

senseGUI – A DECLARATIVE WAY OF GENERATING GRAPHICAL USER INTERFACES

Mariusz Trzaska

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

Keywords: Graphical User Interfaces, GUI, library, tools, Model-Based GUIs.

Abstract: A declarative way of creating GUIs is also known as model-based generation. Most of existing solutions require dedicated tools and quite complicated knowledge from the programmer. They also use special languages. In contrast, we propose a method which utilizes annotations existing in present programming languages. The method greatly improves generating common GUIs for popular languages. Annotations allow the programmer for marking particular parts of a source code defining class structures. Using such simple annotations, the programmer can describe basic properties of the desired GUI. In the simplest form it is enough just to mark attributes (or methods) for which widgets should be created. There is also a way to define more detailed description including labels, the order of items, different widgets for particular data items, etc. Using a generated form, the application user can create, edit and see instances of data objects. Our research is supported by a working prototype library called senseGUI (Java).

1. INTRODUCTION

According to (Basnyat, 2005), the Model-based development (MBD) is a developing trend in the domain of software engineering that advocates the specification and design of software systems through declarative models. A part of this approach are Model-Based GUIs (MB-GUI) which assume that GUIs are automatically generated by the software system. Specification of the generation is described in a model. (da Silva, 2000) discusses different kinds of models. All the models are necessary if one would like to define entire applications, including some parts of activity specification (i.e., business logic). Unfortunately such description is quite complex and time consuming, hence it is not so popular in the development of commercial applications.

Generally, developers utilize three main approaches to creating graphical user interfaces:

§ Defining GUIs using manually written source code. Every popular programming language has its own dedicated libraries. In case of Java it could be Swing (Walrath, 2004) or SWT

(Guojie, 2005). C# developers have WinForms (Sells, 2006);

§ Utilizing dedicated visual editors (designers) which allow for “drawing” a GUI and for generating an appropriate source code. The quality of such generators varies considerably. Some of them allow for round-trip engineering (i.e. (Jigloo, 2008)). In contrast, there are also solutions which act as pure generators;

§ Using a special declarative approach including MB-GUI. The idea is to focus on “what to do” rather than “how to do it”. A recent, commercially used example of such an approach is MS XAML. Particular GUI items are defined using a dedicated programming language (or a description language).

Unfortunately, all of the presented approaches require quite serious involvement from the programmer. Starting from the first one, which is the most time-consuming (and needs also specified knowledge), through the second one, which of course saves time but needs a lot of attention, up to the last one which cuts some effort, but a little bit. Hence our idea was to develop a solution which is very easy to use yet powerful and does not require so much programmer’s effort. Our research is

supported by a working prototype implemented for the Java language as a library, called senseGUI. The library could be applied to any program written in Java. It is worth mentioning that following our concepts it is possible to create implementations for any language which supports reflection.

The rest of the paper is organized as follows. To fully understand our motivation and approach some general information are presented in section 2. Next sections briefly discuss key concepts of our research: utilization capabilities (Section 3) and technical overview (Section 4). Section 5 concludes.

2. RELATED WORK

In general terms, an ordinary application's user needs a Graphical User Interface as an:

- § Input. To fill a data (model) elements with some content. To achieve this, a programmer creates widgets (i.e. text box) and connects them with data. When a user enters some data to the widget, a dedicated part of the program, writes them to the model;
- § Output. To show a content of data or a model. To accomplish this, a programmer writes a code which reads a part of the application's model and writes it to a widget.

The most common way of fulfilment input/output needs is utilizing a GUI library delivered with a given programming language. Most of Java's GUIs are implemented using Swing (Walrath, 2004) or SWT (Guojie, 2005) libraries. The following listing contains definition of a simple Java class:

```
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() { // [...] }
}
```

In case of creating GUI for the class, we need to write a source code performing the following steps (aside adding necessary "model" methods):

- § Create an empty form;
- § Add a layout manager;
- § For each needed attribute add a widget which will show its content and will allow edition;

- § For each widget add a describing label;
- § For each widget add a code which will read the value of a particular attribute and will put it into the widget;
- § Add "Accept" button which will read widgets' contents, update appropriate attributes and will hide the form;
- § Add "Cancel" button hiding the form.

