Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications (SEA'08), November 16 – 18, 2008, Orlando, Florida, USA. pp. 112-118. ISBN: 978-0-88986-776-5.

AUTOMATICALLY CREATING GRAPHICAL USER INTERFACES USING EXTENDED senseGUI LIBRARY

Mariusz Trzaska
Polish Japanese Institute of Information Technology
Koszykowa 86, Warsaw
Poland
mtrzaska@pjwstk.edu.pl

ABSTRACT
Creating GUIs for data-intense application is a time consuming task. A promising approach, saving the developer's time, assumes a declarative way of generating such interfaces. In this paper we present an extended version of our library called the senseGUI. Using simple annotations of the source code, the library is able to automatically generate common business-oriented windows. As a result, an application's user is able to create, update or just see appropriate parts of the data model. Moreover, the extended version of the library gives a programmer an easy way to internationalize the application's interface, validate the input or create GUIs which are not connected to a formal model (a particular class). The simplest usage scenario requires only marking attributes or methods for which widgets should be created. It is also possible to define a more detailed description including different widgets for particular data items, the order of items, labels, etc. Contrary to many existing solutions, our proposal does not require complicated tools. The implemented prototype is dedicated for the Java language but could be easily ported to other languages like, for example, Microsoft C#.

KEY WORDS
Graphical User Interfaces, GUI, Annotations, Model-Based Development, MBD, Attribute-Oriented Programming.

1. Introduction

Finding a usable solution helping in creating Graphical User Interfaces (GUI) is not an easy task. Such efforts have been made from the very beginning of GUIs. As a result, we have plenty of different approaches, libraries and tools. One of the very promising solutions is based on a declarative way. A programmer focuses on what to do rather than how to do it. Choosing the right implementation is a responsibility of a computer system. Perhaps, the most successful examples of the TAK, "the" **J**declarative approach (however, not connected with GUIs) is the SQL query language. A user formulates a query and the system delivers an answer. Using the declarative approach in the database field saves a lot of the programmer's effort because one complicated SQL query could be equivalent to tenths or even hundreds lines of the source code in an "ordinary" language (like Java, C++ or C#). We believe that similar benefits are possible during the process of developing graphical user interfaces. A declarative approach to generating GUIs is also associated with Model-Based Development (MBD) or Attribute-Oriented Programming. Model-Based GUIs (MB-GUI), which assume that GUIs are automatically generated by the software system, are included in the MBD area. Specification of the generation is described in a model. [1] discusses different kinds of models: the Application Model (AM), the Task/Dialog Model (TDM), the Abstract Presentation Model (APM), and the Concrete Presentation Model (CPM). All models are necessary if one would like to define entire applications, including some parts of activity specification (i.e., business logic). Unfortunately, such a description is quite complex and time consuming, hence it is not very popular in the development of commercial applications.
The rest of the paper is organized as follows. To fully understand our motivation and approach some general information is presented in Section 2. Next sections briefly discuss key concepts of our research: utilization capabilities of the senseGUI previous version (Section 3) and its new features (Section 4). Section 5 contains a short description of the utilized architecture and section 6 concludes the paper.

2. Related Work

A researcher designing a new library helping in GUI development has a difficult task, especially in the area of declarative solutions. A potential user expects that the library will be able to create a GUI automatically as the declarative approach promises. At the same time, the user requires conformity to his/her needs regarding functionality, usability, performance or even aesthetics. It is quite obvious that such expectations could not be fulfilled entirely. The library needs some information defining details of the GUI. We believe that the key to success of a given approach is a right balance between a programmer's engagement and universality of the solution. The programmer has to accept that some GUI's facilities will be imposed by the library and will be common for all GUIs generated by the particular solution. Such an approach guarantees that the programmer will

have to pass only a few parameters detailing the result. This could be the case of many business-oriented applications where there are a lot of similar forms.

