Improving Designer-Developer Workflow for Better User Experience

Dániel Dávid Nagy, Bence Kővári

Department for Automation and Applied Informatics
Budapest University of Technology and Economics
Goldmann György tér 3, H-1111 Budapest, Hungary
madve2@gmail.com, beny@aut.bme.hu

Abstract: While creating aesthetically convincing user interfaces has been time-consuming and resource-demanding for a long time, recent improvements in hardware and application platforms finally made it possible to bring the appeal and usability provided by well-presented and professionally designed user interfaces to almost all types of client software. Development of fat client applications, however, still lacks a proper and productive process which could aid both developers and user interface designers achieving their goals without setting back each others progress. In this paper, some of the currently popular applications platforms are introduced and examined in terms of helping development teams apply such a process. It is also shown how it is possible to utilize one of them, Windows Presentation Foundation, using a few simple patterns and guidelines – like Presentation Model – to improve designer-developer workflow, and hence the user experience, significantly.

Keywords: user experience, user interface, designer-developer workflow, windows presentation foundation, presentation model

1 Introduction

Since the introduction of personal computers – that is, computers intended to be used by a wider audience regardless of their level of education instead of qualified researchers – the way the user can communicate with the machine became a question almost as important as the features and capabilities to be accessed themselves. Without a usable and accessible user interface (UI) a piece of software is like being nonexistent. The case of Office 2003 is an expressive demonstration of that. After the release of Office System 2003 the most of the feature requests received, asked for features that were actually not absent from the latest release either, clearly indicating that the UI failed to help users achieve what they wanted to do: not only did they not find certain commands, they were even unaware of the availability of some of them. That is why Office developers laid emphasis on reinventing the UI, rather than implementing new features, in their 2007 release. [1]

The first commercially successful graphical user interface, which came with the Macintosh personal computer – and was developed based on the results of the Xerox Palo Alto Research Center [2] –, while looks a bit outdated at first, already had most of the UI elements and interaction mechanisms found in current software products – like buttons, checkboxes, scroll bars and so on. It introduced many of the basic principles of GUI design, like the usage of metaphors to achieve intuitiveness (the most well-known is the so-called desktop metaphor), the concept of forgiveness (that is, providing an undo operation and better error handling to encourage the user to try out things even if that might result in an unexpected or undesired outcome) and even the aesthetic integrity of the application. [3]

While the latter did not get the attention it would have deserved for a long time due to technical limitations of the past, utilizing the capabilities of current Graphical Processing Units (GPUs), today it is only a matter of well-chosen technology, process and people to create the best user experience. Like industrial design was in the 1920s, software design is in a period of transition: it is being realized that designing the look and visual behavior of a product is a profession on its own and usually cannot be done by the same professionals implementing the functionality behind it [4]. That is, designers and developers have to work together in order to make a success of a software product.

Development of thin client applications has been like that for years now: Web Design is a well-known and widely accepted discipline nowadays, and has the tools, standards and processes to separate presentation from application logic. Fat client applications, however, have yet to achieve an efficient designer-developer workflow.

The next section of the paper examines some of the currently popular solutions for creating modern desktop applications, in terms of improving the productivity of both designers and developers – separately and as a team as well. In Section 3 it is shown how one of those platforms can be used to separate responsibilities and work of those very different roles, hence making it easier for them to shape the user experience together.

2 Related Work

From the very first Macintosh Operating System, every notable platform provided its means to support developers to achieve the aesthetic integrity mentioned earlier, usually by exposing Application Programming Interfaces (APIs) to common UI elements as well as input methods (like handling the mouse interaction and so on). While clearly not being as easy to use as solutions of today, even the very first GUI platforms had their impact on the software market by encouraging developers to create graphically presented applications instead of text based ones.

One of the popular APIs besides that of Mac OS is Swing, which is built on an earlier solution, Abstract Window Toolkit (AWT), providing a cross-platform programming interface for developers and a platform independent look and feel for the users. It supports the usage of the Model View Controller (MVC) pattern by design (the built-in controls follow that pattern themselves) and an application-independently changeable look-and-feel mechanism, meaning that it basically makes it possible to design and develop the application by different groups of people [5]. The former activity however, needs a knowledge of the platform and its capabilities (especially the layout managers) so deep which actually cannot be expected from graphic designers but developers only. It is also hard to design and implement controls not being part of the API or drastically change the look or behavior of the built-in ones.

