Wirtschaftsuniversität Wien

IS Projektseminar SS 2011

ao.Univ.Prof. Dr. Rony G. Flatscher

Java GUI Builders
and Script Deployments

Seminar Paper

Andreas Mulley
0354023

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer anderen Prüfung noch nicht vorgelegt worden.

Wien, am 21. Juni 2011

Andreas Mulley

Java GUI Builders and Script Deployments

Within the scope of this paper a range of software to develop graphical user interfaces with Java will be presented and evaluated. The integration of script languages into Java code is also handled, to enable the developer to execute scripts via the graphical user interface. Therefore a seamless embedding is one of the main points to ensure that the developer is able to insert scripts without needing to write a single line of Java code. This will be demonstrated in some nutshell examples.

Java GUI Entwicklung und der Einsatz von Skript-Sprachen

Im Rahmen dieser Arbeit wird eine Auswahl von Programmen zur Entwicklung graphischer Benutzeroberflächen mit Java vorgestellt und evaluiert. Außerdem wird die Integration von Skript-Sprachen in Java-Code behandelt, um es dem Entwickler zu ermöglichen, Skripts über die Benutzeroberfläche ausführen zu können. Hauptaugenmerk wird dabei auf eine nahtlose Einbettung gelegt, wodurch der Entwickler in der Lage sein soll, Skripts einzubauen ohne auch nur eine Zeile Java-Code schreiben zu müssen. Dies wird in einigen kurzen Beispielen demonstriert.

Table of Contents

Table of Figures

1. Introduction

The subject of the following paper is the presentation and evaluation of Java GUI Builders. Special focus is set on the integration of script languages. The GUI developer should be able to execute scripts without the need of writing Java code.

In the first step several software solutions for generating graphical user interfaces with Java are pointed out. The most user-friendly ones of the free available products are selected for further analysis.

Afterwards some nutshell examples are created to illustrate the potentials and features of these tools. The objective of each example is to generate a window that triggers one or more scripts when a special event – like a clicked button, pressed key or a mouse movement over a special area – happens. To minimize the Java code interaction of the GUI developer is another purpose of these exercises. The implementation with every selected GUI builder is demonstrated as well.

After that some evaluation criteria concerning the functionality and usability are considered on the basis of which these tools can be analyzed and compared. Finally the advantages and shortcomings of each software are summarized.

2. Overview of available Tools

There are a lot of different tools, that help Java developers to create graphical user interfaces more comfortable. There are stand alone programs as well as integrated GUI builders or plug-ins for so-called IDEs (integrated development environments) such as Eclipse [Ecl11a] or NetBeans [Ora11a]. Examples for stand alone builders are JFrameBuilder, SpeedJG or Visaj [IST11], which is an easy to use Java GUI Builder to create Java applications and applets. Due to the fact that a Visaj license costs several hundred UK-Pounds, it is rather designed for commercial use. GUI builder plug-ins for Eclipse are for instance JFormDesigner, Jigloo, Visual Editor, Window Builder Pro, Matisse4MyEclipse, JAXFront and JBuilder. NetBeans provides an integrated Java GUI Builder.

2.1 Description of Tools

A short selection of free available tools will be presented more detailed in this chapter. Starting with the simplest GUI builders the later ones will be more sophisticated.

2.1.1 JGB

The program with the simple name “Java Gui Builder” was a project from sourceforge [Gee11] to define the graphical user interface in an XML-file and generate the Java code out of the XML. This allows the users to adapt the design as they like without needing access to the program itself. It is realized by a *jgb*-package and the definition of an XML-structure. Due to the fact, that there is no graphical user interface to create the XML-file, this software is not suited for further consideration. [Bea03]

2.1.2 SpeedJG

The GUI builder SpeedJG is a tool that offers an interface to create and edit the available *Swing* classes and their available attributes, so the programmer is not forced to look up the *Swing*-API and can lessen the time needed to type code. It also offers an XML-export and dynamic Java code generation via some GUI classes in SpeedJG.jar. In the free test version there is no Java code export, therefore a script integration is not possible. [Woe11]

2.1.3 Java Gui Builder 1.0

This product is more of less a WYSIWYG[1]-Editor, which means that there is an graphical interface where Java *Swing* or *AWT* elements can be positioned by drag and drop on a window. The labels of these elements are not displayed in the editor and there is no possibility to change the attributes of an element. After all elements are added, the Java code can be generated. [CBS11a]

2.1.4 JFrameBuilder

The JFrameBuilder also provides a preview window and some *Swing* elements, which can be added to the preview. What is more there is a hierarchical structure overview of the elements and the attributes can be edited. The Java code will be generated automatically, so the actual code can be displayed when switching to the code view, but there is no possibility to change the generated code within the JFrameBuilder. [CBS11b]

2.1.5 Jigloo

This is the first of two GUI builder plug-ins for Eclipse that is handled in this paper. Jigloo is a product of Cloud Garden, which is free for private use. Just like the Eclipse IDE itself Jigloo does not need to be installed. Simply downloading the zip-file, copying the sub folder of the *plugins* folder into Eclipse's *plugins* folder and the sub folder of the *features* folder into the *features* folder of Eclipse is enough. But there is also the possibility to install it via the update manager of Eclipse.

1 What You See Is What You Get

Jigloo supports both *Swing* and *SWT* classes, but be careful with the button, which toggles the GUI between them, because already added elements would be damaged. Due to the fact, that it offers a wide range of features, Jigloo is selected to be evaluated in this paper. For the examples and further discussion Jigloo 4.6.4 is used in combination with Eclipse 3.6 called Helios. [Clo10]

2.1.6 Visual Editor

Another GUI builder plug-in for Eclipse is the Visual Editor. Unlike Jigloo, when installing this tool it is very important to check whether the version of the Visual Editor is compatible with the used Eclipse version. For Eclipse Helios – which is used for all examples corresponding with this paper – the Visual Editor 1.5 is needed. To install this tool, choose "Help – Install New Software ..." within the main menu. Click the *Add...* button and put the URL into the field *Location* or select the *Archive...* button and search for the already downloaded zip archive. Check the box next to *Visual Editor* and follow the wizard.

The Visual Editor offers *Swing*, *AWT* and *SWT* elements as well as a nice preview and other features. For this reason it is also selected for the nutshell examples and the evaluation. [Ecl11b]

2.1.7 NetBeans

The NetBeans IDE has an integrated GUI builder available, which provides some GUI forms. By means of these forms *Swing* and *AWT* windows can be created. There is also a large assortment of additional features, like the so-called *Connection Mode*. With this feature it is possible to make the event of an element trigger any method of any other element within the GUI, without typing any line of code. Therefore NetBeans and its GUI forms are selected for the evaluation as well. All examples are made with NetBeans 6.9.1 Java SE. [Ora11a]

2.2 Script Interfaces

There are some libraries that allow people to trigger script code from within a Java application. For simplicity only two of them will be handled in this paper.

2.2.1 JavaScript

The easiest way to integrate script code into a Java program is to use the script language JavaScript. Although it is likely to think that this language is related to Java, it is a completely different programming language, the name of which was chosen – more or less inexpertly – to gain attention because of the fame of Java. The classes needed to trigger the JavaScript code are part of the *javax.script* package, which is part of the Java Platform, Standard Edition [Ora11b] since version 6. The following lines of Java code will create a script engine that executes JavaScript code. Of course the first line has to be placed above the class definition together with all other import statements. [OCo06]

```
import javax.script.*;

ScriptEngineManager factory = new ScriptEngineManager();
ScriptEngine jsEngine = factory.getEngineByName("JavaScript");
try {
      jsEngine.eval("JAVASCRIPT-CODE[2]");
} catch (ScriptException exc) {
      exc.printStackTrace();
}
```

2.2.2 BSF4ooRexx

The second script language, which is discussed in this paper, is Open Object Rexx [Rex09]. The integration of this language is realized via the so-called Bean Scripting Framework, which provides a script engine that can handle ooRexx scripts. Both ooRexx and BSF4ooRexx [Gee09] have to be installed to use this feature. For all examples described in this paper ooRexx 4.0.1 was used. The Java code below creates a *BSFEngine*, which passes on a GUI element to the ooRexx script and executes it. Such an element can only be handed on through a vector. It is also possible to add more than one element to this vector, for each a line of code will be needed. If the script does not require any external elements, the two code lines, where the vector is created and the element is added, can be left out and the name of the vector – in this example *guiElements* – must be replaced by *null*.