There are a lot of systems like Teallach [2], Teresa [3], JUST-UI [4], SUPPLE[5] which have features facilitating creating really powerful applications. Unfortunately, most of them work with dedicated model definition languages (i.e. a pattern definition language in the case of [4] or UIML for [3]) and their own platforms. It means that a programmer has to learn something new and quite complicated. Another drawback is the necessity to work with a special platform or system (i.e. Teresa). This is just opposite to our approach which is dedicated to popular languages like Java. Sometimes, new knowledge introduced by the systems is much more complex than the basic problem (GUI to create) which has to be solved. Unfortunately, most of the proposed solutions seem to be too complicated for an average programmer and average tasks.

There are also attempts to introduce a declarative approach in widely used commercial solutions. One of them is Microsoft XAML (Extensible Application Markup Language) [6] which is heavily utilized in Microsoft .NET 3.0, and especially in Windows Presentations Foundation (WPF). It allows defining:

- GUI items: 2D, 3D,
- data binding,
- events,
- special effects: rotation, animation.

Particular GUI items are described using a dedicated description language based on the XML.

```
<Button Content="Click me">
   <Button.Margin>
      <Thickness Left="10" Top="20" Right="10" Bottom="30"/>
   </Button.Margin>
</Button>
<Page    xmlns=http://schemas.microsoft.com/[...]
         xmlns:x=http://schemas.microsoft.com/[...]
         x:Class="MyNamespace.MyPageCode">
   <Button Click="ClickHandler" >Click Me!</Button>
</Page>
```

Fig. 1 Sample MS XAML code

Figure 1 shows a sample XAML code. As it can be seen it contains similar information to the one embedded in the C# source code. In fact, XAML has a direct reflection in the programming language code. Thus, a programmer creating a GUI does not benefit from the approach too much because s/he has to provide a lot of information. The only difference is that the information is given using dedicated XAML file rather than C# code.

3. Basic Capabilities

To fully understand improvements made to our library, this section contains a short description of the existing proposal. More information could be found in [7].

Our goal was to create a library which will automatically generate a GUI form for an ordinary Java class. Using that form, an application's user will be able to enter new data, update existing information or just see the data. For instance, a programmer has the Person class (shown on figure 2) and he/she does not want to manually create a GUI for it. Our basic assumption was that the entire process will be performed without any additional involvement of the programmer.

```
public class Person {
    private String firstName;
    private String lastName;
    private Date birthDate;
    private boolean
    higherEducation;
    private String remarks;
    private int SSN;
    private double
    annualIncome;
    public int getAge() {//
    [...]}
}
```

Fig. 2 Java source code of the Person class

We have created a prototype implementation illustrating our approach as a Java library called the senseGUI. The library needs to read classes' structures in order to automatically generate a GUI for them, which is possible using a technology called reflection. This functionality is available for all modern programming languages including Java, C# and partially C++. Besides reading classes' structures, it allows instantiating their objects.

Unfortunately, it is not possible, in a generic case to automatically discover which part of a data class should be visualized. Hence we had to introduce some kind of markers. The markers were responsible for tailoring the generated GUI to the programmer's needs. Finally, after some research, we decided to use custom Java annotations as markers in our library. They exist for the Java Programming Language and MS C# and allow describing a class or its content. We tried to make them as simple as possible, but the number of the annotation's parameters grew to 11. Fortunately, all of them had some default values and did not need to be changed each time. We introduced two basic kinds of annotations: `GUIGenerateAttribute`, `GUIGenerateMethod`. The former is dedicated to marking attributes and the latter to marking methods. Each of them has additional parameters (with different default values):

- `label`. Describes a label for a widget. If it is an empty string (default) then a name of the attribute or the method (without a prefix get/set) will be used.
- `widgetClass`. Widget class which will be used to handling editing of the attribute or method. Default value is a JTextBox.
- `getMethod`. A method to read a value of the attribute. The default value employs a standard setters/getters approach.