An important step forward solving those problems is Flash – initially designed to be an animation tool, became famous as a Rich Internet Application (RIA) platform, now available as a fat client platform. Being designed for graphic professionals in the first place, it is one of the first solutions providing the opportunity for designers to work on their own, to create the look and feel of the application themselves, rather than trying to make software developers to make their visions come true. It took user experience possibilities to a whole new level: providing highly customizable UI elements, multimedia capabilities and animation being a core part of the system, the classic design principles (like metaphors, direct manipulation, see-and-point etc., as described in Macintosh Human Interface Guidelines [3], as well as being good-looking, of course) can be achieved much easier and more deeply than ever [6].

The biggest problem with Flash lies in its biggest advantage: being primarily a designer tool extended with scripting support and some basic features to help developers implementing the desired behavior, it really cannot compete with other software platforms in terms of development tools, programming possibilities and performance. It is also hard to separate the application logic from the user interface, thereby making it difficult to test and to maintain the software. In conclusion, it does not really improve the designer-developer workflow itself, because it makes designing easier at the expense of the development process.

Windows Presentation Foundation (WPF) was designed with the needs of both developers and designers in mind. It introduced the concept of having two distinct set of tools for the two different roles, without losing the ability to work on the same set of files. That is, if a developer implements an application with a very basic look, the designer should be able to completely change it without harming the application logic; similarly, if a designer creates a set of screens and other visual resources, the developer should be able to use them immediately and put the desired behavior into practice without having to recreate them in a different set of tools (potentially losing some details here and there, which is not uncommon to other software platforms). The look and the behavior of the application should be able to evolve almost independently [7].

As a result, WPF was released as a part of the .NET Framework 3.0 platform, allowing developers to work with the libraries and tools they know and are productive with, along with a new family of software called Expression Studio, targeting designers wanting to work on .NET projects. Expression Blend, like Flash, was designed with them in mind, using similar look, metaphors and capabilities common to modern professional vector graphics software, including tools to create animations, customized user controls and so on.

To make this possible, the WPF team separated application logic from presentation by design: UI elements are no longer created imperatively from code but are defined in a separate markup file, with a so-called code-behind file attached to it, implementing its behavior. That model not only makes it possible to have two distinct tools working on the same project without harming each others work, but the concept of using a markup language[1] is much easier to grasp for most designers, because they usually know a markup language of some kind (most likely HTML/XHTML), rather than having to understand constructors, destructors and object references.

No software platform can be complete without accessible guidelines on how to use it properly, and WPF is no exception. While having a separate code file makes it easier to separate application layers, it still is not a complete solution by itself: since the code-behind file is linked to the markup file, application logic implemented in it cannot be reused without reusing its UI too; making it impossible to create multiple views of it or to separate it completely from the graphical interface (to unit test it, for example). Code-behind should be used only as a bridge between the UI markup and the pure application logic.

One of the most complete and continuously improved guidelines is Composite Application Guidance, also known as Prism [8][9], containing not only patterns and best practices, but some reusable libraries as well, making it possible to solve the problem of code-behind files, and even to create completely modularized and dynamically composed applications as well.

To avoid the difficulties described above, no external libraries are required of course. The Composite Application Guide recommends the Presentation Model (PM) pattern [8][9], first published by Martin Fowler in 2004 [10]. It introduces the concept of Presentation Model classes, which are abstract representations of

[1] Along with WPF was introduced XAML (eXtensible Application Markup Language), which actually defines UI by mapping XML tags and attributes to .NET objects and properties. It can be used to define other kinds of object graphs too, like workflows or web services.

views, relying heavily on commanding and data binding to keep in sync with them. It solves separation by not letting the view to directly communicate with the actual Model layer, and not letting other parts of the application to reference the view directly. The state and behavior of a view is implemented in its Presentation Model rather than its code-behind, making views changeable and even completely omissible.

