The parallel development of a Silverlight application on multiple devices

Bachelor’s Thesis

Koos van Strien

Supervisor(s): Willem Meints, Dick van Albada
Signed: -
1 Introduction

This thesis reports on a Bachelor’s Thesis research project for the University of Amsterdam. The research is performed at Info Support B.V., an IT services company focusing on the development and administration of high quality software solutions.

When we develop an application for multiple devices, it is quite common to develop two separate applications. Info Support expects a growing demand for mobile applications that mirror the functionality of desktop applications, and therefore wants to investigate ways to develop an application for multiple devices in a more efficient way. One way to reach this efficiency is to reuse as much code as possible. With less code duplication, maintenance will be easier too.

In our research we develop one Use Case from a fleet management solution for company cars, where both a desktop and a mobile phone application are being built. The two applications share parts of the functionality, but each has its own platform-specific requirements. After developing the applications, the problems we encounter will be discussed and some recommendations made. These recommendations will not only be about further research, but also about what problems usually appear in the development of an application for multiple platforms / devices.
2 Scientific positioning

2.1 Related work

Some research has been done in the area of parallel development of two similar applications on different devices. Berti, Paternò, and Santoro [2006] describe the work on Migratory User Interfaces, which are geared towards completing a task on a device different from the one where the task is started. Earlier, Moore and Rugaber [1993] have done research in the area of migrating textual user interfaces to graphical user interfaces, successfully porting a textual MS-DOS program to a graphical MS Windows program. Their conclusion, “Our goals are to define methods, abstractions and mapping mechanisms to ease the transition, and to refine automated methods of migration.”, shows that the research was primarily meant to be a stepping stone for other researches. Myers and Rosson [1992] have conducted a survey on user interface programming. They discovered around 48% of “today’s” application’s code is devoted to the user interface portion, and 45% of the time is spent on the User Interface (UI) portion. Furthermore, they concluded that the use of design tools already reduced the amount of time spent on UI’s, and that it can be reduced even further if tools get better. Svensson and Magnusson [2004] have done research about UIs for services in a wireless network, where the UI is derived from the service’s possibilities and geared towards the capabilities of several devices. Ipadeola, Iyilade, Adigun, and Xulu [2008] also focus on providing a UI on a (often beforehand unknown) device, but use the UI to access grid services instead of wireless network services. Specific research about the display of web pages is performed by Wong, Chu, and Katagiri [2002] and MacKay, Watters, and Duffy [2004], but no out-of-browser applications were considered. Braun, Hartl, Kangasharju, and Mühlhäuser [2004] address the need of defining UIs when users use multiple (heterogeneous) devices simultaneously, coming close to the Migratory User Interface research mentioned earlier. Bisignano, Di Modica, and Tomarchio [2006] propose a solution for generating UI’s for multiple devices using a XML-derivation for defining a UI, where the tasks a user can perform are described using a so-called “Application Interaction System”.

Much of the related work is primarily focused on the automation of creating or translating User Interfaces. A large part of the related work focuses also on shared functionality of different devices. This leaves the device-specific strengths and capabilities out of sight.

2.2 Our research

The development of an application for a specific platform involves making use of the platform-specific strengths, and evading platform-specific weaknesses. When developing an application for multiple platforms, there will be similarities as well as differences between these platforms. Earlier research focused on sharing every aspect of an application (including interfaces) and doing “just” another compile run to generate the platform-specific application. Although this sometimes succeeded, it looks like the techniques are stuck in laboratories: none of the major programming environments have embraced the idea of generating one UI for more multiple devices. The main reason for this seems to be the fact that the result doesn’t weigh out the efforts involved: Many times mentioned as “Particularly difficult” was the “time-consuming and error-prone” [Ipadeola et al., 2008] task of developing a UI for multiple devices. Because of this, in our research we will initially let the platform-specific designs (such as UI designs) exist apart from each other, but share the parts that can be shared (as large parts as possible). We also try to address the earlier underexposed subjects of explicitly making use of device-specific strengths as
well as circumventing platform-specific weaknesses.

We will also investigate which problems occur when developing two applications in parallel, one for a desktop computer and one for a smartphone. The main goal of our research is to look at the problems that arise while developing an application in parallel, so our research is more like a case study or an investigation than a fundamental research. When developing the application, we try to:

- maximize code reuse between two instances of an application
- make use of the device-specific strengths

Furthermore, we want to point out which parts of the parallel development of two applications turn out to be particularly hard. We want to conclude our research with an advice about the effort and benefits of sharing as much code as possible.
3 Approach

3.1 The parallel development of two applications

To be able to study the parallel development of two applications inside a system, we choose to implement a Use Case of an existing application design. The Use Case is directed towards a Desktop implementation, but we will alter this Use Case slightly, making it a realistic Phone implementation as well. Besides just seeing what problems we will meet, we want to focus on the following main points in parallel development:

- Handling “pure” shared functionality (for example: sharing self-developed libraries)
- Handling almost shared functionality (for example: using third-party libraries across different platforms)

3.2 Description of the Use Case

In our research we look at the implementation of a Use Case from Info Support’s “Garage Management System” (GMS). The GMS is a fictitious application about a car leasing company wanting to manage their fleet. From the GMS we picked one Use Case, Use Case 20 (changing the owner of a car, Appendix B). This Use Case involves a user in the role of an Administrative Worker. An Administrative Worker has the possibility to change the owner of a car. Beside the Administrative Worker, there is also a customer - which can be every customer of the company. A customer can own a car, but doesn’t have to in every case.

3.3 Architecture

3.3.1 Server

At the back-end of the application is a REST server written in the Python programming language. In a typical RESTful way, this server provides a way to access its resources using URLs. The structure of the server makes it easy to add extra resources.

3.3.2 Desktop client

The front-end on Desktop side is a Silverlight Desktop client. Although a Windows Presentation Foundation-based client would extend the graphical possibilities of the UI, we choose to use Silverlight because of the compatibility with Windows Phone. Because Silverlight is available on the desktop as well as on the phone, the User Interface components are available as well on both platforms. Using Silverlight instead of Windows Presentation Foundation on the desktop greatly reduces the effort involved in user interface translation. This enables us to focus more on the key issues as described in subsection 2.2.