- `setMethod`. A method for writing the value.
- `showInFields`, `showInTable`, `showInSearch` Flags telling if this item should be visible in a form with fields, table (grid) view, search criterion view.
- `order`. A number defining order in a form.
- `readOnly`. If true, then the widget is read only.
- `scaleWidget`. Indicates if this widget should change size during resizing the form.

Our generic solution is capable of working with languages common types: numbers, strings, booleans, dates and enumerations (`enum` type).

Default implementation for most of them uses text boxes, such as widgets, with two exceptions:

- booleans work with a special version of a check box (see further),
- enumerations are processed using combo boxes automatically filled with all possible values (read via language reflection).

Our sample code (the `Person` class), modified using our approach is shown on figure 3. Notice that all we had to do was add annotations telling which parts of the class should have their own GUI.

```
public class PersonAnnotated {
    @GUIGenerateAttribute
    private String firstName;
    @GUIGenerateAttribute
    private String lastName;
    @GUIGenerateAttribute
    private Date birthDate=new Date();
    @GUIGenerateAttribute
    private boolean higherEducation;
    @GUIGenerateAttribute
    private String remarks;
    @GUIGenerateAttribute
    private int SSN;
    @GUIGenerateAttribute
    private double annualIncome;
    @GUIGenerateMethod
    public int getAge() { // ...}
}
```

Fig. 3 Annotated Person class

The library has a few public methods, some of which get an instance of the annotated class. After a single method call, a programmer can show a generated GUI presented on the figure 4. Using the GUI an application's user is able to update the data or just see it.

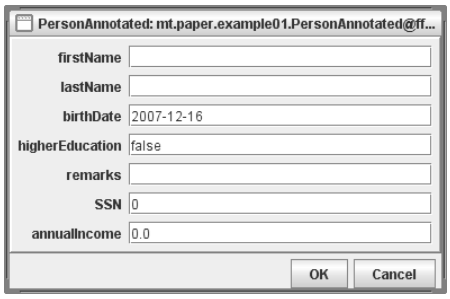

Fig. 4 A form automatically generated for the code from the figure 3.

As can be seen, using annotations with only default values could lead to improper overall form visualization (labels, order, widgets). Thus, a programmer should utilize some of the annotation parameters. A sample source code using annotations with modified parameters is shown on the figure 5.

```
public class PersonAnnotated {
      @GUIGenerateAttribute(label = "First
      name", order = 1)
      private String firstName;
      @GUIGenerateAttribute(label = "Last name",
      order = 2)
      private String lastName;
      @GUIGenerateAttribute(label = "Birth
      date", order = 3)
      private Date birthDate = new Date();
      @GUIGenerateAttribute(label = "Higher
      education",   widgetClass=
      "mt.mas.GUI.CheckboxBoolean", order = 5)
      private boolean higherEducation;
      @GUIGenerateAttribute(label = "Remarks",
      order = 50, widgetClass=
      "javax.swing.JTextArea", scaleWidget
      =false)
      private String remarks;
      @GUIGenerateAttribute(order = 6)
      private int SSN;
      @GUIGenerateAttribute(label = "Annual
      income", order = 7)
      private double annualIncome;
      @GUIGenerateMethod(label = "Age",
      showInFields = true, order = 4)
      public int getAge() { // ...}

      // Standard getters/setters methods
}
```

Fig. 5 The Person class annotated with modified parameters

And the result of generating a form for the code from the figure 5 is presented in the figure 6.

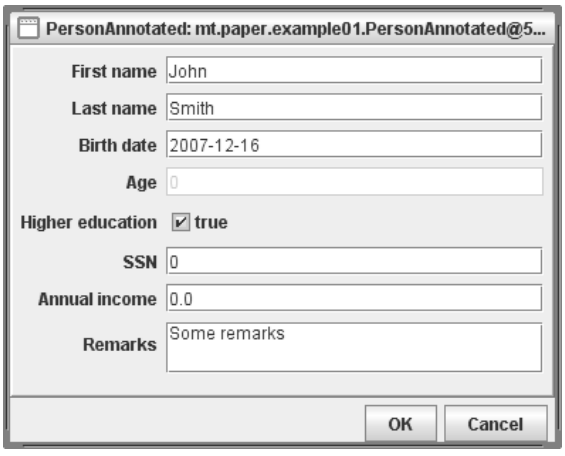

Fig. 6 A form automatically generated for the code from the figure 5.