While PM solves most problems developers had, it is not without its flaws of course: the reference the Presentation Model keeps to the view can easily result in referencing too many of its UI parts, basically having a code-behind-like class in the end, making problems reappear again.

3 Proposed Solution

A simple solution would be to completely omit any references to views, even from Presentation Models. Since data binding can handle most necessary communication scenarios (thanks to the availability of two-way binding and command binding), mostly there is no need for the programmer to access or manipulate views directly[2].

Sometimes, however, this is not the case. Not all properties are dependency properties, nor is it required of all UI components to implement the INotifiyPropertyChanged interface – hence there are scenarios that cannot be solved by data binding only.

Silverlight 2, which is a RIA platform evolved from WPF, introduced the so-called States and Parts model, which basically makes it possible for UI-independent implementation classes to have basic assumptions about views they have to work with, like a Button class can assume that its view has at least one FrameworkElement, which can be clicked on [12].

WPF, however, lacks the support for Visual States and Parts by itself. Although there is an external library available (called WPF Toolkit) to substitute the missing features required, and the next version of WPF is planned to have built-in support [13], developers wanting to work with the current version need a similar, but simpler solution.

[2] The Model-View-ViewModel (MVVM) pattern, which is a more specific version of the PM pattern, is actually based on that idea [11]

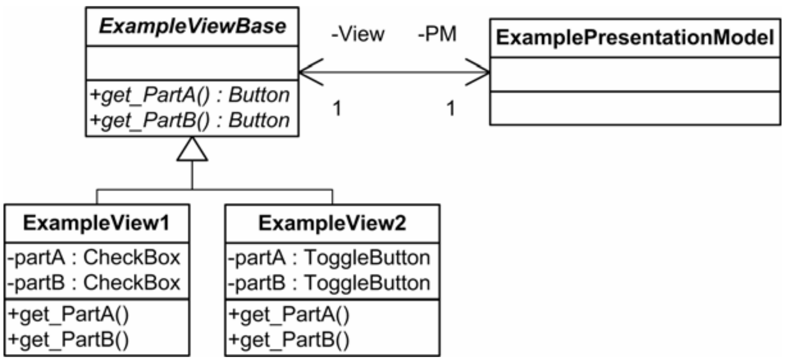

Figure 1
Proposed class hierarchy

Figure 1 shows our proposed pattern to keep letting the Presentation Model class having a reference to the view without losing the ability to change or omit it.

The idea is having a common abstract base class for all the possible views a Presentation Model can be the abstraction of, defining the parts it may assume being present as typed getters (using C# those can be properties of course). This way it can be made sure that the Presentation Model class will not reference more UI elements than it was intended to do originally, nor will it access more specific interfaces to them than necessary.

The Presentation Model gets its reference to the view as a constructor parameter, making it possible for other parts of the application to decide which view they should use. That view might be developed completely independently of course and it can be just a mock, provided by a unit test class, too. To be able to instantiate a view in XAML, without having to add additional code anywhere to create a Presentation Model for it and set both references, it is recommended that the view should create the Presentation Model for itself[3] in its constructor. It makes using that piece of the UI safely usable, too, because its dependencies will always be fulfilled and hence it always can work properly (while having to do it from code might be a source for some hard-to-find errors).

One of the common scenarios this refinement to the PM pattern can be applied to is when it is necessary to work with third-party UI components without property change notification of any kind (like dependency properties or PropertyChanged events). Take the class hierarchy shown in Figure 2 as an example.

[3] While a Presentation Model class might work with different views, a view is expected to work with a specific Presentation Model only, hence letting the view create that for itself does not make the application less flexible.

