8

GUIs without Pain – the Declarative Way

Mariusz Trzaska
Polish Japanese Institute of Information Technology
Poland

1. Introduction

Graphical User Interfaces (GUIs) are required by almost all modern applications. Generally, developers utilize three main approaches to creating them:

- 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, 2009)). In contrast, there are also solutions which act as pure generators;
- Using a special declarative approach. 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, most of the presented approaches require quite serious involvement from the programmer. The first one, is definitely the most time-consuming and also needs specified knowledge. The second one, saves some time but needs a lot of attention during designing process. The last one utilizes probably the easiest approach for having a decent user communication layer in an application. Following the declarative way, a programmer focuses on what to do rather than how to do it. Such a method saves time and ensures less programming errors in the final product.

In this chapter we would like to:

- Present existing declarative solutions,
- Briefly describe our previous proposal for the Java language: the senseGUI library (Trzaska, 2008) and fully discuss the new one called the GCL language.

Both of them have been implemented and are publicly available (together with source codes) using the following addresses: http://go.mtrzaska.com/?sensegui and http://gcl-dsl.googlecode.com/.

The first prototype called the senseGUI utilizes annotations existing in the Java language (they also exist in other programming languages like MS C#). The 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) in an ordinary Java class for which widgets should be created. There is also a way to define more detailed

Source: User Interfaces, Book edited by: Rita Mátrai,
ISBN 978-953-307-084-1, pp. 270, May 2010, INTECH, Croatia, downloaded from SCIYO.COM

descriptions 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 newest proposal is a Domain Specific Language (DSL) called GCL. The language has been implemented as a library which mimics syntax of another language. We took into account our experiences gathered during design and implementation of the senseGUI library. As a result the new library is much more flexible and does not require modifications (marking with annotations) of the source (model/data) code. Hence it is possible to use it even with Java programs for which we do not have source code.
It is also worth mention that both solutions could be easy ported to other popular languages like Microsoft C#.
The rest of the chapter is organized as follows. To fully understand our motivation and approach some related solutions are presented in Section 2. The next section briefly introduces the concept of Domain Specific Language, which has been utilized for the GCL. The 4-th and 5-th sections describes the GCL functionalities and sample utilizations. The last one concludes the chapter.

2. Related solutions

2.1 The typical way

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.

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

Listing 1. A sample Java class

Let's consider a simple Java class presented on Listing 1. 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 Figure 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).

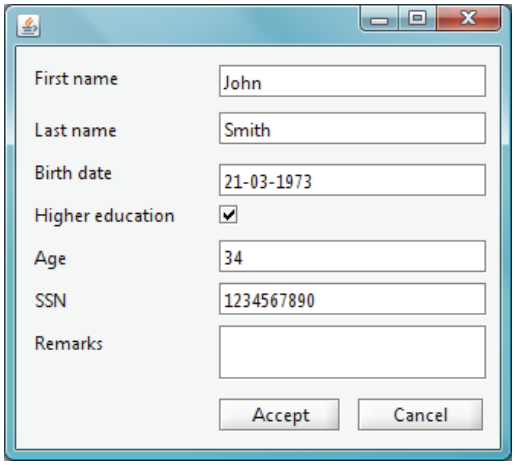

Fig. 1. A sample form designed using Jigloo GUI editor

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.

2.2 The declarative way

In our opinion the most useful declarative solutions are those which raise the level of abstraction. Such an approach considerably simplifies a programmer's job and decreases the number of errors. However, the common side effect is some kind of similarity of the generated GUIs. This is caused by the fact that the majority of the GUI appearance and behaviour is defined inside the library and the programmer only "guides" the tool with some details. Of course, it is possible to create much more customizable library.

Unfortunately that means providing a lot of details by a programmer, which could cause complexity similar to the classical methods.
In most cases utilizing a declarative approach means introducing some kind of a Domain Specific Language (DSL). The DSLs are quite extensive area (much bigger than just GUIs) thus they deserve a dedicated section.

