GCL – An Easy Way for Creating Graphical User Interfaces

Mariusz TRZASKA
Polish Japanese Institute of Information Technology
Warsaw Koszykowa Str. 86, 02-008, Poland

ABSTRACT

Graphical User Interfaces (GUI) can be created using several approaches. Beside using visual editors or a manually written source code, it is possible to employ a declarative method. Such a solution usually allows working on a higher abstraction level which saves the developers' time and reduces errors. The approach can follow many ideas. One of them is based on utilizing a Domain Specific Language (DSL). In this paper we present the results of our research concerning a DSL language called GCL (GUI Creating Language). The prototype is implemented as a library for Java with an API emulating the syntax and semantics of a DSL language. A programmer, using a few keywords, is able to create different types of GUIs, including forms, panels, dialogs, etc. The widgets of the GUI are built automatically during the run-time phase based on a given data instance (an ordinary Java object) and optionally are to be customized by the programmer. The main contribution of our work is delivering a working library for a popular platform. The library could be easily ported for other programming languages such the MS C#.

Keywords: Graphical User Interfaces, declarative languages, Domain Specific Languages (DSL), model, data, generic programming.

1. INTRODUCTION

According to [1], Domain Specific Languages (DSL) offer an expressiveness power usually focused on a particular application or technical domain. They introduce special syntax and semantics allowing for working on a quite high level of abstraction. DSLs often employ a declarative approach which means specifying a result to be achieved rather than steps that lead to the result. In effect, a person using a DSL expects improvement in the process of developing software. The improvement could mean saving 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 [2] we can find information about roots of the DSLs in the 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 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 [3]. The language provides a set of primitives and other structures combining 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 [4] has been designed to generate prototypes especially for testing usability. Thanks to 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 a developed application;
- To develop distributed Web-based applications. The paper [5] presents a system where domain experts directly contribute to the development process by utilizing dedicated DSLs. Hence a web application is composed from various blocks which behavior is specified with the languages;
- To test software. In the case of a system described in [6], a DSL has been used for the "mocking" process. It means mimicking the behavior of some real objects linked with tested objects;
- To create Graphical User Interfaces. This area is discussed in Section 2.

The purpose of our work is creating a DSL named GCL (GUI Creating Language), which will significantly reduce a programmer's involvement during creating a GUI for a business-intensive application. We believe that the created DSL for Java is easy-to-use yet powerful and allows to define various types of GUIs.

The rest of the paper is organized as follows. To fully understand our motivation and approach some related solutions are presented in Section 2. Section 3 briefly discusses key concepts of the language and its implementation. Section 4 contains sample utilizations of GCL. Section 5 concludes.

2. RELATED SOLUTIONS

In our opinion, raising the level of abstraction is the most significant goal of declarative solutions. Such an approach considerably simplifies programmer's job and decreases the number of errors. However, the common side effect is some kind of uniformity of generated GUIs. This is caused by the fact that the majority of the GUI appearance and behavior 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. However that means providing a lot of details by a programmer, which could cause the complexity comparable to the classical methods.

The paper [7] introduces an interesting DSL called SWUL (Swing User-interface Language). The language has been developed using MetaBorg which provides a concrete syntax for domain abstractions. It is based on 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 1 presents a sample SWUL code.

Listing 1. A simple SWUL code

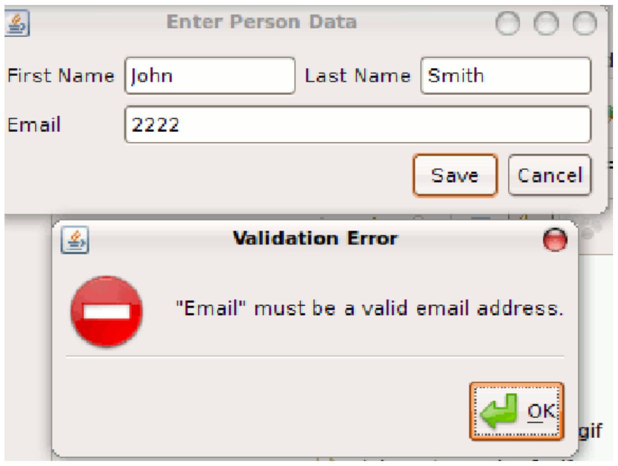
```
JFrame frame = frame {
  title = " Welcome !"
  content = panel of border layout {
    center = label { text = "Hello World"
  }
    south = panel of grid layout {
      row = {
        button { text = " cancel " }
        button { text = "ok" }
      }
    }
  }
};
```

The readability of the code is much better than a Java code with 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 that 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.
The paper [8] 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 behavior of an application. The system allows for defining rules which could concern different aspects including an automatically generated GUI. For instance Listing 2 shows rules describing some components relations among customers and a shop.

