A Proposal to Delegate GUI Implementation using a Source Code based Model

Marco Monteiro
School of Technology and Management, Polytechnic Institute of Leiria
Campus 2, Morro do Lena - Alto do Vieiro, Apartado 4163, 2411-901 Leiria, Portugal
marco@estg.ipleiria.pt

Paula Oliveira
Engineering Department, University of Trás-os-Montes e Alto Douro
Quinta de Prados, Apartado 1013, 5001-801 Vila Real, Portugal
poliveir@utad.pt

Ramiro Gonçalves
Engineering Department, University of Trás-os-Montes e Alto Douro
Quinta de Prados, Apartado 1013, 5001-801 Vila Real, Portugal
ramiro@utad.pt

Abstract

In this paper we propose an architecture whose main goal is to improve productivity in user interface development for data-intensive applications. This objective is to be achieved by defining a high level model that describes the user interface structure. That model will be integrated in the source code through non-functional language extensions. Our final goal is allowing developers to define user interface model by adding language extensions to the source code and then acquiring an external software package to which they delegate the implementation of the concrete user interface.

1. Introduction

As the size and complexity of information systems increases, building maintaining and integrating its applications is getting harder. To keep up with that complexity, there's a constant need to improve productivity of the development process in the software industry.

Driven by that need, in this paper we propose an architecture whose main goal is to improve productivity in Graphical User Interface (GUI) development for data-intensive applications. Nowadays developers tend to create GUI by composition of various components. Our final goal is allowing developers to define GUI using language extensions and then acquiring an external software package (which we'll call smart template) to which they delegate the implementation of the concrete GUI.

This paper is organized as follows. Research problem and alternative solutions are discussed in section 2. Proposed solution is presented in section 3 and conclusions on section 4.

2. Overview

Currently, a large number of projects use Component Based Development (CBD), which allows applications development by assembling a set of pre-manufactured components. Each component is a black-box entity, which can be deployed independently and is able to deliver specific services [1].

GUIs are composed of various graphical elements, such as buttons or input fields. When developing GUIs, both the presentation and behavior aspects of those elements are to be considered. Presentation aspects concern the appearance and layout of GUI elements and behavior is related to the interaction between themselves or between them and the underlying code. Using CBD, each GUI element is mapped to a component and presentation or behavior aspects are defined by its properties, methods and events. Also, by using Rapid Application Development (RAD) tools, GUI layout design is made visually through composition of components. Compared to older processes and methodologies the advent of CBD and RAD tools has certainly increased the productivity of GUI development. However, CBD still

hasn't redeemed its promises of reuse and flexibility [2] and there's still a lot of risks, challenges and unresolved issues in CBD [3].

The issue that drove us to study and propose a solution to improve productivity in this area, is related to the process of components composition and configuration. On large or very large applications, the same component can be reused several times on different contexts, which is the main factor for the productivity improvement accomplished by CBD. However, as the number of instances and complexity of components increases, developers time is increasingly spent on the tedious tasks of composing layouts, configuring components and maintaining consistency in presentation and behavior aspects of the GUI components through the entire application.

Next, we'll present some alternative solutions for the described problem, namely Cascade Style Sheets and Templates on section 2.1, Frameworks on section 2.2 and Automatic GUI generation on section 2.3.

2.1. Cascade Style Sheets and Templates

In 1994, Håkon Wium Lie [4] proposed the Cascading Style Sheets (CSS) language, to describe the presentation of a document written in a markup language, usually Hypertext Markup Language (HTML). It enables the separation of document presentation from document content and ensures visual consistency through central configuration. Another concept used in web applications development is the page template, that's a pre-developed page layout used to create new pages from the same design. It's a concept adapted by Microsoft ASP.Net 2 Master Pages. As CSS, page templates also allows developer to create consistent layout and presentation through entire sites or group of pages. Unlike CSS that acts on individual HTML elements, templates acts on entire pages. Both CSS and page templates are great to define presentation aspects, but very limited when defining behavior or interactions between graphical elements. Håkon Wium Lie himself stated that *"CSS was primarily designed to present documents, not user interfaces"* [5].

2.2. Frameworks

The word framework has a lot of meanings, depending of the context. Within Object-Oriented (OO) design perspective, a framework is a set of cooperating classes that makes up a reusable design for a specific kind of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instance of framework classes [6][7].

Frameworks are used in several application domains and at different levels of abstractions. In GUI development context, frameworks can control components creation, deployment, layout and configuration, leaving developer free of those repetitive tasks. Contrary to CSS and page templates, frameworks aren't limited to presentation issues, as they can handle GUI behavior aspects. In the GUI domain, frameworks can be generic, like Apache Struts or MonoRail (Model-View-Controller based frameworks), they can be available from commercial supplier to complement and integrate with their own components catalog or they can be custom made for a specific application or set of applications. In any of those cases, frameworks are proprietary, each having its own architecture, interface, classes and idiosyncrasies, therefore, requiring long learning processes. Also, code produced is interconnected with frameworks hierarchy of classes, which is against separation of concerns and creates non portable code.