The Desktop client is built within Silverlight 4, a major development platform for end-user web- or desktop-applications.
3.3.3 Phone client

The front-end on the Phone side is a Windows Phone-client. During our research, Windows Phone used Silverlight 3.5 for its UI. Silverlight 3.5 is a hybrid version between Silverlight versions 3 and 4, somewhat tailored to mobile platforms. Although many controls of Silverlight 4.0 are also available on Silverlight 3.5, the appearance of the mobile version is re-designed for the Windows Phone design system named Metro \cite{Mic2010}. As UI sharing is not what we’re looking for in the first place, this won’t cause significant problems for us.

3.4 Process

As stated in subsection 2.2 our research is much like a case study. Therefore we will look at the development process of a simple application that has two versions: one for the Windows desktop and one for the Windows Phone. As long as the functionality is shared, the following process is executed:

1. Develop an aspect for one platform
2. Develop the same aspect for another platform
3. Combine inside a platform-agnostic library

Any encountered problems are then written down, combined with possible solutions, and the solution chosen by us during our research.
4 Results & Discussion

4.1 Setup

In our research, we start out with a small application which retrieves a list of cars from the server and displays it. Cars are represented by Car objects, which are structured as follows:

- Every Car object has two properties:
  - an ObservableCollection<Detail> of Detail objects
  - a property called Signplate, acting as identifier

- a Detail, acting as a hashtable entry, has two properties:
  - a Key
  - a Value.

To display the list of Car objects on screen, we choose to store the objects inside an ObservableCollection. The extended data binding possibilities of the ObservableCollection datatype makes it a perfect candidate for our storage needs1 (the Detail objects inside the Car class are stored in an ObservableCollection for the same reason).

The Phone- and Desktop-specific applications both have their own project inside the same solution. Those projects are called PhoneGarageManagement and DesktopGarageManagement. This implies that from within code, it is possible to refer to classes within another library or application, even when this library or application is not yet compiled.

4.2 Creating "platform-agnostic" libraries

When developing software projects, large parts of code are usually stored in (reusable) libraries. While reusing libraries on a single platform is quite easy, creating reusable libraries across multiple platforms is somewhat harder - not all third-party libraries and API's available on one platform are available on the other. In this subsection we'll look at problems making it desirable to have platform-agnostic libraries and ways to provide in this need and handle ensuing problems.

4.2.1 The need for platform-generic libraries

After we developed the initial phone application (only displaying a list of cars), the first problems appear when developing the Desktop counterpart of the application. Because the data structures are already in the Windows Phone project (PhoneGarageManagement) inside the same solution, we try to access the PhoneGarageManagement project. At this point, DesktopGarageManagement (the Desktop project) rejects a reference to the PhoneGarageManagement project: it is not a real Silverlight project. The fact that this solution would rely on an expected but non-guaranteed binary compatibility between two platforms would have made it a questionable solution anyway, but the barrier is hit: Windows Phone projects cannot be referenced directly from within a "regular" Silverlight project. To solve this problem, we have to create a separate project - a platform-generic library.

---

1For a description of the M-V-VM pattern using ViewModels and the description of the concept of data binding, see subsubsection A.2.3 and subsubsection A.2.4
Although this rejection is a Silverlight-specific issue, outside our specific domain (Silverlight / Windows Phone) this will happen in every environment where the attempts are made to reuse compiled libraries across platforms: not every single bit of an application on one platform can be used on any other platform, so platform-generic libraries will be needed.

4.2.2 Microsoft’s Solution: Portable Libraries

Having concluded that there is a need for platform-generic libraries raises a question: what should be the target platform for this library, as it would be “platform-agnostic”? If we just want to run the same binary code on all platforms, an obvious option could be creating a “regular” Silverlight library. However, referencing a standard Silverlight library from a Windows Phone project is not guaranteed to work. It is possible, though, and at this moment the easiest method.

A neater way is creating a so-called “Portable Library”. In an update on January 19th, Microsoft announced the “Portable Library Tools CTP” [Patten, 2011]. A Portable Library can safely be referenced from within the platforms it is selected to run on (for example, Silverlight 4 as well as Windows Phone as well as XBox 360). Applied to our case, this results in three projects in one solution: the shared library, the Silverlight client for Desktop and the Silverlight client for WP7.

A problem that arises in this case, is that some parts of the Silverlight API are not available on both platforms. Or, sometimes, they are available on both platforms, but not in the “platform-agnostic” API. A concrete example of this we encounter when we try to move the Car datatype to the platform-agnostic library. The ObservableCollection is not available in the Portable Library Tools. The generic question lying behind this problem is: How can we store datatypes inside a platform-agnostic library, when those datatypes aren’t accessible or visible for this type of library?

4.2.3 Injecting platform-specific datatypes into a platform-agnostic library

To inject datatypes with a platform-specific implementation into a platform-agnostic library, Burke [2011] combines two techniques to get a solution:

- using a Dependency Injection (DI) pattern
- type the class’ properties using supertypes of the object; thus creating a “supertyped container place”

The second technique, using a supertype as container place, does the work only partially. It only provides a place where the actual object finally can be stored, even when the type of this object is usually not accessible in that place. In our case the platform-agnostic library has a property Cars of type IList<Car>, as visible in Listing 1. IList is a superclass of ObservableCollection, so an ObservableCollection<Car> object will “fit” in an IList<Car> variable. But how and where will this insertion find place?

The DI pattern states that a class - call it class A - should provide only a place where objects can be stored. This place is typed using a contract (or an Interface) according to the requirements of the object - call it of type B - that will be stored. Class A is then instantiated by a factory, which also

---

2As of 3/31/2011, Microsoft released an update to the Portable Library Tools. This made the System.Windows namespace (which contains the ObservableCollection datatype) accessible from within Portable Libraries, as long as the only target platforms are Silverlight 4 and Silverlight for Windows Phone 7.

3A basic knowledge of Design Patterns is expected from the reader - more information about Design Patterns in general and the DI pattern in particular can be found in Gamma, Helm, Johnson, and Vlissides [1994].
public class PortableViewModel : INotifyPropertyChanged {
    private IList<Car> carList;

    public IList<Car> Cars {
        get {
            if (carList == null) {
                carList = PortabilityFactory.Current.CreateList<Car>();
            }
            return carList;
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;
}

Listing 1: PortableViewModel.cs

private class PhonePortabilityFactory : PortabilityFactory {
    public override IList<T> CreateList<T>() {
        return new ObservableCollection<T>();
    }
}

Listing 2: few lines from App.xaml.cs

inserts objects of type B in the placeholders. This way, class A has no knowledge about the creation of other objects contained, thus the class is less dependent on other classes. This is one way to decrease coupling.