Listing 2. Definition of component relations in DEUCE

```
  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>).
```

The idea is interesting but requires further research, especially, considering performance for real-world applications. Another uncertain 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 [9] (the successor of the XUI), the Swing JavaBuilder [10], eFace [11]. 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 3 contains sample code in YAML [12] and Figure 1 presents the generated dialog window. Notice a dedicated section for binding names with GUI controls and validators.



"Email" must be a valid email address.

Fig 1 The dialog window generated by the code from Listing 3

Two commercial technologies are also worth mentioning: JavaFX [13] and WPF (with XAML for the MS C# language) [14]. Both are claimed to be declarative and both are based on a similar idea. A created GUI is defined using a separate file and a special syntax. Although syntaxes are different, semantics and the 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 a source code.
The above solutions are useful and in some cases provide higher level of abstraction than pure Java. But even using such a DSL, 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 simpler.

3. DESIGN AND IMPLEMENTATION

Our first attempt at declarative user interfaces (see the senseGUI[1] library described in [15]) was not based on a DSL, but utilized Java annotations . 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 a source code. Such a change is very useful for a programmer:

- The process of defining the GUI takes place in one location: a 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) a model (data) source code by the programmer. The code is not always accessible (it could be shipped as e.g. a Java jar file) and even if it is, modifications should be avoided wherever possible.

[1] The senseGUI library is freely available at: http://www.mtrzaska.com/sensegui

Listing 3. Sample code in the YAML (Swing JavaBuilder).

```
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}
```

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

- The number of different constructs is to be minimized,
- Most of the customization information is 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) and 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 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 to modify it. The design is language independent and could be implemented for any language which supports reflection.

Listing 4. The GCL root statement

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

Listing 4 presents the root statement of the GCL language. The *containing* part is optional; if it is omitted, then 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,
- *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; this is a simple customization for the way particular type values are to be presented,
- *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 a 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 the 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 [16].

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

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

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

- As we mentioned earlier, particular parts of the API has 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 has to be replaced with something else;
- 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 [12].

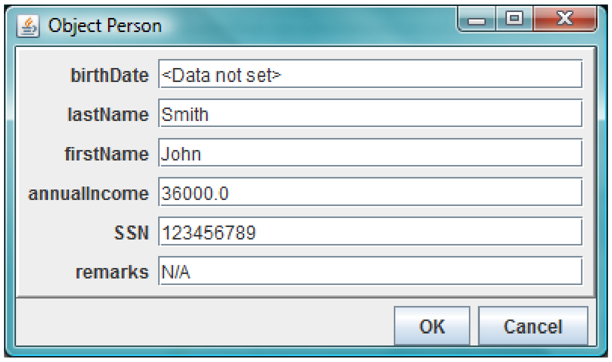
4. SAMPLE UTILIZATIONS

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):

- 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 2). The *usingOnly* statement is a shortcut for the *using(person).containing()* (Listing 5) with an empty containing part.

Listing 6. Simplest GCL utilization #1

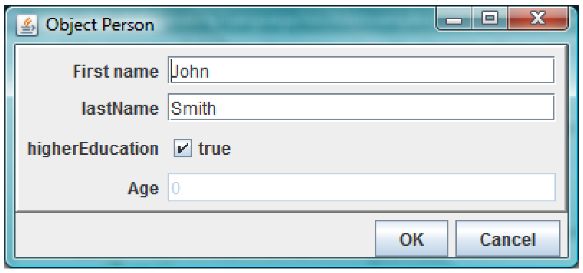
```
JFrame frame1 =
create.frame.usingOnly(person);
```



Object Person

birthDate	<Data not set>
lastName	Smith
firstName	John
annualIncome	36000.0
SSN	123456789
remarks	N/A

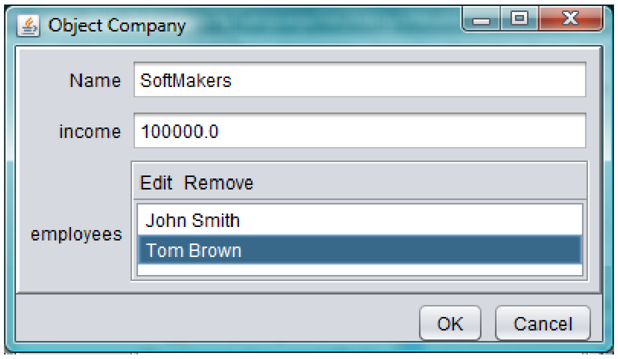
Fig 2 The window generated by the code from Listing 6



