists and templates to visualize multiple objects. The `span` element inside the `div` will be copied and inserted as a new child of the `div` once for each item of the `items` collection. The `name` property of each item object will be used to provide the inner content for the `span` element.

```
<div ng-repeat='item in items'>
  <span>{{item.name}}</span>
</div>
```

Advanced bindings and Converters

Converts are an important part of MVVM. AngularJs does have a Converter implementation similar to the one we learned before in the theory chapter. AngularJs is calling it Filters instead of Converters, but their behavior is nearly identical.

AngularJs includes lots of bundled filters to help developers write their code faster. New filters can be easily added by writing JavaScript code.

The following code shows how to use the Filter `numberSquare` to manipulate the binding's content by calculating the squared value before visualizing the manipulated value. Filters can also work the other way around when used with two-way bindings for UI elements that support user input.

```
<span>{{numer | numberSquare}}</span>
```

In addition to filters AngularJs enables in line calculations like the following.

```
<span>{{numberA * numberB}}</span>
```

Events and Commands

Events can be used with AngularJs like the following.

```
<button ng-click="action()">Action</button>
```

In AngularJs, like with most web based binding frameworks, there is no command pattern, as it is described in the theory section. Instead the actions are directly bound and the “is enabled” mechanism from the command can be implemented with an additional binding.

Additional features

AngularJs is more than just a binding framework. It adds lots of convenient functionality to manage and structure the code.

3.5.2 KnockoutJs

KnockoutJs is a popular and well known open source binding framework. It provides binding functionalities similar to AngularJs. The main difference between KnockoutJs and AngularJs is that KnockoutJs does not provide its own application design pattern. This is not necessarily a bad thing, because the lack of an included application design structure makes it more flexible and compatible to other libraries and patterns like MVC, MVP or MVVM.

Bindable data objects

KnockoutJs uses a different concept than AngularJs for binding data objects to the UI. We have seen that AngularJs does not require the manipulation of data objects to support bindings. KnockoutJs on the other hand requires additional code to make data objects bindable. Data variables can have bindable and non bindable properties. Non bindable members look like any JavaScript member but bindable ones have to be of an observable type.

The observable types supported by KnockoutJs are:

`ko.observable`

This type is used for single observable objects like numbers, string or user defined objects including other observables.

```
var observableString = ko.observable('value');
var observableNumber = ko.observable(42);
```

`ko.observableArray`

When working with collections of data objects the observable-array type is used.

```
var someObservableArray = ko.observableArray();
someObservableArray.push('value');
```

`ko.computed`

This type is used for computed observables. With it a method can be used to generate or manipulate data each time the UI asks for a new value. This is similar to the Filter approach used by AngularJs.
In the code below, the UI will be notified of every change of the used observables A and B and gets a new calculated value.

```
var observableComputed = ko.computed(function () {
  return this.A() * this.B();
}, this);
```

Binding syntax

On the HTML side there are a few additional properties that enables bindings support.

The most important addition is the `data-bind` property. It defines all the bindings with its own syntax inside the value of the HTML property. Many different binding scenarios are available with the included bindable HTML and CSS properties like `visible`, `text`, `css`, `style`, `checked`, `enabled`, and others. For binding types that are not natively supported by KnockoutJs, there is a mechanism to write custom binding.

The following code shows a simple data binding. The visibility of the `div` element is bound to the `isVisible` property of the ViewModel, Controller or Presenter.

```
<div data-bind="visible: isVisible">
```

To specify the binding context of DOM elements, KnockoutJs uses the `with` keyword inside the `data-bind` property. In the sample below the context of the `div` and all its children will be the `contextObject`.

```
<div data-bind="with: contextObject">
```

Lists and templates

KnockoutJs has a template system that works similar to the one we have seen of AngularJs. For collections of data objects there is a mechanism that uses a template defined with HTML. That template HTML code will be copied and inserted into the DOM for each item inside the `ko.observableArray`. The binding context of each copied template is the corresponding data object itself.

The following HTML code shows how to use the `foreach` binding mechanism of KnockoutJs. In the case below the `ul` element specifies, the content of the UI template for a collection of items. The source variable that includings all items is `items`. The `li` will be copied `items.count` times and placed inside the `ul` element. The property `a` of each item inside the collection is shown as inner HTML for each `span` element.

```
<ul data-bind="foreach: items">
    <li>
        <span data-bind="text: a"/>
    </li>
</ul>
```

Advanced bindings and Converters

KnockoutJs supports inline calculations similar to AngularJs.

```
<span data-bind="text: a * b"/>
```

For more complex calculations it is recommended to use computed observables as described in the beginning of this section.

Events and Commands

Binding commands or actions works like binding any other property of the data context. The `click` binding used below, as well as others, can be declared within the `data-bind` property as well.

```
<button data-bind="click: action">Action</button>
```

Additional features

KnockoutJs is a pure binding framework and does not provide application design and lifecycle management tools like AngularJs.

When using data objects originating from a remote server, there are mostly no observable objects used. Therefore, the "Mapping" plugin [31] for KnockoutJs automatically converts pure JavaScript variables and arrays to KnockoutJs observables.

3.5.3 DurandalJs

DurandalJs is a nice framework that combines jQuery, KnockoutJs and RequireJs. Its tools can be used to create MVC, MVP and MVVM architectures. The following list of features originates from the official DurandalJs website [22] and provides a good explanation of the benefits DurandalJs offers.

- Clean MV* Architecture
- JS & HTML Modularity
- Simple App Lifecycle
- Eventing, Modals, Message Boxes, etc.
- Navigation & Screen State Management
- Consistent Async Programming w/ Promises
- App Bundling and Optimization
- Use any Backend Technology
- Built on top of jQuery, Knockout & RequireJS
- Integrates with popular CSS libraries such as Bootstrap and Foundation
- Make Your Own Templatable and Bindable Widgets
- Fully Testable

3.5.4 WinJS

WinJs uses JavaScript code, in addition to other things, to provide a mechanism to bind properties of UI controls to properties of data objects. WinJs specifies the bindings not with HTML like AngularJs or KnockoutJs do, instead it uses JavaScript code. This often results in more code that is worse structured. However, WinJs was designed to work well with other frameworks. Frameworks like AngularJs and KnockoutJs can be easily used in combination with WinJs. How the binding mechanism of WinJs works can be seen on the TryWinJs website [18] that offers lots of interactive code samples for most features that WinJs offers.

3.6 Technologies to create and deploy mobile apps

Not so many years ago there was only one possible deployment scenario of web based applications, hosting it on a web server. With the raise of mobile platforms many different scenarios for the usage of web applications have evolved. There are application platforms that only accept web applications like ChromeOS [32] and FirefoxOS [33]. Other platforms like Windows and Windows Phone applications support three different native development platforms with one being web based.

Other platforms like Android [34] and iOS have no built in web development support, but they all have one special native UI control. This control is similar to a conventional browser and can be used to host web applications. Therefore the creation of web based applications is theoretically possible on most mobile platforms. Finished web applications can be downloaded from the platform's application store and behave mostly like native

apps. The only limitation is the lack of native UI controls which often results in applications without native look and feel. More complex frameworks add a custom API wrapper to let developers use the native controls within their web applications.

Some of the most interesting deployment scenarios and frameworks will be described and compared in the following sections.

3.6.1 PhoneGap

PhoneGap [35] is an open source project supported by Adobe and successfully used by many developers worldwide. It is a HTML, CSS and JavaScript based cross operating system mobile development platform. PhoneGap provides libraries and tools to create application packages for the following platforms.

- Amazon Fire OS [36]
- Android [34]
- BlackBerry 10 [37]
- Firefox OS [33]
- iOS [38]
- Ubuntu [39]
- Windows Phone [40]
- Windows 8 [23]
- Tizen [41]

In addition to the application packaging, PhoneGap provides API wrappers for most of the devices' native features that are not available with the standard JavaScript APIs, like accessing the camera, reading and writing on shared storage, using the phone's payment abilities and others. PhoneGap tries to unify the APIs by combining similar APIs into one PhoneGap API, to help developers to cross develop their applications.

Unlike other solutions PhoneGap does not provide APIs for native UI controls. All UI inside a PhoneGap application is written with pure HTML. This makes it easy to port existing web applications to PhoneGap, but the applications may not feel as native and well performing as they could, when developed with the native tools of each platform.

Building PhoneGap applications require the development tools for the targeted platforms installed on the development device. Installing all platform development tools can be a lot of work and does not work with every operating system. Therefore, Adobe provides the PhoneGap Build service [42], which let developers build their applications on Adobe's servers instead of local development devices. Furthermore, this allows the creation of application packages for multiple platforms within minutes.

Applications developed with PhoneGap have the same performance limitations as other applications developed for mobile browsers. For most applications this becomes less and less important though, due to the rapid increase of performance of mobile hardware and the continuous improvements of mobile web engines.

3.6.2 Titanium

Titanium [43] is a cross platform open source web development solution that has its focus on creating performance optimized and native looking applications. Unlike PhoneGap, Titanium does not use HTML and CSS to create the UI of the application, instead it provides APIs to use native UI controls from the target platform by writing JavaScript code.

Due to the more complex and resource intensive nature of Titanium's internal design, Titanium does not support as many platforms as PhoneGap does. At the time of this work, Titanium supported the following platforms.

- Android [34]
- iOS [38]
- Blackberry 10 [37]
- Windows Phone [40]
- Windows 8 [23]

To make development with Titanium easier and more comfortable Appcelerator, the company behind Titanium, provides an IDE that is based on Eclipse.

Due to the usage of the platforms native controls Titanium applications offer good user experience and potentially better performance than PhoneGap applications. However, they are mostly not as well performing as purely native developed applications.

4 Cross platform development with .NET

When searching for cross platform development tools .NET is most likely not the first choice to look at, since .NET is developed by Microsoft and has the reputation of being closed source with a proprietary license. However, this is mostly untrue. Many parts of .NET are open source and can be used for open source projects as well as commercial projects under the Apache License 2.0. In 2014 Microsoft made another big step to make even more Parts of .NET open source, including the new compiler platform Roslyn, the Entity Framework and many other parts of .NET. A list of all open source projects that are managed by the .NET foundation can be found on the official .NET Foundation website [44]. Xamarin, a company focused on .NET development tools for non-Microsoft platforms provides a solid cross platform development experience for .NET developers. The combination of Microsoft's own platforms and the Xamarin platforms cover a wide range of mobile and desktop devices.

Microsoft: Windows Phone, Windows desktop, Windows Store, Silverlight for web, ASP.NET web applications, and xBox games and applications.

Xamarin: Android, iOS and Mac OS

With this set of supported platforms .NET development is easily possible for a large part of mobile and desktop devices sold today. When looking at the statistics from the motivation section of this document we can see that .NET covers about 95% of all mobile and desktop devices, which is not as much as the more than 99% that web platforms can cover, but is a more than decent ammount.

This means, if an individual developer or a company needs to support multiple platforms with one of the platforms has no support for .NET, they either not use .NET at all or use a cross platform .NET solution in addition to web or native applications. Often it is easier and faster to find websites than it is to install an application. Therefore, it is beneficial to create a web application in addition to .NET applications even if all required platforms are supported by .NET.

4.1 Development tools

For developing .NET applications good development tools are even more important than they are with web development. A good IDE can make the development process much easier and faster. Unlike for web development there are not many different IDE's to choose from. For cross platform development with .NET there are only two IDE's available. The first one is Microsoft's Visual Studio, which was already described in the web section. The other one is Xamarin Studio.

To support all possible platforms where .NET can be used, it is necessary to either work with both IDE's or use only Visual Studio, since each of the platforms are supported by Visual Studio in combination with the Xamarin plugins, yet not all platforms are supported by Xamarin Studio.

A short comparison of Visual Studio and Xamarin Studio can be found in the following table.

	Visual Studio	Xamarin Studio
License	Commercial	Commercial
Developing platforms	Windows	Windows, Mac
Supported target platforms	Windows Desktop, Windows Store, Windows Phone, Silverlight, Xbox, ASP.NET, Android[1], iOS[1], Mac[1]	Android, iOS, Mac, ASP.NET
Source control integration	Subversion, Mercurial, Git, Perforce, TFS	Subversion, Mercurial, Git, TFS
Debugger	Integrated	Integrated

***Table 4**: Comparison of .NET IDEs*

1) Android, iOS and Mac platforms need the commercial plugin from Xamarin for use inside Visual Studio

4.2 Programming languages and the .NET framework

.NET is a framework that supports three different languages C#, VB.NET and F#. In addition to them platform specific markup languages like XAML or Android XML are used for graphical user interface (GUI) declarations. For this document we focus on C# as well as the two markup languages XAML and Android XML.

In this section we will discuss some language feature of C#, as well as possible scenarios to use them. All of the following examples are in some way relevant for MVVM development.

4.2.1 Language syntax

The syntax of C# looks familiar to C++ and Java with lots of nice additions like for example LINQ, which is included in the language itself.

4.2.2 Classes, interfaces and inheritance

With C# the developers have full featured support of classes, interfaces and inheritance as known from other object oriented languages. The following example shows how to implement a simple animal inheritance with C#. The sample uses the base class `Animal` to provide common functionality for all animal classes like the `Cat` class below.

```
public class Animal
{
    public string Name { get; set; }

    public Animal(string name) {
        this.Name = name;
    }
}

public class Cat : Animal
{
    Cat(string name) : base(name) {
    }
}

Cat cat = new Cat("Pinky");
```

The same example implemented with the use of an interface instead of the `Animal` base class could look as follows.

```
public interface Animal
{
    string Name { get; set; }
}

public class Cat : Animal
{
    public Cat(string name)
    {
        Name = name;
    }

    public string Name { get; set; }
}

Cat cat = new Cat("Pinky");
```

4.2.3 Lambda expressions

Lambda expressions [45] are a short way of writing anonymous functions by using the following syntax.

```
(input parameters) => expression
```

The following list includes some examples of lambda expressions as well their explanations.

`(int x, int y) => x*y`	Simple lambda expression that returns a function to multiply two integers.
`(x, y) => x * y`	The types of the parameters are not required in many cases where the compiler is able to determine them by looking at the functions context.

`a => {`
`  int b = a + 1;`
`  Console.WriteLine(b);`
`}`
	The curly brackets are used for multiline functions.

`() => variable`
or
`() => Property`
	A Lambda expression that always returns a specific variable or property.

***Table 5**: Examples for lambda expressions*

4.2.4 Properties

Properties are an interesting construction that are used to replace the `get` and `set` methods by combining them into one concept. This makes it possible for the data binding components to know which get and set methods belong together.

Properties in C# can have the accessibility options, public, protected, internal and private. Additionally, the property can be set to read-only or write-only by removing either the get or set method.

The code below shows a simple property implementation.

The property reads and writes the `value` to and from a `private` variable, which is also called “backfield” variable.

```
private string _property;
public string Property {
    get { return _property; }
    set { _property = value; }
}
```

Properties can be used on the left or right side of an assignment.

When used on the right side of an assignment, the `get` method is called and its return value is assigned to the statement of the left side of the assignment.

```
var value = object1.Property;
```

When used on the left side of an assignment, the `set` method will be called and the `value` variable will be the result of the right side of the assignment.

```
object1.Property = value;
```

When both the `get` and `set` methods have the default functionality as shown above, the definition of a property can be abbreviated.

```
public string Property { get; set; }
```

When using data bindings with C# it is often required to implement the `INotifyPropertyChanged` interface. This interface makes it possible to subscribe to the `PropertyChanged` event to get notified when a property has changed. The notification can be done with the helper method `RaisePropertyChanged`. In the following example the `RaisePropertyChanged` method is called on the end of the property’s `set` method.

```
public class BindableClass : INotifyPropertyChanged
{
    private string _property;
    public string Property
    {
        get { return _property; }
        set {
            _property = value;
            RaisePropertyChanged("Property");
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
    private void RaisePropertyChanged(string name) {
        if(PropertyChanged != null)
            PropertyChanged(this,
                new PropertyChangedEventArgs(name));
    }
}
```

This is a convenient way to implement a bindable class but it has one major disadvantage. Since the name of the property is specified as a string value it is not checked by the compiler if a property with this name even exists. This results in a possible source of hard to find typing errors. Another problem is that the IDE's refactoring functionality, which can automatically rename the property's definition and all of its usages, will not rename the string value and therefore, introduce errors to the code.