2.3. Automatic GUI generation

Automatic generation is potentially the most productive method to develop GUI, since by definition it allows developers to delegate GUI creation to an external application. Proposed solutions to generate GUI automatically are mainly model-based systems, that attempt to formally describe the tasks, data, and users that an application will have, and then use this formal models to guide the generation of the GUI. Some systems automatically design the GUI and others provide design assistance to developers[8]. These models are abstract models, meaning that they don't specify exactly how the GUI is going to look, but rather what elements are to be shown and how they should behave. Systems will then use that abstract description to generate concrete interfaces for various devices. Trying to fit the same application on different devices with different capabilities [9], is what makes automatic GUI generation so complex. Despite a lot of research, model-based automatic GUI generation still hasn't become common in GUI development, in part because building models is an abstract process and better results are often achievable by a human designer in less time [10]. Abstract models can be complex to build and maintain, thus keeping models and applications concrete GUI synchronized can be problematic.

There are also some commercial tools, like Oracle Designer, that generates not only the GUI but complete applications, embracing all development cycle, which is not appropriated to development methodologies like extreme programming [11], where constant changes and very fast prototyping are required. They're usually based on relational database (DB), and use its metadata to generate layouts for data representation. However, resulting applications have behavior limitations, because DB are typically restricted to create, retrieve, update and delete (CRUD) operations.

3. Proposed solution

According to our initial motivation, main goal is to improve productivity in GUI development for data-intensive applications. Based on that premise and on preliminary studies, it was defined a set of 5 guidelines for the solution. First, it should integrate seamlessly with GUI commercial components, avoiding the need to recreate code to generate GUI elements. Second, it should be as easy to use as possible, allowing developers to concentrate on producing business related code. Separation of concerns should be encouraged. Third, it should be flexible, allowing rapid changes at any phase of the product development. Fourth, it should produce very fast prototypes, preferably in a fully automated manner. Fifth, it can't be limited to work with relational DB or XML as data source. Data persistence shouldn't be of any concern to the solution, as stated by separation of concerns principle.

Next, we'll present a description of the proposed solution, namely the general architecture in section 3.1, the GUI Model in section 3.2, the Binding Framework in section 3.3 and Smart Templates in section 3.4.

3.1. Architecture

There are 2 main characteristics for the proposed system architecture that distinguish it from others. First, it'll use an abstract model to characterize GUI, but instead of relying on specialized GUI models, based on XML or any other format, it'll be integrated with the source code through language extensions, which we'll call Graphical User Interface Language eXtensions (GUILX). Second, instead of generating GUI automatically, system will delegate that responsibility to external software packages, which we'll call smart templates (check Figure 1). System will provide the necessary resources to bind the source code model with external smart templates, who will handle GUI implementation for the business code included in the model.

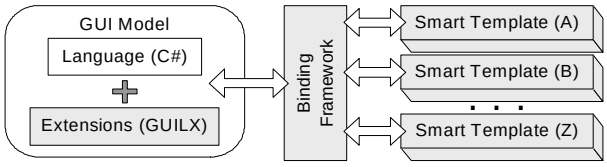

Figure 1. Proposed architecture.

3.2. GUI Model

Although it's not a common technique, using the source code as a model to generate GUI is not original. For instance, in 2004 Jelinek [9], used annotated source code to generate GUI. Despite some similar concepts, Jalinek model uses a tree-rewrite based language, as our model will use a mainstream OO language. That choice was made to facilitate integration with comercial GUI components. We'll use C#, but all concepts are also applicable to the Java language. Source code based models main advantage is the proximity between the model and the code we want to execute. This approach turns development process more flexible, as synchronization between business code and GUI model is no longer required. In our solution, the GUI model is composed of standard C# source code that defines business operations and GUILX language extensions to define the GUI related aspects. These extensions are declarative as they allow developers to define what GUI they want, instead of defining how to build it.

To better comprehend the model, lets analyze data-intensive applications. In those applications, GUI elements such as textboxs or grids, provide data for the user to read or write. Also, users can perform operations by activating events on GUI elements, like clicking on a button or a menu item. Comparing this reality with OO languages, such as C#, there are some similarities, as objects also have data, which can be encapsulated as properties and have associated operations called methods. Objects data and behavior can be mapped in GUI elements or set of elements. For example, considering a business class called "Book" with a read-only string property called "ISBN", a string property called "Title", a boolean property called "Rented" and a method called "Sell". By analyzing source code at run-time through applications metadata, we can generate the GUI elements and layout needed to represent instances of Book objects. GUI elements are chosen by the kind of language elements and accordantly to properties types and accessability (check Figure 2).