However, because ObservableCollection isn’t available at all in the Portable Library, creating a factory inside the library that delivers this class will not work. This is solved by creating a supertyped factory, which is overridden in the platform-specific project. This way, the factory that would return an IList<T> in reality returns an ObservableCollection<T> (see Listing 2), so the injection work is done inside the platform-specific implementation.

It may seem strange that a List, which doesn’t implement the INotifyCollectionChanged interface, would notify databound elements of changes inside the collection. In fact, the databinding engine doesn’t refer to Cars as a List at all, but sees an ObservableCollection. Because the databinding engine uses reflection to check for INotifyCollectionChanged on the collection itself rather than the object being used as the DataContext, it matters what type that object actually is (as opposed to the type of its container) [Burke, 2011].

4.3 Doing platform-specific operations in a platform-agnostic way

In subsubsection 4.2.3 we covered how to use platform-specific datatypes inside platform-generic libraries. Besides datatypes, there is also functionality that has to be available in the (platform-generic) library, but can only be reached from within a platform-specific library. Rephrased in a question: how to handle functionality that is platform-agnostic, but require platform-specific operations to succeed? This “platform-specific operation” can for example be any library call to a third-party library that’s not platform-agnostic. In our Garage Management case, the synchronize operation is a nice example: synchronization is the same on all platforms. The operations forming a synchronization operation are:
1. Download list of cars from server
2. Update ViewModel

Communication with the server is handled slightly different in Silverlight 4 than in Silverlight 3.5. Some of the differences are caused by the immaturity of some Silverlight 3.5-libraries (for example, it is not possible to set the `Content-Length` HTTP-header within the Silverlight 3.5 library). Other differences stem from the fact that some namespaces simply won’t be available on Silverlight 3.5 (such as the `System.Windows.Browser` namespace, which is targeted to the in-browser usage of Silverlight and not useful for UI’s on the Windows Phone). Those differences add new challenges for a shared codebase.

There are several ways to cope with a proper separation of platform-specific and platform-generic implementation details, each having their advantages and disadvantages. In our research we look at three ways of handling this:

1. A general code-base, from which platform-specific libraries are compiled
2. A platform-independent general library, to be extended in the platform-specific libraries
3. Our “hybrid” solution

4.3.1 General code base, platform-specific libraries

This method aims at creating one code-base, containing exceptions for each target platform. These exceptions are distinguished by compiler directives, and by passing options to the compiler, the target platform is selected. This is depicted in Figure 1. Instead of rewriting different methods for a specific target platform, only the code inside the methods which really differs is “split out” into platform-specific cases. This keeps a nice uniform interface to a programmer using the libraries. However, there are two major drawbacks in this method:

1. It’s impossible to interchange compiled libraries between platforms
2. Datatypes that look similar (the same name, namespace, properties and methods) are in reality different

The first drawback - incompatibility with all platforms but the one targeted at compile time - is because this “compiler-directives” splits the “one” code-base out into two generated code bases - see Figure 2. Those libraries are incompatible with each other, so they are not interchangeable between the platforms.

An example of this effect can be seen when trying to use the open source RestSharp library, which provides ways to communicate to RESTful services from within Silverlight 4 and Windows Phone. To address the differences between Silverlight 4 and Windows Phone (using Silverlight 3.5), preprocessor directives are used. Inside the source code this has benefits: all source code is in one place, which makes changes to the library easy to commit and overseeable. However, when trying to use this project in a Portable Library, it turns out to be impossible: although the Portable Library accepts the Phone-specific library as reference, Silverlight 4 (“Desktop-Silverlight”) uses different libraries to connect to the REST server so the library doesn’t work.

The second drawback reveals itself when one doesn’t know the differences between the platform-specific libraries. Without knowing about the differences, it may be tempting to pass an (outwardly identical) object of one platform-specific implementation of the library to another.
In our case, an example is passing a `RestRequest` object of the Silverlight 4-library to a method inside the Portable Library accepting a `RestRequest` object being defined by the Windows Phone-library. This “injection” of a Silverlight 4-specific `RestRequest` object into the shared library turns out to be impossible: the classes are in reality not the same, although their name, methods and properties may be. This is depicted in Figure 3.

4.3.2 General code base, extended in specific libraries

In contrast to the solution of building whole platform-specific libraries, it is also possible to build one “base” library. This library contains all shared functionality, as long as it is platform-independent. It is well possible that this will be just a collection of some interfaces or abstract classes, being contracts of what derived classes should look like. In contrast to the approach described in subsection 4.3.1, this compiled library can be shared across all platforms. The types defined inside this library will be the same on all platforms, making the sharing of code much easier. However, this method has its own major drawbacks:

1. The library which can be shared has to be extended, resulting in even more types
2. It’s impossible to refer to platform-specific libraries

The second drawback needs some explanation. It might look quite obvious, because of the definition that everything the library contains should be “platform-independent”. The implication is that libraries already written with the “general code-base, platform-specific libraries” approach, simply cannot be referred to. Applied in our case: the RestSharp libraries are compiled specifically for this platform, and don’t have platform-agnostic type specifications. This makes the usage of RestSharp-libraries impossible inside this method, which in turn means that much shared functionality cannot be included inside other libraries.

4.3.3 Our “hybrid” solution

There is a third solution combining the two methods mentioned above. The method described in subsection 4.3.2 will result in portions of “shared functionality” being moved to the platform-specific classes. When applying the method of subsection 4.3.1 on these (derived) classes this will keep not only the platform-agnostic functionality, but also type specifications inside platform-independent classes. Platform-specific implementations or derivations of the functionality are then defined inside platform-specific classes.

This solution fits well in our context: it is quite possible that the RestSharp-library will start to make use of Portable Libraries. Until that time, we define an interface through which we communicate with the (platform-specific) RestSharp library. When RestSharp decides to start using Portable Libraries, it is easy to move functionality back to the platform-independent part of our project.