To resolve those issues the following methods can be used instead of the above `RaisePropertyChanged` implementation.

```
public void RaisePropertyChanged<T>(Expression<Func<T>> selector) {
    if (PropertyChanged != null) {
        PropertyChanged(this, new PropertyChangedEventArgs(
          PropertyHelper.GetPropertyName(selector)));
    }
}

public static string GetPropertyName<T>(Expression<Func<T>> property){
    var lambda = (LambdaExpression)property;
    MemberExpression memberExpression;

    var body = lambda.Body as UnaryExpression;
    if (body != null) {
        var unaryExpression = body;
        memberExpression = (MemberExpression)unaryExpression.Operand;
    }
    else {
        memberExpression = (MemberExpression)lambda.Body;
    }

    return memberExpression.Member.Name;
}
```

Instead of specifying the string value this implementation of RaisePropertyChanged is called with a lambda expression that has no input parameters and a method that returns the property as its function body. The `GetPropertyName` method extracts the name of the property from the lambda expression and returns it as a string value.

```
RaisePropertyChanged(() => Property);
```

Now the compiler and the refactoring tools can associate the property with its usage as the parameter for the `RaisePropertyChanged` method.

4.2.5 LINQ (Language Integrated Query)

LINQ is an included features in the .NET language itself and provides an SQL like syntax for writing queries. These queries have the great advantage that they can be checked by the compiler in contrast to queries that are specified as string values. LINQ providers are used to interpret and translate the LINQ expressions to interact with the underlying data source.

The .NET framework by itself already includes some LINQ providers. Furthermore, new providers can be created by implementing a set of LINQ interfaces. Four included link providers are listed in the following table.

LINQ to Objects	LINQ to Objects is the included LINQ provider for in-memory collections. It can be used for collections that implement the `IEnumerable<T>` interface.
LINQ to XML	LINQ to XML provides the LINQ functionality for a collection of objects with the type `XElement`. With the `XDocument` class that is included in the .NET framework a tree shaped collection of `XElements` can be created easily.
LINQ to SQL	LINQ to SQL is the LINQ provider for the Microsoft SQL server connection. It translates LINQ expressions internally into SQL statements and returns the results of the query as .NET objects.
LINQ to DataSets	LINQ to DataSets can work with ADO.NET databases and works with the Microsoft SQL servers, as well as with other databases.
LINQ to *	In addition to the four built-in LINQ providers, many third-party providers can be used to connect LINQ with different data sources.

***Table 6**: List of LINQ providers*

The following code shows a simple examples of how LINQ to Objects can be used on a collection of animals. The `names` variable will be a collection of the names of all female animals that are less than ten years old.

```
class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }
    public bool Male { get; set; }
}

var animals = new List<Animal>{...};
var names = from a in animals
            where a.Age < 10 && a.Male == false
            select a.Name;
```

In addition to this simple use case, LINQ offers lots of SQL like functionality, as for example ordering, filtering, merging and grouping objects.

4.2.6 Async and await

The concept of async and await was introduced to the .NET platform not long ago and got very well accepted by the .NET community by now. The concept is an alternative to conventional asynchronous calls, which use a call-back function to return the result of the application. These conventional asynchronous calls often create lots of code that is difficult to read. The async and await language feature of the .NET framework can simplify asynchronous calls and reduce the lines of code.

The following code shows how to make use of the async and await concept to simplify asynchronous calls within .NET applications. Instead of two methods and one additional delegate the async and await implementation uses only one method that can be used like a synchronous method and is therefore easy to use and debug.

Asynchronous method with callback

```
public delegate void AsyncCallback(Result r);

void Main(){
    Task.Run(()=>AsyncCall(Callback));
}

void Callback(Result r) {
    Use(r);
}

public void AsyncCall(AsyncCallback callback) {
    var r = ResourceIntensiveTask();
    callback(r);
}
```

Async and Await

```
async void Main() {
    var r = await AsyncCall();
    Use(r);
}

public async Task<Result> AsyncCall() {
    return await ResourceIntensiveTask();
}
```

4.2.7 Reflection

The .NET language provides advanced support for reflection and has lots of classes that help to make the usage of reflection easy and powerful. Reflection is an important concept that is used within many parts of modern software. One good example for the extensive use of reflection is the inversion of control (IOC) mechanism that many MVVM frameworks use to add better code structure and produce cleaner code. This IOC components rely heavy on reflection and could not be implemented without it.

4.3 Libraries for .NET

The .NET framework is one of the most powerful development frameworks with lots of built-in libraries. In addition to the included functionality there are many high quality libraries available from third-party sources. Most of those libraries can be added to Visual Studio and Xamarin Studio projects using the NuGet package manager. NuGet is a handy tool to manage and update third-party libraries, as well as some parts of the .NET framework itself.

The range of available third-party libraries include a large number of commercial and open source libraries. Since the different .NET platforms are not always compatible with each other, some libraries are not available for all platforms. Since the introduction of portable class libraries this has become less of a problem. Portable class libraries simplify the process of producing platform independent libraries and are widely used by the .NET community.

Portable class libraries can be used for different platforms by using only a common subset of APIs that are available on every supported platform. The developer of the portable class library can specify which platforms to support. Portable class libraries are not only available for Microsoft platforms but also for third party platforms like the ones supported by Xamarin.

4.4 MVVM frameworks

It is easy to develop a simple MVVM architecture without the use of any additional framework, but in most cases it can be beneficial to use one of the many availabe open source MVVM frameworks. The following list contains some of the most established open source MVVM frameworks for .NET applications.

- PRISM [46]
- Caliburn Micro [47]
- **Simple MVVM Toolkit** [48]
- Catel [49]
- **MVVM Light** [50]
- **MVVM Cross** [51]

The following section will give a detailed description of the three highlighted frameworks. The sample code used in this section is only one of many use cases for each of the MVVM frameworks. The examples are chosen to give a short but meaningful overview to compare the frameworks with each other.

All of the cross platform MVVM frameworks that will be described in the following sections will use a similar structure when used for cross platform development. The here described project structure for Visual Studio (or Xamarin Studio) applies to all three sample applications using either MVVM Light, the Simple MVVM Toolkit or MVVM Cross. The solution includes a “core” or “portable” project that contains the Model the ViewModels and all other code that can be shared across platforms. For each platform there will be a dedicated project that includes the Views, as well as all other platform specific code.

Solution

- Project.Core (Model, ViewModels, …)
- Project.Platform1 (Views, …)
- Project.Platform2 (Views, …)

  …

4.4.1 MVVM Light

MVVM Light is a small and simple MVVM framework that includes basic, but useful functionality for the creation of MVVM applications. This section will include a very basic sample application developed with the help of MVVM Light.

Supported Platforms

MVVM Light depends on a few Windows specific APIs and is therefore only available for Windows platforms. Xamarin platforms do not provide a binding framework by default. Even if it would be possible to replace the windows specific API calls from the MVVM Light source code, MVVM Light would not be able to serve non-Microsoft platforms very well without the addition of an external binding framework for Android, iOS and Mac.

The MVVM Light CodePlex page [50] promises that MVVM light supports the following platforms:

- Windows Presentation Foundation (WPF)
- Silverlight
- Windows Store with Windows Runtime (WinRT)

- Windows Phone with WinRT

However, when looking at the source code we will find iOS and Android implementations as well.

Model

Our Model contains a collection of animals. All data classes should implement the `INotifyPropertyChanged` interface to fully support data bindings. The base class `ObservableObject` provided by the MVVM Light framework offers a solid implementation of the `INotifyPropertyChanged` interface by providing the `RaisePropertyChanged` method.

```
public class Animal : ObservableObject {
    private string _name;
    public string Name {
        get { return _name; }
        set {
            _name = value;
            RaisePropertyChanged(()=>Name);
        }
    }
}
```

Messaging

MVVM Light includes a full featured messenger that enables loosely coupled communication within the application. The messenger has two important methods, one to register for a specified token and another to broadcast updates.

Register:

```
void Register<TMessage>(object recipient, Action<TMessage> action);
```

Publish:

```
void Send<TMessage>(TMessage message, object token);
```

Commands

MVVM Light offers the `RelayCommand` class, which is an implementation of the `ICommand` interface that is used to bind commands to UI-controls. In addition to the `Action` that specifies the command's execution behavior a `canExecute` method can be used to show if the command can be executed or not.

```
public RelayCommand(Action execute);
public RelayCommand(Action execute, Func<bool> canExecute);
```

ViewModel

The code below shows the full Implementation of our ViewModel. The ViewModel will subscribe to the `MessengerTokens.Animals` token and will receive the list of animals from our little setup code that is executed at the startup of the application.

```
public class MainViewModel : ViewModelBase {
    // Bindable Properties
    private ObservableCollection<Animal> _animals;
    public ObservableCollection<Animal> Animals {
        get { return _animals; }
        set { _animals = value; RaisePropertyChanged(()=>Animals); }}
```

```
        // Commands and Actions
        public RelayCommand Command { get; set; }

        private void Action() { /* ... */ }

        // Message callback methods
        private void AnimalsChanged(ObservableCollection<Animal> animals) {
            Animals = animals; }

        // Constructor
        public MainViewModel() {
            //Message registrations
            Messenger.Default.Register<ObservableCollection<Animal>>
                (this, MessengerTokens.Animals, AnimalsChanged);

            // Command setup
            Command = new RelayCommand(Action);

            // Model setup
            var animalService = new AnimalService();
            var animals = animalService.GenerateRandomAnimals();
            Messenger.Default.Send(
                new GenericMessage<ObservableCollection<Animal>>(animals));}
    }
```

View

Views with MVVM Light can be of any UI control's type like Window, Page or UserControl. The only connection to the ViewModel is the specification of the controls DataContext. To specify the `DataContext` for the used control, the following XAML binding can be used. The `StaticResource` with the name `Locator` is an instance of the ViewModelLocator that holds an instance of every ViewModel and is defined in a global resource file.

```
DataContext="{Binding MainViewModel, Source={StaticResource Locator}}"
```

We have only one View that displays the list of animals from our Model. It uses a `ListBox` with a simple `DataTemplate` to visualize the name of each animal. The `Button` will bind to the ViewModel's command. The following implementation shows the View for the Windows Phone platform. It can look very similar on other Microsoft platforms that use XAML as a markup language. For Xamarin platforms this code looks quite different and will be discussed in the MVVMCross section as well as in the practical part later in this document.

```
<StackPanel DataContext="{Binding MainViewModel,
            Source={StaticResource Locator}}">
  <ListBox ItemsSource="{Binding Animals}">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding Name}"></TextBlock>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
  <Button Command="{Binding Command}" Content="Tap me"></Button>
</StackPanel>
```

Impression and Conclusions

When comparing the three MVVM frameworks MVVM Light is the smallest and easiest to use. It provides solid tools for messaging, commands and other MVVM concepts. When developing only for Microsoft platforms this framework will be a good choice. Especially

smaller teams and developers with little experience can benefit from this MVVM framework. This framework is probably not the best choice, when targeting non-Microsoft platforms.

4.4.2 Simple MVVM Toolkit

Simple MVVM Toolkit is a simple, but useful framework to help with the development of MVVM applications. It provides a little more functionality than MVVM Light but might require more time to get used to.

Supported Platforms

The Simple MVVM Toolkit is available for most .NET platforms as the following list shows:

- WPF
- Silverlight
- Windows Store (RT)
- Windows Phone
- Xamarin.Android
- Xamarin.iOS

The framework does not include binding functionality for Android and iOS development. It is recommended to use the MVVMCross.Core library to support bindings for non-Microsoft platforms.

Model

The Simple MVVM Toolkit provides advanced functionality for data model objects. It includes the base class `ModelBase` that has some useful functionality like an advanced implementation of the `INofifyPropertyChanged` interface.

```
public class Animal : ModelBase<Animal> {
    private string _name;
    public string Name {
        get { return _name; }
        set {
            _name = value;
            NotifyPropertyChanged(m => m.Name);
        }
    }
}
```

Messaging

Similar to MVVM Light, the simple MVVM Toolkit includes a full featured messenger that enables loosely coupled communication within the application. The two methods described below are implemented by the base class of all ViewModels. In addition to the methods below, there are other implementations with different parameters available.

Registration:

```
void RegisterToReceiveMessages(string token,
    EventHandler<NotificationEventArgs> callback);
```

Publish:

```
void SendMessage<TOutgoing>(string token,
    NotificationEventArgs<TOutgoing> e);
```

Commands

The `DelegateCommand` included in the Simple MVVM Toolkit offers the same functionality as MVVM Light's `RelayCommand`.

```
public DelegateCommand(Action<T> executeAction);
public DelegateCommand(Action<T> executeAction,
    Func<T, bool> canExecute);
```

ViewModel

The simple MVVM Toolkit offers two different base classes for ViewModels to choose from.

MainViewModel

The first one (`ViewModelBase<TViewModel>`) is similar to the one from MVVM Light. Bellow's example of a simple MVVM Toolkit ViewModel offers the same functionality as the ViewModel from the sample implementation with MVVM Light.

```
public class MainViewModel : ViewModelBase<MainViewModel>
{
    // Bindable Properties
    private ObservableCollection<Animal> _animals;
    public ObservableCollection<Animal> Animals
    {
        get { return _animals; }
        set { _animals = value; NotifyPropertyChanged(m => m.Animals); }
}

    // Commands and Actions
    public DelegateCommand<Animal> AnimalSelectedCommand { get; set; }

    private void AnimalSelectedAction(Animal animal)
    {
        SendMessage<Animal>(MessengerTokens.SelectedAnimal.ToString(),
           new NotificationEventArgs<Animal>("Animal changed", animal));
    }

    // Message callback methods
    private void AnimalsChanged(object sender,
        NotificationEventArgs<ObservableCollection<Animal>> e) {
        Animals = e.Data;
    }

    // Constructor
    public MainViewModel(IAnimalService animalService) {
        //Message registrations
        RegisterToReceiveMessages<ObservableCollection<Animal>>
            (MessengerTokens.Animals.ToString(), AnimalsChanged);

        // Command setup
        AnimalSelectedCommand =
              new DelegateCommand<Animal>(AnimalSelectedAction);

        // Model setup
        var animals = animalService.GenerateRandomAnimals();
        SendMessage(MessengerTokens.Animals.ToString(),
            new NotificationEventArgs<ObservableCollection<Animal>>
                ("Animals changed", animals));
    }
}
```

AnimalDetailViewModel

The second one (`ViewModelDetailBase<TViewModel,TModel>`) is specialized on data visualization and manipulation for a specific data model object. The second generic type (`TModel`) specifies the type of the model object that provides the data for this ViewModel. This base class implements the `IEditableObject` Interface that supports a concept for object manipulation. The `IEditableObject` interface provides an easy and unified way to add editing with rollback functionality. A simple example for this could be a detail page that provides data manipulation and has two buttons, "OK" and "Cancel". In case the user presses the "OK" button the ViewModel saves all changes. When the "Cancel" button gets pressed, all changes get rolled back.

```
public class AnimalDetailViewModel :
  ViewModelDetailBase<AnimalDetailViewModel, Animal>
{
    // Properties
    public string Name {
        get { return Model != null ? Model.Name : null; }
        set { Model.Name = value; NotifyPropertyChanged(m=>m.Name); }
    }

    // Commands and Actions
    public DelegateCommand SaveCommand { get; set; }
    public void SaveAction() {
        EndEdit();
    }

    public DelegateCommand CancelCommand { get; set; }
    public void CancelAction() {
        CancelEdit();
    }

    // Message callback methods
    private void SelectedAnimalChanged(object sender,
      NotificationEventArgs<Animal> e) {
        Model = e.Data;
        BeginEdit();
        NotifyPropertyChanged(m => m.Name);
    }

    // Constructor
    public AnimalDetailViewModel() {
        // Command setup
        SaveCommand = new DelegateCommand(SaveAction);
        CancelCommand = new DelegateCommand(CancelAction);

        //Message registrations
        RegisterToReceiveMessages<Animal>
            (MessengerTokens.SelectedAnimal.ToString(),
            SelectedAnimalChanged);
    }
}
```

View

With the Simple MVVM Toolkit it is possible to use all UI controls as a View. The following XAML markup shows the Implementation of the two Views corresponding to the above described ViewModels.