Implementing the above steps means writing a few tenths lines of code (7 attributes multiplied by 5 to 10 lines per widget plus handling layout, control buttons, etc), which are quite similar to each other.

Different approach has been utilized in the GUI editors concept. One of them is Jigloo GUI Builder working with the Eclipse IDE platform. Using the editor one can visually draw a form by placing appropriate widgets. An example, for our sample `Person` class, is presented on fig. 1. For the figure, the editor has generated 105 lines of Java code. This number is without a code needed to read/write values from/to the data instance, which should be written manually. Comparing to hand coding GUI, using an editor is a big facilitation. However, the programmer has to spend some time on placing widgets in a window, adding "data code" and handling resizing the window (which is not always easy to achieve).

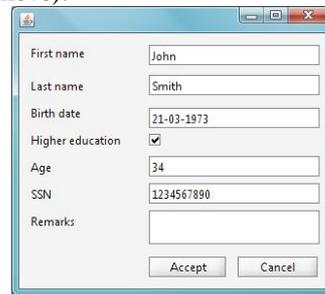


Figure 1: A sample form designed using Jigloo GUI editor

We believe that, in the case of typical graphical user interfaces, i.e. forms for editing or entering data, the most promising approach is the declarative one. The reason is that a programmer focuses on defining what he/she would like to achieve, rather than how to do it.

Systems like Teresa (Mori, 2004), JUST-UI (Molina, 2002), SUPPLE (Gajos, 2004) support a declarative approach and have features which facilitate creating really powerful applications. Most of them work with dedicated model definition languages (i.e. pattern definition language in case of (Molina, 2002) and their own platforms. It means that a programmer has to learn something new and quite complicated. Sometimes, the 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.

3. senseGUI CAPABILITIES

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 and he/she does not want to manually create a GUI similar to the one in fig. 1. Our basic assumption was that the entire process will be performed without any additional involvement of programmer. To make sure if our library is useful we have decided to create a working sample of a typical business application: a software for a video store.

We had to solve three main problems:

- how to read a class content (structure)?
- what kind of widgets should be generated for particular class items (attributes)?
- how to connect generated widgets with data?

The first problem was quite easy to solve. We have decided to employ technology called reflection. This functionality is available for all modern programming languages including Java, C# and partially C++. It allows reading classes' structures and instantiating their objects.

Answers to the rest of the questions are presented in the next sections.

3.1 Basic usage

Roughly speaking 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 generated GUI to programmer's needs. The open question was how to connect the markers with a program, framework and a working GUI? There were some options like configuration files (i.e. XML or Java-properties) or passing parameters to the library. Finally, after some research, we decided to use another approach: annotations. They exist for the Java and MS C# and allow describing a class or their content. We tried to make them as simple as possible, but during the implementation of our sample application, the number of 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 first one is dedicated to marking attributes and the later 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 `JTextField`.
- § `tooltip`. A short text presented to the user when a mouse cursor hovers over the widget.
- § `getMethod`. A method to read a value of the attribute. 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). For objects see section 3.2.

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

- § booleans work with special version of 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 below. Notice that all we had to do was adding 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() { // ...}
}
```

The result of calling appropriate (single) senseGUI method showing the generated window is shown on fig. 2

Figure 2: A form generated automatically by the senseGUI

As it can be seen, using annotations with only defaults values could lead to improper overall form visualization. Thus, a programmer should utilize some of the annotation parameters. Part of the appropriate sample code is shown below and its result frame is presented on fig. 3.