A few things should be noted:

- All widgets have proper labels,

- An age value is read-only which means that a user is not able to edit the text field,
- Information about higher education is presented using a check box, rather than a text box,
- All widgets are placed in an order explicitly defined by a programmer,
- The current value of the item is automatically placed in the widget,
- The remarks area could have many lines and is properly scaled during resizing the form.

Besides working with basic types and enumerations, our library supports more advanced data types and data-intense operations. The senseGUI library together with our another prototype called senseObjects support a programmer in important data management areas:

- managing the class extent. Every instantiation of the data class is automatically added to the proper extent. Moreover, the instance is added to all extents of super classes, i.e. programmers – employees – persons.
- links between objects. It is possible to create a bidirectional link using just one method call. Furthermore, there are also possibilities of creating:
  - compositions,
  - qualified links,
  - links with the {xor} constraint,
  - links with the {subset} constraint.
- persistency of the classes' extents including all existing connections. It is just one method call to save or load all extents to or from a given file.

More information could be found in [7].

4. New Features

We thought that there are some important aspects of the Graphical User Interfaces which were not covered by our library. Hence we have decided to improve it in the area of:

- internationalization (i18n),
- validation of the entered data,
- GUIs which do not need a dedicated model and are necessary for an application.

The following sub sections describe each aspect of the extended library in details.

4.1 Internationalization

There are many approaches to introducing internationalization (i18n) to an application. Internationalization applies to various aspects of an application such as:

- Messages, labels, texts shown to a user,
- Culturally-dependent data like dates, currencies, numbers formats, sorting order, etc.

Each programming language or framework has its own solution. The same situation is for the Java language, for which our prototype has been developed. Hence we have decided to follow the Java rules.

The new version of the prototype library focuses on translating GUI items like labels, messages, etc. Our goal was to introduce the new functionality to the library in such a way that backward compatibility will be preserved.

As Java designers suggest we used a `ResourceBundle` class, which is some kind of a mapper between a key (i.e. name of the label) and its value (the label's translation in the particular language). An instance of the class is passed to the overloaded method which creates a GUI. If the method detects an invalid resource bundle then there are no changes in the label's processing. However, if the passed bundle is valid, then the label parameter is specially processed. Text entered as a label is treated as a key in the bundle, which means that the translated value is retrieved and used in the GUI. Such an approach guarantees that previously written programs will work with the new feature.

4.2 Validation of the input

Verifying data entered by a user is an important aspect of almost every graphical user interface. Of course, it could be performed on the model level (i.e. inside the setter method) but this solution could mean problems in communicating with the GUI. It is better to assume that the validation will be on the GUI level before the data is/are passed to a model.

Designing a right validators' architecture implies finding answers to the following questions:

- how to define validation rules?
- how to connect the rules with a particular widget and its data?

Because one of the goals in designing our library was ease of use, we decided to use an ordinary programming language (Java) for describing the rules. A programmer does not have to learn special validation languages or platforms. All he/she has to do is implement a very simple interface (see figure 7) in a validator object.

```
public interface Validator {
    public String isInvalid(String labelName,
                            String enteredData);
}
```

Fig. 7 Java interface for validators objects

The interface has only one method which returns a message describing the invalidated data, or null if the entered data is correct. Initially the method had only one parameter passing the just entered data. But we have found out that it is useful to use the passed label names in the message because we can create more generic validators.