MainView

```
<StackPanel DataContext="{Binding MainViewModel,
  Source={StaticResource Locator}}">
  <ListBox ItemsSource="{Binding Animals}">
```

```
      <ListBox.ItemTemplate>
        <DataTemplate>
          <Button Content="{Binding Name}"
            Command="{Binding MainViewModel.AnimalSelectedCommand,
            Source={StaticResource Locator}}"
            CommandParameter="{Binding}"/>
        </DataTemplate>
      </ListBox.ItemTemplate>
    </ListBox>
  </StackPanel>
```

AnimalDetailsView

```
  <StackPanel DataContext="{Binding AnimalDetailViewModel,
    Source={StaticResource Locator}}">
    <TextBox Text="{Binding Name, Mode=TwoWay}"/>
    <Button Command="{Binding SaveCommand}" Content="Save" />
    <Button Command="{Binding CancelCommand}" Content="Cancel" />
  </StackPanel>
```

Impressions and Conclusions

The Simple MVVM Toolkit has some nice features and is easy to use. It supports a wider range of platforms than MVVM Light, offers good documentation and has a large community. It does not provide an own binding framework for Android or iOS. That is why it is dependent on MVVM Cross or other frameworks that provide binding functionality for non XAML platforms.

4.4.3 MVVM Cross

MVVM Cross is unlike MVVM Light and the Simple MVVM Toolkit a rather large framework that can offer much more than Model, View, ViewModel, Messenger and Command implementations. Due to its modularity the individual core package can be extended with numerous packages that provide cross platform solutions for common problems MVVM developers are facing. It was built with great care to support cross platform MVVM development as best as possible and does that with lots of success.

Supported Platforms

MVVMCross is available for most modern .NET platforms including the following.

- WPF
- Windows Store
- Windows Phone
- Xamarin.Android
- Xamarin.iOS
- Xamarin.Mac

This framework includes advanced binding functionality for all non-Microsoft platforms and therefore can provide a full featured MVVM experience for all supported platforms without the use of any external library.

Packaging system

MVVM Cross consists of a required core that can be extended with additional plugins. Those plugins can be easily created and reused across multiple MVVM cross applications. MVVM Cross offers numerous plugins for the most common use cases via individual NuGet packages. The following list contains some of the available plugins [51].

- Accelerometer

- Bookmarks
- Color
- DownloadCache
- Email
- File
- Json
- JsonLocalisation
- Location
- Messenger
- MethodBinding
- Network
- PhoneCall
- PictureChooser
- Reflection
- ResourceLoader
- Share
- SoundEffects
- SQLite
- ThreadUtils
- Visibility
- WebBrowser

Model

The data objects can use the `MvxNotifyPropertyChanged` base class to inherit an implementation of the `INotifyPropertyChanged` interface. This base class offers similar methods like the corresponding classes of the other two already mentioned MVVM frameworks.

```
public class Animal : MvxNotifyPropertyChanged
{
    private string _name;
    public string Name {
        get { return _name; }
        set
        {
            _name = value;
            RaisePropertyChanged(()=>Name);
        }
    }
}
```

Messaging

Messaging with MVVM Cross can be done with the messenger plugin that is available from the MVVM Cross repository or via NuGet. The messages sent with the MVVM Cross Messenger will be identified by the type of the message and not by tokens like it was the case for the other two applications developed with MVVM Light and the Simple MVVM Toolkit.

The following method is used to publish new messages that are required to have the base type `MvxMessage`.

```
void Publish(MvxMessage message);
```

To subscribe to a message type, there are three methods available. Depending on what thread should execute the notification a different method can be chosen. The subscribe methods return an object of type `MvxSubscriptionToken`, which can be used to manage the subscription. In case the token gets collected by the garbage collector or the `Dispose()` function gets manually called, then the recipient is no longer registered and will not receive any further messages for the registered type.

The `Subscribe` method always executes the `deliveryAction` on the publishing thread.

```
MvxSubscriptionToken Subscribe<TMessage>
    (Action<TMessage> deliveryAction, MvxReference reference =
    MvxReference.Weak, string tag = null) where TMessage : MvxMessage;
```

The `SubscribeOnMainThread` method always executes the `deliveryAction` on the platform's main-thread/UI-thread. This is useful if the `deliveryAction` directly manipulates UI controls or changes the values of bindable properties.

```
MvxSubscriptionToken SubscribeOnMainThread<TMessage>
    (Action<TMessage> deliveryAction, MvxReference reference =
    MvxReference.Weak, string tag = null) where TMessage : MvxMessage;
```

The last option is used for resource intensive code inside the `deliveryAction` what should always run on a thread pool thread.

```
MvxSubscriptionToken SubscribeOnThreadPoolThread<TMessage>
    (Action<TMessage> deliveryAction, MvxReference reference =
    MvxReference.Weak, string tag = null) where TMessage : MvxMessage;
```

Commands

Commands are a part of the MVVM Cross core and do not need an additional package. The `MvxCommand<T>` included in MVVM Cross offers similar functionality as the `RelayCommand` class used within MVVM Light and the `DelegateCommand` from the Simple MVVM Toolkit.

```
public MvxCommand(Action<T> execute);
public MvxCommand(Action<T> execute, Func<T, bool> canExecute);
```

Application setup

Unlike MVVM Light and the Simple MVVM Toolkit, the MVVM Cross framework provides a unified way for initializing the application.

In the common portable class library project the `App` class can be used to execute common initialization code. The following code sample is the default implementation of this class. It registers all classes with the ending `"Service"`, as a lazy loading singleton and calls the `RegisterAppStart` method that executes some internal setup code and navigates to the `MainViewModel`.

```
public class App : Cirrious.MvvmCross.ViewModels.MvxApplication {
    public override void Initialize() {
        CreatableTypes().EndingWith("Service").AsInterfaces()
            .RegisterAsLazySingleton();
        RegisterAppStart<ViewModels.MainViewModel>();
    }
}
```

The `Setup` class in each platform specific project is used to make all platform specific initializations. The `CreateApp` method is usually executed within the native initialization

events of each platform. The default implementation of the `Setup` class for Windows Phone looks like the following.

```
public class Setup : MvxPhoneSetup {
    public Setup(PhoneApplicationFrame rootFrame) : base(rootFrame){
    }

    protected override IMvxApplication CreateApp(){
        return new Core.App();
    }

    protected override IMvxTrace CreateDebugTrace(){
        return new DebugTrace();
    }
}
```

ViewModel

The base class of all ViewModels (`MvxViewModel`) used in MVVM Cross has similar functionality as the one used by MVVM Light or the Simple MVVM Toolkit. Additionally, it has an automated dependency injection mechanism that can find the right Service implementation when the interface of a service is used as constructor parameter of the ViewModel.

```
public class MainViewModel : MvxViewModel {
    // Messenger subscription tokens
    private MvxSubscriptionToken _animalsChangedToken;

    // Used services
    private readonly IMvxMessenger _messenger;

    // Bindable Properties
    private ObservableCollection<Animal> _animals;
    public ObservableCollection<Animal> Animals {
        get { return _animals; }
        set { _animals = value; RaisePropertyChanged(() => Animals); }
    }

    // Commands and Actions
    public MvxCommand<Animal> AnimalSelectedCommand { get; set; }

    private void AnimalSelectedAction(Animal animal) {
        _messenger.Publish(
        new SelectedAnimalChangedMessage(this, animal));
    }

    // Message callback methods
    private void AnimalsChanged(AnimalsChangedMessage m) {
        Animals = m.Animals;
    }

    // Constructor
    public MainViewModel(IMvxMessenger messenger,
        IAnimalService animalService)
    {
        //Message registrations
        _messenger = messenger;
        _animalsChangedToken =
            _messenger.Subscribe<AnimalsChangedMessage>(AnimalsChanged);

        // Command setup
        AnimalSelectedCommand =
            new MvxCommand<Animal>(AnimalSelectedAction);

        // Model setup
        var animals = animalService.GenerateRandomAnimals();
```

```
            _messenger.Publish(new AnimalsChangedMessage(this, animals));
        }
    }
```

View

MVVM Cross requires UI controls to implement the interface `IMvxView`, as well as some other interfaces to be used as a View. MVVM Cross also includes a special page control class, which can be used instead of each platform's native page control. This class extends from the native page control and implements the `IMvxView` interface as well as a different interface on each platform. For example, the `IMvxStoreView` interface is used for Windows Store applications.

The ViewModelLocator of MVVM Cross does not look like the one of MVVM Light and the Simple MVVM Toolkit. With MVVM Cross, the UI control's DataContext can be used to identify the ViewModel, since the `DataContext` of an `IMvxView` does always point to its ViewModel by default. The design time property `d:DataContext` can be used to benefit of compile time type checking, since the type of the ViewModel will otherwise not be known until the application is running.

```
<views:MvxPhonePage
  d:DataContext="{d:DesignInstance viewModels:MainViewModel }">
  <ListBox ItemsSource="{Binding Animals}">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <Button Content="{Binding Name}"
            Command="{Binding DataContext.AnimalSelectedCommand,
            Source={RelativeSource TemplatedParent }}"
            CommandParameter="{Binding}"/>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</views:MvxPhonePage>
```

Binding frameworks for Android, iOS and Mac

One big disadvantage of iOS and Android development when using Xamarin compared to Microsoft's XAML platforms is that they do not support data bindings by default. Therefore, MVVM Cross provides a solid binding framework for those platforms, which is similar to web based binding frameworks like KnockoutJs or AngularJs. This binding framework can not only be used within iOS, Android and Mac applications, but also as an alternative to the native binding concept for XAML based applications.

The MVVM Cross bindings provide functionality similar to the XAML bindings and offer a more compact syntax. Later in the practice section we will show some examples of how the MVVM Cross binding framework can be used to enable data bindings inside an Android application developed with the Xamarin runtime.

Impression and Conclusions

MVVM Cross is without a doubt one of the most powerful MVVM frameworks, especially when used for cross platform development. It offers a lot of functionality and a solid application design for cross platform development. The included binding framework works well with XAML and non XAML platforms and enables MVVM development for all supported platforms. Due to the modular design of MVVM Cross and its plugin system it is suitable for small, as well as extensive applications. It allows to integrate many different plugins when needed, but also keeps the app binary as small as possible. Furthermore,

through their plugin system the developers of MVVM Cross encourage third party developers to create additional plugins.

This all sounds very amazing, but developing with MVVM Cross comes at a small cost. All the included concepts and the complex application design makes MVVM cross more difficult to understand than MVVM Light or the Simple MVVM Toolkit. Choosing MVVM Cross is probably not the easiest way to create someone's first MVVM application.

4.5 MonoGame, a cross platform graphics framework

Sometimes the platform's UI features are not enough to create graphically advanced applications and games. Graphically intense code is often written with native libraries, once for every target platform. Nowadays there are lots of frameworks that provide tools to develop performance oriented graphical applications that are able to run with little changes on multiple platforms.

MonoGame is one of these cross platform graphic frameworks. It can easily be used together with an existing MVVM application to enhance the user experience by adding animations or advanced user controls. MonoGame lets developers create fast and powerful applications for iOS, Android, Mac OS X, Linux, Windows, Windows Store and Windows Phone. Currently it is under constant development and support for even more platforms is planned for the future.

4.6 Xamarin for Android, iOS and Mac

The company Xamarin [52] has adopted the .NET platform to provide .NET's great features for development of non-Microsoft platform applications. Xamarin uses the open source parts of .NET and implements most proprietary components by themselves. Furthermore, they add .NET wrappers for platform specific functionality. This concept works very well, with decent performance similar to native developed applications. Xamarin does not replace the native UI controls with new implementations. The native controls get wrapped into .NET controls and will behave like the native controls when running on the target device. This makes it easy to create .NET applications that run on Android, iOS or Mac with a great native user experience. In most cases even experienced developers cannot tell the difference between a native application and an application developed with the help of a Xamarin platform without looking at the application's source code.

For developing Xamarin Applications the developer can use Xamarin's own IDE, Xamarin Studio or a plugin for Visual Studio.

Until 2014 the development for Xamarin platforms was quite similar to writing native platform code with the benefit of using .NET's features as an addition to the existing programming paradigms.

In 2014 Xamarin adopted the "write once, run everywhere" concept. This means that the entire application is written once and does not include any platform specific code. In addition to the existing development environments Xamarin has introduced an extension library called Xamarin.Forms that includes most common controls for the targeted platforms in the form of cross platform controls. With that an application can be written by using these controls instead of the previously used wrapped native components. The main difference to the new controls is that they can be used across platforms. The controls will be translated to native controls for all targeted platforms to get the familiar

fluent user experience and performance as known from Xamarin platforms. The Xamarin.Forms controls do also support data bindings, therefore no third party binding framework is needed. How to use these new controls is not within the scope of this work and will not be discussed by this document. A good description of the Xamarin.Forms library can be found on the Xamarin homepage [53].

4.7 Windows' unified application architecture

In spring 2014 Microsoft has enabled the possibility to develop unified apps. Up to this point it was only possible to share code across platforms with the use of a portable class library. Portable class libraries are limited to pure code files and do not support XAML markup files or other application resources.

The Unified app architecture is a very new concept introduced by Microsoft. At the time this document was composed, the unified app architecture was available for Windows Phone 8.1 and Windows 8.1 with Update 1. Microsoft is still working to further improve and expand this technology in terms of features and supported platforms.

With the use of unified apps, source code can be shared across multiple platforms more easily than before. With the further unification of Windows Phone and Windows it is now possible to share more than 90% of the applications code for many real world applications. For some types of applications it is even possible to have no unique code at all. The MVVM architecture can help to get the most out of this new and useful concept.

5 Practice MVVM

This section describes the implementation of a simple data intensive application called ShoppingList. The application is the implementation of a very simple shopping list application where users can add and manage products in lists. The functionality of the application is only partly implemented to show how MVVM can be used on different platforms.

Each implementation of ShoppingList provides similar functionality and has a similar structure to better compare individual implementations with each other. The sample application consists of the following three Views.

ListsView This View displays the shopping-lists and lets the user select one of the available lists. When a list is selected the application navigates and displays the ProductsView.

ProductsView The ProductsView shows a list of all products from the selected list. It offers functionality to open the AddProductView to add a new product to the list. In addition to that, a cleanup function can be executed, which removes all bought products from the selected list.

AddProductView The AddProductView allows the user to create a new product and add it to the currently selected list. It should only be possible to add a product with a name that has a minimum length of two characters.

5.1 MVVM with web technologies

As described in the previous section for MVVM development with web based applications, there is the need of several tools, libraries and frameworks to make the development process as fast and comfortable as possible.

This section will describe the used frameworks in different ways. At first there will be an explanation about why this frameworks and tools were chosen, followed by the example implementation of the ShoppingList application which includes solutions for many common problems. To demonstrate how easy it is to introduce an additional visualization, there will be two different implementations of this application one for mobile devices with touch screens and one for conventional desktop devices that are controlled with mouse and keyboard.

To ensure that our new application behaves as expected, there will be a short explanation on how unit-tests could be added to our application. Since MVVM is a software design pattern that is especially suitable for unit testing, it will be easy to test most of the application's logic with unit-tests.

The full source code of the examples used in this section is available on GitHub under the MIT license.

https://github.com/Lupin1st/ShoppingList.Web

5.1.1 Used frameworks and tools

Web based technologies are a great way to create applications for multiple platforms but as discussed above JavaScript, HTML and CSS alone are often not the best tools to create big and data intensive applications. Therefore, most web based applications use different kinds of libraries and extensions to provide better developing experiences. The following list includes all external libraries and tools that were used to develop the ShoppingList application. In addition to a description there will be a justification of why each tool was chosen for the development process of the application.

Programming Tools (Visual Studio with extensions)

The web application was implemented with the help of Visual Studio, since it provides a lot of features that help with developing web applications. Especially when using TypeScript as a JavaScript replacement, the usage of Visual Studio is encouraged. There are also a large number of additional extensions for Visual Studio that can increase developing comfort and add new features to the IDE. The following list contains all relevant extensions for VisualStudio that were used to develop this sample application.

Web Essentials Web Essentials [54] extends Visual Studio with lots of useful additions to help developers create web applications easier, faster and supports them to write cleaner code.