3. Domain specific languages

According to the (Deursen, 2000) Domain Specific Languages (DSLs) offer an expressiveness power usually focused on a particular application or technical domain. They utilize a special syntax, semantics and work on a quite high level of abstraction. DSLs often employ a declarative approach which means specifying the job to do rather than describing how it should be done. As a result, a person using a DSL expects improvement in the process of developing software. The improvement could mean saving a programmer's effort, better quality of the system, shorter time to market, fewer errors, and, last but not least, less typing.
The DSL concept is not quite new. In (Visser, 2008) we can find information about roots of the DSLs in Fortran language in late 1950s. Even one of the most successful examples of the idea, the SQL query language has been defined in 1970s but is still widely used nowadays. Since the 2000s we can observe the rising popularity of DSL languages in a wide range of fields and utilizations:

- as visualization tools. An interesting example developed within the purely functional language Haskell is described in the (Borgo, 2008). The language provides a set of primitives and other structures combing them into bigger structures. As a result, it is possible to create different post-processing of images together with animations;
- to specify content and behavior of advanced HMIs (Human – Machine - Interactions). The language described in the (Bock, 2006) has been designed to generate prototypes especially for testing usability. Thanks to the simple visual syntax and semantics the DSL acts as a common layer for all members of an interdisciplinary software production team allowing them to understand major aspects of the application;
- to develop distributed Web-based applications. The paper (Nussbaumer, 2006) presents a system where domain experts directly contribute to the development process by utilizing dedicated DSLs. Hence the web application was composed from various blocks which behaviour were specified with the languages;
- to test software. In the case of the system described in the (Freeman, 2006), the DSL has been used for the "mocking" process. It means mimicking the behaviour of some real objects linked with tested objects;
- to create Graphical User Interfaces. Some of them are described in the following paragraphs.

The (Bravenboer, 2004) introduces an interesting DSL called SWUL (Swing User-interface Language). The language has been developed using MetaBorg which provides concrete syntax for domain abstractions. It utilizes a preprocessor concept: a programmer utilizes a dedicated tool to transform a defined DSL language into a "real" language, which is further compiled using its native tools. Listing 2 presents a sample SWUL code.
The readability of the code is much better than the Java with the Swing components. The structure of the GUI is more explicit and roles of particular constructs are self-explanatory. However, the level of abstraction is quite similar to the one represented by Java. A

programmer who would like to implement a typical GUI – model interaction (Create/Retrieve/Update/Delete) has to write a similar amount of code like in pure Java. Another disadvantage is the special pre-compiler which has to be utilized every time before the "real" Java compilation occurs.

```
JFrame frame = frame {
  title = " Welcome !"
  content = panel of border layout {
    center = label { text = " Hello   World " }
    south = panel of grid layout {
      row = {
        button { text = " cancel " }
        button { text = "ok" }
      }
    }
  }
};
```

Listing 2. A simple SWUL code

```
group(customerInfo, <nameInput, ageInput>).
group(nameInput, <customerLabel, customerInputField>).
group(ageInput, <ageLabel, ageInputField>).
above(customerInfo, shoppingBag).
above(shoppingBag, checkOutButton).
oneColumn(nameInput).
oneColumn(ageInput).
oneRow(<nameInput, ageInput>).
```

Listing 3. Definition of component relations in DEUCE

The (Goderis, 2007) describes the DEUCE framework which utilizes another DSL called SOUL defined on top of Smalltalk. The two languages are used to implement the entire structure and behaviour of an application. The system allows for defining rules which could concern different aspects including an automatically generated GUI. For instance Listing 3 shows rules describing some components relations among customers and a shop.

The idea is interesting but requires further research. Especially, considering performance for real-world applications. Another unsure aspect is the ability and usefulness to describe the whole system using just rules.

There is also a big group of solutions introducing different DSLs based mostly on the XML syntax. Interesting examples are Aria (Aria , 2009) (the successor of the XUI), the Swing JavaBuilder (The Swing JavaBuilder, 2009), eFace (eFace, 2009). They utilize a dedicated file containing a definition of the GUI which is created during run-time by the library. In most cases there is also support for data-binding which connects parts of the model and a widget. Listing 4 contains sample code in the YAML (YAML , 2009) and Figure 2 presents generated dialog window. Notice a dedicated section for binding names with GUI controls and validators.