2 JAVASCRIPT-CODE has to be replaced by the actual JavaScript code

```
import org.apache.bsf.*;

BSFManager mgr = new BSFManager();
try {
      BSFEngine rexxEngine = mgr.loadScriptingEngine("rexx");
      java.util.Vector guiElements = new java.util.Vector();
      guiElements.addElement(GUI-ELEMENT-NAME[3]);
      rexxEngine.apply("", 0, 0, "OOREXX-CODE[4]", null, guiElements);
} catch (BSFException exc) {
      exc.printStackTrace();
}
```

Unlike the JavaScript integration for BSF4ooRexx it is necessary to add two *jar* files to the library path, when working with an IDE. In Eclipse this can be done by going to the main menu and selecting “Project – Properties”. Within the popped up window the tree element *Java Build Path* as well as the tab *Libraries* have to be chosen. There it is possible to add the files below via the button *Add External JARs ...* – these two files should be found at the location where BSF4ooRexx was installed within the *BSF4ooRexx* folder.

- bsf-rexx-engine.jar
- bsf-v400-20090910.jar

There is another point which is important for the BSF integration in Jigloo. There is an icon with the title *Builds and runs the generated code*, which is the third one from the left at the top of the element structure screen. If this button is clicked, the library path will be overwritten and the above mentioned library files cannot be found on runtime. To remove this setting go to the main menu and select “Run – Run Configurations ...” A dialog appears, where the previously built frame can be selected in the tree view on the left side. Then choose the *Arguments* tab and remove the library path within the *VM arguments* field. The changes will be accepted when the *Apply* button is clicked.

Within the NetBeans IDE the library files can also be added via the main menu by selecting the menu item “File – Project Properties”. After choosing the tree element *Libraries* and the tab *Compile*, there is a button called *Add JAR/Folder*, which enables the adding of the two files mentioned above.

3 GUI-ELEMENT-NAME has to be replaced by the name of a Java object
4 OOREXX-CODE has to be replaced by the actual code

3. Usage Examples

In this chapter some nutshell examples are given how to use the three selected GUI builder tools – Jigloo, Visual Editor and NetBeans' GUI Forms – for creating graphical user interfaces in Java. Special focus is kept on the integration of script languages and on the minimization of user interaction with the Java code itself. Therefore there have also been written some Java classes to extend these tools.

3.1 Description of Scenarios

In the following the different scenarios that should be realized with each of the tools will be explained.

3.1.1 Button with JavaScript Action

In the first nutshell example a simple *JFrame* in the standard *BorderLayout* with a label and an *OK* button shall be created. If the button is clicked, the following JavaScript code is triggered, which creates a *JOptionPane* that appears in front of the *JFrame*, as shown in the figure below.

```
importPackage(javax.swing);
var optionPane = JOptionPane.showMessageDialog(null, 'JavaScript dialog');
```

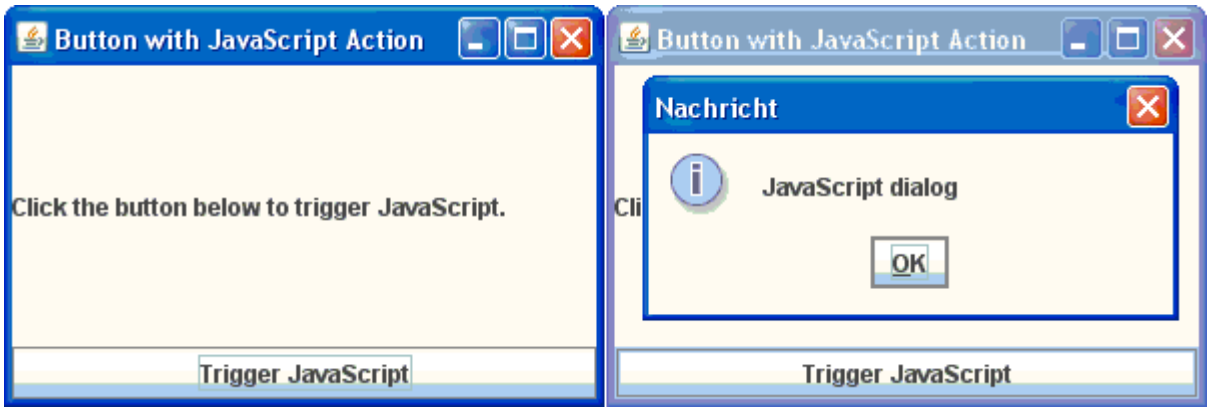

Figure 1: Frame with JavaScript generated dialog

3.1.2 Button with BSF4ooRexx Action

This example is a little bit more tricky than the first one. The created *JFrame* should have a *GroupLayout* or something equivalent where several input elements like text fields and radio boxes are displayed. After the *OK* button was pressed, the ooRexx code below generates another *JFrame*, that shows all the entered data (see figure 2). Therefore the input elements have to be passed on to the ooRexx script.

```
parse arg firstName, lastName, male, username, password;
fn=bsf.wrap('<O>' || firstName);
ln=bsf.wrap('<O>' || lastName);
ml=bsf.wrap('<O>' || male);
un=bsf.wrap('<O>' || username);
pw=bsf.wrap('<O>' || password);
if ml~isSelected then
      salutation='Mr.';
else
      salutation='Ms.';
jframe=.bsf~new('javax.swing.JFrame',
      'Hallo '||salutation||' '||fn~getText()||' '||ln~getText()||'!');
label=.bsf~new('javax.swing.JLabel');
label~setText('You have registered as '||un~getText()||
      ' with password '||pw~getText()||'.');
jframe ~~add('Center', label);
jframe~setLocation(450,450);
jframe~setSize(350,150);
jframe ~~setVisible(.true) ~~toFront;

::requires BSF.CLS;
```

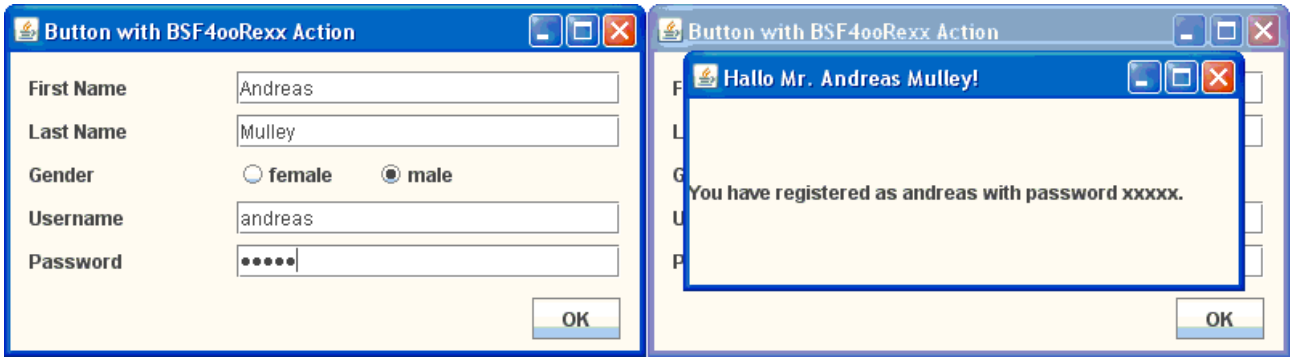

Figure 2: Form data read and displayed by ooRexx script

3.1.3 Menu Bar with Script Actions

The subject of this example is the creation of a menu bar. Two different menus should be added to the *JFrame*. The first menu does only have one menu item with a *Quit* action, which closes the window. The second one has two items, one triggers the JavaScript code from the first example, the other an ooRexx script, that is shown below. The following figure represents the result of this script.

```
frame=.bsf~new('javax.swing.JFrame','BSF4ooRexx Frame');
panel=.bsf~new('javax.swing.JPanel');
label=.bsf~new('javax.swing.JLabel');
label~setText('This is a simple BSF4ooRexx dialog');
panel ~~add('Center',label);
frame~add(panel);
frame~setLocation(500,500);
frame~setSize(300,80);
frame ~~setVisible(.true);
::requires BSF.CLS;
```