In addition to basic improvements it provides tools and enhancements for CSS, HTML, JavaScript, TypeScript and CoffeeScript.

NuGet NuGet [55] is a package management system for libraries. It delivers a solid solution for managing libraries inside Visual Studio. With NuGet, developers can add new libraries quickly to their projects. The NuGet package manager also makes updating to the newest version of libraries more convenient.

For companies that have large numbers of internal libraries NuGet can also be used to manage the internal code and distribute the code within the company.

ReSharper ReSharper [56] is one of the best known and most used extensions for Visual Studio. It adds countless tools and improvements to Visual Studio and provides great enhancements to the developing experience of Visual Studio for .NET, C++ and web applications.

Alternative programming language (TypeScript)

When working with JavaScript there are nowadays plenty different approaches to create code. As described previously the three programming languages CoffeeScript, Dart and TypeScript are good candidates to replace JavaScript and enhance the developing experience.

Between these three languages to create JavaScript code, Typescript was chosen because of multiple reasons.

- TypeScript is near to JavaScript and therefore, easier to debug
- Good integration in the used developing environment Visual Studio
- Clean and well-structured object oriented syntax
- Good compatibility with existing JavaScript libraries

- Future-proof compatibility to ES6

HTML tools and extensions (jQuery)

The usage of the two UI frameworks that are described below are dependent on jQuery and require the jQuery library. The DOM manipulation mechanisms of jQuery will not be used directly within our sample application since the binding framework makes this obsolete, in many cases.

UI frameworks (jQueryUI and jQueryMobile)

For creating modern web applications there are plenty of good UI frameworks to start with. For this implementation the two frameworks jQueryUI and jQueryMobile were chosen because of their openness, easy usage, good documentation and large community.

Only one of each framework will be used simultaneously. This will show how MVVM can be used to create different interfaces for the same web application while sharing a large amount of code.

Binding framework (KnockoutJs):

KnockoutJs will be used to provide a full featured binding framework for the ShoppingList sample application. Bindings are an essential part of MVVM and will link our Views, written in HTML, to the ViewModels, written in TypeScript. KnockoutJs was chosen because of its solid implementation and the flexibility to use an external software architecture.

MVVM Framework:

For web applications the MVVM architecture is not as widely used as it is for .NET development. There are only a view MVVM frameworks available and none of them seems suitable for this sample application. Therefore, all MVVM classes as well as an IOC system and a messenger implementation were especially created for this application.

5.1.2 Implementation (Core)

The core application includes all shared code that can be used for the mobile and the desktop implementation. It contains the data Model, the ViewModels, all service interfaces, as well as the implementation of all shared services like the messaging service and a class for initializing the application. For this example the core application does not contain any HTML or CSS code, yet for larger applications it can be useful to share some HTML and CSS code as well.

Model

The model class itself includes all data relevant for the application. In this case it is a collection of all shopping-lists (`DataList`). For real world applications a service could load and store the model's state to the local storage or onto a remote server. The use of an interface makes it easy to change the model implementation for multiple reasons later on.

```
export interface IDataModel {
    lists: Array<DataList>;
}

class DataModel implements IDataModel {
    public lists: Array<DataList> = Array<DataList>();

    constructor() { /* adding lists for first time use */ }
}
```

The `DataList` class has two properties. The first is its name, which is a knockout observable object of the type `string`. The second one is an observable collection that contains all products associated with the list. It is important to use knockout observables, since data objects are later used for data bindings, which would not work otherwise.

```
class DataList {
    public name: KnockoutObservable<string> = ko.observable("");
    public products: KnockoutObservableArray<DataProduct> =
        ko.observableArray<DataProduct>();

    constructor(name: string) {
        this.name(name);
    }
}
```

The `DataProduct` class, like the `DataList` class, uses observable types for its three properties.

```
module ShoppingList.Model {
    export class DataProduct {
        public name: KnockoutObservable<string> = ko.observable("");
        public amount: KnockoutObservable<string> = ko.observable("");
        public bought: KnockoutObservable<boolean> = ko.observable(false);

        constructor(name: string, amount: string, bought: boolean) {
            this.name(name);
            this.amount(amount);
            this.bought(bought);
        }
    }
}
```

Services

Services are used to create well structured, testable and reusable classes to help with common tasks.

Messaging

The very basic messaging service provides only the two methods register and publish. The implementation of the messenger uses a map to store the tokens as its keys and a list of functions with the signature of `MessageRecievedCallback` as its values. When new content gets published all subscribers get notified through the execution of their callback function.

```
interface MessageRecievedCallback {
    (content: any): void;
}

interface IMessagingService {
    register(token: MessagingToken, messageRecievedCallback:
        MessageRecievedCallback): void;
    publish(token: MessagingToken, content: any): void ;
}

class MessagingService implements IMessagingService { /* ... */ }
```

Navigation

The navigation service is used for page navigation, it can navigate forward and backward on the navigation stack. Its implementation is highly dependent on the used UI framework and can therefore not be shared between the mobile and desktop version of our application.

```
interface INavigationService {
    navigateTo(containerId: string): void;
    navigateBack(containerId: string): void;
}
```

Notification

The `NotificationService` allows us to show message boxes with the design of each UI framework and will be implemented by the platform specific parts of the application.

```
interface MessageBoxFinishedCallback {
    (result: MessageBoxResult): void;
}

enum MessageBoxResult { Ok, Cancel }

interface INotificationService {
    showMessageBox(title: string, content: string,
        callback: MessageBoxFinishedCallback): void;
}
```

ViewModels

The ViewModels are located inside the core part of the application and will be shared across all implementations. As described in the theory section of this document, the ViewModel provides some of the Model's data for the View and manages the visualization states of the View. Its properties are all observable, so they can be used for data binding.

ViewModel locator

The ViewModelLocator holds static variables of all the ViewModels. In this basic scenario all ViewModels get created at startup and will live until the application gets closed. For larger applications it is often required to unload some of the ViewModels to save memory. This is especially important for ViewModels that hold references to large objects.

ViewModel base class

This base class can be used to provide functionality that is often used inside ViewModels. In this sample application the `ViewModelBase` class is responsible for managing the `this` pointer's scope. It automatically changes the behavior of the `this` pointer to always point to the ViewModel and not the calling object.

ListsViewModel

The `ListsViewModel` provides the data for the ListsView it has two observable properties. One is the collection of lists that will be displayed and the other one is the currently selected list that is later used by the View for highlighting the selected list.

```
class ListsViewModel extends ViewModelBase {
    // Properties
    public lists = ko.observable<Array<Model.DataList>>();
    public selectedList = ko.observable<Model.DataList>();
```

The `selectList` action is responsible to tell other ViewModels about changes of the selected list and will navigate to the `ProductsView` after the selection of a list.

```
    // Actions
    public selectList(list: Model.DataList) {
        Services.ServiceLocator.MessagingService.publish(
            Services.Messaging.MessagingToken.SelectedListChanged, list);
```

```
            Services.ServiceLocator.NavigationService.
                navigateTo("ProductsView");
        }
```

The message actions are called when someone publishes changes about all available lists or the currently selected list.

```
        // Message actions
        private listsChangedMessageAction(lists: Array<Model.DataList>) {
            this.lists(lists);
        }

        private selectedListChangedMessageAction(list: Model.DataList) {
            this.selectedList(list);
        }
```

The constructor registers the ListsViewModel for the two message-tokens `ListsChanged` and `SelectedListChanged`.

```
        // Constructor
        constructor() {
            super();

            Services.ServiceLocator.MessagingService.register(
                Services.Messaging.MessagingToken.ListsChanged,
                this.listsChangedMessageAction);

            Services.ServiceLocator.MessagingService.register(
                Services.Messaging.MessagingToken.SelectedListChanged,
                this.selectedListChangedMessageAction);
        }
    }
```

ProductsViewModel

The ProductsViewModel has only the selected list as an observable property.

```
        // Properties
        public selectedList = ko.observable<Model.DataList>();
```

It provides actions for toggling the bought state, adding new products or removing the already bought products from the list.

```
        // Actions
        public buyProduct(product: Model.DataProduct) {
            product.bought(!product.bought());
        }

        public addProduct() {
            Services.ServiceLocator.NavigationService.
                navigateTo("AddProductView");
        }
```

```
        public cleanup() {
            Services.ServiceLocator.NotificationService.showMessageBox(
              "Cleanup", "Remove all bought products?", (result) => {
                 if (result == Services.MessageBoxResult.Ok) {
                     var products = this.selectedList().products;

                     for (var p = 0; p < products().length; p++) {
                         if (products()[p].bought()) {
                             products.splice(p, 1);
                             p--;
                         }
                     }
                 }
            });
        }
```

To receive changes for the selected list, the `ProductsViewModel` listens like all other ViewModels to all messages that are published with the `SelectedListChanged` massage-token.

```
// Message actions
private selectedListChangedMessageAction(list: Model.DataList) {
    this.selectedList(list);
}
```

The constructor registers the ViewModel for messages published with the `SelectedList-Changed` message token.

```
//Constructor
constructor() {
    super();
    Services.ServiceLocator.MessagingService.register(
        Services.Messaging.MessagingToken.SelectedListChanged,
        this.selectedListChangedMessageAction);
}
```

AddProductViewModel

The `AddProductViewModel` has three properties, the first is the selected list where the new products will be added to after the View executes the `addProduct` action. The other two are used to store the current state of the name and the amount of the new product.

```
class AddProductViewModel extends ViewModelBase {
    // Properties
    public selectedList = ko.observable<Model.DataList>();
    public name = ko.observable<string>();
    public amount = ko.observable<string>();
```

The `addProduct` action is responsible for creating a new product by using the two temporary properties `name` and `amount`. After a successful creation of the new product the `Navigationservice` will navigate back to the previous page.

```
        // Actions
        public addProduct() : void {
            var product = new ShoppingList.Model.DataProduct(
                this.name(), this.amount(), false);

            this.selectedList().products.push(product);

            this.name("");
            this.amount("");
```

```
            Services.ServiceLocator.NavigationService.
                navigateBack("AddProductView");
        }

        // Message actions
        private selectedListChangedMessageAction(list: Model.DataList) : void{
            this.selectedList(list);
        }

        // constructor
        constructor() {
            super();

            Services.ServiceLocator.MessagingService.register(
                Services.Messaging.MessagingToken.SelectedListChanged,
                this.selectedListChangedMessageAction);
        }
    }
```

Setup

The `AppBase` class is the base class for the platform specific initialization classes and contains common code to setup the application. This class tells KnockoutJs about the binding relation between the HTML controls and the ViewModels.

```
    export class AppBase {
        public initialize() {
            this.registerCommonServices();
            this.registerServices();

            var views = ["Lists", "Products", "AddProduct"];
            this.applyBindings(views);
        }

        public registerCommonServices(): void {
            ShoppingList.Services.ServiceLocator.MessagingService =
            new ShoppingList.Services.Messaging.MessagingService();
        }

        public applyBindings(names: Array<string>): void {
            var outstanding = names.length;

            ViewModel.ViewModelLocator.initialize();

            for (var i in names) {
                var viewContainer = $("div[data-view = '" +
                    names[i] + "View']")[0];
                var viewModel = ShoppingList.ViewModel.
                    ViewModelLocator[names[i] + "ViewModel"];
                ko.applyBindings(viewModel, viewContainer);
            }

            var model: Model.IDataModel = new Model.DataModel;

            Services.ServiceLocator.MessagingService.publish(
                Services.Messaging.MessagingToken.ListsChanged, model.lists);
        }
    }
```

5.1.3 Implementation (Desktop)

The following implementation is platform specific and will only be included in the desktop implementation. It includes Views, services as well as code to setup the desktop specific parts of the application.

Setup

The setup class `AppDesktop` is derived from `AppBase` and uses the `ServiceLocator` to register service implementations that are specific for the desktop application.

```
class AppDesktop extends AppBase {
    public registerServices(): void {
        ShoppingList.Services.ServiceLocator.NavigationService =
            new ShoppingList.Services.NavigationServiceJQueryUi();
        ShoppingList.Services.ServiceLocator.NotificationService =
            new ShoppingList.Services.NotificationServiceJQueryUi();
    }
}
```

Services

The desktop implementation has two specific services, the `NavigationService` and the `NotificationService`.

Navigation

This implementation of the `NavigationService` uses jQueryUI to manage all pages and dialogs. It calls the jQueryUI method "`$.dialog`" to open and close dialog windows.

```
export class NavigationServiceJQueryUi implements INavigationService {
    navigateTo(containerId: string): void {
        $("#" + containerId).dialog();
    }

    navigateBack(containerId: string): void {
        $("#" + containerId).dialog("close");
    }
}
```

Notification

The notification service for the desktop implementation uses a jQueryUI dialog to show themed message boxes.

```
export class NotificationServiceJQueryUi implements INotificationService {
    showMessageBox(title: string, content: string,
        callback: MessageBoxFinishedCallback): void {

        $("#messageBox").attr("title", title);
        $("#messageBox #messageBoxContent").html(content);

        $("#messageBox").dialog({
            resizable: false, height: 250, modal: true,
            buttons: {
                "Ok": function () {
                    callback(MessageBoxResult.Ok);
                    $(this).dialog("close");
                },
                Cancel: function () {
                    callback(MessageBoxResult.Cancel);
                    $(this).dialog("close");
                }
            }
        });
    }
}
```

The following HTML code is nested within an invisible container and will only be shown when jQueryUI displays it as a dialog window.

```
<div id="messageBox" title="">
```

```
    <p>
      <span class="ui-icon ui-icon-alert"
            style="float:left; margin:0 7px 20px 0;"></span>
      <span id="messageBoxContent"></span>
    </p>
  </div>
```

The message box will be used to ask if the user wishes to remove all bought products from the list and will look like the following screenshot.

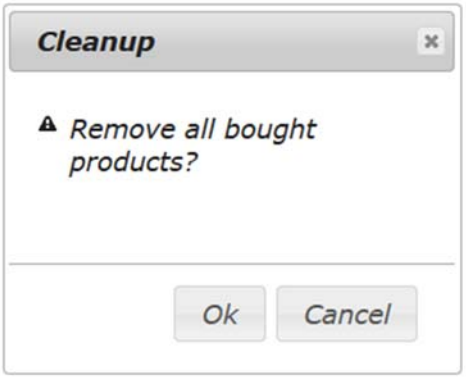

***Figure 14**: Screenshot of a MessageBox (Desktop)*

Views

The desktop implementation has one View for each ViewModel and uses jQueryUI to provide a solid experience for desktop users.

ListsView

The root `div` uses the CSS class `view-container-lists` to inform the setup code of the `AppBase` class, which HTML control is the root container of the ListsView. It uses KnockoutJs bindings to bind the UI to the ListsViewModel's collection of all lists. For each list in the collection there will be one `li` item inside the `ul` container. A data binding for the CSS property will highlight the selected list by applying a different style for selected and not selected lists.

```
<div data-view="ListsView" class="view-container-lists">
  <div data-role="header " data-position="inline"><h2>Lists</h2></div>
  <div data-role="content" data-theme="a">
    <ul data-bind="foreach: lists">
      <li data-bind="text: $data.name, click: $root.selectList,
                     css: { list_container_selected: $data ==
                     $root.selectedList() }" />
    </ul>
  </div>
</div>
```

ProductsView

The `ProductsView` displays a list of all products from the currently selected list and provides two buttons one for adding a new product and a second one for removing all bought products.

```
<div data-view="ProductsView" class="view-container-products">
  <div data-role="header"><h2>Products</h2></div>
  <button data-bind="click: addProduct, visible: selectedList()">
    Add Product
  </button>
```

```
    <button data-bind="click: cleanup, visible: selectedList()">
      Cleanup
    </button>
    <!-- ko with: selectedList -->
    <ul data-bind="foreach: products">
      <li data-bind="text: $data.amount() + ' ' + $data.name(),
                     click: $root.buyProduct,
                     css: { product_container_bought: $data.bought }" />
    </ul>
    <!-- /ko -->
  </div>
```

The desktop application displays the ListsView and the ProductsView next to each other on the starting page of the application. The result looks like the following screenshot.