There are also two commercial technologies worth mentioning: JavaFX (Topley, 2010) and WPF (with XAML for the MS C# language) (Nathan, 2006). Both of them claims to be declarative and are based on similar idea. Created GUI is defined using a separate file and a special syntax. Although syntaxes are different, semantics and amount of information provided by a programmer are similar. Roughly speaking even with a data binding technology a programmer has to write quite a lot of source code.

```
JFrame(name=frame, title=frame.title, size=packed,
defaultCloseOperation=exitOnClose):
- JLabel(name=fNameLbl, text=label.firstName)
- JLabel(name=lNameLbl, text=label.lastName)
- JLabel(name=emailLbl, text=label.email)
- JTextField(name=fName)
- JTextField(name=lName)
- JTextField(name=email)
- JButton(name=save, text=button.save,
onAction=($validate,save,done))
- JButton(name=cancel, text=button.cancel,
onAction=($confirm,cancel))
- MigLayout: |
[pref] [grow,100] [pref] [grow,100]
fNameLbl fName lNameLbl lName
emailLbl email+*
>save+*=1,cancel=1
bind:
- fName.text: this.person.firstName
- lName.text: this.person.lastName
- email.text: this.person.emailAddress
validate:
- fName.text: {mandatory: true, label: label.firstName}
- lName.text: {mandatory: true, label: label.lastName}
- email.text: {mandatory: true, emailAddress: true, label:
label.email}
```

Listing 4. Sample code in the YAML (Swing JavaBuilder).

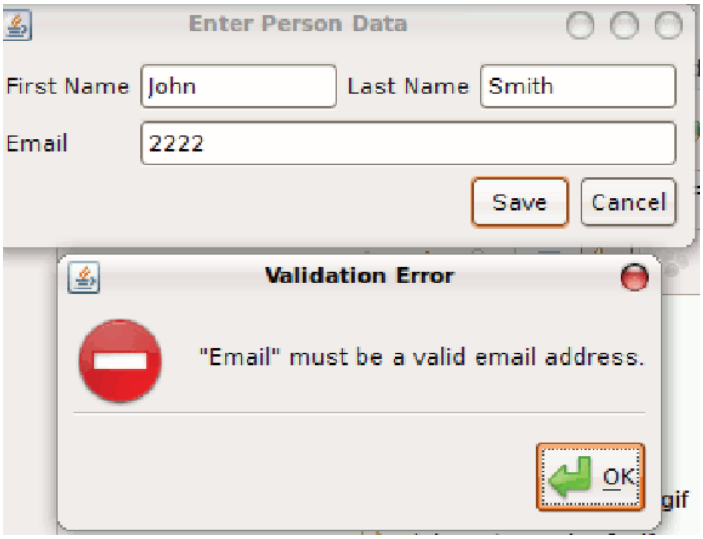

Fig. 2. The dialog window generated by the code from Listing 4

The above solutions are useful and in some cases provide higher level of abstraction than pure Java. But, even using such DSLs, a programmer has to spend a lot of time on GUI creation. We believe that our approach is sometimes a bit less powerful but much more simpler.

4. The design and implementation

Our first attempt at declarative user interfaces (see the senseGUI library described in the (Trzaska, 2008)) was not based on a DSL. It utilized annotations of the Java programming language. The implemented library, based on the annotated model (Java classes), was able to generate different types of GUIs (frames, dialogs, panels). In our current proposal, also for Java, we have decided to use a dedicated DSL rather than marking source code. Such a change is very useful for a programmer:

- The process of defining the GUI takes place in one location: the GCL statement. In the senseGUI library it was split between a model definition and a library's method call;
- There is no need for modifying (marking with annotations) model (data) source code by a programmer. The code is not always accessible (it could be shipped as i.e. Java jar file) and even if it is, modifications should be avoided wherever possible.

During the design process of the language we tried to make it as simple, yet powerful. Hence we defined the following general requirements:

- The number of different constructs has to be minimized,
- Most of the customization information has to be optional. It could be achieved using some (carefully chosen) default values,
- Orthogonality and reuse wherever possible, i.e. embedded fields should be defined using "ordinary" fields properties.
- Support for important GUIs facilities like internationalization (i18n), validators.

Such an approach significantly reduces the number of special cases and thus the size of documentation.

