GUI GENERATION BASED ON LANGUAGE EXTENSIONS
A Model to Generate GUI, based on Source Code with Custom Attributes

Marco Monteiro
School of Technology and Management, Polytechnic Institute of Leiria, Leiria, Portugal

Paula Oliveira, Ramiro Gonçalves
Engineering Department, University of Trás-os-Montes e Alto Douro, Vila Real, Portugal

Keywords: Attribute-Oriented Programming, Source Code Model, GUI Generation, Data-Driven Application.

Abstract: Due to data-driven application nature and its increasing complexity, developing its user interface can be a repetitive and time-consuming activity. Consequently, developers tend to focus more on the user interface aspects and less on business related code. In this paper, we're presenting an alternative approach to graphical user interface development for data-driven applications, that allows developers to refocus on the source code and concentrate their efforts on application core logic. The key concept behind our approach is the generation of concrete graphical user interface from a source code based model, which includes the original source code metadata and non-intrusive declarative language extensions that describes the user interface structure. Concrete user interface implementation will be delegated to specialized software packages, developed by external entities, that provides complete graphical user interfaces services to the application. When applying our approach, we're expecting faster graphical user interface development.

1 INTRODUCTION

In this paper we propose an alternative approach to Graphical User Interface (GUI) development for data-driven applications. Nowadays developers tend to create GUI by composition of various components. Our final goal is to allow developers to define GUI by adding non-intrusive declarative language extensions to the original source code and then acquire an external software package to which they delegate the implementation of the concrete GUI.

We start by introducing the research problem on section 2, followed by a description of the proposed model on section 3 and conclusions on section 4.

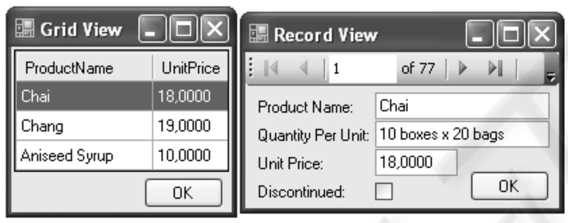
2 OVERVIEW

Currently, a large number of projects use Component Based Development (CBD), which allows application 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 (Szyperski, 1998).

GUIs are composed of various graphical elements, such as buttons or input fields, each with its own presentation and behavior aspects that needs to be considered. Presentation 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, the advent of CBD and RAD tools has increased GUI development productivity.

However, CBD still hasn't redeemed its promises of reuse and flexibility (Bruin and Vliet, 2002) and there's still a lot of risks, challenges and unresolved issues in CBD (Vitharana, 2003). One of those issues is related to the process of component composition and configuration. On large or very large applications, the same component can be reused several times on different contexts, which is a major factor to the productivity improvement accomplished by CBD. However, as the number of instances and complexity of components increases, developer's 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. Developers tend to focus more on GUI aspects and components internals and less on application core logic or business related code. While there are some applications where that makes sense, because User Interface (UI) is the most critical factor for its success, on many others it's important to refocus development time and resources to the core functionalities, as they're the main factor to the applications success. This is particularly true for most data-driven applications[1].

No matter how or where data comes from, as long as applications devices uses the GUI paradigm, data presentation and manipulation follows some patterns that are easily recognizable. For example, there are two typical ways of presenting data: record views and grid views. Record views allows the presentation of all attributes of only one entity instance, while grid views usually presents the most relevant attributes of various instances simultaneously (Figure 1).



ProductName	UnitPrice
Chai	18,0000
Chang	19,0000
Aniseed Syrup	10,0000

Product Name: Chai
Quantity Per Unit: 10 boxes x 20 bags
Unit Price: 18,0000
Discontinued: ☐

Figure 1: Grid and Record View.

The main goal for the solution we're proposing is to improve GUI development by capturing the presentation and behavior aspects of these patterns and therefore reducing or eliminating the need to configure every different view. Analogous principles are used by solutions like Cascade Style Sheets (CSSs), templates, specialized or custom frameworks and automatic GUI generation.

2.1 Automatic GUI Generation

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 (Nichols and Faulring, 2005). 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 (Myers et al., 2000). Abstract models can be complex to build and maintain, thus keeping models and applications concrete GUI synchronized can be problematic.

[1]In this article, we consider data-driven applications as applications that allow users to access and manipulate large amounts of complex data, located on data repositories.