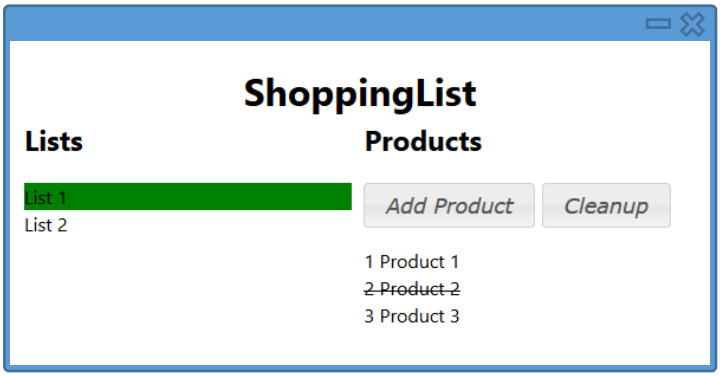

Figure 15: Screenshot of ListsView and ProductsView (Desktop)

AddProductView

The AddProductView provides two input fields. One for the name and one for the amount of the new product. The View also contains a button to submit the changes and add the entered values as a new product to the selected list.

```
  <div data-view="AddProductView">
    <div id="AddProductView" title="Add Product" style="margin: 12px">
      <p>
        Name: <input data-bind="value: name, valueUpdate: 'afterkeydown'" />
      </p>
      <p>Amount: <input data-bind="value: amount" /></p>
      <button data-bind="click: addProduct, enable: name() &&
                         name().length >= 2">
        Add
      </button>
    </div>
  </div>
```

The screenshot below shows the AddProductView after the user added name and amount of a new product.

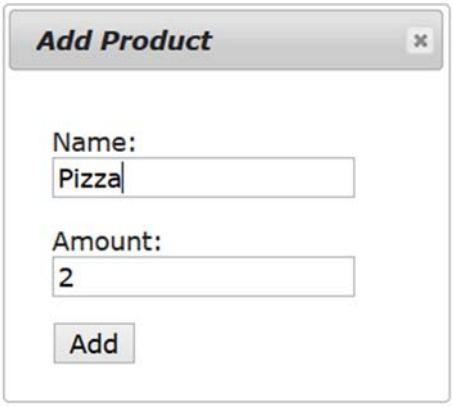

***Figure 16**: Screenshot of the AddProductView (Desktop)*

5.1.4 Implementation (Mobile)

The following implementation is platform specific and will only be included in the mobile implementation. It includes Views, services as well as code to setup the mobile specific parts of the application.

Setup

The setup for the mobile implementation looks the same as the one for the desktop implementation, with the difference of registering service implementations that use jQueryMobile instead of jQueryUI.

Services

The mobile implementation has to implement the same services as the desktop implementation. For the case that one platform does not provide all the features that are used in another implementation, it is recommended to use a dummy implementation for this service as well. This makes the code cleaner and provides the possibility of an easy adoption when the APIs gets available in the future.

Navigation

JQueryMobile does provide a different page navigation experience as jQueryUI. It navigates through the application with the use of pages instead of showing popup windows. The `NavigationServiceJQueryMobile` calls the jQueryMobile page navigation methods to switch pages.

```
class NavigationServiceJQueryMobile implements INavigationService {
    navigateTo(containerId: string): void {
        $.mobile.navigate("#" + containerId);
    }

    navigateBack(containerId: string): void {
        window.history.back();
    }
}
```

Notification

For displaying a message box with jQueryMobile a dialog is used, which contains the information. JQueryMobile's dialogs are structured similar to pages, with the difference that they are displayed as modal windows instead of whole pages. In the example below

jQuery is used to manipulate the `div` element that will later be used as content for the dialog. This could also be implemented with the help of KnockoutJs and an additional ViewModel.

```
class NotificationServiceJQueryMobile implements INotificationService {
    showMessageBox(title: string, content: string,
        callback: MessageBoxFinishedCallback): void {
        $("#MessageBox [data-field=HeaderContent]").html(title);
        $("#MessageBox [data-field=BodyContent]").html(content);

        $("#MessageBox [data-button=ButtonOk]").unbind('click');
        $("#MessageBox [data-button=ButtonOk]").click(() => {
            callback(MessageBoxResult.Ok);
            window.history.back();
        });

        $("#MessageBox [data-button=ButtonCancel]").unbind('click');
        $("#MessageBox [data-button=ButtonCancel]").click(() => {
            callback(MessageBoxResult.Cancel);
            window.history.back();
        });

        $.mobile.navigate("#MessageBox");
    }
}
```

The mobile implementation of the message box looks like the following screenshot when displayed by the cleanup command.

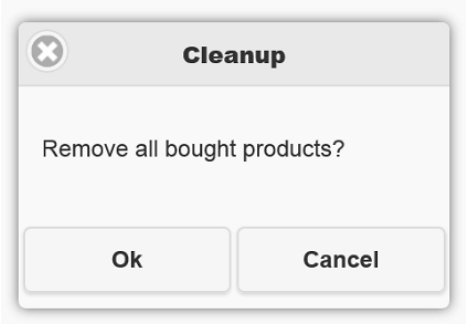

***Figure 17**: Screenshot of a MessageBox (Mobile)*

The following HTML code contains the content for each message box. It will be filled with data from the `NotificationService` each time a message box needs to get displayed.

```
<div id="MessageBox" data-role="dialog">
  <div data-role="header">
    <h1 data-field="HeaderContent">
      Add product
    </h1>
  </div>
  <div data-role="main" class="ui-content">
    <p data-field="BodyContent"></p>
  </div>
  <div>
    <table style="width: 100%">
      <tr>
        <td style="width: 50%">
          <button data-button="ButtonOk">Ok</button></td>
        <td style="width: 50%">
          <button data-button="ButtonCancel">Cancel</button>
```

```
            </td>
          </tr>
        </table>
      </div>
    </div>
```

Views

The Views for the mobile implementation use jQueryMobile themed UI controls like pages, dialogs, buttons and others to provide a solid user experience for mobile devices.

For the mobile implementation, the ListsView and ProductsView are displayed on separate pages while using the same ViewModels as the desktop implementation.

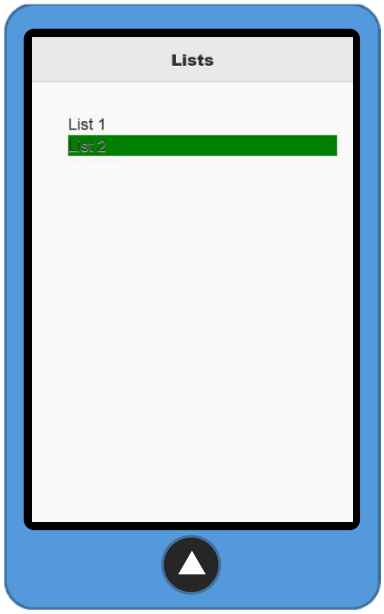

***Figure 18**: Screenshot of the ListsView (Mobile)*

***Figure 19**: Screenshot of the ProductsView (Mobile)*

The screenshot below shows the AddProductView of the mobile implementation after the user added name and amount of a new product.

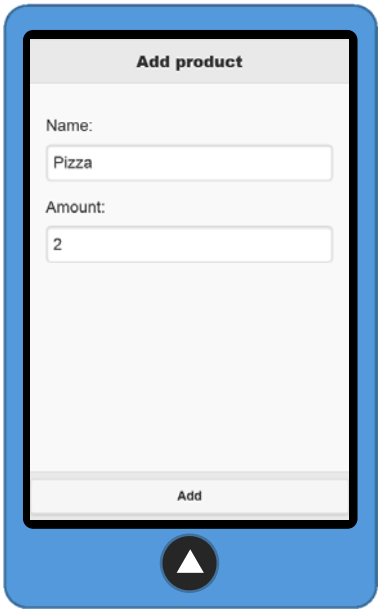

Figure 20: Screenshot of the AddProductView (Mobile)

5.1.5 Tests

The unit testing platform for our sample application will be the qUnit framework. It is one of the most common and powerful unit testing frameworks for JavaScript and can be used comfortably with TypeScript as well.

It is important, as always by writing unit tests, that the tests do not depend on or change the global states of the application. When writing unit tests for an application with a structure based on MVVM, testing the ViewModels is an important part, since most of the applications logic is located within them.

It is often beneficial to create test implementations of all services that use platform specific APIs to be able to run tests for all ViewModels on the desired testing platform. This platform is most likely a browser from a development device or a test runner on a continuous integration server.

Platform specific tests can obviously only run on the individual platform and can therefore not easily run on a continuous integration system. Although, it is often possible to find a solution for running platform specific code from a continuous integration server, it is rarely that important. Since platform specific code is located within services and should only abstract some of the platform's APIs, which hardly ever change. Furthermore, the usage of platform specific services should be reduced to a minimum and should never contain logic that could be placed in the common part of the application.

The following sample is a test for the `AddProductViewModel`. It creates instances of service implementations that are designed to work specifically well for testing. Those test implementations can offer additional functionality for testing through exposing some of the internal states.

The messaging service can be handy to simulate the behavior of ViewModels with conditions similar to the conditions when the application is actually running on the user's device.

```
test("AddProductTest", ()=> {
    ShoppingList.Services.ServiceLocator.MessagingService =
    new Services.Messaging.MessagingService();
    var navigationService = new Services.NavigationServiceTest();
    navigationService.NavigationStack = ["addProductView"];

    ShoppingList.Services.ServiceLocator.NavigationService =
    navigationService;

    var addProductViewModel =
        new ShoppingList.ViewModel.AddProductViewModel();

    var list1 = new Model.DataList("ListTest");
    list1.products.push(
        new Model.DataProduct("TestProduct1", "1", false));

    ShoppingList.Services.ServiceLocator.MessagingService.publish(
        Services.Messaging.MessagingToken.SelectedListChanged, list1);

    addProductViewModel.name("TestProduct2");
    addProductViewModel.amount("2");

    addProductViewModel.addProduct();
```

```
        QUnit.ok(list1.products().length == 2);
        QUnit.ok(list1.products()[1].name() == "TestProduct2");
        QUnit.ok(list1.products()[1].amount() == "2");

        QUnit.ok(navigationService.NavigationStack.length == 0);
    });
```

QUnit has a nice testing interface, but can also be run within other test environments like the integrated test runner of Visual Studio or most continuous integration services.

5.1.6 Challenges

Although the development of the two web applications was easier and more comfortable than expected there were some unexpected challenges and problems that had to be solved. The following table will show some positive and negative impressions of the development process.

Positive	Negative
TypeScript has major advantages over JavaScript especially, for developing larger applications. Also the use of object oriented design patterns like MVVM and others can benefit a lot from TypeScript.	TypeScript is still not as powerful and its usage not as comfortable compared to major high level object oriented frameworks like .NET or Java. Especially the "this" pointer's scope of JavaScript that is still present within TypeScript can be bothersome for developers with .NET or Java background.
Good frameworks like jQuery make it easy to manipulate HTML code.	Browser inconstancies can make it more difficult to run the application on different devices.
KnockoutJs is a powerful, reliable and easy to use binding framework.	The amount of UI frameworks is large but none of them seem as useful as most native tools of today's mobile and desktop platforms. Let us see how WinJs turns out, maybe it will change this in the near future.
There are a great number of tools and a large community of web developers. Even the TypeScript community is quite large and active**,** for such a new technology.	Integrating different UI frameworks with completely different concepts can be difficult to manage with shared code.
Great support through Visual Studio for the entire development process.	
NUGET package management makes it easy to manage web libraries as well as others.	

***Table 7**: Positive and negative impressions (Web)*

5.1.7 Benefits of MVVM

When developing web applications there are many advantages when using MVVM. The following list will show some of them.

- MVVM can offer a clean application architecture with defined places for nearly every type of code.
- The possibilities for structuring web applications are endless and it is always difficult for developers to work on already existing source code. With MVVM new developers can adopt more easily especially, if they already have experience with MVVM.
- Data bindings provide an easy, flexible, robust and unified way to update the HTML code.
- Views can be easy replaced or extended.
- Multiple Views with the same ViewModel are always in sync without additional syncing mechanisms.
- With the use of a messaging service the data flow within the application is always well defined and errors can be easily found.
- Services provide a clean concept for common functionality and are easy to maintain, replace or extend.

5.1.8 Results and conclusions

The following numbers represent the amount of code used by the individual components of our sample application.

Projects	**Model**[1]	**Views**[2]	**Services**[3]	**ViewModels**[4]	**setup**[5]	**overall**
Core	44	0	60	110	42	212
Desktop	0	55	32	0	16	103
Mobile	0	52	30	0	16	98
					Sum:	413

***Table 8**: Lines of code used by the sample application (Web)*

1) The Model includes all data classes
2) Views contain markup code (HTML) and TypeScript/JavaScript code that is used for visualization
3) Services and the implementation of the messenger
4) Code inside ViewModels
5) Code used to initialize the application

The above numbers show that the individual implementations for mobile and desktop use a similar amount of code. The data classes (Model) the ViewModels and a part of the services are used across all implementations while the Views are implemented once for every platform.

The numbers below are calculated using the lines of code of the table above and show the distribution of the code across the individual platforms. The column for shared code shows the numbers of one individual implementation and ignores the line numbers of the other one.

Projects	overall	shared code
Core	51,33%	
Desktop	24,94%	67,30%[1]
Mobile	23,73%	68,39%[2]

***Table 9**: Distribution of code across the individual implementations (Web)*

1) The Value is computed with: $\frac{Desktop}{Core+Desktop}$
2) The Value is computed with: $\frac{Mobile}{Core+Mobile}$

As can be seen in the table above, about 70% of the code can be shared across platforms. This value can likely be achieved for real world implementations as well. These numbers could be increased even further when concepts for sharing HTML code get introduced.

5.2 MVVM with .NET

MVVM pattern appeared the first time on the .NET platforms Silverlight, and WPF and has not changed much since its beginning. The adoption of MVVM is wide and many of the great .NET applications today benefit from the structure of MVVM.

The following application was developed to show how MVVM can be used to enhance .NET applications on different platforms. The sample application will use MVVM for extensive cross platform development to show how MVVM can help to develop applications for multiple platforms with a large amount of shared code.

The developed sample application has similar functionality as the web application but was extended with full localization capabilities and the ability to store the state of the application across user sessions.

The full source code that is described in this section is available on GitHub and licensed under the MS-PL.

https://github.com/Lupin1st/ShoppingList.Net

5.2.1 Used frameworks and tools

For the implementation of this sample application Visual Studio was used as the main development environment. Without a doubt it provides the most powerful and advanced features for developing .NET applications. Especially, when used with the ReSharper extension.

The MVVM framework used to develop this application is MVVM Cross, due to the following reasons:

- It is the only framework that provides binding features for non-Microsoft platforms
- It is extremely flexible and has an extendable architecture
- It has a lightweight application design because of its plugin system
- There are lots of plugins for most common tasks
- There is great support from a large community

MVVM Cross provides more features than most other MVVM frameworks especially, when used for cross platform development. The MVVM Cross framework and its plugins are extensively used in all parts of this sample application.

The core of MVVM Cross, as well as its plugins can be installed as NuGet packages. The following list shows the plugins, which were used to develop this sample application.

- **Messenger**: a messenger implementation
- **Localization**: for localizing the application (only used for the Android implementation)
- **File**: for reading and writing files to the local file system
- **Json**: for serializing and deserializing objects to and from strings

5.2.2 Structure

Within Visual Studio the application is separated into multiple projects. A “Core” project, which is a portable class library is used to contain all platform independent code. Furthermore, there is one additional project for Windows Desktop (WPF) and one for Android. Applications for Windows RT platforms have a special structure, since the implementation makes use of the unified Windows platform, introduced with the Update 2 of Visual Studio 2013. The full implementation for Windows Phone and Windows Store needs three additional projects, one shared project and one project for each platform. For the sample application only the Windows Store part will be implemented, since most of the UI code is placed in the shared project. Therefore, a Windows Phone application could be added with little effort.

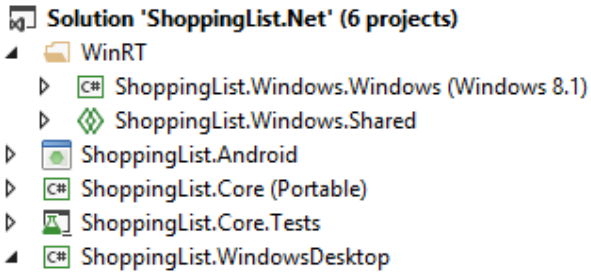

***Figure 21**: Screenshot of the project structure within Visual Studio (.NET)*

5.2.3 Implementation (Core)

The core implementation contains all data classes, ViewModels as well as most of the applications UI independent logic. Furthermore, it provides interfaces for platform specific code (services) and contains the resource files used for localization.