The overall goal of the GCL language is saving a programmer's time by generating a GUI. The library automatically creates necessary controls based on the given model. The model is defined by ordinary Java classes. A programmer passes a model's instance (a Java object), optionally customizes it and the library generates a widget. Using the widget, an end user of the application is able to see the object's content and modify it. The design is language independent and could be implemented for any language which supports reflection.

```
Create ComponentType for DataInstance containing (Field01Type
Field01Descriptor, Field02Type Field02Descriptor, ...)
```

Listing 4. The GCL root statement

Listing 4 presents the root statement of the GCL language. The containing part is optional; if it is omitted, then only default values will be used. Below are descriptions of all parts of the statement:

- The ComponentType could be one of the following:
  - frame – an instance of the `JFrame` class,
  - internalFrame – an instance of the `JInternalFrame` class (same as 'frame' but utilized in the MDI applications),
  - panel – an instance of the `JPanel` class; a panel could be embedded in any other Java GUI,
  - dialog – an instance of the modal `JDialog` class.
- The DataInstance is just the Java object for which we need a GUI;
- The FieldType is one of the following:
  - attribute - describes a given attribute, i.e. attribute("firstName"),

    - method - describes a given method, i.e. method("getAge"),
- The FieldDescription is a combination of the following modifiers:
    - `resizeWidget(boolean)` - Sets the widget's resizing behavior wherever it should be resized horizontally and vertically,
    - `setMethod(String)` - Sets the method used to modify the item's value (with the String parameter),
    - `as(String)` – Sets a label for the item. It could be achieved directly by providing the text or taking into account i18n by giving a key in a language bundle (standard Java approach),
    - `asComplex(Field01Description, Field02Description, ...)` - Treats the item as a complex one (a field embedded in a field) and allows passing additional information about an internal widget.
    - `order(int)` - Sets an order for the item,
    - `usingWidget(String)` - Sets a name of the Java class (with a full package) which will be used as a widget for showing the item,
    - `validate(Validator)` - Sets a validator for the item,
    - `readOnly(boolean)` - Indicates if the item should be read-only,
    - `value(String)` - Sets the default value. Used by Ad Hoc GUI (see further). Ignored in GUIs based on existing data models,
    - `type(Class<?>)` - Sets type of the field (in the case of attributes it is the attribute's type; for methods type of the returned valued). Normally, the type is read from the structure of the data object. Hence, this method is useful in Ad Hoc GUIs where there is no data object connected,
    - `getMethod(String)` - Gets the method,
    - `buttons(MultiObjectsListButton...)` - Defines additional buttons for multi-objects list. Ignored in other cases.

In the case of popular programming languages like Java or MS C#, a DSL could be implemented using one of the following approaches:

- String-based. All DSL constructs are passed to a library as strings. This way most implementations of the SQL (including JDBC) work. Obvious disadvantages include: lack of type-control, no context-sensitive help, no compilation time errors checking, etc.;
- API-based. The idea makes use of a special design of the library providing a DSL: classes, methods, interfaces. All of them have special names which read separately sound quite strange, but after connecting them together emulate statements of the DSL language. All the concepts and constructs are described in the (Fowler, 2010).

We believe that the second approach is more useful for a programmer, hence we have implemented our GCL that way. Sample statement in the Java implementation could look like the code in Listing 5 (the right side of the equal character).

```
JFrame frame = create.frame.using(person).containing();
```

Listing 5. Sample GCL statement in the API-based implementation

It is worth noting that:

- As we mentioned earlier, particular parts of the API have quite strange names, i.e. the containing method, but reading the whole statement makes them sensible;
- Due to the Java restrictions we had to change a bit our syntax. The “for” keyword had to be replaced with something else (`using`);

- Another problem was caused by the fact that the return value type of the whole statement (in the API-based implementation – the `containing` method) is determined by the second part – the type of the widget (i.e. `frame`). In terms of the Java API it means that the return type of the last method (`containing`) should be determined by another element of the language. To solve the issue we introduced different "paths" – each for every returned type;

This section described details specific to the design and implementation of the DSL part of the library. General information about analyzing business class structures, generating GUI, etc. could be found in the (Trzaska, 2008).

5. Sample utilizations

Below we present a few sample utilizations of the GCL language, together with short descriptions and snapshots of the generated GUIs (the person is an instance of the typical business Person class). All of them are available on the project page (http://gcl-dsl.googlecode.com/).