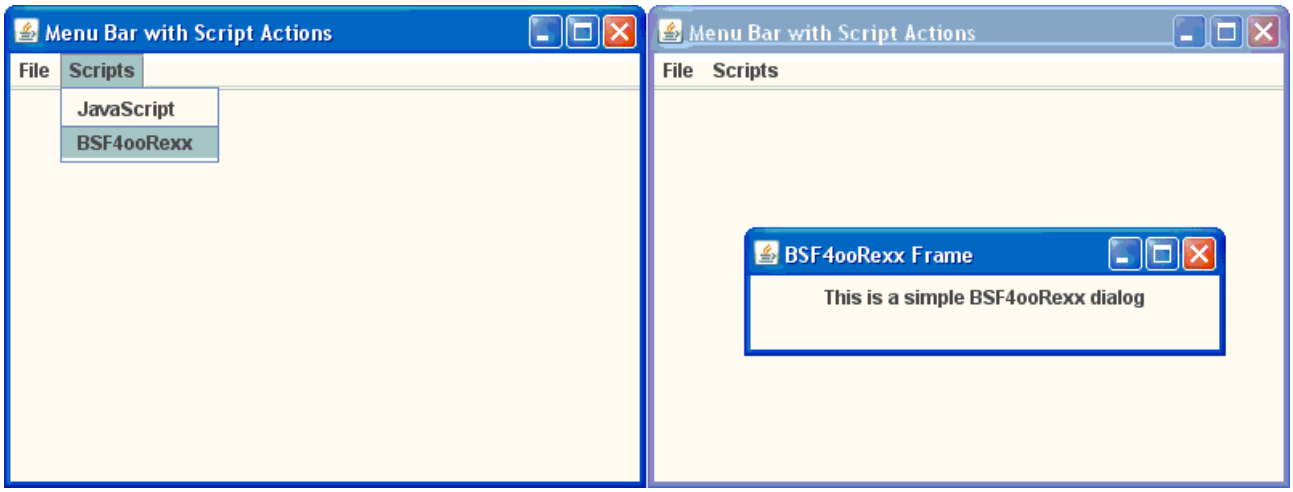

Figure 3: Menu with script created frame

3.1.4 Script Code triggered by Key Event

To execute script code via shortcuts is the topic of the next example. If the combination *CTRL-J* is pressed on the keyboard, a JavaScript triggered pop-up window appears, for *CTRL-R* it is produced by ooRexx. These shortcuts are displayed next to the menu items as illustrated below.

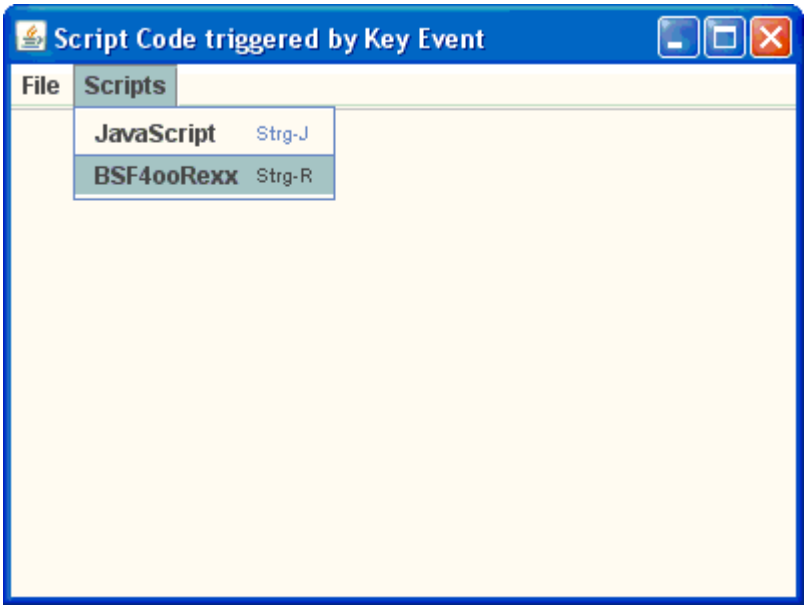

Figure 4: Shortcuts for script execution

3.1.5 Script Code triggered by Mouse Event

How to change existing Java elements via BSF4ooRexx is shown in this example. The *JFrame* contains a *JLabel*, the content of which will be changed when the user is moving the mouse over it, clicking on it or moving the mouse apart from the label (see figure below). This is realized with the following ooRexx script, to which the label must be passed on.

```
parse arg label;
if COMPARE(label,'<O>')<4 then
      label=bsf.wrap('<O>' || label);
else
      label=bsf.wrap(label);
label~setText('Mouse is moved over me!');
::requires BSF.CLS;
```

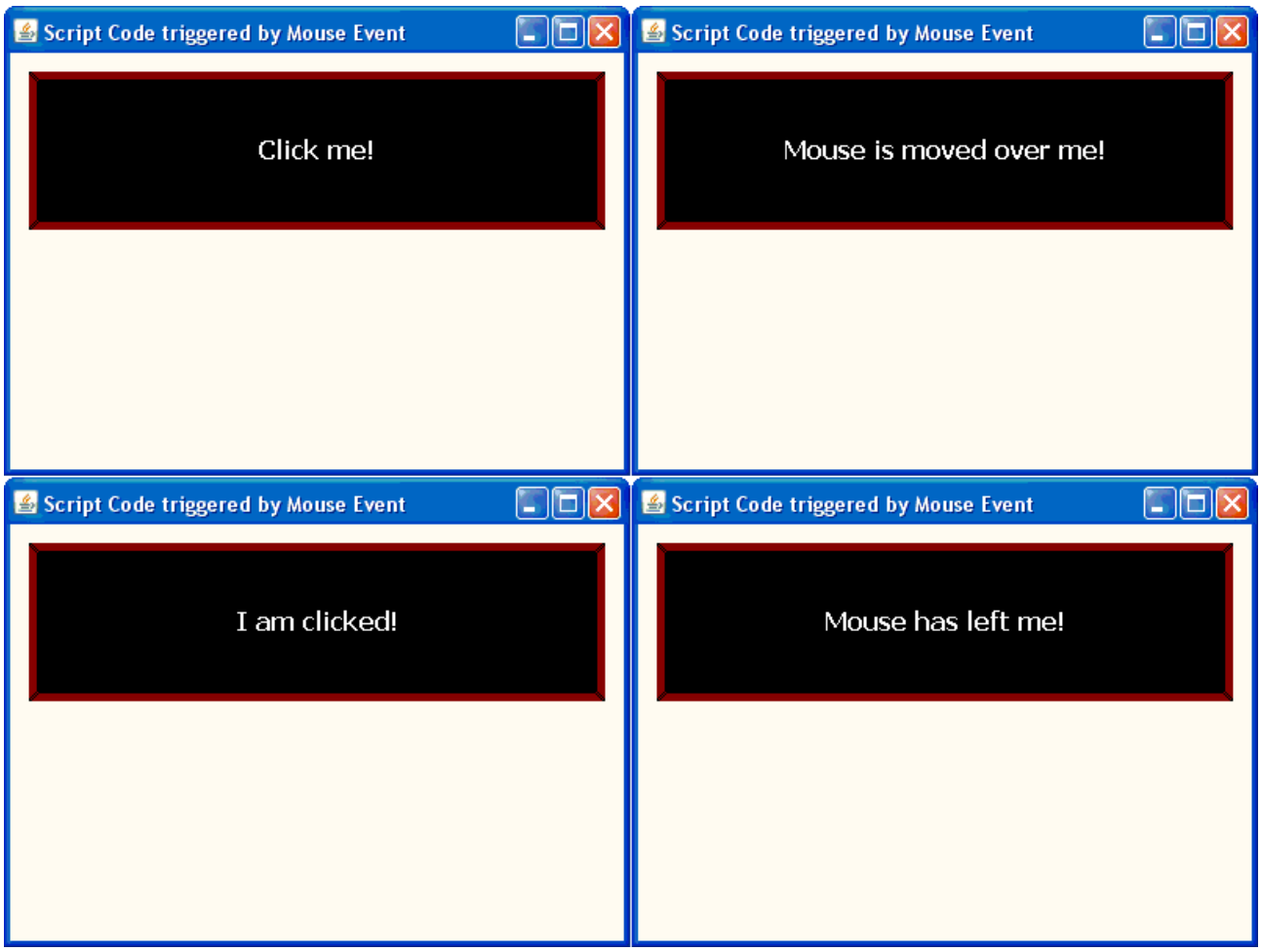

Figure 5: Script tells what the mouse does

3.1.6 JavaScript Extension

To realize the examples mentioned before the GUI developer always will have to put in some lines of Java code manually, to load the script engine (see section 2.2). For this reason a GUI builder extension needs to be implemented, so that the GUI developer only has to click a button – or something like that – and fill out a form providing the favored JavaScript code. The given code should be executed when the selected button is pressed.