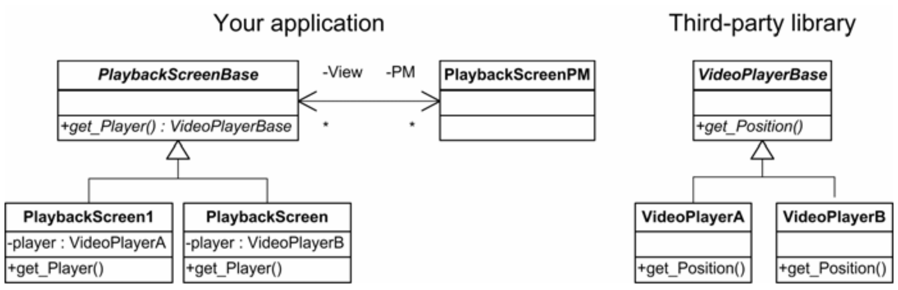

Figure 2
Example scenario

Suppose there is a family of third-party video player controls which have to be used by multiple views of the application (playback screens) – all of which behave the same but look completely different; the only thing they have in common is the presence of any of the video players from the third-party library. The video players have a Position property, without change notification of any kind, which has to be taken into account by the application logic.

The refined Presentation Model pattern described above solves the problem by defining a common abstract base class for all the playback screens (PlaybackScreenBase), with a getter for the player control, constraining its base class only (VideoPlayerBase) – hence making it possible to implement the behavior of all the playback screens in one common – and unit testable – class (PlaybackScreenPM), which, in turn, can reference the video player (regardless of its concrete type and the presence other elements on the UI) and access its regular – non-bindable – properties (like Position), without losing the flexibility of being able to create more and more views for it.

Like every design pattern, the refined Presentation Model pattern has its liabilities. Most of them are present in the original PM pattern too (like having to do the synchronization between the view and the Presentation Model class, as well as between the Presentation Model and the Model layer itself [8] [11]), but there are two notable new ones as well:

- The common abstract base class has to be designed carefully. Adding a common UI part later means all the existing views have to be edited to provide the new getter.
- Not being able to data bind all the important properties can make synchronization between the view and the Presentation Model more complex, which can be the source of inconsistency between the model and the view. Hence it is recommended to access all the bindable properties through data binding, referencing directly the other ones only.

Conclusions

Using WPF with the Presentation Model pattern can siginificantly improve the designer-developer workflow. With small refinements to the pattern, it becomes possible for both designers and developers to work on their parts independently, with tools they know and are productive with, and without having to worry about harming each others work. For this reason, along with the wide range of multimedia capabilities WPF has to offer, creating stunning and good-looking applications, while keeping them intuitive and accessible, becomes easy and natural, making software vendors able to provide the best user experience.

References

[1] Jensen Harris: The Story of the Ribbon, in Proceedings of MIX 08, Las Vegas, United States, March 5-7, 2008

[2] Robert Elliott, Emilie Herman, John Atkins et al.: The Essential Guide to User Interface Design (Second Edition), Wiley Computer Publishing, 2002

[3] Macintosh Human Interface Guidelines, Addison-Wesley Publishing Company, 1992

[4] Bill Buxton: Sketching User Experiences, Morgan Kaufmann Publishers, 2007

[5] Matthew Robinson, Pavel Vorobiev: Swing (Second Edition), Manning Publications, 2003

[6] Michael Kemper, Guido Rosso, Brian Monnone: Advanced Flash Interface Design, friends of ED, 2006

[7] Todd Fine: Windows Presentation Foundation [Online], http://www.dotnet-u.com, 2006

[8] Blaine Wastell, Bob Brumfeld, David Hill et al.: Composite Application Guidance for WPF and Silverlight [Online], www.codeplex.com/CompositeWPF, 2009

[9] David Hill: Building Silverlight & WPF Applications With Prism, in Proceedings of Tech·Ed 2009, Durban, South Africa, August 2-5, 2009

[10] Martin Fowler: Presentation Model, Essay, July 2004

[11] Jonas Follesø: Model-View-ViewModel, in Proceedings of Norwegian Developers Conference, Oslo, Norway, June 17-19, 2009

[12] Karen Corby: Silverlight 2 Control Model, in Proceedings of Professional Developers Conference, Los Angeles,United States, October 27-30, 2008

[13] Jeff Wilcox: Sharing Skills and Code with Silverlight & WPF, in Proceedings of MIX 09, Las Vegas, United States, March 18-20, 2009