- The simplest possible utilization of the GCL. A generated widget (in this case a frame/window) is totally based on default values (Listing 6 and Figure 3). The `usingOnly` statement is a shortcut for the `using(person).containing()` (Listing 5) with an empty containing part.

```
JFrame frame1 = create.frame.usingOnly(person);
```

Listing 6. Simplest GCL utilization #1

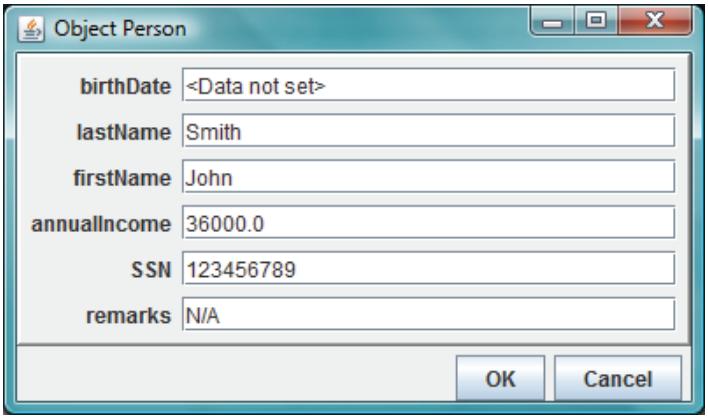

Fig. 3. The window generated by the code from Listing 6

- A customized frame for the same Person object with a validator (Listing 7 and Figure 4). Thanks to the Orthogonality principle utilized during the design process, validators could be applied to any filed in the same manner like other modifiers.
- A default frame showing automatically generated content for the given instance of the Company class is presented on Figure 5 and the code on Listing 8. One of the Company class attribute called employees is a list with references to employees. This case is reflected in the frame as an automatically generated (and populated) list box with

buttons. Two of them are provided by the library and allows editing or removing linked objects. A programmer is also able to define custom buttons with various actions, i.e. creating another employee.

```
JFrame frame =  create.
                frame.
                using(person).
                containing(
                    attribute("firstName").as("First name"),
                    attribute("lastName").validate(new
ValidatorNotEmpty()),
                    attribute("higherEducation"),
                    method("getAge").as("Age"));
```

Listing 7. Sample GCL utilization #2

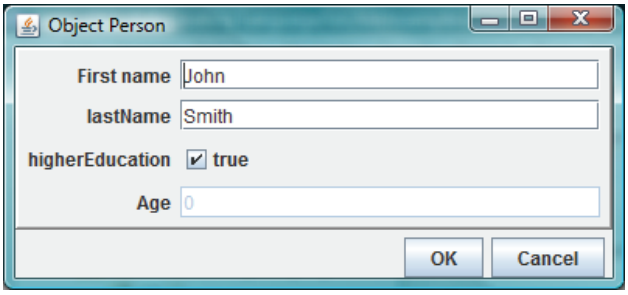

Fig. 4. The window generated by the code from Listing 7

```
JFrame frame =  create.
                frame.
                using(company).
                containing(
                       attribute("name").as("Name"),
                       attribute("income"),
                       attribute("employees"));
```

Listing 8. Sample GCL utilization #3

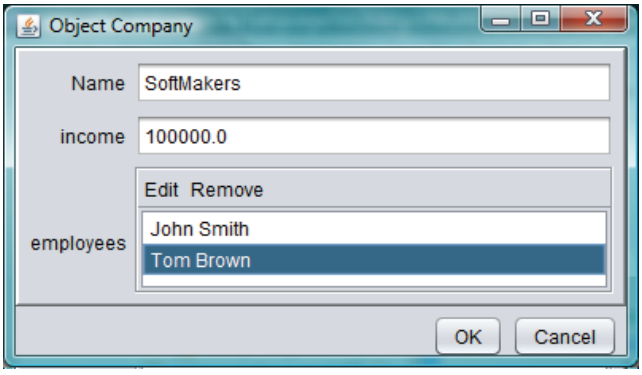

Fig. 5. The window generated by the code from Listing 8

- Ad Hoc GUIs. Aside of GUIs required by existing data structures (i.e. Person class), a typical business application also needs different dialogs and windows which do not have explicit data structures. For instance a login dialog or a database connection wizard usually do not utilize a dedicated data (model) class. Such cases could be processed by the GCL functionality called Ad Hoc GUIs. A user creates a statement which generates a widget according to the given definition. Of course it is possible to use all GCL constructs like validators or many types of customizations. An example is presented on Listing 9 and Figure 6. Note that:
  - Interface `AdHocActionPerformed` gives a possibility of executing a custom method when a user clicks the OK button.
  - It is possible to provide default values,
  - Different data types are processed using different widgets (i.e. an `enum` with a combo box – the `Colors` class in the example).

```
AdHocActionPerformed processAccept = new AdHocActionPerformed(){
        @Override
        public void Accept(Map<String, String> enteredData) {
            // Do something with the fields...
        }
       };
       frame = create.
              frame("Data", processAccept, "OK").
              containing(
                attribute("firstName").as("First
name").value("Martin"),
                attribute("lastName").validate(new
ValidatorNotEmpty()),
                attribute("higherEducation").type(boolean.class),
                attribute("values").type(Colors.class));
```