3.1.7 BSF4ooRexx Extension

This example does nearly the same like the previous one. But for BSF4ooRexx it should be possible to provide the name of a Java element, the attributes of which can be read and changed within the ooRexx script below.

```
parse arg label;
if COMPARE(label,'<O>')<4 then
      label=bsf.wrap('<O>' || label);
else
      label=bsf.wrap(label);
text=label~getText;
if COMPARE(text,'Toggle')<7 then
      label~setText('Toggle me!');
else
      label~setText('I was toggled!');
```

3.2 Examples with Jigloo

First of all it is necessary to create a new Java project by going to the menu and clicking on “File – New – Java Project”. Then you can choose a project name, select an adequate Java Runtime Environment and set the Project Layout to *Use project folder as root for source and class files* if the Java files should be saved within the same folder as the compiled class files. By going on *Next >* you can add the needed BSF libraries (bsf-rexx-engine.jar and bsf-v400-20090910.jar) and finalize the creation with the *Finish* button.

A new *JFrame* can be generated via the menu button “File – New – Other ...”. If Jigloo is correctly installed there is a folder called *GUI Forms* in the tree. Within this folder there is a sub folder *Swing*, where you can choose the *JFrame* and go to the next step. There it is necessary to enter a package name as well as a class name before the *JFrame* can be added to the project.

After that there is a split screen in the center – in the upper part is an empty frame and in the lower one the associated Java code. To enlarge these two parts, that are the most important ones for working with the GUI builder, double click on the tab with the new created class name. This step

can be undone by double clicking the tab again. Now two additional screens have appeared on the right – the upper one is an hierarchical structure of the GUI elements and the lower one displays the properties of the currently selected GUI element.

3.2.1 Button with JavaScript Action

First of all the title of the *JFrame* will be set. This is done by scrolling the properties down and writing a text in the value field next to the property title and accepting it with the return button. Now the chosen title is visible in the frame.

The next step is to add the label to the frame. Directly above the frame view there is a tabbed icon list, which contains the available Java *Swing* classes. Choose the *Components* tab and click on the *JLabel* icon, which is located approximately in the center of the list. If you click on the *JFrame*, the form to edit the basic properties for new components will open. There the internal name of the component as well as the text can be declared. The label text can also be edited afterwards by double clicking the *JLabel*.

After that click on the *JButton* icon, which is the first one of the *Components* tab, and click on the frame. Just like for the label the internal name and text for the button can be chosen in the pop-up form. But now it is also necessary to change the *Layout Parameters*. Click the plus next to *Constraints*, then click on *direction* and a select arrow appears on the right. Choose *South* in order that the button will be inserted below the label. If you want to change the layout later on, just click on the button and choose the *Layout* tab on the properties screen.

Until now the button does not have any action listener. Choose the first icon of the *More Components* tab, which is the *AbstractAction* icon and click on the just created *JButton*. The text of the *AbstractAction* will overwrite the text of the button, so define the label of the button here. Now it is necessary to put the Java code that creates a JavaScript engine (see section 2.2.1) into the *actionPerformed* method. To find this method within the Java code, click on the *AbstractAction* within the GUI element structure screen on the upper right. After you have copied the Java code, you insert the JavaScript code (see section 3.1.1) and the first example is finished. Save the Java class and run it – the icon with the white triangle in a green circle.

3.2.2 Button with BSF4ooRexx Action

After adding a new *JFrame* and choosing a title just like mentioned before, the layout has to be set. For this reason click on the *GroupLayout* icon, which is the third on the right side in the *Layout* tab, and then on the empty *JFrame*. Now the elements can be added to the frame.

At first some labels for these elements will be needed on the left side. Click the *JLabel* icon, then go to the upper left corner of the frame until two red broken lines are visible – just like shown in the graphic below – and click.

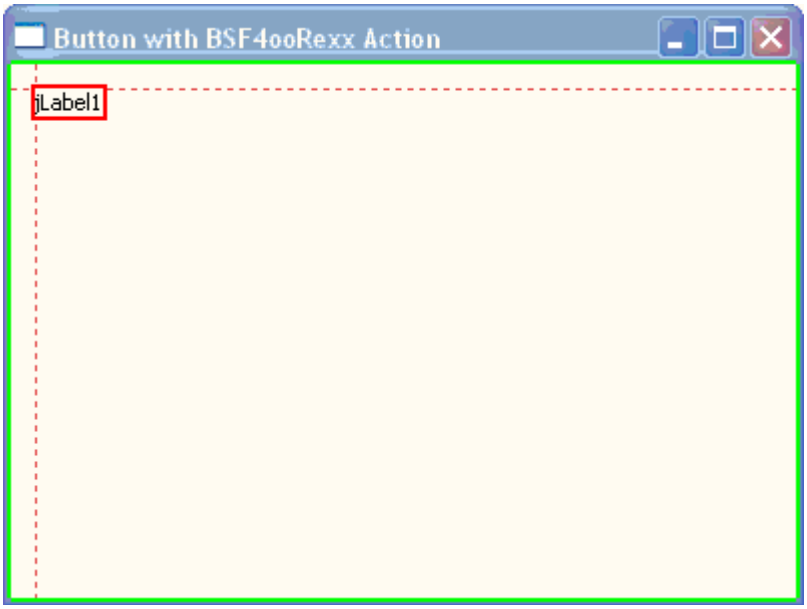

Figure 6: Element positioning assistance with broken lines

Change the text to “First Name” and accept. At the moment the label is too short to display the whole text, but this will be handled later on. Add the next *JLabel* below the first one, click when the red broken line left to the label is visible and change the text to “Last Name”. Do the same for the three additional labels “Gender”, “Username” and “Password”. After that the size of the labels can be edited. Therefore mark all labels by clicking on each of them while keeping *CTRL* pressed. Then enlarge the labels with the usual behavior to the right until they reach a third of the frame.

Further the input elements will be placed. Click the *JTextField* icon in the components tab and move it next to the first label until the two red broken lines appear like in the graphic below.

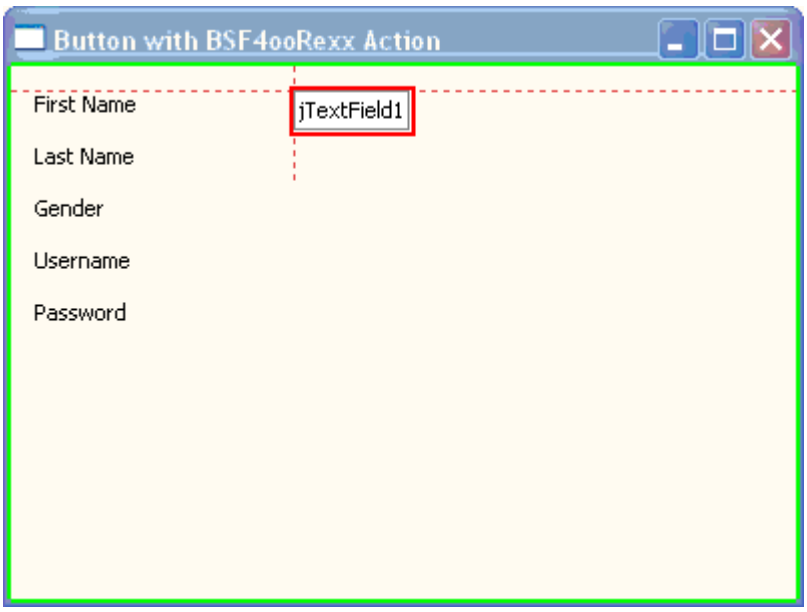

Figure 7: Column arrangement within the *GroupLayout*

Remove the text, to provide an empty text field and set the component name to "firstName". Do the same for the "lastName" input field. To add the radio buttons is a little bit more difficult. First of all click the *JRadioButton* icon in the *Components* tab and add two radio buttons next to the gender label, one with text and component name "female" the other one with "male". After that choose the *ButtonGroup* icon in the components tab, click anywhere in the frame and set the component name to "gender". Now it is possible to select both radio buttons and change the value for *buttonGroup* in the properties screen to *gender*. Then add another *JTextField* for "username" and a *JPasswordField* – which is part of the *More Components* tab – for the "password" input field.