```
@GUIGenerateAttribute(label =
"First name", order = 1)
private String firstName;
@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;
@GUIGenerateMethod(label = "Age",
showInFields = true, order = 4)
public int getAge() { // ...}
```

A few things should be noted:

- § All widgets have proper labels,
- § An age value is read-only,
- § Information about higher education is presented using a check box,
- § All widgets are placed in an order explicitly defined by a programmer,
- § Current value of the item is automatically placed in the widget,
- § Remarks area could have many lines and is properly scaled during resizing the form.

Figure 3: A form generated automatically by the senseGUI based on modified annotations.

3.2 Advanced possibilities

We have mentioned that our library works with some common data types (including the basic ones). What if a programmer would like to show the content of a custom defined type embedded in another custom type (i.e. person has a job information described using a dedicated class)? Generally speaking, there are two ways of achieving such a functionality.

The first one takes advantage of another capability of our solution. As we said, a programmer can generate an entire window for a class in just one method call. However, behind the scene, the call utilizes other methods. One of them is a method which creates a panel containing all data widgets. Then, the panel is merged with some other control widgets and embedded inside a window. The window is shown to a user, allowing performing requested operations. The idea is based on utilizing just the widget panel and embedding them in a custom designed window.

Another approach requires cooperation with our other library called senseObjects. The library is a result of our research regarding supporting common data management using a solution native for a particular programming language (i.e. without using a database). Similarly to the senseGUI, the current prototype is developed for the Java and requires minimal programmer's involvement. In order to use it, a programmer has to:

- § inherit his/her data objects from a particular class,
 - § call the constructor from the super class.
- The senseObjects library supports 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.

Going back to our problem with presenting in one window information described using another class (i.e. jobs “within” a person’s window), we can link instances of Job class with our Person object. Thanks to this approach it is possible to utilize the part of the senseObjects functionality, which is dedicated to managing links.

The data management library (senseObjects) together with the senseGUI add an extra automatic functionality regarding links. Using a special annotation (similar to the one dedicated to attributes) it is possible to manage links between data objects using automatically generated GUI. Special annotation (@GUIGenerateAssociation) describing class’s connections consists of an array with definitions of particular links. We had to use an array because a class could have many associations. Each of the link’s definition contains the following elements:

- § Name of the role (UML semantic),
- § Name of the reverse role,
- § Cardinalities,
- § Qualified name of the target class,
- § Information if a link is read-only.

Note that the PersonAnnotated class inherits from the senseObject class and calls the constructor from the super class.

The following code (without attributes) allows generating a form presented on the figure 4.