A straightforward implementation is the creation of multiple Visual Studio projects inside one solution. In our case there are three projects:

1. Portable Library
2. Phone application
3. Desktop application

The main code file for many cross-platform classes is placed inside the Portable Library. This code file is then linked to within other projects, where it is extended to certain platform-specific needs.
public partial class PortableViewModel : NotifyPropertyChanged {
    public override void dispatch(Action toExecute) {
        Deployment.Current.Dispatcher.BeginInvoke(toExecute);
    }

    public override void dispatch<T>(Action<T> toExecute, T argument) {
        Deployment.Current.Dispatcher.BeginInvoke(toExecute, argument);
    }
}

Listing 3: PortableViewModel (excerpt)

4.3.4 Application of the “hybrid solution”

Although a ViewModel is a Model of a View, between two platforms even ViewModels overlap. The main reason is that ViewModels do not contain information about the View, but the View “binds” itself to the ViewModel. This has several huge advantages and opportunities, all based on the fact that the operational code is clearly separated from the UI declarations, and the View “subscribes” to change notifications on the ViewModel, so the ViewModel doesn’t have to know which elements to update when changes occur.

ViewModels can be changed from within almost all classes and methods. To avoid collisions when two separate threads try to change the same property on the ViewModel, a system with mutual exclusive locks for changing properties could have been implemented. The easier (and non-blocking!) way is to require that changes on UI-related properties are all committed from within one thread. This is possible because by default, classes inside a Silverlight application can easily dispatch method calls to a Dispatcher, which places a method call in queue for the UI thread. However, within a platform-agnostic library there is no Dispatcher available, as a platform agnostic library should not know about “default” running threads. In our case, even the Dispatcher class, System.Windows.Dispatcher, is not available in a Portable Library.

In this case, we not only want to keep the functionality of updating the ViewModel inside the platform-agnostic library, but keep the ViewModel itself inside the platform-agnostic library. This implies that the method described in subsubsection 4.2.3 using a portability factory cannot be used, because the Dispatcher interface does not extend or implement any class or interface. And unlike the case of subsubsection 4.2.3, no View is “looking into” places where a Dispatcher could be stored.

It is possible, however, to mimic the way we would use a Dispatcher. We do this by creating a method with signature public virtual void dispatch(Action a) (see Listing 3). The default action of this method is just calling the Action that was passed along (an Action is basically any method that returns void). Because we declare this method virtual, the user of the PortableViewModel has the choice to extend the class and override this method (if a Dispatcher is available in the target application), or to leave it this way (thus executing the Action in the same thread it was called). Listing 3 gives an overview of the code. This is an application of our “hybrid solution”: providing a uniform code base which is the same on all platforms (and inside a platform-agnostic library), with the possibility extending or overriding parts of it when needed or possible.
public partial class AppliedPortableViewModel : PortableViewModel {
    public virtual void dispatch(Action toExecute) {
        toExecute.Invoke();
    } 
    
    public virtual void dispatch<T>(Action toExecute, T argument) {
        throw new NotImplementedException();
    } 
    
    public void request<T>(RestRequest request, Action<T> dispatchedOperation) where T : new() {
        Client.ExecuteAsync<T>(request, (response) => {
            this.dispatch<T>(response.Data, dispatchedOperation)
        });
    }
}

Listing 4: AppliedPortableViewModel extends PortableViewModel (excerpt)

4.4 User Interface sharing

Although we explicitly stated that we would not focus on User Interface generation and reusability, we still want to take a close look on the interface definitions. In Listing 5 and Listing 6 are two interface definitions for presenting a list of Cars displayed: Listing 5 for Windows Desktop, Listing 6 for Windows Phone. When studying the definitions, it appears that the main differences are:

- style differences (the orientation of the StackPanel, font layout and general style)
- action differences (which handlers to call if a detail changes)
- a difference in UI, which we will initially ignore and is thus commented out (the part between <!-- and -->).

This raises the question if there will be benefit in sharing one code-base for certain UI elements. Note that this is a different approach than earlier research about User Interfaces: earlier research focused on the automatic generation of User Interfaces for several devices. We already have those User Interfaces, but see possibilities for sharing parts of the code, thus making changes easier (and honouring the DRY principle).

We are aware that the implementation of a User Interface is a highly vendor- as well as platform-specific issue. It is well possible that the problems and solutions found by a close investigation on this topic will be specifically pointed to the Microsoft platform and it will be hard to draw platform-generic advices or conclusions from it. Still, we encounter code duplication that can possibly be removed, so we chose to include it in our research. In the following sections, ways of reusing this code will be examined.

4.4.1 Direct inclusion

The easiest way to share User Controls between platforms would be making them outwardly identical (using the same names for bindings, themes et cetera). The controls could then easily be extracted to another source file, which would be included on the spot where the control first appeared. Because XAML doesn’t support compile-time inclusion of other files, this is impossible, but even if it was possible it would not have been a neat solution. We are convinced that inclusion is not a good practice for the following reasons:
1. The interface definition would have been cluttered by inclusions. A literally inserted element is a static entity, not being able to interact with, or to be used by, other controls. This imports an alien species inside your User Controls, making code less readable and less maintainable.

2. In interface definitions, styles are often provided by the Operating System. In Listing 6, one of these styles is PhoneTextLargeStyle. Those styles are not necessarily available on other platforms, thus making direct inclusion impractical. It is possible to work around this though: by using user-defined styles that inherit from those Operating-System styles, an “abstraction layer” of styles could be created. The coupling with the OS theming (being Windows Desktop or Windows Phone) would then occur inside a “style sheet”. However, this would obfuscate which resources / styles the element is really using.

4.4.2 Creating new User Controls

A second solution is “abstracting the control away” in a new User Control. This is quite easy in Silverlight. All controls are “just” classes deriving from the UserControl class, and creating a new User Control is as easy as describing the control in relation to the UserControl inside a XAML-file, and defining the properties (and possible bindings) inside a code-behind file. In our case, we will try to create a custom User Control called CarSelector. To customize it according to the different target devices, we derive from Listing 5 and Listing 6 that the following properties should be available:

- Style property of the ListBox
- ItemsSource property of the ListBox
Listing 7: Passing the list of Cars as DataContext to a User Control

- Style of the TextBlock and StackPanel inside a ListBoxElement
- Binding of the TextBlock inside a ListBoxElement

The Style property of ListBox is an easy one: the encapsulating control has only one “child”: ListBox. So in the code-behind file accessors are created which hand all requests regarding the Style property to the ListBox.