Currently the spaces between and the size of the single elements might not be the same, so they should be adjusted. So select all input elements and labels and click the button *Make selected elements same height*, which is located left of the frame. There are also some buttons to align selected elements if they are not correctly arranged yet. Then enlarge the password field to the right end of the frame until there is a red broken line. Tag all input elements except the radio buttons and click the button *Make selected elements same width*. In order that these input elements will adjust their size to the window size, click on the black triangle at the upper right of each element and check *Expands Horizontally*. Maybe the *male* radio button disappears, because it is moved to the right outside the frame. If so, select it on the elements structure screen and move it back to its previous position.

The last step of this example is to set a *JButton* below the password field on the right side. Add an *AbstractAction* and change the text to "OK" like in the previous example. Insert the Java code from

section 2.2.2 to create a *BSFEngine*. To pass the input fields on to the ooRexx script, add the five elements *firstName*, *lastName*, *male*, *username* and *password* to the *guiElements* vector. Then insert the BSF4ooRexx code from section 3.1.2 and the frame is complete.

3.2.3 Menu Bar with Script Actions

For this example almost exclusively the *Menu* tab is needed. Select the first icon with the name *JMenuBar* and click anywhere into the frame. Automatically the first menu will be added too. Double click it to change the text to “File”. Then add another *JMenu* next to the file menu and call it “Scripts”. Now a *JMenuItem* can be added to the file menu and two of it to the scripts menu by simply clicking on the chosen menu. Just as for buttons the text of the menu items will be overwritten by the action's text, so it is not necessary to define it here.

The second part of this exercise are the actions. There are at least two possibilities to add an *AbstractAction* to a menu item. The first is to choose the icon in the *More components* tab and then click on the menu item within the GUI elements structure screen. The other one is to select the menu in the frame screen, so that its menu items are shown. Then you can add the action – just like for a button – by clicking on the menu item directly. Now add three actions to the three menu items and call them “Quit”, “JavaScript” and “BSF4ooRexx”.

To close the *JFrame* by choosing the *Quit* menu item it is necessary to add a short method that returns the actual frame object. Therefore go to the quit action by clicking it within the element structure screen and put the following code before the method with the name *getAbstractAction1* – if the component name of the quit action was changed the name of the method would be that name with the prefix *get*. Of course the returned data type must be the previously defined class name of the frame – here it is *MenuBarFrame*.

```
private MenuBarFrame getFrame() {
      return this;
}
```

Now put the following line of Java code into the *actionPerformed* method of the quit action and the frame will close when the quit menu item is clicked.

```
getFrame().dispose();
```

To keep it simple use the JavaScript code from the example in section 3.1.1 and paste it – in combination with the Java code that creates the JavaScript engine (see section 2.2.1) – into the *actionPerformed* method of the JavaScript action. For the BSF action certainly a *BSFEngine* (see section 2.2.2) is needed. To just let pop up another frame take the ooRexx code from section 3.1.3.

3.2.4 Script Code triggered by Key Event

In this exercise the *JFrame* from the last example is used as starting point. The goal is to execute the previously defined actions via key events. Therefore choose the actions within the element structure screen. They are displayed below each menu item, but they can also be found under the folder *Non-visual components*. Select the quit action, go to the properties screen and click next to *accelerator*, which should be the first property. Type on the keyboard the button *Q* then *CTRL* and confirm the input with the *return* key. If the frame is rendered now, it can be closed by simply typing *CTRL-Q* into your keyboard. Do the same with the button *J* for the JavaScript action and with *R* for the BSF action. By now the user do not need to use the menu bar anymore, but if he does so, the short cuts will be shown to the right of the menu items.

3.2.5 Script Code triggered by Mouse Event

First of all create a new *JFrame*, set a title for it and change the layout to *GroupLayout* just like mentioned before. The layout could also be set by right clicking on the frame and choosing the sub menu *Set Layout*. After that go to the *Components* tab and add a *JLabel* called "Click me!" to the frame. Enlarge the label so that it fills nearly half of the frame – just like shown in the following picture.

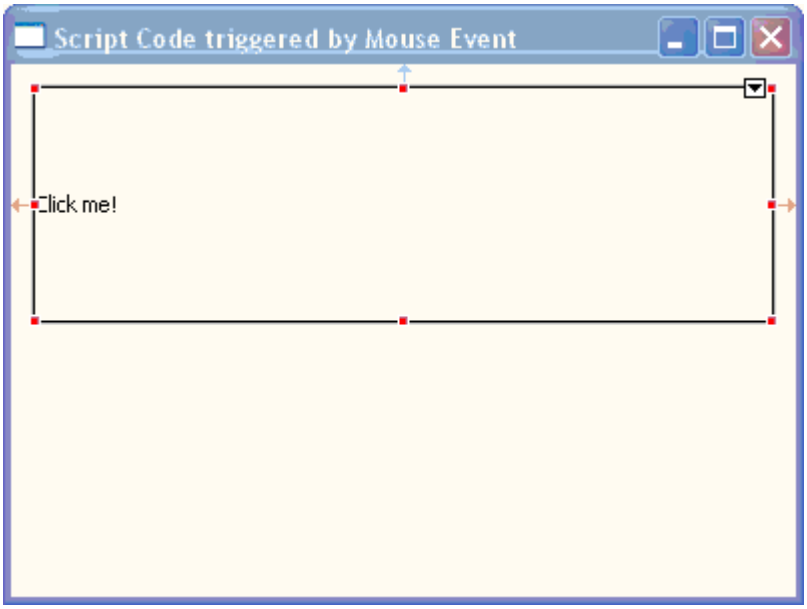

Figure 8: Element resizing through drag and drop

Due to the fact that the label has the same background color as the frame, it would not be easy to recognize whether the mouse is over the label or not. For this reason the color needs to be changed. This could be done by selecting the label, then clicking on the property *background* and pressing the button "...", which appears at the right side of the *background* property. A dialog pops up where the desired color can be chosen. But after that the background of the label is still the same as before. The reason for this is that most components are transparent as default. To change this scroll down the properties and click on *opaque* to activate this property.

Another way to change the background of a component would be to add a panel behind the component and set the background of the panel. This can be achieved by a right click on the component and the selection of *JPanel* within the sub menu *Surround by container*. Because of the fact that the panel has the same size as the label it can only be selected within the element structure screen. If the background of a container is changed there is no need to set the *opaque* property.

After the background is satisfying change the text color by selecting the label and clicking on the property *foreground*. It is exactly the same like with the background. To get the text into the middle of the label area, set the *horizontalAlignment* property to *CENTER*. The text should be more prominent, so click the *font* property and the button that appears on the right. Then change the text settings accordingly.

To draw a clear dividing line between the label and the frame a border line could be used as well. But it is also possible to add a border in addition. Therefore choose the value *LineBorder* within the *border* property. A little "+" appears to the left of the property. Click it to show all sub properties and change them as desired.

Now the mouse events will be added. Therefor choose the *Events* tab within the properties screen when the label is selected. Open the sub events of the *MouseListener*, click on *mouseEntered* and change the value to *handler method*. On the code screen appears a new method, which is called like the name of the label followed by the term "MouseEntered". Change the content of this method to the BSFEngine generating Java code from section 2.2.2. To pass the label object to the BSF4ooRexx script from section 3.1.5 add it to the vector using its internal name – most probably "jLabel1". After that do the same with the *mouseClicked* and the *mouseExited* sub event, but change the text, that is set in the ooRexx code to "I am clicked!" and "Mouse has left me!". If this is finished run the class and move the mouse over the label, click it and move the mouse away. The label will always say, what is done to it.