```
@GUIGenerateAssociation(definitions=
{"Employer; Employee; 1; *; mt.paper.
example01.Company; readOnly=false"})
public class PersonAnnotated
extends SenseObject {
    public PersonAnnotated() {
        super();
    } // Attributes with annotations
}
```

Using buttons generated by the library it is possible to modify information about the connections:

- § Select an existing target object,
- § Add a new target object (create it),
- § Edit target object’s information,
- § Remove a selected link.

Figure 4: The generated window with links management.

Another useful functionality provided by the library is a table (grid) view of a group of objects. The dedicated method, which provides an access to the function works with just two parameters:

- § a collection containing objects to show,
- § a class definition object (subclass of the Java Class class).

A sample window showing information about instances of the PersonAnnotated class (see previous source codes) is presented on the fig. 5. The window has been shown using only one line of Java code. It also supports selecting a particular object (from the shown ones) which could be processed later.

First name	Last name	Birth date	Age	Higher ...	SSN	Annual in...	Remarks
John	Smith	2007-12-...	0	false	123456789	48000.0	Good worker
Alex	Brown	2007-12-...	0	true	987654321	53000.0	Excellent id...
Tom	White	2007-12-...	0	false	543219876	26000.0	

Figure 5: The table window generated by the library.

The last (but not least) capability of our solution, which is worth mentioning, is automatic searching of objects based on a parameters entered by a user. It is really easy for a programmer to show a dedicated dialog which allows inputting values of searched items. We use the term “item” rather than an attribute, because it is also possible to search within results of a method (if they were properly annotated – they have own widgets). During the search process, which could be time-consuming, an application’s user see an animated progress bar.

We would like to emphasize that from a programmer’s point of view, all of these functionalities have been run using only a few

simple annotations in data classes and a couple lines of Java code (needed to show the generated form).

4. DESIGN AND IMPLEMENTATION

From the very beginning, our solution has been designed as a generic one. It means that most of the functionality could be utilized with any Java classes. Moreover, our approach may be implemented for other languages supporting reflection, i.e. MS C#.

One of the problems which had to be solved, was exchanging data between class items (attributes and methods) and widgets. After some research we decided to use the following approach:

- § For every annotated attribute, there must be a read method named as the attribute with get prefix, i.e. `getLastName()`. Annotated attributes could be of any basic type, Data, String or enumeration.
- § If the marked attribute is writeable (not read-only), then there must be a similar method but with a “set” prefix, i.e. `setLastName(...)` with an appropriate parameter.
- § The above rules could be modified by explicitly defined methods.
- § A widget used to visualize/modify the item has to have methods: `setText(...)` and `getText()` with parameter and return types of String. The methods are used to set and read widget’s state (based on the states of the connected item). Most of existing widgets (i.e. `JTextField`) work that way. However, if a widget does not have the methods a simple wrapping must occur. This is the case of `boolean` type and `JCheckBox`. We have provided a new class `CheckBoxBoolean` which simply overrides the mentioned methods.
- § A special case are enumerations. We have created a generic widget (`ComboBoxEnum`) which automatically works with all enumerations defined by a programmer.

The similar situation occurs for generating GUI for methods (`GUIGenerateMethod` annotation).

5. CONCLUSION

We have presented the result of our research regarding automatic generation of Graphical User

Interfaces for business applications. The generated GUI is based on annotations of data items processed by a developed application. A programmer chooses which attributes or even methods should be reflected as widgets and writes only a couple of source code lines. Then, our library automatically generates windows for creating, editing or presenting data.

Our contribution could be evaluated from two points of view. Firstly, our approach could work with popular technologies (i.e. it works with Java and Swing) and would not require any dedicated systems or sophisticated interface description language. Secondly, the working prototype has been developed for the Java language and Swing GUI library. It is possible to develop similar libraries for other programming languages supporting reflection.

Our future work will focus on formal usability and performance tests. However, informal benchmarks show that the performance is good enough for small and medium size systems.

In our future work we would like to focus on improving our approach, because, in our opinion, the declarative (model-based generation) way of creating GUIs has a big potential which could save a lot of programmers’ time.

REFERENCES

- Basnyat S., Bastide R., Palanque P.: Extending the Boundaries of Model-Based Development to Account for Errors. MDDAUI '05. 2005.
- da Silva P.: User interface declarative models and development environments: a survey. Proceedings of DSVIS 2000, 2000, pp. 207–226.
- Gajos K., Weld D.: SUPPLE: Automatically Generating User Interfaces, in Proceedings of IUI04, Funchal, Portugal, 2004, pp.83-100.
- Guojie J. L.: Professional Java Native Interfaces with SWT/JFace. ISBN: 978-0470094594. Wrox. 2005.
- Jigloo SWT/Swing GUI Builder:
<http://www.cloudgarden.com/jigloo/>.
- Molina P., Meliá S., Pastor O.: JUST-UI: A User Interface Specification Mode, in Proceedings of CADUI 2002, Valenciennes, France, 2002, pp.63-74.
- Mori G., Paterno F., Santoro C.: Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions, IEEE ToSE, 30(8), 2004, pp.1-14.
- Sells Ch., Weinhardt M.: Windows Forms 2.0 Programming. ISBN: 978-0-321-26796-2. AWPddison Wesley Professional. 2006.
- Walrath K., Campione M., Huml A., Zakhour S.: The JFC Swing Tutorial (2nd Edition). ISBN 0201914670. Prentice Hall. 2004.