Binding the ItemsSource property is somewhat harder: Initially, the CarSelector has no ItemsSource property available. It is possible to create a custom ItemsSource property, but “linking” it to the ListBox’s ItemsSource adds extra complexity and code, while an easier solution is available. The easy solution is passing the list of Car objects as DataContext. This way, every binding inside the CarSelector will start at that list, which makes binding to it very easy, as shown in Listing 7.

The Style property of the TextBlock and StackPanel are not as easy to reach as the Style property inside the ListBox. The main reason for this is the fact that both elements are inside a DataTemplate. Because a template is just a blueprint, elements inside a template do not really “exist”. As a result, passing on data elements like we did with ListBox’s Style property is not possible. However, even when it was possible, this would be an undesirable way to define the styles: all Styles would have to be defined in the top-most element, thus putting knowledge about child elements (namely ListBoxItem elements) in a container element (namely ListBox, or our “extracted control” containing ListBox).

This shows us where creating new User Controls in this case leads to. Because the styles need to be applied to elements inside the CarSelector element, the styles have to have knowledge about CarSelector’s inner working, as depicted in Figure 4. This makes the abstraction less useful, and implies more tight coupling: when something changes inside CarSelector it will lead to a change on all device-specific applications of the element.

4.4.3 Extracting properties from the User Control

When defining User Interfaces in XAML, it is possible to separate Styles and DataTemplates from Controls. This is a good way to reuse Styles and DataTemplates, but also separates the (platform-dependent) styles from the (platform-independent) Control (in this case ListBox). Doing so, it creates the interesting situation visible in Listing 8: the actual interface definition (below the vertical dots) contains only one element.

When applying the same practice to the Phone-specific project, this results in an almost identical control, as visible in Listing 9. But two single elements that are almost the same do not have benefit in
Figure 4: a custom UI element representing CarSelector

Listing 8: Separating Styles DataTemplate and Controls

Listing 9: The result of removing the Styles and DataTemplate for the Phone
sharing code: replacing two single elements with two abstractions of one single element is only useful if the abstraction is either significantly smaller or more maintainable than the original elements were. The two elements already are quite small, and maintainability is not an issue given the clear separation of styles and templates from controls, so the need for replacing the element disappears.

4.4.4 User Interface sharing: Conclusion

The research around sharing parts of the User Interface has taken a surprising turn: after restructuring (and decoupling) Styles and Templates in the XAML sources, it turns out there is not as much duplication as we estimated. When duplication remains after the decoupling of this code, it will have been split out into Templates, Styles and nested elements. When Templates or Styles are identical, it is fairly easy to put them in a separate Resource File. When duplicate nested elements remain in the code files, no further element extraction is useful. The reason for this is depicted in Figure 5 and Figure 6. In Figure 5 are two similar nested elements depicted, but the “inner elements” of those elements have their own styles. Now, when extracting element A with nested B into a new platform-agnostic element A’, the platform-specific style definition has to know about the inner workings of element A’ - as depicted in Figure 6. In our opinion this takes away one of the huge advantages of abstraction.
5 Conclusion & Future Work

In this research we looked at the parallel development of an application on two platforms. We focused in our research efforts on the following three aspects:

- Sharing libraries across platforms
- Separating platform-generic and platform-specific operations
- The need for sharing UI

We present patterns for sharing code between different platforms, and make recommendations for it. We describe a way to inject platform-specific datatypes into another platform. But libraries and APIs are subject to change: after we figured out how to inject platform-specific datatypes, the Portable Library Tools were changed. This made one of our patterns obsolete. We describe a way to handle with third-party platform-specific libraries, and compare ways to write libraries in a platform-agnostic way. In addition to the Object-Oriented (OO) patterns and practices we also take a pragmatic point of view and argue that linking source code files within the filesystem sometimes is the best way of sharing code between projects. Finally, we look at the possibilities for UI sharing, and conclude that the need for sharing UI code in our example is gone once we restructure the elements inside the UI and make a clear division between code, styles and templates. From this we learn that a clear separation of concerns sometimes solves many imagined problems, and helps to clarify the real problem.

Our overall conclusion is that code and class sharing is achievable and a good practice. Comparing our development issues with third-party libraries, we conclude that it is better to take a platform-agnostic approach while writing libraries that will be used across platforms, because translating or (im)porting libraries and types involves significant effort.

One of the goals described in subsection 2.2 that we did not meet was the investigation of making use of device-specific features. A way to investigate this inside our current project would be to make use of the Global Positioning System of a smartphone to detect nearby customers. This issue remains open to investigation.

Because the libraries we used were under full development, things are likely to change. Therefore it is worthwhile to revisit some issues of this research after the release of the Portable Library Tools, as well as the Windows Phone Mango update, which will bring a slightly boiled-down Silverlight 4 version to Windows Phone. This will make it interesting to look at reusing “plain” Silverlight 4 libraries, without using the Portable Library Tools. Although “plain” Silverlight libraries are not guaranteed to work, it will have the benefit of being able to experiment with the full Silverlight 4 library across platforms. Finally, it is worthwhile to further investigate User Interface code sharing. Not looking for automation of this sharing, but how to “abstract away” UI elements, while at the same time applying new styles. We expect that the use of Silverlight 4 libraries across those platforms will make this easier.
A An introduction to the Microsoft platform

In our research, we use the Microsoft development environment. This is primarily because the recent introduction of Windows Phone 7 (featuring Silverlight as development platform) raises questions that we try to address in our research. Because knowledge about the Microsoft platform is not considered as general Computer Science knowledge, we think it to be necessary to include a section on the Microsoft platform and development environment.

A.1 .NET

The .NET framework is a software framework providing a runtime environment and base class library inside the Windows platform. .NET is an initiative Microsoft, Intel and Hewlett-Packard started in 2000. It was released in 2002. The complete source code of the .NET framework libraries is available since late 2007 [Guthrie, 2007]. This doesn’t mean it’s free software, for the licenses still apply, but the sources can be viewed, which makes it easy to know what’s going on inside the libraries.

A.1.1 Structure of .NET

.NET provides a managed environment where .NET programs run on a .NET Common Language Runtime (CLR). This managed environment is called the Common Language Infrastructure (CLI).

The CLR is comparable with the Java VM (or Java Runtime), which JIT-compiles and executes Java programs. Sourcecode in the .NET-platform is, just as Java code on the Java platform, not compiled to native code, but to an intermediate language. In the Java environment this is called “bytecode”, in the Microsoft environment it is called Intermediate Language (IL) or Common Intermediate Language (CIL).