Listing 9. Sample GCL utilization#4

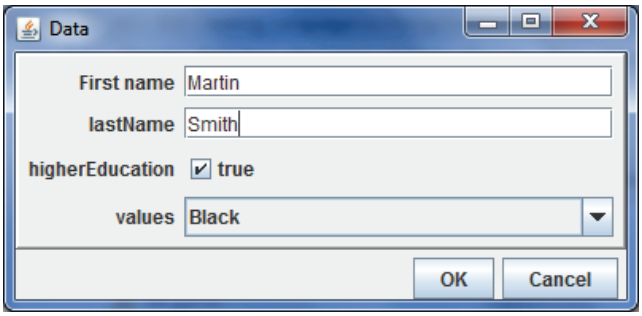

Fig. 6. The window generated by the code from Listing 9

- This sample is very similar to the one presented on Listing 7 but supports internationalization (I18n): An internationalized (using the Java message bundle) and customized frame for the Person object with a validator (Listing 10 and Figure 7).
- The last sample is similar to the one presented on Listing 8 but provides a custom button. Listing 11 contains appropriate GCL code (notice the `buttons` modifier) and Figure 8 the generated window. The `buttons` modifier expects an object implementing

the `MultiObjectsListButton` interface (containing just 2 methods). Listing 12 presents the utilized (partial) implementation which creates a new employee and connects him with the company. Notice that the implementation uses the GCL itself to get the new employee data.

```
dialog =  create.
          dialog.
          using(person).
          containing(resourceBundle,
            attribute("firstName").as("Person.firstName"),
            attribute("lastName").as("Person.lastName").
validate(new ValidatorNotEmpty()),
            attribute("higherEducation").
   as("Person.higherEducation"),
            method("getAge").as("Person.getAge"));
```

Listing 10. Sample GCL utilization #5

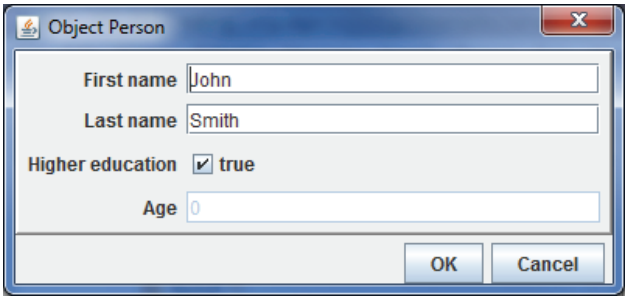

Fig. 7. The window generated by the code from Listing 10 (with i18n)

```
frame =   create.
          frame.
          using(company).
          containing(
              attribute("name").as("Name"),
              attribute("income"),
              attribute("employees").
                 buttons(new ButtonCreateEmployee()).asComplex(
                      attribute("lastName").as("Last name")
                 )
               );
```

Listing 11. Sample GCL utilization #6

6. Conclusions and future work

We have presented a Domain Specific Language called GCL. The purpose of the language is to facilitate creation of Graphical User Interfaces. Our research has been supported by the working implementation for the Java platform. However, utilized approach and design are generic enough to create the language for other platforms (like MS .NET and C#).
To our best knowledge, the GCL is the only solution offering such a high level of automation in creating typical, business-oriented GUIs. In the simplest case, a programmer,

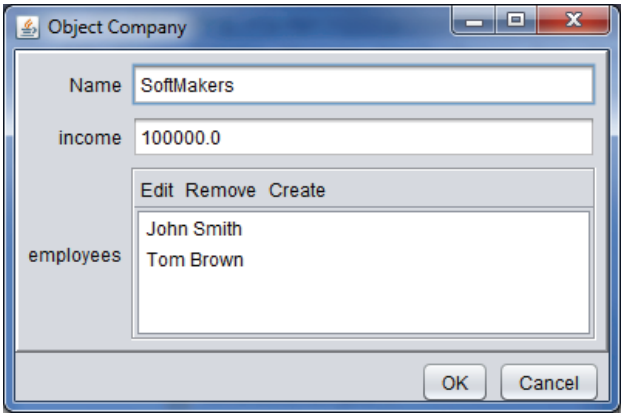

Fig. 8. The window generated by the code from Listing 11

```
class ButtonCreateEmployee implements MultiObjectsListButton {
   public String getButtonLabel() {
     return "Create";
   }

   public void process(JList multiObjectsList, Collection<Object>
objects) {
     Employee emp = new Employee();
     dialog =  create.
               dialog.
               using(emp).
               containing(attribute("firstName"),
                          attribute("lastName"));
       // [...]
   }
}
```