Another issue which had to be solved was connecting implemented validators with appropriate parts of a model and the widget. At the beginning we plan to introduce a

dedicated parameter for the annotation. The parameter would have type of the `Validator` interface. Unfortunately, Java's annotations parameters could be only of primitive types, String, Class, enum or array of the above types. Hence our idea is not implementable directly. Of course, we could pass the name of the class as a string and utilize the reflection to instantiate the object. But how to define additional parameters for the validators? For instance, we would like to limit a number's range and we need to provide minimum and maximum values which will be different for various classes. Using just strings seems very complicated.

In that situation, we have changed our approach and have abandoned the idea of using annotations for validating input. Instead, we have introduced a map containing labels' names as keys and validators as values. The validators could be added in two ways:

- by passing a map to the method showing a GUI (we made another method's override just like in the case of internationalization),
- by calling a method in the class defining the generated GUI (returned by the above method).

Such an approach ensures a high level of flexibility because simple validators could be created using anonymous types (figure 8) and more sophisticated ones could be reused.

```
SingleObjectPlusFrame personFrame =
        GUIFactory2.getObjectFrame(person, resourceBundle);

// Add an ad-hoc custom validator
personFrame.addValidator("firstName", new Validator() {
    @Override
    public String isInvalid(String labelName, String enteredData)
    {
        if(enteredData == null || enteredData.length() < 1) {
            return labelName + " must be provided!";
        }

        return null;
    }
});
```

Fig. 8 Adding a validators using anonymous types

We have implemented a few sample validators checking if an input is not empty, is a number, a number is within a given range or a string conforms to a given regular expression. The figure 9 shows utilization of such a validator.

```
personFrame.addValidator("annualIncome", new
        ValidatorRange(20000, 100000));
```

Fig. 9 Adding a predefined validator

The validation procedure is started before updating the model. If it fails appropriate message is shown to the application's user describing all validation problems. The model is only updated if the user's input is successfully verified.

4.3 Ad Hoc GUIs

Usually we need a GUI for a model or part of the model. This kind of GUI could be automatically (based on annotations) generated by the first version of our library. But sometimes there is a need to get some information from a user for which we do not have a complete model. For instance, our system needs to connect to an outside database and we need a way to provide its location and port. Such GUIs we call Ad Hoc GUIs and we have created a dedicated functionality to work with them. The functionality is available via a simple method (which has a few overloaded versions), with two main parameters:

- an array of strings describing widgets to create. Each description contains a label and optional default value. It is worth mentioning that labels also support the new internationalization feature of our library.
- an instance of the `AdHocActionPerformed` interface (figure 10).

Similarly to our previous interfaces, this one is also very simple and contains only one method.

```
public interface AdHocActionPerformed {
    public void Accept(Map<String, String> enteredData);
}
```

Fig. 10 The Java interface utilized by Ad Hoc GUIs

The method is called when a user clicks the first button (with a customized label) of a generated GUI and passes the entered data. The body of the method should contain some actions working with the entered data. The second button simply cancels/hides the window.

```
String[] definitions = {"Database.addres",
        "Database.port:3306"};
SingleObjectPlusFrame adHocFrame =
GUIFactory2.getAdHocFrame(definitions, "Database
information", "Connect", new AdHocActionPerformed() {
    @Override
    public void Accept(Map<String, String>
    enteredData) {
        String enteredDbAddress =
                enteredData.get("Database.addres");

        // Do something with entered data, i.e.
        connect to the database
    }
}, resourceBundle);

// Add validators
adHocFrame.addValidator("Database.addres", new
        ValidatorNotEmpty());
adHocFrame.addValidator("Database.port", new
        ValidatorRange(0, 65535));
```

Fig. 11 The code for creating a sample Ad Hoc window

The figure 11 contains a sample Java code with a method which will be called when an application's user clicks the "Connect" button. The generated GUI is presented on the figure 12.