2.2 Attribute Oriented Programming

Attribute oriented programming is a program-level marking technique, that allows developers to mark language elements (*e.g.* classes, methods, and properties) in the source code, to indicate that they maintain application or domain specific semantics. By hiding the implementation details of those semantics from source code, attributes increase the level of programming abstraction and reduce programming complexity, resulting in simpler and more readable programs (Wada and Suzuki, 2005). Attributes can change application runtime behavior by transforming its logic, with the assistance of a supporting generation engine. Dependencies on the underlying middleware are thus replaced by attributes, acting as weak references (Rouvoy and Merle, 2006). With the advent of .Net attributes and Java annotations, attribute oriented programming is now an widespread technique, used in broad-range domains like logging, web services, persistence or security, but also on specific domains like fault-tolerance (Schult and Polze, 2002) or system level modeling and simulation environment (Lapalme et al., 2004).

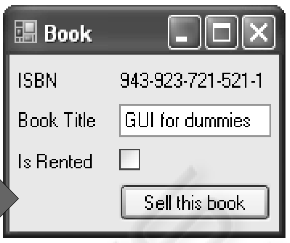
In the context of data-driven applications, attributes can be used to address the problem of object-relational impedance mismatch, which occurs due to the fact that object-oriented and relational paradigms represent information in manners that are quite different from each other (Lodhi and Ghazali, 2007). Object Relational Mapping (ORM) tools, deals with the impedance mismatch problem by providing mappings between the object model and the relational model, which are usually defined by external configuration resources. However, on tools like Hibernate Annotations, Castle ActiveRecord or DevExpress Persistent Objects for .Net, mapping configuration is implemented directly on the source code, using attributes. This approach allows developers to concentrate only on the object-oriented paradigm and delegate to ORM tools the responsibility of handling persistence issues. This leads us to the following question *"why don't we apply the same principles to handle presentation issues"*? That was the question that droves us to study and propose the solution presented in next section.

3 PROPOSED MODEL

The key concept behind our approach is the automatic generation of concrete GUIs from source code based models, as opposed to specialized GUI models used by most automatic GUI generation tools. In 2004, Jelinek (Jelinek and Slavik, 2004) also used annotated source code to generate GUI, but using a tree-rewrite based language. Our model will use a mainstream language (C#), with the purpose of reusing available components and therefore simplifying concrete GUI development. Source code based models main advantage is the proximity between the model and the code we want to execute, improving the integration and interaction between application core logic and application GUI. Also, the process of model creation is quicker and simpler, because part of the model is already defined by original source code. However, mixing business and GUI related code in the same module, it's against separation of concerns principle. To minimize this problem, the model will be defined by new language elements (attributes) that are declarative only, meaning that they will not interfere with original structures or execution flows. Also, to avoid cluttering of source code with GUI related details, it's important to restrict the scope of the model so that it'll describe only structural information about the GUI.

The proposed model, explores some conceptual similarities between data-driven applications GUI and object-oriented languages. On the applications, there are GUI elements such as textboxs or grids that provide data for the user to read or write. Objects can also provide data to external entities, through structures like C# properties. Application users perform operations by activating events on GUI elements, like clicking on a button. Objects also perform operations using methods, which are accessible to external entities through its interface. It's possible to map source code structures into GUI elements, using source code metadata. GUI elements are chosen according to the language element kind, data type or accessibility. However, as original metadata is not enough to describe the GUI completely, language extensions (custom attributes) were added to enrich the metadata with structural information about GUI. Analyzing the example in Figure 2, we can verify that GUI elements were generated according to: the kind of language element, where methods (*e.g.* `Sell`) maps into buttons and properties into labels, textboxs or checkboxs; the data type, where string properties (*e.g.* `Title`) and boolean properties (*e.g.* `Rented`) generates different type of GUI elements; the accessibility, where read-only properties (*e.g.* `ISBN`) are mapped differently of read and write properties. The `Show` attribute (which is our model first attribute) affected the GUI by allowing to define what language elements are meant to be available (*e.g.* `Keywords` property is hidden, because it doesn't have a `Show` attribute) and defining which text is used to describe it.

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


Book

ISBN 943-923-721-521-1

Book Title GUI for dummies

Is Rented ☐

Sell this book

Figure 2: GUI generation from extended source code.

3.1 System Architecture

To prove the viability of the proposed model, a prototype system was developed on Microsoft .Net Framework using C# language. This system conceptual architecture (Figure 3), is composed of three parts or layers: the GUILX Model, the Binding Framework and Smart Templates.