3.2.6 JavaScript Extension

Writing an extension for Jigloo is very easy. Everything that needs to be done is to create a Java class. If some parameters should be provided by the GUI developer – in this example one of these parameters is the JavaScript code – there must be a constructor that can handle them. The following Java class was written to add an *AbstractAction*, that automatically executes the given JavaScript code, to a button or a menu item.

```
package gui;

import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.script.*;

public class JavaScriptAction extends AbstractAction {
	public String javascriptCode;

	public JavaScriptAction(String name, String code) {
		super(name);
		javascriptCode = code;
	}
	public void actionPerformed(ActionEvent arg0) {
		ScriptEngineManager factory = new ScriptEngineManager();
		ScriptEngine javascriptEngine =
				factory.getEngineByName("JavaScript");
		try {
			javascriptEngine.eval(javascriptCode);
		} catch (ScriptException e) {
			e.printStackTrace();
		}
	}
}
```

To use this class go to the menu and choose “File – New – Class”. Then insert the class name, which is *JavaScriptAction*, and continue. Then replace the code of the new class by the Java code above – eventually the package name has to be adjusted. To shorten the explanation take once again the frame from the example in section 3.2.3 and remove the already added actions by selecting all actions under the *Non-visual components* folder and pressing the *delete* key. After that the new class

can be tested. Therefor choose the button *Add custom class or layout...* within the *Custom* tab and click on the menu item with the name "JavaScript". A frame appears. Type the name "JavaScriptAction". While typing, all possible classes where displayed. Double click on the *JavaScriptAction* class and the dialog to create a new element appears. There is also the possibility to choose a suitable constructor by pressing the button *Change...* next to the *Constructor* field. Now a dialog pops up where the just mentioned activity can be done as well as setting the parameters, which will be passed on to the constructor. Because there is only one constructor available, the correct one is already selected. Unfortunately the parameter names – name and code – are not taken from the constructor definition, so there are two parameters called *param00* and *param01*. Type "JavaScript" into the value field next to *param00*, which will overwrite the text of the menu item. Then click on the "..." button on the right of the field below and copy the JavaScript code from the sample in section 3.1.1 into the popping up frame. Confirm all three windows and the action is finished. Within the *Custom* tab there is a new icon, with the same symbol as the *AbstractAction* icon. If the mouse is moved over this icon the help text tells, that it is a short cut for the just used *JavaScriptAction*, which has been automatically generated by Jigloo.

Now add a button somewhere on the frame. Because the frame has a *BorderLayout*, it might be prettier to change the *direction* of the button to *South*, like mentioned in the first example. Then click the new *JavaScriptAction* icon and add an action to the button just like for the menu item. After that run the frame and the *JavaScriptAction* will execute the corresponding JavaScript code when the menu item or the button is clicked.

3.2.7 BSF4ooRexx Extension

This example is relative similar to the previous one, but what is more the Jigloo extension, that generates a BSF4ooRexx action, will be able to undertake a Java *Swing* element, which can be edited by the ooRexx script. The Java class that is needed for this purpose is called *BSFAction* and shown below.

```
package gui;

import java.util.Vector;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import org.apache.bsf.*;

public class BSFAction extends AbstractAction {
	public String ooRexxCode;
	public Vector guiElements = null;

	public BSFAction(String name, String code) {
		super(name);
		ooRexxCode = code;
	}
	public BSFAction(String name, Object o, String code) {
		super(name);
		ooRexxCode = code;
		guiElements = new Vector();
		guiElements.addElement(o);
	}
	public BSFAction(String name, Object[] args, String code) {
		super(name);
		ooRexxCode = code;
		guiElements = new Vector();
		for (int i = 0; i < args.length; i++) {
			guiElements.addElement(args[i]);
		}
	}
	public void actionPerformed(ActionEvent arg0) {
		BSFManager mgr = new BSFManager();
		try {
			BSFEngine rexxEngine = mgr.loadScriptingEngine("rexx");
			rexxEngine.apply("", 0, 0, ooRexxCode +
					"::requires BSF.CLS;", null, guiElements);
		} catch (BSFException e) {
			e.printStackTrace();
		}
	}
}
```

Import this class and take the frame from the recent example. Then select the file menu to make the quit menu item visible, choose the button *Add custom class or layout...* from the *Custom* tab once more and click on the quit menu item. Type *BSFAction* into the popped up frame and double click the class name, as soon as it is displayed in the list below. Now select the *Change...* button in the upper right corner. On the next frame a different constructor must be chosen, because the default one does not support the transfer of GUI elements. Take the second one in the list, which has three parameters – of data type *String*, *Object* and *String*. The first one is the label, so put the word "Quit" into it. The second is the element, which will be transferred to the ooRexx script. Jigloo provides a list of all elements of the data type *java.lang.Object* within the actual class. Choose the term *this*, because the frame – which is the actual class – should be closed by the new action. After that click on the third parameter and open the text editor frame with the "..." button. Fill it with the script below and accept the three open windows. By now the the frame can be closed through the script.

```
parse arg f;
frame=bsf.wrap('<O>' || f);
frame~dispose;
```

Now there is only one menu item, that does not have an action behind it. So add a new *JLabel* to the frame and change the *text* to "Toggle me!". Then add a *BSFAction* to the menu item via the new icon within the *Custom* tab. Choose the same constructor as before, set an adequate name for the first parameter and select the new created label as second one. When the script from section 3.1.7 is inserted as the last parameter, run the frame and the label can be changed by the script.

3.3 Examples with the Visual Editor

Because the Visual Editor is a plug-in for Eclipse just like Jigloo the creation of a new project is exactly the same. Though the creation of a new frame differs from Jigloo a little bit. On the toolbar there is an icon with a white "C" within a green circle. Push the little triangle next to this icon to open the select menu and select the menu item with the name *Visual Class*. Then the frame to create a new Java class appears. The Visual Editor provides a tree view on the left side of this frame where the class type can be chosen. Select the *Frame* item within the *Swing* folder, fill out the fields *Package* and *Name* and submit the form to add the new frame to the project.

This tool offers a split screen between GUI preview and code as well, but the element icons are on the right side. Just click on an element group and the available elements within these group will be shown. By double clicking on the tab with the class name only these two screens and the elements are visible, but the *Properties* and the hierarchical structure of the GUI elements – here under the label *Java Beans* – are available by clicking on the little icons on the lower right below the element icons.

3.3.1 Button with JavaScript Action

After creating a new frame, the first step is to change the title of it. Therefore go to the *Properties* and search the *>title* entry and set it accordingly. Then select the *JButton* icon within the *Swing Components* element group and move the mouse over the frame. Now the frame is split into the five areas of a *BorderLayout – North*, *West*, *Center*, *East* and *South*. Move the mouse to *South* and click. After that a dialog appears where the internal name of the new element could be changed. Check the box next to "*Do not ask again*", because this name can be changed within the *Properties* as well as via the context menu, if it is really necessary. The text of the button can be set by a right click on it and the menu item *Set Text* or by clicking the button twice with a little time-lag.

The next strep is the label, so choose the *JLabel* icon, also within the *Swing Components*, and move the mouse over the frame. Now the lower part of the frame is grayed out, because this area is used by the button. Click the mouse when it is over *Center* and change the text either via the context menu or by clicking the label twice – just like for the button before.

Then add an action to the button by right clicking it, choosing the *Events* menu item and the *actionPerformed* sub menu item. Finally go to the code screen and replace the content within the *actionPerformed* method – a console output and a comment – by the script engine generating Java code from section 2.2.1 including the JavaScript code from section 3.1.1.

3.3.2 Button with BSF4ooRexx Action

Create a new frame and change the title like mentioned before. Then the layout must be changed, so that an input form can be generated. Therefore right click on the content panel of the frame, which means anywhere within the frame except its header, choose the menu item *Set Layout* and select the

GridLayout. Go to the context menu once again and click the *Customize Layout...* item. Set the *Number of columns* to 2 and the *Number of rows* to 7 and the grid spacing to 1 in both directions.

Now add the elements to the frame. First of all insert seven *JLabels*, three *JTextFields*, two *JRadioButtons*, one *JPasswordField* and one *JButton*, all of them can be found under the *Swing Components* element group. Then sort them by drag and drop until it looks like in the graphic below, the selected one is the password field. This might be a little bit tricky, because elements can only be dropped in the first row or as the very last element.