In contrast to the Java environment, the .NET environment wasn’t designed with the “one language fits all” philosophy. With the introduction of .NET came two specifications which cover the boundaries of the .NET languages. On the one hand, everything that is possible inside languages running on .NET is specified in the Common Type Specification (CTS). On the other hand, everything a .NET language must minimally adhere to is described in the Common Language Specification (CLS). The CLS is thus a subset of the CTS.

All those three-letter abbreviations can be somewhat overwhelming, so they’re listed in Table 1.

<table>
<thead>
<tr>
<th>abbreviation</th>
<th>full name</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR</td>
<td>Common Language Runtime</td>
<td>Provides secured execution environment; manages memory</td>
</tr>
<tr>
<td>CIL</td>
<td>Common Interpreted Language</td>
<td>“bytecode” for .NET</td>
</tr>
<tr>
<td>CLI</td>
<td>Common Language Infrastructure</td>
<td>Code-to-CIL and executing CIL infrastructure</td>
</tr>
<tr>
<td>CLS</td>
<td>Common Language Specification</td>
<td>All .NET languages must obey this</td>
</tr>
<tr>
<td>CTS</td>
<td>Common Type Specification</td>
<td>All .NET language possibilities</td>
</tr>
</tbody>
</table>

The connection between C# and the .NET framework is the fact that C# is designed for the CLI. Therefore C# is one of the most efficient ways of accessing the .NET framework capabilities.
Listing 10: “Traditional” in-code way of defining a UI element

```csharp
ListBoxItem lb1 = new ListBoxItem() { Name = "item1", Content = "Item 1" };
ListBoxItem lb2 = new ListBoxItem() { Name = "item2", Content = "Item 2" };
ListBox lb = new ListBox() { Name = "myListBox", Items = { lb1, lb2 } };
myGrid.Children.Add(lb);
```

Listing 11: “Traditional” in-code way of defining a UI element - (almost) no shorthands

```csharp
ListBoxItem lb1 = new ListBoxItem();
lb1.Name = "item1";
lb1.Content = "Item 1";
...
ListBox lb = new ListBox();
lb.Name = "myListBox";
//lb.Items.Clear();
lb.Items.Add(lb1);
lb.Items.Add(lb2);
myGrid.Children.Add(lb);
```

A.2 WPF and Silverlight

In the .NET Framework 3.0, Microsoft announced a new graphical subsystem, codenamed “Avalon”. Avalon was built to provide a consistent programming interface, and a clear separation between the user interface and the business logic. When released, Avalon was renamed to Windows Presentation Foundation (WPF). In 2007, Microsoft released Silverlight. Silverlight is a graphical subsystem just like WPF and was meant to be used on (but not limited to) webpages. The key to Silverlight is the “Everywhere” part: it runs inside browsers, as native applications on desktop computers, but is also used as the graphical subsystem of the Windows Vista “gadgets” (desktop widgets) and for providing the Windows Phone UI.

A.2.1 Separation of concerns

WPF introduces the use of an declarative language called XAML to define the user interfaces. Behind the scenes, XAML just instantiates WPF-classes and gives their properties values. To see how this works, in Listing 10 is a “traditional” example given of an in-code interface definition. This code is a shorthand for Listing 11. Note that every constructor is empty: every user interface element class should have a parameterless constructor.

Now if all these classes can be instantiated with parameterless constructors, and properties can be set, this opens possibilities for doing this in XML (or an XML derivant, like XAML). The elements will be class instantiations of classes, the properties will be properties. This is exactly what happens: the “procedural” code inside Listing 10 has the same result as the “declarative” code in Listing 12. Listing 12 can be shortened even more: because it is intuitive that the direct “child-property” of a ListBox is the collection of ListBoxItems, and the direct “child-property” of a ListBoxItem is the Content property, the code can be written in a shorter notation, Listing 13. WPF-elements do this by defining a default “child-property”.

This way of defining “interface classes” enables WPF developers to create a clear division between UI and business code.
A.2.2 Code-behind files

In a WPF or Silverlight application in C#, every XAML file is backed by a “code-behind” file. This file has the same filename as the XAML file, and if a method call is done from within the UI, the first place to look for a handler is here. The code-behind file is part of the View, and is in Silverlight often (not always!) the place where the reference to the ViewModel is kept. The code-behind file greatly enhances the possibilities of “scripting up” a UI without messing up the design code. Changes done by designers (inside the XAML file), cannot mess up the event handlers living in code. However, code-behind files are easily misused, resulting in code in the code-behind file while it should be in the ViewModel.

A.2.3 The M-V-VM pattern

While the question how to separate business code and user interface has long bothered developers, several patterns have emerged. Model-View-Controller (MVC) is one of the most well-known, however the term is much misused for the most used pattern: the Model-View-Presenter (MVP) pattern. Inside WPF applications, the Model-View-ViewModel (M-V-VM)-pattern is preferred. In the mentioned patterns, the Model is everywhere the same. However, the View has different tasks:

- In **MVC**, a View looks directly to the Model for data
- In **MVP**, the View handles user input (whereas in a MVC-pattern the Controller would handle this)
- In **M-V-VM**, the View “binds” itself to a ViewModel and changes itself if the ViewModel changes

The last part, the Controller / Presenter / ViewModel each have their own function:

- a **Controller** handles user input, and changes the Model if needed
- a **Presenter** alters Views, and gets called by the View on user input
- a **ViewModel** is an abstraction layer of the Model, on which the View can act

The huge advantage of the M-V-VM and MVC patterns over the MVP pattern is the fact that a ViewModel or Controller shouldn’t know anything about a View. The advantage of M-V-VM over MVC is that it offers one interface to the View, so the View only has to interact with the ViewModel.
Listing 14: Binding Cars to a ListBox

instead of both the Controller and the Model. This makes it easier to attach User Interfaces to backend code, easier to replace User Interfaces, but it makes testing much easier too.

The “strict” separation of business code and user interface inside the M-V-VM pattern is only possible because of the extensive data binding features WPF and Silverlight offer on UI side. For this reason, the M-V-VM pattern is not widespread in other development platforms.