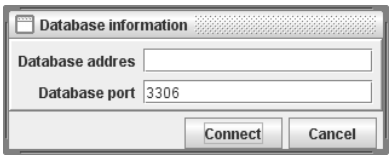

Fig. 12 The Ad Hoc GUI generated automatically for the code from the figure 11

A few things should be noted:

- There is access to entered data inside the provided method. Entered values are available via the map based on the labels.
- Ad Hoc GUIs work also with validators (bottom part of the code from the figure 11) as the rest of the senseGUI library,
- The form's title and the button's label are customizable.

5. Design and Implementation

The senseGUI library contains more than 30 classes so this section describes only a small part of the overall architecture. Figure 13 shows a simplified view of the components constituting the senseGUI typical window. The window consists of:

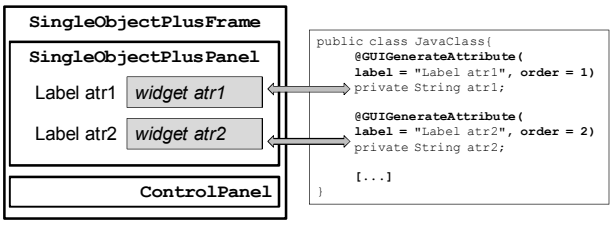

Fig. 13 A simplified view of the components constituting the senseGUI typical window

- an instance of the `SingleObjectPlusFrame` class which describes the generated window,
- `SingleObjectPanel` storing all generated labels and widgets for annotated parts of the model (class). The panel could be utilized independently, which greatly improves reuse and customization of the final window. The panel works with methods:
  - reading class structure,
  - getting initial values from associated model,
  - validating entered data,
  - updating the model.
- an object of the `ControlPanel` class which contains typical buttons like "Accept", "Cancel" and associated business logic.

It is worth noting that we tried as much as possible to follow the Java guidelines. Such an approach makes easier using and modifying the library by a developer already familiar with the Java platform.

6. Conclusion

We have presented new capabilities of the senseGUI library. The library allows automatic generation of forms for typical business-oriented applications. The GUI is generated for the Java classes annotated with simple markers. The markers (annotations) have some parameters which allow tailoring the result. A developer is able to show such a form using just one method call. Thanks to our library, an application's user is capable of seeing data and modify them.

The new features significantly extend the senseGUI's existing functionality, namely:

- internationalization allows preparing different translations of the GUI,
- validation performs verifying data entered by a user. The process has been designed to be straightforward yet powerful (a programmer needs to implement only one interface with one method),
- Ad Hoc GUIs make possible creating forms for entering or modifying data which do not have a formal (a class) model. With one simple method call, passing fields definition and method to execute, a programmer is able get input from a user.

Our future work will continue on researching declarative ways of creating Graphical User Interfaces because we believe that there is a bright future for the approach.

References

[1] P. da Silva, User interface declarative models and development environments: a survey. *Proceedings of DSVIS 2000*, Limerick, Ireland, 2000, 207–226.

[2] T. Griffiths, P. Barclay, et al, Teallach: A model-based user interface development environment for object databases, *Interacting with Computers, vol.1*, 2001, 31-68.

[3] G. Mori, F. Paterno, C. Santoro: Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions, *IEEE Transactions on Software Engineering, 30*(8), 2004, 1-14.

[4] P. Molina, S. Meliá, O. Pastor, JUST-UI: A User Interface Specification Mode, *Proceedings of CADUI 2002*, Valenciennes, France, 2002, 63-74.

[5] K. Gajos, D. Weld, SUPPLE: Automatically Generating User Interfaces, *Proceedings of IUI'04*, Funchal, Portugal,2004, 83-100.

[6] M. MacDonald, *Pro WPF in C# 2008: Windows Presentation Foundation with .NET 3.5 Second Edition* (Apress, 2008).

[7] M. Trzaska, senseGUI – a Declarative Way of Generating Graphical User Interfaces. *Proceedings of the Third International Conference on Software and Data Technologies (ICSOFT 2008),* ISBN: 978-989-8111-51-7, Porto, Portugal, July 5 – 8, 2008, 71-76.