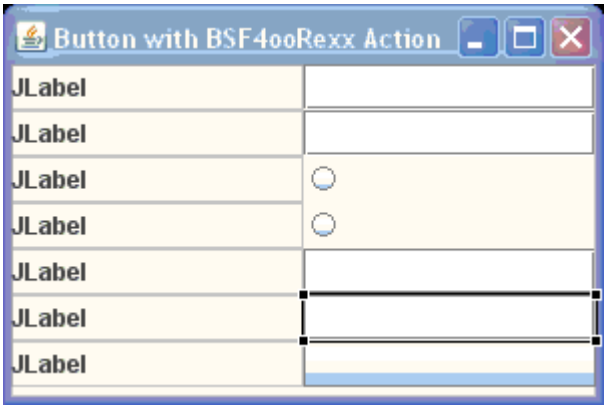

Figure 9: Arranging elements within the *GridLayout*

When this is done, change the text of the labels to “First Name”, “Last Name”, “Gender”, “”, “Username”, “Password” and “”. The fourth and the last label are empty, because they are only place holders within the *GridLayout*. Then set the text of the button to “OK”. After that change the internal names for the input fields by right clicking on each, choosing *Rename Field* and setting their values to “firstName”, “lastName”, “username” and “password”. In the end select each radio button, go to the *Properties* and set the values for *>field name* and *text* for the first radio button to “female” and for the other one to “male” as well as the value for *name* to “gender” for both.

Now the BSF action will be added. This could be done like in the example before, but there is also another possibility. Select the button, go to the *Properties*, click on *action* and then on the “...” button, that appears on the right side of the value field. Double click on *AbstractAction* and the new *actionPerformed* method will be visible within the code screen. In this way there is no content within the method, that has to be deleted, so insert the Java code from section 2.2.2 in combination with the additional code from section 3.1.2 into the empty method and the frame is ready to use. The result is displayed in the following graphic.

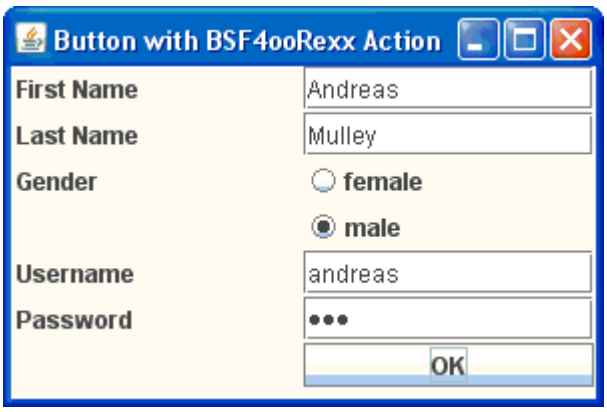

Figure 10: Input form created with the Visual Editor

3.3.3 Menu Bar with Script Actions

First of all open the *Swing Menus* element group, select the *JMenuBar* icon and click on the header of the frame. Then add the first *JMenu* by choosing its icon and clicking the mouse at the narrow line under the frame header, which tries to represent the menu bar. Now the menu bar slowly becomes visible. To make it the normal size, click at the left end of the bar to select the new menu and set its text to "File". Add another menu to the right of the file menu and call it "Scripts". After that the menus need to be completed with *JMenuItems*, one for the file menu and two for the script menu. Unfortunately they are not displayed in the frame preview, when the corresponding menu is clicked. For that reason they only can be selected via the hierarchical structure. So go to *Java Beans* – which is the icon next to *Properties* on the left side – and set the text for each menu item via the context menu and *Set Text*. Call the menu item of the file menu "Quit" and the other ones "JavaScript" and "BSF4ooRexx".

When this is done set an action for the quit menu item also via the context menu – just like shown in the first example. Click into the code screen and exchange the content of the *actionPerformed* method by this line of code:

```
getFrame().dispose();
```

Now the term *getFrame* will be red underlined, because this method does not exist yet. So put the Java code from section 3.2.3 after the *getJMenuItem* method and the red underlining will disappear. Then do the same for the JavaScript menu item and the BSF menu item, but replace the line within the *actionPerformed* method with the corresponding code described in section 3.1.3.

3.3.4 Script Code triggered by Key Event

This example shows how to implement the accelerator feature, which is needed to execute actions via short cuts no matter where the mouse or the cursor is located. Therefore take the frame from the last example, select the quit menu item via the *Java Beans* icon and switch to the *Properties*. Until now there is no property called *accelerator*. So click the icon *Show Advanced Properties*, which is the third from the left within the right upper corner of the *Properties* tab. Now the *accelerator* property appears in the list. Click the "..." button, that appears when clicking into the value field. The *Java Property Editor* will be opened – maybe this frame does not pop up in front of the Eclipse window, but it should be available on the task menu. Select the key *Q*, check the box next to *Control* and submit it. After that do the same for the other menu items, but change the key to *J* for the JavaScript menu item and *R* for the other one. By now, when the class has been run, the scripts can be triggered by *CTRL-J* and *CTRL-R* as prescribed in the scenario description.

3.3.5 Script Code triggered by Mouse Event

Add a new frame to the project and set the layout to *FlowLayout* via the context menu just like shown before. Then add a *JPanel* to the frame, which can be found under the *Swing Containers* element group. Go to the *Properties* and set the *preferredSize* to "250,100" – which means that the panel will have a width of 250 pixels and a height of 100 pixels – and the *>layout* to *BorderLayout*. In addition change the *background* and add a *border*. For both of them appears the *Java Property Editor*, which provides a range of different configurations. When this is done place a *JLabel* into the center of the panel to make it fill out the whole panel. Then change the *Properties* of the label. Set the *horizontalAlignment* to *CENTER*, the *>text* to "Click me!" and choose an appropriate *font*.

Now the mouse listeners will be generated. Right click on the label and choose the menu item *Events*, the the *Add Events ...* sub item. Click the "+" next to *Mouse* on the appearing frame. Choose the *mouseEntered* event and click the *Finish* button. Then go to the code screen and change the content of the *mouseEntered* method according to the descriptions in section 2.2.2 and 3.1.5, but replace the parameter given to the vector with the internal name of the label – most probably "jLabel". Repeat these activities for the *mouseClicked* and the *mouseExited* event, but do not forget to change the text within the BSF code to "I am clicked!" and "Mouse has left me!".

3.3.6 JavaScript Extension

To write an JavaScript extension for the Visual Editor, it is necessary to create a so-called "JavaBean". This is a Java class, which provides a constructor that does not require any parameters as well as a *get* and a *set* method for every class variable. So the variables – like the JavaScript code – cannot be declared as parameters of the constructor when creating a new instance, but they can be set via the *Properties* afterwards. For this reason the *JavaScriptAction* class looks a bit different than before as shown in the code below.

```
package gui;

import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.script.*;

public class JavaScriptAction extends AbstractAction {
	public String javascriptCode;

	public JavaScriptAction() {
		super();
		javascriptCode = "";
	}
	public String getJavascriptCode() {
		return javascriptCode;
	}
	public void setJavascriptCode(String code) {
		javascriptCode = code;
	}
	public void actionPerformed(ActionEvent arg0) {
		ScriptEngineManager factory = new ScriptEngineManager();
		ScriptEngine javascriptEngine =
				factory.getEngineByName("JavaScript");
		try {
			javascriptEngine.eval(javascriptCode);
		} catch (ScriptException e) {
			e.printStackTrace();
		}
	}
}
```

Import this class into your project as described in section 3.2.6 and use the frame created in section 3.3.3 as starting point. First of all remove the already existing actions by going to the *Java Beans* icon, selecting all *actionPerformed* nodes in the tree view and pressing the *delete* key. Now select the *JavaScript* menu item and switch to the *Properties*. Click the "..." button that appears on the right side when choosing the *action* field. Then type "JavaScriptAction" and double click the class *JavaScriptAction* as soon as it is visible within the *Matching Items* field. A new *JavaScriptAction* is added to the menu item. By pressing the "+" next to *>action* the properties of the new *JavaScriptAction* will be displayed. Type some JavaScript code – for instance the script from the first example – into the value field next to *javascriptCode*, but be aware that only one line at once can be copied into this field. Then add a new *JButton* to the frame, give it a name, select it and add an action just like for the menu item before.