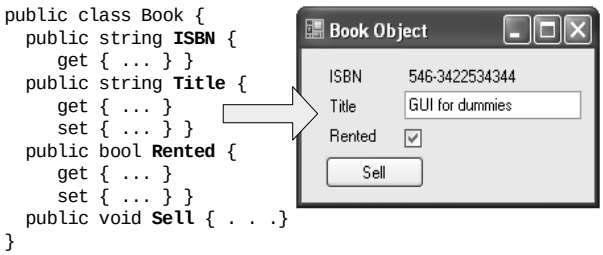

Figure 2. Generation GUI from source code.

Although language metadata has already some useful information, it's not enough for defining a model to generate GUI. Filling that gap is GUILX language extensions responsibility, by enriching the metadata with structural information about GUI. This extensions are implemented by annotating the source code through .Net custom attributes, which provide a way for developers to extend native language by associating declarative information within the source code. "Show" is the first attribute defined on GUILX, indicating which language elements are meant to be available and with what description. If we analyze

second example in Figure 3 and compare it with the first one, we can verify that the checkbox is not shown because "Rented" property doesn't have the "Show" attribute. Also, "Title" property and "Sell" method have different descriptions.

```
public class Book {
  [Show]
  public string ISBN
     { get { ... } }
  [Show("Books title")]
  public string Title
     { get { ... }
       set { ... } }
  public bool Rented
     { get { ... }
       set { ... } }
  [Show("Sell this book")]
  public void Sell() { ... }
}
```

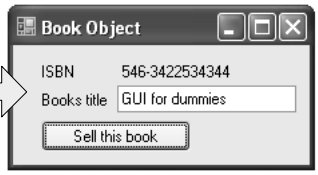

Figure 3. Usage of GUILX "Show" attribute.

Definition of GUILX is still in progress but is critical for the success of the solution. It must be rich enough to ensure that a complete prototype (even if a very basic one) can be generated from the GUI model but simple enough to avoid cluttering the source code with GUI related details.

3.3. Binding Framework

The "Binding Framework" will be responsible for the connection of smart templates and the GUI model. It'll allow the smart template to query the GUI model, to create object instances and to invoke methods of that objects. Also, it'll serve as a controller, maintaining the execution context for the GUI elements, thus controlling navigation through entire application. It must always know what object instance is the user viewing and where to go or what to show next. Every time there's a need to map an object instance to some GUI element, framework will notify the smart template to change interface accordantly.

3.4. Smart Templates

Proposed solution is designed to support various smart templates, one at a time. The idea is allowing developers to define a GUI model and then adquire a smart template to which they delegate all GUI implementation. Smart templates are specialized frameworks, developed by external entities that provide complete GUI services to the GUI model, by complying with the rules specified by the binding framework. There can be smart templates developed by different suppliers, for different devices and using completely different methods. One can generate GUI automatically, other can generate GUI partially and another can generate GUI from manual definitions.

4. Conclusion

Although the proposed solution is still in a embryonic state, we're expecting productivity improvements by allowing developers to focus on business code development and reducing the repetitive tasks of composing layouts and configuring GUI components. Proposed solution is expected to be easier to use than custom frameworks, because learning a declarative language (GUILX) is easier than learning a complete class hierarchy. Also, in the proposed solution, business code doesn't use any direct code related to GUI development, thus ensuring the separation of concerns principle. Compared to other methods of automatic GUI generation, we're also expecting easier development, due to the fact that the model is integrated in the source code, therefore being easier to create and maintain that an abstract model. However, the success of the proposed solution depends on the definition of the GUILX language, which requires a correct balanced between simplicity and flexibility.

References

[1] C. Szyperski, *Component Oriented Programming*. Springer, 1998.

[2] H. de Bruin and H. van Vliet, "The future of component-based development is generation," 2002.

[3] P. Vitharana, "Risks and challenges of component-based software development," *Communications of the ACM*, vol. 46, no. 8, pp. 67–72, 2003.

[4] H. W. Lie, "Cascading html style sheets; proposal." published 10 Oct 1994. Available from: http://www.w3.org/People/howcome/p/cascade.html; accessed on 28/01/2007.

[5] H. W. Lie, *Cascading Style Sheets*. PhD thesis, Faculty of Mathematics and Natural Sciences, University of Oslo, 2005.

[6] I. Jacobson, M. Griss, and P. Jonsson, *Software reuse: architecture, process and organization for business success*. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA, 1997.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1995.

[8] J. Nichols and A. Faulring, "Automatic interface generation and future user interface tools," *ACM CHI 2005 Workshop on The Future of User Interface Design Tools*, 2005.

[9] J. Jelinek and P. Slavik, "Gui generation from annotated source code," in *TAMODIA '04: Proceedings of the 3rd annual conference on Task models and diagrams*, (New York, NY, USA), pp. 129–136, ACM Press, 2004.

[10] B. Myers, S. Hudson, and R. Pausch, "Past, present, and future of user interface software tools," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 7, no. 1, pp. 3–28, 2000.

[11] K. Beck, "Embracing change with extreme programming," *Computer*, vol. 32, no. 10, pp. 70–77, 1999.