First name John
lastName Smith
higherEducation ✓ true
Age 0

Fig 3 The window generated by the code from Listing 7

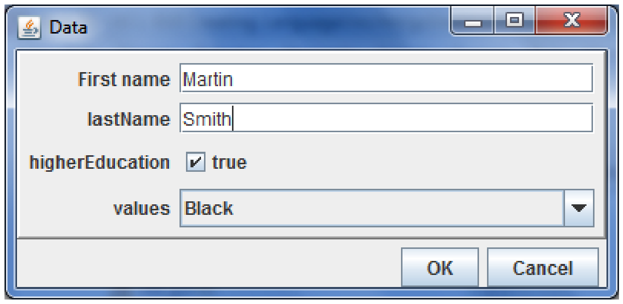
- A customized frame for the same Person object with a validator (Listing 7[2] and Figure 3). Thanks to the orthogonality principle utilized during the design process, validators could be applied to any field in the same manner like other modifiers.
- A default frame showing automatically generated content for



Name SoftMakers
income 100000.0
employees John Smith, Tom Brown

Fig 4 The window generated by the code from Listing 8

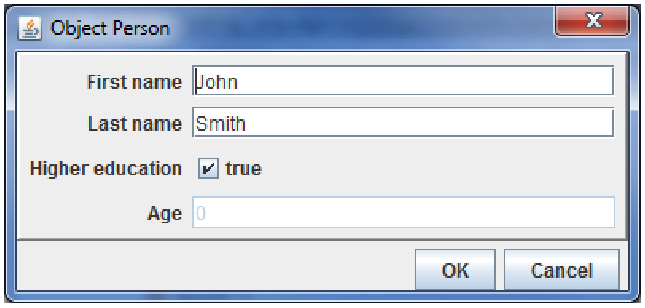
the given instance of the Company class is presented in Figure 4 and the code in Listing 8. One of the Company class attribute called *employees* is a list with references to employees. This case is reflected in the frame as an automatically generated (and populated) list box with buttons. Two of them 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.
- 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 5. Note that:
    - Interface *AdHocActionPerformed* gives a possibility of executing a custom method when the 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).



First name Martin
lastName Smith
higherEducation ✓ true
values Black

Fig 5 The window generated by the code from Listing 9

- This sample is very similar to the one presented on Listing 7 (and Figure 3) 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 6).

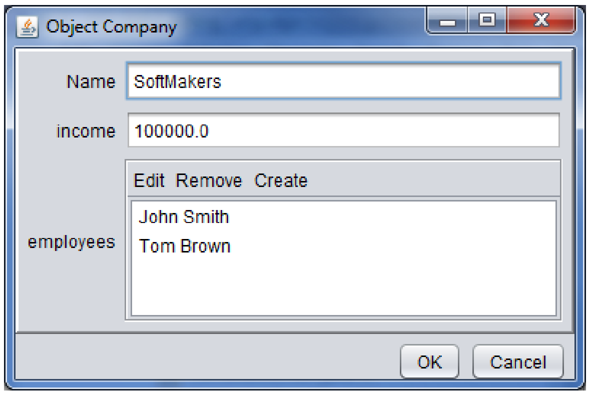


First name John
Last name Smith
Higher education ✓ true
Age 0

Fig 6 The window generated by the code from Listing 10

- 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 7 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.

[2] Due to the readability, listings 7 – 12 are placed together at the end of this section.


Object Company

Name: SoftMakers

income: 100000.0

employees:
- John Smith
- Tom Brown

Fig 7 The window generated by the code from Listing 11

Listing 7. Sample GCL utilization #2
```
JFrame frame =  create.
                frame.
                using(person).
                containing(
                    attribute("firstName").as("First name"),
                    attribute("lastName").validate(new ValidatorNotEmpty()),
                    attribute("higherEducation"),
                    method("getAge").as("Age"));
```

Listing 8. Sample GCL utilization #3
```
JFrame frame =  create.
                frame.
                using(company).
                containing(
                       attribute("name").as("Name"),
                        attribute("income"),
                        attribute("employees"));
```

Listing 9. Sample GCL utilization #4
```
AdHocActionPerformed processAccept = new AdHocActionPerformed() {
        @Override
        public void Accept(Map<String, String> enteredData) {
          // Do something with the fields...
        }
};
```

Listing 9 – *cont.* Sample GCL utilization #4
```
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 10. Sample GCL utilization #5
```
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 11. Sample GCL utilization #6

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

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

```
  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"));
        // [...]
    }
```

5. 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 accomplished by the working implementation[3] for Java. However, the utilized approach and design are generic enough to adopt GCL for other platforms (like MS .NET and C#).

To our best knowledge, GCL is the only solution offering such a high level of automation in creating typical, business-oriented GUIs for a popular platform. In the simplest case, a programmer, 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. Our next goal is to modify GCL to allow its utilization in web-oriented technologies like Google Web Toolkit (GWT).

[3] The GCL prototype is available at: http://gcl-dsl.googlecode.com/

6. REFERENCES

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

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

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

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

[12] YAML: http://en.wikipedia.org/wiki/YAML.

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

[14] A. Nathan, *Windows Presentation Foundation Unleashed* (Sams, 2006).

[15] 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..

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