Model

The model consists of the two data objects `DataList` and `DataProduct` and the `DataModel` class itself. All three objects extend the base class `MvxNotifyPropertyChanged`.

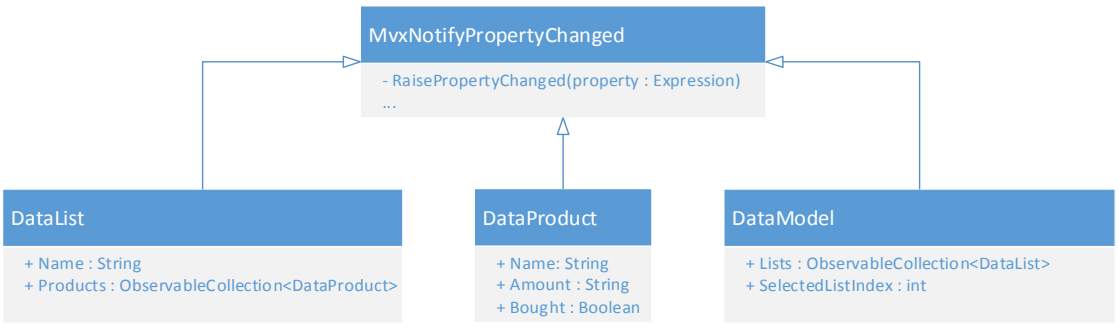

***Figure 22**: Inheritance structure of data classes (.NET)*

Services

The core project contains service interfaces for all platform specific code that needs to be called from the core project, as well as interfaces for individual functionalities that are implemented in the core project itself. For convenience the implementation of those interfaces have strict naming rules. All services end with the "Service" keyword plus the name of the platform. This is later used to register all services as singleton instances with only a view lines of code.

StateService

The StateService is responsible for preserving the state of the application across application sessions. It has two properties that hold the current state of the application and two methods to store and restore these properties to the local file system.

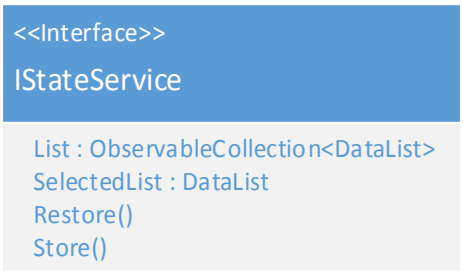

***Figure 23**: UML diagram of the IStateService interface*

Unlike most services the StateService is implemented only once in the core project. For better extendibility, consistency and use with the MVVMCross IOC system it has an interface like all other services. To simplify testing there is an implementation of the StateService specifically for testing.

Restore

The Restore method reads the file that includes the last state of the application from the local file system using the File plugin of MVVM Cross. If the file exists, then it uses the MVVM Cross Json plugin to deserialize the content to an object of the type `DataModel`. If the file was never written to the file system, it creates a new `DataModel` object and fills it

with sample data. Now the loaded state can be accessed from everywhere within the application using the two properties of the StateService. The last view lines of the `Restore` method publish the values of the same two Properties using the MVVM Cross messenger, to make sure that all parts of the application have the current reference to the state objects.

Store

The `Store` method serializes the local `DataModel` variable and saves its content to the devices file system.

NotificationService

The NotificationService is a typical service that abstracts device specific functionality to be used from within the core project. It provides only one method to show a simple message box with two buttons on the targeted platform.

***Figure 24**: UML diagram of the INotificationService interface*

Since some ViewModels will need the NotificationService within their constructor, it was necessary to implement a design-time dummy for this service that is only used when an XAML application is in design-mode.

Application setup and ViewModelLocator

The MVVM Cross framework provides a built-in concept for ViewModel location. The integrated ViewModel locator uses the IOC system of MVVM Cross to dynamically create ViewModels when needed. This leads to lots of performance and convenience improvements over a ViewModel locator that creates all ViewModels at the start of the application.

However, at its default implementation it is not compatible with design-time data on Windows platforms. Therefore, the default implementation was slightly changed for the sample application. This was done by overwriting the `MvxDefaultViewModelLocator` class with an implementation that holds an additional property for each ViewModel. For Windows platforms an additional `ApplicationContext` class was created that holds a reference to the ViewModelLocator and provides methods for both design-time and run-time initialization. An instance of this class was added as an application resource to the App.xaml file. With that change an instance of the `ApplicationContext` will be created when the application starts or when the designer of Visual Studio performs its design-time initialization. With that the `ApplicationContext` instance can be used to access the `ViewModelLocator` and its ViewModels from each View on windows platforms. At run-time only the run-time initialization will be used on all platforms.

The initialization code initializes all MVVM Cross components and registers all used services as a singleton instance with the help of the IOC system from MVVM Cross.

To restore the state of the last application session and to save the state when the application closes, the individual platform events had to be used, since these events behave different on every platform. The `Store` and `Restore` methods of the StateService are used to read and write the application's states to and from the local file system.

ViewModels

With MVVM the ViewModels are always the central part of the application and connect the individual Views of each platform with the common data Model. Since ViewModels, most of the time, have similar properties and methods it is recommended to structure the code of each ViewModel in the same way to increase maintainability.

ViewModels use services for common tasks and do not contain platform specific code. However, sometimes it is required to include some lines of code that are not relevant to all platforms. In that case these lines could be added to the ViewModel with caution. When adding platform specific code to ViewModels, it is important to keep in mind that this might lead to different behavior on each platform. Furthermore, this could drastically reduce overview of the application's behavior and complicate testing. Due all these reason introducing platform specific code should be avoided, if possible.

The ViewModels communicate with each other, as well as with services by using the messenger provided by MVVM Cross.

For better testing capabilities and overview of which services are used by which ViewModel, all consumed services are added as constructor parameters (constructor dependency injection) to the ViewModel.

The base class for all ViewModels

For common functionality used in ViewModels the class `ViewModelBase` was introduced as the base class for all ViewModels.

It provides the `BackCommand` that is needed in every ViewModel for back navigation. For consistency this command will replace the devices default back navigation mechanism even if no different behavior is required.

Furthermore, the `ViewModelBase` contains a `TextSource` property that is used for the MVVM Cross multi-language support. This property is used only by the Android application. The multi-language features will be described in detail later on.

ListsViewModel

The `ListsViewModel` is the first ViewModel that is created after the start of the application. This ViewModel provides functionality for the View to visualize the different shopping lists. Moreover, it offers a mechanism to select one shopping-list.

Properties

`ObservableCollection<DataList> Lists:`
This collection will always contain all current shopping-lists of the application and will be updated when another part of the application changes the collection. The Views will use this property to bind it to the items-source of the platform's list control.

`DataList SelectedList:`

This property holds a reference to the currently selected list. When the property's setter is called the property sends a message containing its new value to notify all other ViewModels. After the value got successfully changed the `NavigationService` is used to navigate to the ProductsViewModel. This navigation will have different behavior on each platform.

Commands and actions
The ListsViewModel does not have any command and therefore, also no actions.

Messaging actions
The ListsViewModel registers for the two messages `ListsChangedMessage` and `SelectedListChangedMessage` and sets the corresponding local property to the value received from the message.

Constructor
Like all ViewModels the constructor registers for messages and initializes the local properties. In this case it fetches the shopping-lists, as well as the selected list from the StateService.

ProductsViewModel

The `ProductsViewModel` provides data for the `ProductsView` and is responsible for providing functionality to display the products of the currently selected list, as well as adding a new product and removing all bought products.

Properties

`DataList List:`
The `ProductsViewModel` provides a property for the currently selected shopping-list that is used by the Views to display the right products.

Commands and Actions

The `AddProductCommand` navigates to the `AddProductViewModel` to let the user create a new product.

```
public MvxCommand AddProductCommand { get; private set; }
private void AddProductAction() {
    ShowViewModel<AddProductViewModel>();
}
```

The `CleanupCommand` shows a message box and removes all bought products, if the "OK" button was clicked by the user.

Since it uses the async method `ShowMessageBox` from the NotificationService, this command cannot be of the type `MvxCommand` that is provided by MVVM Cross and used for all non-async commands. Since MVVM Cross does not provide an implementation of the ICommand interface that is able to call async methods inside its action, the `AsyncCommand` was introduced. The implementation of the `AsyncCommand` will allow us to call any async method inside the command's action and provides functionality similar to the implementation of the `MvxCommand`.

After displaying the message box the product is being removed from the list. It is important that this is done while running on the UI-thread. Changing products while running on a worker thread can result in a problem because this action will automatically trigger

the UI to visualize the new products. Changing UI-controls is often only possible when running on the UI-thread and will likely throw an exception otherwise. The `RequestMainThreadAction` from the cross platform dispatcher that is provided by MVVM Cross allows us to ensure that the later part of the action will run on the platform's UI-thread.

```
public AsyncCommand CleanupCommand { get; private set; }
private async Task CleanupAction(object parameter) {
    var result = await _notificationService.ShowMessageBox(
        LocalizationResources.DeleteBought,
        LocalizationResources.CleanupMessageBody);

    if (result == QuestionResult.Ok) {
        var productsToDelete = List.Products.Where(
            product => product.Bought).ToArray();

        Dispatcher.RequestMainThreadAction(() => {
            foreach (var product in productsToDelete) {
                List.Products.Remove(product);
            }
        });
    }
}
```

Lastly the `ProductsViewModel` overrides the `BackAction` from the base class (`ViewModelBase`) to clear the selected list when the user navigates to the previous View. This is only relevant to Android, since Windows Store and Windows Desktop, display both `ListsView` and `ProductsView` within one page or window. The `Close` method is part of the navigation concept of MVVM Cross and will be described later.

```
protected override void BackAction() {
    _messenger.Publish(new SelectedListChangedMessage(this, null));
    Close(this);
}
```

AddProductViewModel

The `AddProductViewModel` is responsible for the `AddProductView` and offers functionality to add a new product to the current shopping-list.

Properties

The `AddProductViewModel` has three properties. One represents the currently selected list (`DataList List`). The other two properties (`Name` and `Amount`) are of type `String` and are used to temporarily store information about the new product. They both will be data bound to the View's text editing controls.

Commands and Actions

The `AddProductViewModel` has only one command, which creates a product with the entered values for name and amount and adds the new product to the current shopping-list. The command also has a `CanExecute` method that is used to determine if the application is in a state where the command can successfully be executed. That is the case, if the entered name of the product has a minimum length of two characters. To tell the command that a valid state is reached, the command's `RaiseCanExecuteChanged` method is called every time the `Name` property of this ViewModel has changed.

```
public MvxCommand SaveProductCommand { get; private set; }
private void SaveProductAction() {
    List.Products.Add(new DataProduct {
        Amount = Amount,
        Name = Name
    });

    Amount = "";
    Name = "";
}

private bool SaveProductCanExecute() {
    return Name != null && Name.Length >= 2;
}
```

Localization

MVVM Cross includes functionality to localize the application by using JavaScript object notation (JSON) files that store the translations as pairs of key and value. This however, has two disadvantages. The first one is that editing JSON is not the most comfortable way to manage string resources, unlike for example the resource file editor of Visual Studio. The second disadvantage is that the localization plugin of MVVM Cross does not work at design-time.

Furthermore, MVVM Cross provides interfaces for most of its internal functionality and it is easy to replace individual parts of the framework. To replace the default JSON localization plugin the `IMvxTextProvider` interface is used [57]. The new `ResxTextProvider` is only used for the Android application. For all XAML applications the resource class generated from Visual Studio is used directly as a binding source for localizing the UI controls.

Navigation

For writing native looking cross platform applications, the navigation concept is often one of the most challenging parts to implement. The application should call all navigational methods from within the ViewModels, yet individual code for each platform is not wanted there. However, MVVM Cross provides a solid concept for navigating between pages. The simple default navigation implementation, is convenient for small screen applications, where one View correlates with one page. For applications that are designed for devices with large screens the default implementation would not be appropriate. Luckily MVVM Cross provides Individual classes for all platforms that can be overwritten to get customized navigation behavior. For this sample application an alternative implementation of the navigation concept was realized for the Windows Store and the Windows Desktop implementation, since both are required to work well with larger display sizes. When supporting a platform, which supports many different display sizes, different implementations can be determined statically by providing individual application packages, dynamically during the startup of the application or even every time the display size changes.

5.2.4 Implementation (Windows Unified)

The implementation for the unified application platform contains two projects. One shared project that contains files that are used for all platforms. The other one is the platform specific Windows Store project. As already mentioned, for this sample application only the Windows Store project was implemented, since extending the application with and additional Windows Phone project would be straight forward.

Due to the usage of the Windows unified architecture most of the files can be located within the shared project and used for both, the Windows Store and the Windows Phone

implementation. For our example application the Windows Store project contains only two files, the implementation of the presenter and the MainView.

Services

The implementation of the `INavigationService` uses the `MessageDialog` from the common WinRT and can be placed in the shared project.

Views

The shared project contains one View for each ViewModel. In addition to that, the Windows Store implementation includes an additional View that has no own ViewModel. It simply uses the Views from the shared project and displays them on one page. That is the reason why the Windows Store project needs a different navigation concept. This job is done by the presenter class, which navigates only to the `MainView` instead of all other Views.

MainView

The `MainView` has no own ViewModel and is used to place the three shared Views on the screen. The `ListsView` and `ProductsView` are placed next to each other on the screen while the `AddProductView` is hidden and shown as a flyout by the press of a button.

Flyouts are a UI control available only in Windows Store applications and therefore, cannot be placed within the shared project. However, Flyouts are directly associated with a button control and that is why we located the two buttons “new” and “cleanup” within the `MainView` of the Windows Store project instead within the `ProductsView`. Through this method we were able to use flyouts for the Windows Store application, yet also include the `ProductsView` in the shared project to maximize code sharing. The following code is the `MainViewModel`’s markup without the parts relevant only for the design.

To let the UI controls know which ViewModel is responsible for them the `DataContext` of the root UI control needs to be set. All children of the control will inherit the same `DataContext`.

For localization there is an application resource named `Localization` that holds a reference to the resource dictionary that includes all translations.

```
<Grid>
  <views:ListsView/>
  <Grid>
    <views:ProductsView/>
    <Grid DataContext="{Binding Locator.ProductsViewModel,
          Source={StaticResource App}}">
      <Button Command="{Binding AddProductCommand}"
              Content="{Binding Strings.New,
              Source={StaticResource Localization}}">
        <Button.Flyout>
          <Flyout x:Name="FlyoutAddProduct">
            <views:AddProductView />
          </Flyout>
        </Button.Flyout>
      </Button>
      <Button Command="{Binding CleanupCommand}"
              Content="{Binding Strings.DeleteBought,
              Source={StaticResource Localization}}"/>
    </Grid>
  </Grid>
</Grid>
```

The following screenshot shows the `MainView` after clicking the “New” button.