Listing 12. Sample implementation of the `MultiObjectsListButton` interface.

using just one GCL statement, is able to generate a working widget (a window, a dialog or a panel) for a given data instance (a typical Java class). Such an approach does not impose utilizing complex, hard-to-use libraries or modifications of business source codes.

We believe that Domain Specific Languages will gain in popularity because of their simplicity and usefulness. Hence we would like to continue our research in the field of DSLs and, especially, GUIs creation.

7. References

Aria - a framework for building Java and XML based applications. http://www.formaria.org/

Basnyat S., Bastide R., Palanque P.: Extending the Boundaries of Model-Based Development to Account for Errors. MDDAUI '05. 2005.

Bock C., Gorlich D., Zuhlke D.: Using Domain-Specific Languages in the Design of HMIs: Experiences and Lessons Learned. Proceedings of the MoDELS'06 Workshop on Model Driven Development of Advanced User Interfaces. Genova, Italy. 2006.

Borgo R., Duke D., Runciman C., Wallace M.: The 2008 Visualization Design Contest: A Functional DSL for Multifield Data. Manuscript submitted August 1 2008 for IEEE Visualization Design Contest.

Bravenboer M., Visser E.: Concrete syntax for objects: domain-specific language embedding and assimilation without restrictions. Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications OOPSLA '04. October 24-28, 2004. Vancouver, Canada. ISBN 1581138319. pp 365-383.

da Silva P.: User interface declarative models and development environments: a survey. Proceedings of DSVIS 2000, 2000,pp. 207–226.

Deursen A.V., Klint P., Visser J: Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices, 2000. 35(6): p. 26-36.

eFace - XAML/WPF for Java. http://www.soyatec.com/eface/.

Fowler M. Domain Specific Languages (work in progress). http://martinfowler.com/ /dslwip

Freeman S., Pryce N.: Evolving an Embedded Domain-Specific Language in Java. 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. October 22-26, 2006. Portland, Oregon, USA. ISBN:1-59593-491-X. pp 855-865.

Gajos K., Weld D.: SUPPLE: Automatically Generating User Interfaces, in Proceedings of IUI'04, Funchal, Portugal,2004, pp.83-100.

Goderis S., Deridder D., Van Paesschen E., D'Hondt T.: DEUCE - A Declarative Framework for Extricating User Interface Concerns, in Journal of Object Technology, Special Issue: TOOLS Europe 2007, vol. 6, no. 9, October 2007, pages 87-104.

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.

Nathan A., Windows Presentation Foundation Unleashed (Sams, 2006).

Nussbaumer M., Freudenstein P., Gaedke M.: The Impact of Domain-Specific Languages for Assembling Web Applications. Engineering Letters Journal. Volume 13, Issue 3. 2006. ISSN: 1816-093X.

Sells Ch., Weinhardt M.: Windows Forms 2.0 Programming. ISBN: 978-0-321-26796-2. AWPddison Wesley Professional. 2006.

The Swing JavaBuilder. http://code.google.com/p/javabuilders/.

Topley K., JavaFX Developer's Guide (Addison-Wesley Professional, 2010).

Trzaska M.: Automatically Creating Graphical User Interfaces Using Extended senseGUI Library. Proceedings of the Ninth IASTED International Conference on Software Engineering and Applications (SEA'08). November 16 – 18, 2008, Orlando, Florida, USA. ISBN: 978-0-88986-776-5. pp. 112-117.

Visser E.: WebDSL: A Case Study in Domain-Specific Language Engineering. Lecture Notes in Computer Science 5235:291--373. 2008.

Walrath K., Campione M., Huml A., Zakhour S.: The JFC Swing Tutorial (2nd Edition). ISBN 0201914670. Prentice Hall. 2004.

YAML: http://en.wikipedia.org/wiki/YAML.