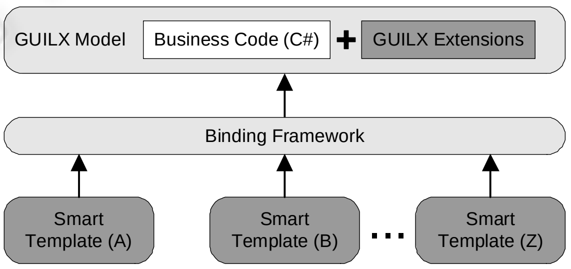


Figure 3: Conceptual architecture.

The language extension model, which we've called the **GUILX Model**[2], incorporates both the original source code (the business code) and language extensions or attributes, as referred on section 3. By principle, these extensions only includes structural GUI information. Concrete GUI details, if applied, should be defined outside the source code model by Smart Templates. However, the GUILX Model must be rich enough to ensure that a functional prototype (even if a very simple one) can be created. Language definition

[2]GUILX is the acronym for **G**raphical **U**ser **I**nterface **L**anguage e**X**tensions.

is still in progress, but besides the `Show` attribute it also includes the `Query` attribute, that's applied only to method parameters. Methods that have at least one parameter marked with the `Query` attribute, are intercepted by GUILX system, so that final application may ask user to input parameter values before method execution (check Figure 4).

```
[Show("Rent this book")]
public void Rent(
[Query("How many Days?", QueryKind.allways)]
int Days)
{
  ...
}
```

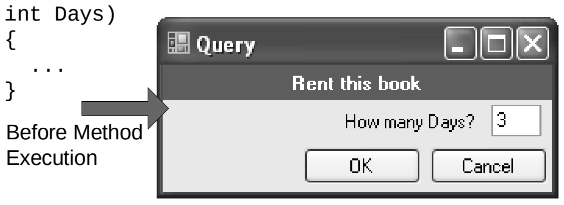


Figure 4: Example of "Query" attribute.

Solution architecture is designed to delegate the implementation of concrete GUI, to external software package, which we've called **Smart Templates**. The idea is allowing developers to define a GUILX Model and then adquire a Smart Template that's suitable for the application GUI. Smart Templates are specialized frameworks, developed by external entities that provide complete GUI services to the GUILX Model. 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 create GUI from manual definitions.

The **Binding Framework** is responsible for the interoperability between Smart Templates and GUILX Model, so that these two layers don't interact directly, thus keeping them independent from each other. It allows the Smart Template to query the GUILX Model metadata, to create object instances and to invoke methods of that objects. Also, it serves as a controller, maintaining the execution context for the GUI elements, thus controlling navigation through entire application.

4 CONCLUSIONS

Proposed model provides an alternative approach to create application GUI, allowing developers to refocus on business code development and delegate complete GUI creation to external software packages, called Smart Templates. Although GUILX language definition is still in an embryonic state, preliminar results already produces functional prototypes, thus proving the viability of our solution. Compared to other methods of automatic GUI generation, we believe our solution is easier to use because it simplifies the process of GUI model creation. Instead of relying on specialized abstract models, it uses a source code based model, which is partially defined by information already present on the original metadata and complemented by the GUILX language extensions.

REFERENCES

Bruin, H. and Vliet, H. (2002). The future of component-based development is generation.

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

Lapalme, J., Aboulhamid, E. M., Nicolescu, G., Charest, L., Boyer, F. R., David, J. P., and Bois, G. (2004). Esys.net: a new solution for embedded systems modeling and simulation. *SIGPLAN Not.*, 39(7):107–114.

Lodhi, F. and Ghazali, M. A. (2007). Design of a simple and effective object-to-relational mapping technique. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1445–1449, New York, NY, USA. ACM.

Myers, B., Hudson, S., and Pausch, R. (2000). Past, present, and future of user interface software tools. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 7(1):3–28.

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

Rouvoy, R. and Merle, P. (2006). Leveraging component-oriented programming with attribute-oriented programming. In *Proceedings of The 11th ECOOP International Workshop on Component-Oriented Programming*, Nantes, France. Monday, July 3, 2006 at ECOOP 2006, (July 3-7, 2006).

Schult, W. and Polze, A. (2002). Aspect-oriented programming with c# and .net. *Object-Oriented Real-Time Distributed Computing, 2002.(ISORC 2002). Proceedings. Fifth IEEE International Symposium on*, pages 241–248.

Szyperski, C. (1998). *Component Oriented Programming*. Springer.

Vitharana, P. (2003). Risks and challenges of component-based software development. *Communications of the ACM*, 46(8):67–72.

Wada, H. and Suzuki, J. (2005). Modeling turnpike frontend system: a model-driven development framework leveraging uml metamodeling and attribute-oriented programming. In *Proceedings of The 8th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, Montego Bay, Jamaica. ISBN: 978-3-540-29010-0.