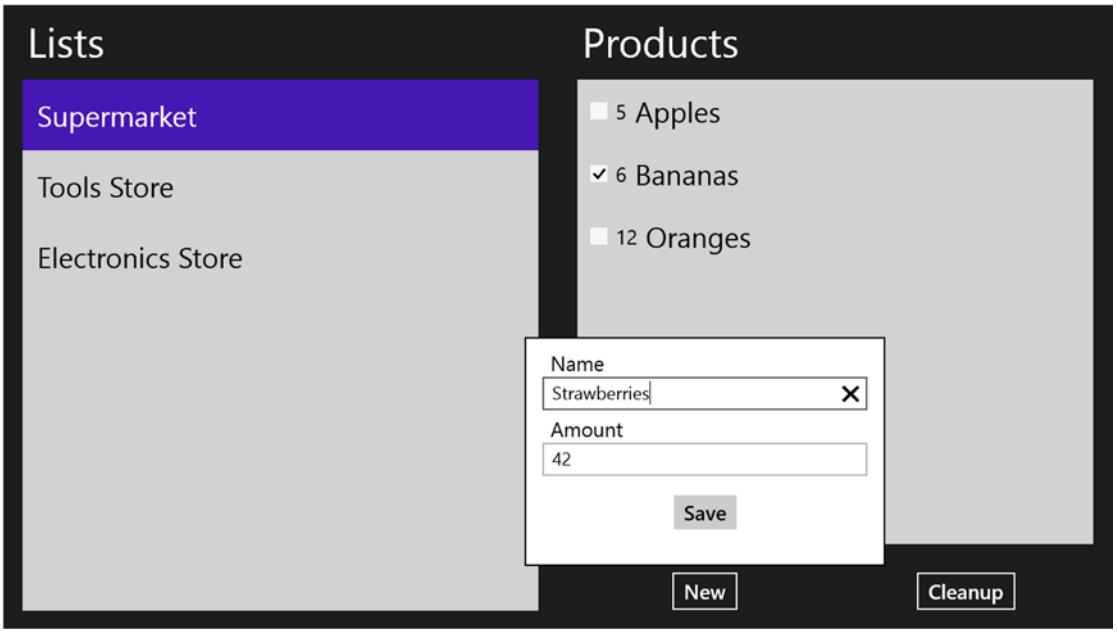

***Figure 25**: Screenshot of the main page (Windows Store)*

ListsView

The `ListsView` only contains one `ListBox` control that displays all shopping-lists. It binds the `SelectedItem` property to the ViewModel’s `SelectedList` property.

```
<Grid>
  <ListBox ItemsSource="{Binding Lists}"
           SelectedItem="{Binding SelectedList, Mode=TwoWay}">
    <ListBox.ItemTemplate>
      <DataTemplate>
        <TextBlock Text="{Binding Name}"/>
      </DataTemplate>
    </ListBox.ItemTemplate>
  </ListBox>
</Grid>
```

ProductsView

For the Windows Store implementation the `ProductsView` only contains one `ListBox` that shows the products of the selected list.

```
<ListBox ItemsSource="{Binding List.Products}">
  <ListBox.ItemTemplate>
    <DataTemplate>
      <StackPanel>
        <CheckBox IsChecked="{Binding Bought, Mode=TwoWay}" />
        <TextBlock Text="{Binding Amount}"/>
        <TextBlock Text="{Binding Name}"/>
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

AddProductView

The AddProductView provides UI for creating a new product. When looking at the `TextBlock` for the name of the product, there is a `TextChanged` event handler registered in addition to the binding.

The reason for that lies with the XAML `TextBlock`'s update functionality. Only when the `TextBlock` loses focus the update method is called. However, for real-time validation of the save button's enabled state, it is necessary to call the update function for every change of the inserted text. A second possible solution for this problem would be the usage of MVVM Cross' binding features that are also available for Windows platforms via an individual NuGet package.

```
<StackPanel >
  <TextBlock Text="{Binding Strings.Name,
             Source={StaticResource Localization}}" />
  <TextBox Text="{ Binding Name }"
           TextChanged="TextBoxName_OnTextChanged" />

  <TextBlock Text="{Binding Strings.Amount,
             Source={StaticResource Localization}}" />
  <TextBox Text="{ Binding Amount, Mode=TwoWay }" />

  <Button Command="{Binding SaveProductCommand}"
          Content="{Binding Strings.Save,
          Source={StaticResource Localization}}"
          HorizontalAlignment="Center" />
</StackPanel>
```

Navigation

The navigation concept for Windows Store applications is exceptionally difficult to combine with applications that use pure page navigation. Windows Store applications usually have only a few pages. Furthermore, they often hide Views within `Flyout` controls, which would be separate pages on platforms with smaller screens.

For this sample implementation it was satisfying enough to overwrite the default presenter in a way that it always shows the `MainView`, instead of the other three Views, and lets the View itself open the `Flyout`. This however, has one big disadvantage. The presenter does not know anything about the `Flyout`s and cannot open or close them. Instead the View itself has to manage them.

It is often necessary to manage the navigation behavior from within the ViewModel instead of the View. This is not possible with the current implementation of the navigation mechanism. To control the navigation within the ViewModels a more complex implementation of the presenter classes could be used.

5.2.5 Implementation (Windows Desktop)

The implementation for Windows Desktops is a WPF application that uses, like the Windows Store implementation, XAML as a markup language. Although both markup languages look very similar, their binaries are not compatible and that is why Views cannot be shared between the two platforms.

Views

The Views of the Windows Desktop implementation look nearly identical to the Windows Store implementation.

The following screenshot shows the ListsView and the ProductsView arranged next to each other by the presenter.

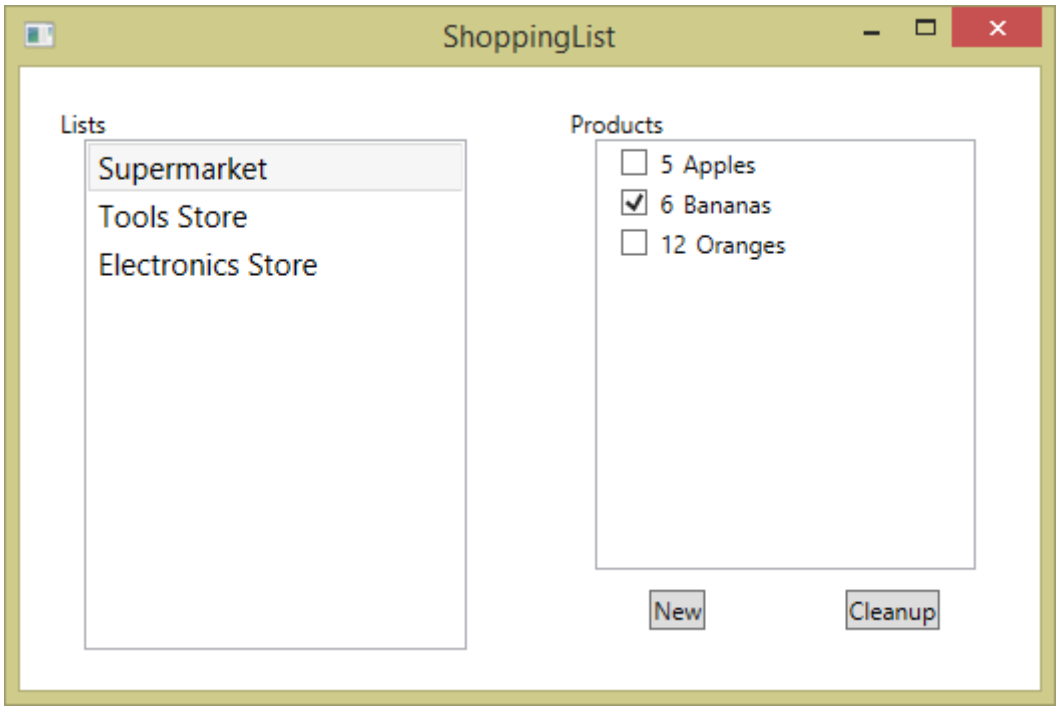

***Figure 26**: Screenshot of the main page (Windows Desktop)*

The AddProductView is displayed as a single View inside a new dialog window.

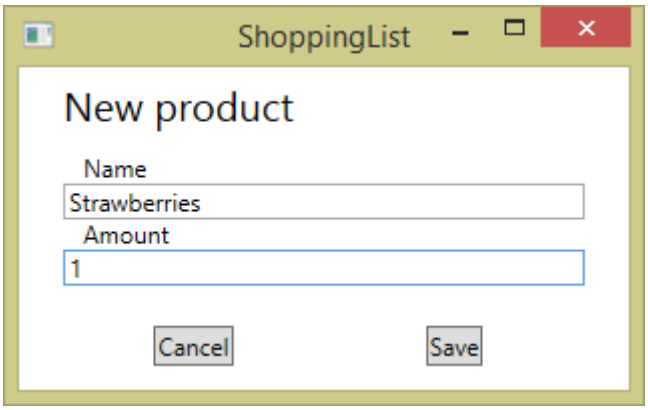

***Figure 27**: Screenshot of the AddProductView (Windows Desktop)*

Navigation

The navigation mechanism of the Windows Desktop implementation uses an implementation of the MVVMCross presenter similar to the one used for the Windows Store project. With the difference that it does not use an additional View to place the `ListsView` and the `ProductsView` next to each other on the screen. Instead the implementation of the presenter adds both Views to the applications root window. All other Views, except the `ListsView` and the `ProductsView`, will be opened as new dialog windows.

5.2.6 Implementation (Android)

The implementation for Android devices uses the Xamarin.Android platform to provide the features of the .NET platform. The Xamarin platforms are able to consume portable class libraries and therefore, can use our Core project.

Services

Like the other implementations, the Android application provides an implementation of the `INotificationService`. When writing applications for the Xamarin.Android platform it is often required to have a reference to the currently active activity, since it is needed for many of the platform's APIs. Placing most of the code inside ViewModels and Services can make it difficult to get hold of this reference. Therefore, MVVM Cross provides an implementation for the `IMvxAndroidCurrentTopActivity` interface that always holds a reference to the top activity and is available as a singleton managed by the IOC system of MVVM Cross.

Views

The Views of the Android implementation use the Android XML files as a markup language. Different to XAML, the Android XML markup language does not provide a native binding framework. Luckily MVVM Cross provides a solid and full featured binding framework, which can be used in a similar way.

ListsView

The `ListsView` of the Android implementation uses the MVVM Cross' binding framework to communicate with its ViewModel. The template for the shopping-list items is located in a separate Android XML file with the path `"@layout/item_list"`.

```
</LinearLayout>
  <Mvx.MvxListView local:MvxBind="ItemsSource Lists;
                                  ItemClick SelectedListChangedCommand"
                   local:MvxItemTemplate="@layout/item_list" />
</LinearLayout>
```

The following template will be used to create the shopping-list's items. The context for the binding framework is the corresponding DataList item.

```
<LinearLayout>
  <TextView local:MvxBind="Text Name" />
</LinearLayout>
```

The following screenshot shows the `ListsView` with the three default shopping-lists generated by the StateService.

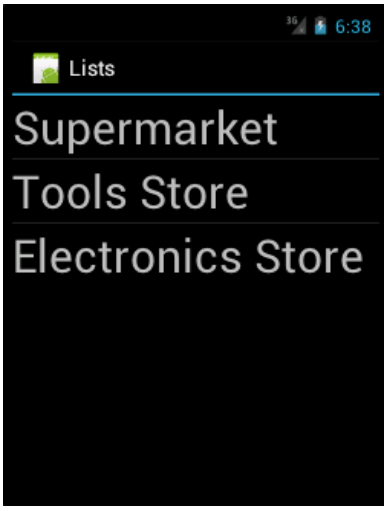

***Figure 28**: Screenshot of the ListsView (Android)*

ProductsView

The `ProductsView` contains an `MvxListView` to show all products from the selected list, as well as two buttons. The `MvxLang` attribute is added by MVVMCross and enabled the support of localization. For this sample application the same RESX files are used, as for the Windows platforms' implementation, to provide the string resources for each language.

```
<LinearLayout>
  <Mvx.MvxListView local:MvxBind="ItemsSource List.Products;"
                   local:MvxItemTemplate="@layout/item_product" />
  <LinearLayout>
    <Button local:MvxBind="Click AddProductCommand"
            local:MvxLang="Text New" />
    <Button local:MvxBind="Click CleanupCommand"
            local:MvxLang="Text DeleteBought" />
  </LinearLayout>
</LinearLayout>
```

The following Android XML code is used as a template for the items of the `MvxListView` control.

```
<LinearLayout>
  <CheckBox local:MvxBind="Checked Bought; Text Amount + ' ' + Name" />
</LinearLayout>
```

Finally, following screenshot shows the `ProductsView` after clicking the first shopping-list on the ListsView.

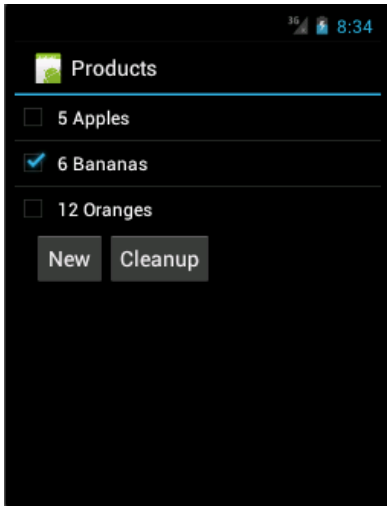

***Figure 29**: Screenshot of the ProductsView (Android)*

AddProductView

The `AddProductView` looks similar to the corresponding implementations of the two Windows platforms. It offers two `EditText` controls as well as two `Buttons`.

```
<LinearLayout>
  <TextView local:MvxLang="Text Name" />
  <EditText local:MvxBind="Text Name" />

  <TextView local:MvxLang="Text Amount" />
  <EditText local:MvxBind="Text Amount" />

  <LinearLayout android:orientation="horizontal">
    <Button local:MvxLang="Text Cancel"
            local:MvxBind="Click BackCommand" />
    <Button local:MvxLang="Text Save"
            local:MvxBind="Click SaveProductCommand" />
  </LinearLayout>
</LinearLayout>
```

The following screenshot shows the `AddProductView` after entering some text into the two `EditText` controls. Similar to the other implementations, the save button will not be enabled until the `EditText` control, for the name of the shopping-list, contains text that is at least two characters long.

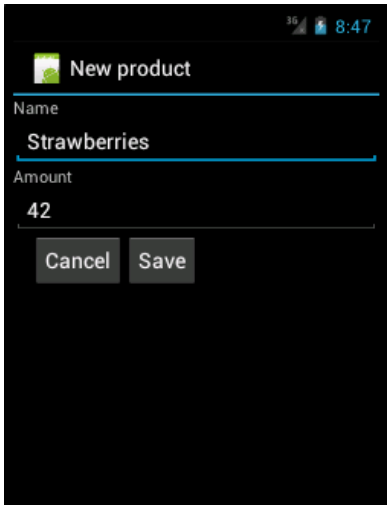

***Figure 30**: Screenshot of the AddProductView (Android)*

Navigation

The navigation concept for the Android implementation is more generic than the one for the Windows Store or Windows Desktop implementation. Every View is shown as a single Android page. Therefore, it is not necessary to override the default presenter implementation of MVVMCross. If a better scaling UI for Android devices with larger screens is required, a different implementation for the presenter would be needed.

5.2.7 Tests

Good testing characteristics are one of the major benefits of MVVM. The loose coupling between the ViewModels and the UI makes it particularly easy to test all aspects of the ViewModel, which contains most of the applications logic.

The concept of putting all common helping mechanisms into individual Services makes it easy to replace them with implementations that can run on the testing platform and provide specialized functionality for unit tests.

The test implementation uses a specialized implementation of the `IStateService` interface to offer better testing properties than the implementation designed for the use at run-time. Such test implementations can be very useful for services that abstract device specific functionality like showing a message box or other things. In such case the test implementation of the service can simulate user input or sensor data and make the testing process a lot easier. Many otherwise needed UI tests can be replaced by unit tests that are more flexible, easier to create and maintain, and will run a lot faster on the testing environment.

For .NET there are multiple unit testing frameworks available and many of them run well from within Visual Studio and on continuous integration platforms. For the following unit test the Microsoft unit testing framework was used.

Testing ViewModels

The `AddProductTest` unit test checks some functionality of the `AddProductViewModel` by setting the `Name` and `Amount` properties and executing the `SaveProductCommand` exactly the same way the user would, when using the UI.