A.2.4 Databinding

Databinding is not Microsoft-specific. It just means binding two data or information sources together, such that they stay synchronized. In the WPF and Silverlight platform Data Binding is especially used to bind properties of UI element to data sources. This happens using the Binding keyword, as visible in Listing 14. This will result in the element looking up for an object Cars and binding itself to Cars’ contents. To know where the element should look, it has a so-called DataContext. By default, the DataContext is the code-behind file of a XAML-file, but it is possible to refine this to a specific object. In Listing 14 is visible that the Template for a ListBoxItem binds itself to a “signplate”. This signplate does not live in the code-behind file, but inside a Car object, that in turn exists inside the Cars collection. By setting the DataContext of the ItemTemplate to Car, all binding “inside” that element refers only to elements inside a Car object.
UC20 - Wijzig voertuigeigenaar

Professional Development Center

Info Support b.v.
Versie: 1.1
Status: Definitief
## Titel
UC20 - Wijzig voertuigeigenaar

## Project
Professional Development Center

## Bestand
GMS - UC20 Wijzig voertuigeigenaar.doc

## Bedrijf
<Bedrijfsnaam>

<table>
<thead>
<tr>
<th>Projectbelang</th>
<th>Formele overeenstemming</th>
<th>Lifecycle</th>
<th>Gebruik</th>
<th>Status</th>
<th>Datum</th>
<th>Versie</th>
<th>Release</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Must</td>
<td>1-Geen</td>
<td>3-Outline beschreven</td>
<td>1-Zeldzaam</td>
<td>Definitief</td>
<td>[Datum statuswijziging]</td>
<td>1.1</td>
<td>1</td>
</tr>
</tbody>
</table>

Kruisboog 42, 3905 TG Veenendaal, Tel. +31 (0) 318 55 20 20 Fax +31 (0) 318 55 23 55
De Smalle Zijde 39, 3903 LM Veenendaal, Tel. +31 (0) 318 50 11 19 Fax +31 (0) 318 51 83 59
Rabobank Bergh 30.59.52.889, Postbank 47.38.593
**Historie**

<table>
<thead>
<tr>
<th>Versie</th>
<th>Status</th>
<th>Datum</th>
<th>Auteur</th>
<th>Verandering</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Definitief</td>
<td>Juli 2006</td>
<td>Info Support b.v.</td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>Definitief</td>
<td>April 2008</td>
<td>Info Support b.v.</td>
<td>Nieuwe lay-out</td>
</tr>
</tbody>
</table>

**Distributie**

<table>
<thead>
<tr>
<th>Versie</th>
<th>Status</th>
<th>Datum</th>
<th>Aan</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>Definitief</td>
<td>Juli 2006</td>
<td>Endeavour release 3.0</td>
</tr>
<tr>
<td>1.1</td>
<td>Definitief</td>
<td>April 2008</td>
<td>Endeavour release 5.0</td>
</tr>
</tbody>
</table>

**Referenties**

<table>
<thead>
<tr>
<th>Code</th>
<th>Bron</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

© Info Support, Veenendaal 2006

Niets uit deze uitgave mag worden verveelvoudigd en/of openbaar gemaakt door middel van druk, fotokopie, microfilm of op welke andere wijze ook, zonder voorafgaande toestemming van Info Support.

No part of this publication may be reproduced in any form by print, photo print, microfilm or any other means without written permission by Info Support.

Inhoudsopgave

1 UC20 - WIJZIG VOERTUIGEIGENAAR .................................................................5
  1.1 Korte beschrijving .....................................................................................5

2 FLOW OF EVENTS .........................................................................................5
  2.1 Hoofdflow wijzig voertuigeigenaar[UC20.BF] ..........................................5
  2.2 Subflow Zoek auto .................................................................................6

3 AANVULLENDE REQUIREMENTS ...............................................................6
  3.1 Functionaliteit ..........................................................................................6
  3.2 Bruikbaarheid .........................................................................................6
  3.3 Onderhoudbaarheid ................................................................................6
  3.4 Betrouwbaarheid ......................................................................................6
  3.5 Efficiency ................................................................................................7
  3.6 Portabiliiteit.............................................................................................7
  3.7 Constraints .............................................................................................7

4 OPENSTAANDE PUNTEN ............................................................................7
  4.1 Openstaand punt 1 ................................................................................7

5 BIJLAGE: VOORBEELDEN ........................................................................8
  5.1 Venstervoorbeelden ..............................................................................8
  5.2 Rapportagevoorbeelden ........................................................................8
1 UC20 - Wijzig voertuigeigenaar

1.1 Korte beschrijving
De Administratief Medewerker kan een eigenaar van een voertuig wijzigen door een voertuig te selecteren en vervolgens een (andere) klant te koppelen aan het voertuig. Indien het een nieuwe klant betreft kan deze worden ingevoerd. Deze klant is dan direct gekoppeld aan het voertuig.

2 Flow of events

2.1 Hoofdflow wijzig voertuigeigenaar[UC20.BF]

Deze hoofdflow begint nadat de Administratief Medewerker heeft aangegeven een voertuig van eigenaar te willen veranderen.

1. Het Systeem start subflow “Zoek auto”.
2. Het Systeem toont KlantEnVoertuig.
3. De Administratief Medewerker geeft aan een andere klant te willen selecteren voor dit voertuig.
5. De Administratief Medewerker zoekt op basis van een combinatie van naam, straatnaam, postcode en woonplaats naar één of meerdere klanten.
6. Het Systeem toont een lijst met klanten die aan de zoekcriteria voldoen.
7. De Administratief Medewerker selecteert een klant
8. Het Systeem toont KlantEnVoertuig met de zojuist geselecteerde klant.
11. De Administratief Medewerker bevestigt het bewaren van de gegevens.

Hier eindigt deze hoofdflow.

2.1.1 Alternatieve Flows

A5.1 Nieuwe klant [UC20.AF01]
De Administratief Medewerker geeft aan een nieuwe klant in te gaan voeren.

i. Het Systeem toont NieuweKlant.
ii. De Administratief Medewerker vult de gegevens in en geeft aan deze te willen bewaren.

Ga verder bij stap 8.

N.B. Deze alternatieve flow is opgenomen uit didactische overwegingen (nuttig als voorbeeld bij de test case beschrijving). Het opvoeren van een klant wordt beschreven in de use case Beheer Klant. Bovenstaande functionaliteit wordt normaliter niet in een use case beschreven maar in het User-Experience model (User-Experience Storyboard).