```
[TestMethod]
public void AddProductTest() {
    var messenger = new MvxMessengerHub();
    var stateService = new StateServiceTest();
    var addProductViewModel =
        new AddProductViewModel(messenger, stateService) {
        Name = "NewProduct", Amount = "1"
    };

    addProductViewModel.SaveProductCommand.Execute();

    var list = stateService.SelectedList;
    Assert.AreEqual(1, list.Products.Count);
    Assert.AreEqual("NewProduct", list.Products[0].Name);
    Assert.AreEqual("1", list.Products[0].Amount);
}
```

5.2.8 Challenges

The above seen cross platform implementation using .NET and MVVM, provides lots of benefits. However, not all parts of the application were implemented without mastering some challenges first. This section will describe the most challenging parts of the development process to better understand the complexity of cross developing applications with the used tools and frameworks.

Our sample application was built on top of MVVM Cross and uses many of the framework's features. The MVVM Cross framework is very well designed in respect to extendibility and modularity, and provides more features than most other MVVM frameworks. It is very powerful, yet not always easy to use. MVVM Cross uses many modern programming paradigms that the developers should be familiar with. It is probably not the best framework for developers starting with MVVM development. For beginners frameworks like MVVM Light or the Simple MVVM Toolkit are better suited. The documentation of MVVM Cross is good, but does not cover every aspect of the framework. However, experienced developers will likely master any challenges that come with using MVVM Cross by looking at its clean and well-structured source code.

Another challenging part was that different platforms often have different ways to load resource files and have different concepts for localization. MVVMCross provides a plugin for resource loading as well as localizing strings. The problem was that MVVM Cross' default implementation does not work well at design-time. All Windows platforms come with a great designer integrated in Visual Studio and work well with the external design environment Blend. Providing design time data for the designer has many benefits for designers as well as developers. Supporting design-time data can only be achieved if the setup of the application works under design-time conditions in a similar way as at runtime. Therefore we had to change the default implementation of MVVM Cross to support design-time data and localization inside Visual Studio and Blend.

Different platforms have different application models and therefore, a different application lifecycle. It is important, especially for platforms like Android and Windows Store to be familiar with the platform and its expected application behavior. That means how and when what parts of the application is being garbage collected and where to store and restore the state of the application, as well as other individual application behavior. This can be a tough challenge, particularly when dealing with cross platform development.

A further challenge was that the used platforms support many different screen sizes and different concepts for navigating through each application. For example the Windows Store platform uses Flyout controls to show content (Views) while an implementation for small Android devices opens each View on a dedicated page. It can be challenging to implement a solid navigation mechanism that can be used from within the ViewModels and woks well on every platform. The navigation concept of MVVMCross can help to create such implementations and can be configured to work well on each supported platform.

Additionally, our sample application provides the same features for all platforms. Usually, when developing larger applications it is often required to use different platform specific features. Since not all features are supported by all platforms it can be a real challenge to add those features and still keep a consistent software architecture.

5.2.9 Benefits of MVVM

MVVM has many benefits when developing .NET applications. Most of these benefits, sometimes even more apply for cross platform development as well. The following list contains some of the many benefits that the MVVM pattern contributes to our sample application.

- The Loose coupling of Views and ViewModels make it easy to switch Views
- MVVM is widely used by many developers and can be learned easily using many different learning resources from the web
- A unified data flow makes it easy to maintain and add additional parts to the application
- Good MVVM frameworks can be used to easily add MVVM to an existing application
- Data bindings are an important part of all XAML languages and can be exploited best with MVVM
- When developing applications that do not use data bindings it is often hard to sync data within the application, when using MVVM this problem does not exist
- Loose coupling of the application's components makes it easy to replace individual parts
- When using services to encapsulate common functionality, it is easy to reuse them for other applications
- ViewModels as well as services are easy to unit test, due to their loose coupling to the rest of the application. Even if the testing environment is not the same as the run-time environment

5.2.10 Results and conclusions

This section will summarize and interpret the results of our sample application in terms of code distribution across the individual projects. It will give conclusions about how much lines of code were needed in each project and what parts of the application use the most lines of code. Keep in mind that each line of code is not equally complicated to write and maintain. That is why the results below are only guidance values and may not reflect the real effort to create and maintain the application.

The following table shows the numbers of lines that had to been written to create our sample application. The numbers do not include empty lines of code, `usings` for C# code, XML namespaces as well as automatically generated code.

Projects	Model[1]	Views[2]	services[3]	VM[4]	setup[5]	transl.[6]	others[7]	overall
Core	70	0	140	240	36	24	69	509
Win. Shared	0	93	13	0	82	0	0	188
Win. Store	0	64	0	0	0	0	0	64
Win. Phone[8]	0	50	0	0	0	0	0	50
Win. Desktop	0	167	13	0	88	0	0	268
Android	0	156	31	0	57	0	0	244
							Sum:	1323

***Table 10**: Lines of code used by the .NET sample application*

1) The Model includes all data classes
2) Views contain markup code (XAML for Windows platforms and Android XML for Android), C# code from the code behind files, as well as code used to place the views on the screen (pre-senter)
3) Services and message classes
4) Code inside ViewModels
5) Code used to initialize the application including saving and restoring the application's states
6) String resources that are used to translate the application (one line of code for each string and language)
7) All other code that is not automatically generated
8) The values for the Windows Phone application are estimated values based on the experience gained from other platforms

Let us take a closer look at the above numbers to see how the code is distributed within our sample application and what the differences between each platforms are. The code that is shared between all platforms and located inside the Core project makes up the largest part and covers all code categories with the exception of the Views, which are always platform specific.

The biggest part of the platform specific code is needed for Views, services and code used to initialize the application. The services part is relatively small for our sample application but will likely be larger for real world applications. MVVMCross is the main reason for the small numbers in the service part, since it offers lots of basic device specific services by default.

The amount of code used to create Views is similar on all targeted platforms, since all of them use an XML based markup language, which is similar to use.

The amount of code inside the individual Windows Store project, as well as the estimated number of code inside the Windows Phone project is quite low since all services and most of the Views are located inside the common Windows Shared project.

The above numbers can be used to create statistics to show the distribution of code across the application's components. For cross platform development there are often numbers used to describe the amount of shared code, which has to be written only once, yet runs on multiple platforms. Since the amount of supported platforms should not be relevant to this numbers, the values are often not calculated like the values of the first

two columns of the table below. Instead the values are calculate like the two columns on the right side of the table below.

The setup code can be seen as a static part of the application since it will not dramatically increase, when crating larger applications. Therefore, the three columns on the right are calculated by ignoring the code used for initializing the application.

Projects	overall	overall (without setup)	shared code (without setup)	
Core	38,47%	44,62%		
Windows Shared	14,21%	10,00%		
Windows Store	4,84%	6,04%	73,56%[2]	88,08%[4]
Windows Phone[1]	3,78%	4,72%	75,20%[2]	90,44%[4]
Windows Desktop	20,26%	16,98%	72,43%[3]	
Android	18,44%	17,64%	71,67%[3]	

***Table 11**: Distribution of code across the individual projects (.NET)*

1) The values for the Windows Phone application are estimated values based on the experience gained from other platforms
2) The Value is computed with $\frac{Core}{Core+Windows\ Shared+P}$, where P is the number of lines inside the platform specific project
3) The Value is computed with $\frac{Core}{Core+P}$, where P is the number of lines inside the platform specific project
4) The Value is computed with $\frac{Core+Windows\ Shared}{Core+Windows\ Shared+P}$, where P is the number of lines inside the platform specific project

In comparison to other cross platform solutions that have values of nearly 100% for overall shared code, the numbers of this sample application seem very low. However, one needs to keep in mind that applications with such high values of shared code will most likely do not have the high performance, good usability and high flexibility that this sample application has. When looking at an application for a single platform about 70% to 75% of its code is located inside the shared project. With our sample application we tried to cover lots of the key problems that have to be solved for real world applications, it is likely that similar numbers can be achieved by real world applications as well.

5.2.11 Extend for more platforms

The sample application targets Android, Windows Desktop (WPF) and the Windows Store platform. Since MVVMCross can be used on many more platforms it is easy to extend the application to support Windows Phone, iOS, Mac and Silverlight as well. Especially the addition of the Windows Phone platform can be achieved with little effort, since most of the code is already implemented inside the shared Windows project.

Right now is an important time for the .NET platform. The .NET source code becomes more and more open source and its capabilities will likely extend to other platforms in the near future. All these new platforms can then be added to our solution with little effort.

List of acronyms

AJAX	Asynchronous JavaScript and XML
API	Application programming interfaces
CPU	Central processing unit
CSS	Cascading style sheets
DOM	Document object model
GUI	Graphical user interface
IDE	Integrated development environments
IOC	Inversion of control
JSP	Java Server Pages
LINQ	Language Integrated Query
MVC	Model View Controller
MVP	Model View Presenter
MVVM	Model View ViewModel
OS	Operating System
SASS	Syntactically Awesome Style-Sheets
UI	User interfaces
WinRT	Windows RunTime

Bibliography

[1] Refsnes Data, "OS platform statistics from W3School's log files," 18 1 2014. [Online]. Available: http://www.w3schools.com/browsers/browsers_os.asp.

[2] Gartner, "Mobile statistics and forecast," 17 1 2014. [Online]. Available: http://www.gartner.com/newsroom/id/2645115.

[3] Gartner, "Collection of Gartner statistics from 2008 to 2014," 17 1 2014. [Online]. Available: http://www.gartner.com/newsroom.

[4] Statistic Brain, "Mobile Phone App Store Statistics," 23 03 2014. [Online]. Available: http://www.statisticbrain.com/mobile-phone-app-store-statistics/.

[5] Appcelerator, "Aptana Studio - official website," 2014. [Online]. Available: http://aptana.com/. [Accessed 24 6 2014].

[6] JetBrains s.r.o., "WebStorm — The smartest JavaScript IDE," 2014. [Online]. Available: http://www.jetbrains.com/webstorm/. [Accessed 24 6 2014].

[7] Microsoft, "Visual Studio - official website," 2014. [Online]. Available: http://www.visualstudio.com/. [Accessed 24 6 2014].

[8] A. MacCaw, The Little Book on CoffeeScript, O'Reilly Media, 2012.

[9] S. L. Chris Bucket, Dart in Action, Manning, 2013.

[10] Microsoft, "TypeScript Language Specification," 6 3 2014. [Online]. Available: http://www.typescriptlang.org.

[11] Sass, "Sass official website - Start Page," 19 2 2014. [Online]. Available: http://sass-lang.com/.

[12] Less, "Less official website - Start Page," 17 3 2014. [Online]. Available: http://sass-lang.com/.

[13] The jQuery Foundation, "jQuery official website," 2014. [Online]. Available: http://jqueryui.com/. [Accessed 5 6 2014].

[14] The jQuery Foundation, "jQueryUI official website," 2014. [Online]. Available: http://jqueryui.com/. [Accessed 5 6 2014].

[15] The jQuery Foundation, "jQuery Theme Roller," 2014. [Online]. Available: http://jqueryui.com/themeroller/. [Accessed 5 6 2014].

[16] The jQuery Foundation, "jQuery Mobile official website," 2014. [Online]. Available: http://jquerymobile.com/. [Accessed 5 6 2014].

[17] Microsoft Open Technologies, "GitHub - WinJs," 9 04 2014. [Online]. Available: https://github.com/winjs/winjs/wiki/Roadmap.

[18] Microsoft, "TryWinJs : Bindings," 18 04 2014. [Online]. Available: http://try.buildwinjs.com/#binding.

[19] Telerik, "Telerik official website - Start Page," 24 2 2014. [Online]. Available: http://www.telerik.com/.

[20] AngularJs, "AngularJs official website," 2014. [Online]. Available: http://angularjs.org/. [Accessed 9 6 2014].

[21] KnockoutJs, "KnockoutJs official website," 2014. [Online]. Available: http://knockoutjs.com/. [Accessed 25 6 2014].

[22] DurandalJs, "DurandalJs official website," 2014. [Online]. Available: http://durandaljs.com/. [Accessed 25 6 2014].

[23] Microsoft, "WinJs official website - Start Page," 4 4 2014. [Online]. Available: http://try.buildwinjs.com.

[24] BackboneJs, "BackboneJs official website," 2014. [Online]. Available: http://backbonejs.org/. [Accessed 9 6 2014].

[25] Derby, "Derby official website," 2014. [Online]. Available: http://derbyjs.com/. [Accessed 25 6 2014].

[26] Ember, "Ember official website," 2014. [Online]. Available: http://emberjs.com/. [Accessed 25 6 2014].

[27] JsViews, "JsViews official website," 2014. [Online]. Available: http://www.jsviews.com/. [Accessed 25 6 2014].

[28] jQXB, "jQXB Expression Binder wensite on Codeplex," 2014. [Online]. Available: http://jqxb.codeplex.com/. [Accessed 25 6 2014].

[29] Meteor, "Meteor official website," 2014. [Online]. Available: https://www.meteor.com/. [Accessed 25 6 2014].

[30] AngularDart, "AngularDart official website," 2014. [Online]. Available: https://angulardart.org/. [Accessed 9 6 2014].

[31] KnockoutJs, "KnockoutJs mapping plugin," 2014. [Online]. Available: http://knockoutjs.com/documentation/plugins-mapping.html. [Accessed 25 6 2014].

[32] Google, "ChromeOS official website," 2014. [Online]. Available: http://www.chromium.org/chromium-os. [Accessed 25 6 2014].

[33] Mozilla, "FirefoxOS official website," 2014. [Online]. Available: http://www.mozilla.org/en-US/firefox/os/. [Accessed 25 6 2014].

[34] Google, "Android official website," 2014. [Online]. Available: http://www.android.com/. [Accessed 25 6 2014].

[35] Adobe, "PhoneGap official website - Start Page," 2 3 2014. [Online]. Available: http://phonegap.com/.

[36] Amazon, "FireOS developer website - Start Page," 15 2 2014. [Online]. Available: https://developer.amazon.com/appsandservices/solutions/platforms/android-fireos.

[37] BlackBerry Limited, "Blackberry 10 developer website - Start Page," 11 2 2014. [Online]. Available: https://developer.blackberry.com/.

[38] Apple Inc., "Apple developer website," 11 2 2014. [Online]. Available: https://developer.apple.com/.

[39] Canonical Ltd., "Ubuntu official website - Start Page," 7 2 2014. [Online]. Available: http://www.ubuntu.com/.

[40] Microsoft, "Windows Phone developer website - Start Page," 8 2 2014. [Online]. Available: http://dev.windowsphone.com/en-us/develop.

[41] Linux Foundation, "Tizen official website - Start Page," 1 3 2014. [Online]. Available: https://www.tizen.org.

[42] Adobe, "Adobe PhoneGap Build," 2014. [Online]. Available: https://build.phonegap.com/. [Accessed 23 06 2014].

[43] Appcelerator, "Titanium official website - Start Page," 5 3 2014. [Online]. Available: http://www.appcelerator.com/.

[44] .NET Foundation, ".NET Foundation official website : Start page," 18 4 2014. [Online]. Available: http://www.dotnetfoundation.org/.

[45] Microsoft, "Lambda Expressions (C# Programming Guide)," 25 3 2014. [Online]. Available: http://msdn.microsoft.com/en-us/library/bb397687.aspx.

[46] Microsoft, "Patterns & Practices: Prism," 24 03 2014. [Online]. Available: http://compositewpf.codeplex.com/.

[47] "Caliburn Micro - GitHub page," 24 3 2014. [Online]. Available: https://github.com/BlueSpire/Caliburn.Micro.

[48] T. Sneed, "Codeplex - Simple MVVM Toolkit," 2014. [Online]. Available: https://simplemvvmtoolkit.codeplex.com/. [Accessed 7 6 2014].

[49] G. v. Horrik, I. A. F. Saúco and R. P. Mounguenge, "Catel MVVM," 24 03 2014. [Online]. Available: http://catelproject.com/.

[50] MVVM Light community, "MVVM Light," 21 3 2014. [Online]. Available: http://mvvmlight.codeplex.com/.

[51] MVVM Cross community, "GitHub : MVVM Cross plugins," 18 4 2014. [Online]. Available: https://github.com/MvvmCross/MvvmCross/wiki/MvvmCross-plugins.

[52] Xamarin, "Xamarin website," 2014. [Online]. Available: http://xamarin.com/. [Accessed 4 6 2014].

[53] Xamarin, "Xamarin website - Forms," 2014. [Online]. Available: http://xamarin.com/forms. [Accessed 04 06 2014].

[54] M. Kristensen, "VSWebEssentials official website - Start Page," 9 3 2014. [Online]. Available: http://vswebessentials.com/.

[55] Outercurve Foundation, "NuGet official website - Start Page," 23 2 2014. [Online]. Available: http://www.nuget.org/.

[56] JetBrains, "ReSharper website," 3 3 2014. [Online]. Available: http://www.jetbrains.com/resharper/.

[57] S. Schöb, "Schöb goes Mobile," [Online]. Available: http://opendix.blogspot.co.at/2013/05/using-resx-files-for-localization-in.html. [Accessed 29 5 2014].

List of figures

List of tables