A6.1 Geen klanten gevonden [UC20.AF02]
Wanneer het Systeem geen klanten vindt die aan de zoekcriteria voldoen, geeft het Systeem hiervan een melding.
Ga verder bij stap 4.
A9.1 **Afrekenen wijzig voertuigeigenaar** [UC20.AF03]
Wanneer de Administratief Medewerker aangeeft de gegevens niet te willen bewaren eindigt deze use case.

A9.2 **Selecteren klant** [UC20.AF04]
Wanneer de Administratief Medewerker aangeeft een andere klant te willen selecteren.
Ga verder bij stap 4.

A11.1 **Afrekenen taak afronden** [UC20.AF05]
Wanneer de Administratief Medewerker aangeeft de taak niet te willen afronden, ga verder bij stap 8.

### 2.2 Subflow Zoek auto

1. Het Systeem toont **SelecterenVoertuig**.
2. De Administratief Medewerker zoekt op basis van kenteken óf merk en/of type.
3. Het Systeem toont een lijst met voertuigen die aan de zoekcriteria voldoen.
4. De Administratief Medewerker selecteert een voertuig.

Hier eindigt deze subflow.

### 2.2.1 Alternatieve Flows

A3.1 **Er is niets gevonden**
Als het Systeem geen resultaat vindt op basis van de zoekcriteria

a. Het Systeem toont de melding

Ga verder bij stap 1

### 3 Aanvullende requirements

Aanvullende requirements die alleen van toepassing zijn op deze Use-Case worden vastgelegd in dit hoofdstuk. De paragrafen van dit hoofdstuk komen overeen met de ISO 9126 indeling.

#### 3.1 Functionaliteit

**3.1.1 Functioneel requirements 1**
Omschrijving van functioneel requirement 1.

**3.2 Bruikbaarheid**
Geen aanvullende requirements.

**3.3 Onderhoudbaarheid**
Geen aanvullende requirements.

**3.4 Betrouwbaarheid**
Geen aanvullende requirements.
3.5 **Efficiency**  
Geen aanvullende requirements.

3.6 **Portabiliteit**  
Geen aanvullende requirements.

3.7 **Constraints**  
Geen aanvullende requirements.

4 **Openstaande punten**

4.1 **Openstaand punt 1**  
Omschrijving van openstaand punt 1.
5 Bijlage: Voorbeelden

5.1 Venstervoorbeelden
De vensters worden nader gedefinieerd in het User-Experience Model. Hieronder staan figuren om een idee te geven hoe de vensters er uit komen te zien.

5.1.1 Venster ZoekVoertuig
Geen voorbeeld beschikbaar.

5.1.2 Venster KlantEnVoertuig

![User Interface Voertuig](image)

5.1.3 Venster SelecterenKlant
Geen voorbeeld beschikbaar.

5.1.4 Venster TaakAfronden
Geen voorbeeld beschikbaar.

5.1.5 Venster SelecterenVoertuig
Geen voorbeeld beschikbaar.

5.2 Rapportagevoorbeelden
Er wordt geen gebruik gemaakt van rapportages.
Glossary

**API** Application Programming Interface, a set of rules and specifications to communicate with a software library or program.

**M-V-VM** The M-V-VM (Model-View-ViewModel) pattern is an architectural pattern describing ways to cleanly separate data backends from operational code and operational code from User Interfaces. It is a variation of the Presentation Model (PM)-design pattern coined by Martin Fowler, which in turn builds upon the Model-View-Presenter (MVP) and Model-View-Controller (MVC)-patterns. The key issue creating the difference between MVC/MVP and M-V-VM is the way the interface is changed: instead of letting the Presenter change the View (MVP) or have the View watching the Model itself (MVC), inside M-V-VM the View is bound to the ViewModel (VM) and only watches the VM for changes. The View doesn’t do this by polling, but instead the ViewModel fires an event whenever a property is changed.

**Metro** Metro is a so-called “design language” - a set of guidelines and rules that define how designs should look. Metro is created by Microsoft, and used in Zune, Windows Media Center and Windows Phone 7.

**Migratory User Interface** interfaces geared towards completing a task across multiple devices - for example entering an e-mail via speech recognition while driving your car, completing the e-mail on your smartphone, adding some attachments and finally sending it from your desktop computer.

**platform-agnostic** When a library is platform-agnostic, we mean that a library hasn’t platform-specific features. Platform-agnostic libraries can be used across different platforms, as opposed to platform-specific libraries. Also used is the term platform-generic, which emphasizes the fact that the functionality in this library is not platform-specific. Of course the libraries can be extended in a platform-specific way.

**platform-generic** See platform-agnostic.

**platform-specific** As opposed to platform-agnostic or platform-generic libraries, platform-specific libraries run on only one platform, because they are built with knowledge about that specific platform.

**project** In Visual Studio, a Project is a set of code files and other resources that mostly result in one library. A Project has a build configuration, determining what compiler options are used when compiling the source.

**REST** REpresentational State Transfer is a architectural style outlining ways to guide communication across the web. A RESTful service (a service which conforms to the REST constraints) is stateless and maximizes the use of the pre-existing interface it uses. In the case of HTTP it uses the verbs defined in the HTTP specifications (such as GET, POST, PUT and DELETE) to operate on URL’s representing server-side resources. One URL always represents the same resource or resource collection (however the contents may change, of course).
**Silverlight** Formerly named Windows Presentation Foundation/Everywhere, Silverlight is Microsoft’s attempt to bring a Rich, WPF-alike UI environment to the browser. Interfaces are defined in XAML and applications can run inside as well as out of browser.

**solution** In Visual Studio, a Solution is a set of projects and cross-project files/settings (e.g. the build order of projects) used to build an application. A solution can contain one or more Projects, each with their own configuration.

**Windows Phone** Windows Phone is an operating system (OS) for smartphones. The OS is the successor of Windows Mobile, and is primarily aimed at the consumer market (as opposed to Windows Mobile aiming at the enterprise market).

**Windows Presentation Foundation** Windows Presentation Foundation (formerly known as “Avalon”) is a system used inside Windows to render User Interfaces. WPF interfaces are defined using XAML, and offer a wide range of possibilities regarding the division of UI and “backend”-code and data binding.

**XML** Extensible Markup Language - a language used to encode documents in machine-readable form. Used widely on all fronts inside Computer Science - including protocols like SOAP (XML-RPC), document storage (DOCX, XLSX), interface definitions (XHTML) and XAML, which is used to initialize structured values and objects.
References