3.3.7 BSF4ooRexx Extension

For the seamless BSF4ooRexx integration for the Visual Editor the Java class *BSFAction* – the definition of which is displayed below – needs to be imported into the Java project.

```
package gui;

import java.util.Vector;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import org.apache.bsf.*;

public class BSFAction extends AbstractAction {
	public String ooRexxCode;
	public Object element = null;
	public Vector guiElements = new Vector();

	public BSFAction() {
		super();
		ooRexxCode = "";
	}
	public String getOoRexxCode() {
		return ooRexxCode;
	}
```

```
        public void setOoRexxCode(String code) {
                ooRexxCode = code;
        }
        public Object getElement() {
                return element;
        }
        public void setElement(Object o) {
                element = o;
                guiElements.clear();
                guiElements.addElement(element);
        }
        public void addElement(Object o) {
                guiElements.addElement(o);
        }
        public void actionPerformed(ActionEvent arg0) {
                BSFManager mgr = new BSFManager();
                try {
                        BSFEngine rexxEngine = mgr.loadScriptingEngine("rexx");
                        rexxEngine.apply("", 0, 0, ooRexxCode +
                                        "::requires BSF.CLS;", null, guiElements);
                } catch (BSFException e) {
                        e.printStackTrace();
                }
        }
}
```

For simplicity continue with the frame, that was just used. Select the *BSF4ooRexx* menu item and add an action to it, but now choose the *BSFAction* class. Type the lines of ooRexx code from section 3.1.7 into the *ooRexxCode* property of the new action. The difference to the Jigloo example is, that the Visual Editor does not support a list of all available GUI objects, when the data type of a parameter is *java.lang.Object*, but it provides the possibility to create a new object. So choose the *element* property and click the "..." button on the right side. Then type "JLabel" and double click the *JLabel* class as soon as it is displayed. Open the sub properties of >*element* via the "+" and set the *text* of the label to "Toggle me!". There is a little bug and the new label is not yet displayed on the preview or within the *Java Beans*, so go to the main menu and select "Edit – Undo" and then "Edit – Redo". The frame is still the same, but the new label has appeared as a little dark rectangle below the frame as shown in the following graphic. Now move the label to the center of the frame and run it. Whenever the BSF menu item is selected the label changes its content.

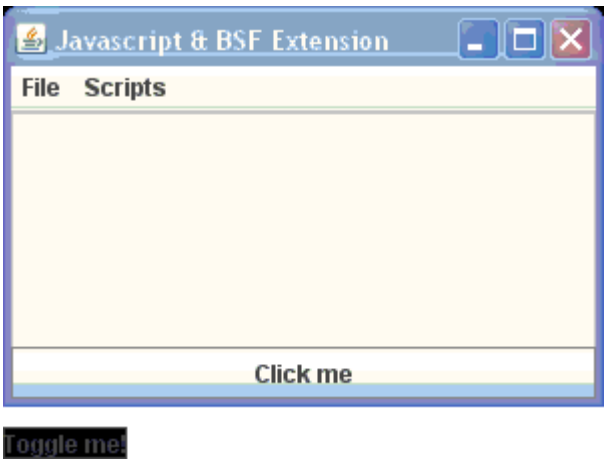

Figure 11: New created label outside the frame

3.4 Examples with NetBeans

The last GUI builder tool, which is be discussed in this paper, is the IDE NetBeans and its integrated GUI forms. Just like for Eclipse, the development of new Java applications is organized through projects. So create a new project by clicking “File – New Project...” within the main menu or simply the second icon from the left on the tool bar. Choose *Java* among *Categories* and *Java Application* among *Projects*, which is the default project type. When selecting *Java Desktop Application* a frame, that already contains a menu bar with a quit option and an about dialog, would be generated. But this would not be very meaningful for the purpose of this section. On the next frame a name for the new project can be set as well as for the main class. Because there will be created some independent frames for demonstrative reasons a main class is not really necessary. So deselect the flag next to *Create Main Class* and finalize it with the *Finish* button. However the project is not completely ready to use, because the BSF libraries are not yet included. So go to the main menu and select “File – Project Properties”. In the tree view choose *Libraries* and click the *Add JAR/Folder* button to add the needed libraries (bsf-rexx-engine.jar and bsf-v400-20090910.jar).

Now create a new frame via the *New File...* button, which is the first one within the tool bar. Choose the folder *Swing GUI Forms* and the file type *JFrame Form*. Into the next window type in a class name and a package name.

Unlike the two Eclipse plug-ins NetBeans does not support a split screen for preview and code, but there are two buttons to switch between the *Design* and the *Source* screen directly above the frame. By double clicking the tab with the class name only the necessary screens are available. In the

Source mode only the code is shown, whereas in the *Design* mode there are three parts. To the left of the frame there is the hierarchical structure of GUI elements, which is called *Inspector*, and on the right side there are the available elements ordered by groups. To change the properties of an element just right click on it and select the menu item *Properties*, which is the last one in the list.

3.4.1 Button with JavaScript Action

At first set the title of the frame, which is done via the *Properties*. Type an appropriate text into the field next to *title*. At the moment the new title is not visible yet, because the preview does only show the content of the frame. However there is an icon with an eye directly above the frame, which is called *Preview Design* and displays a preview of the whole frame.

Due to the fact that the default layout is the *GroupLayout* and for this example a *BorderLayout* is requested, the layout has to be changed. Therefore right click on the frame, choose *Set Layout* and then *Border Layout*. Next add a *JLabel* by selecting the *Label* icon within the *Swing Controls* element group and clicking into the center of the frame. The text can be changed either via clicking it twice with a little time-lag or via the menu item *Edit Text* within the context menu. In case of a label it would also be possible to edit the text by a double click, but if for example a button is double clicked the screen switches automatically to the *Source* mode. Click the *Preview Design* icon to display the frame. The size of the frame is not as expected. So go to the *Properties* of the label and scroll down until the property *preferredSize* is visible. Click the "..." button on the right side and a new frame pops up. Set the *Width* to "400" and the *Height* to "200". Then check the preview again and the frame will be from the expected size. This problem occurred, because NetBeans prioritizes the size of the contents instead of the size of the frame, when the layout is set to *BorderLayout*.

Now the *JButton* will be added. Choose the *Button* icon – which is also in the *Swing Controls* element group – and move the mouse over the bottom of the frame until a rectangle of orange broken lines is visible just like shown in the graphic below. Then place the button with a click into the frame and change its text.

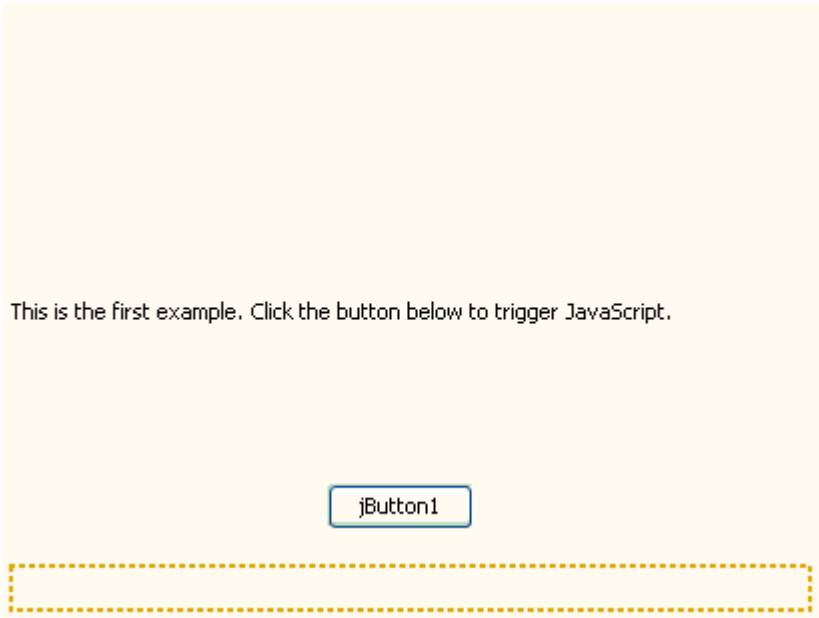

Figure 12: Placing a button into the *South* of a *BorderLayout*

The last step of this example is to add an action to the button. This is done via the context menu of the button and the selection of “Events – Action – actionPerformed” or simply by double clicking the button. As mentioned before the display toggles to the code screen and the cursor is already at the position where the Java code from section 2.2.1 in combination with the JavaScript from section 3.1.1 must be inserted. After that save the frame and run it by going to the main menu and selecting “Run – Run File”. This time it would also be possible to click the icon with the big green triangle within the tool bar, but this icon will always run the first created Java class within the project when there is no main class defined. So for the next examples it would not be feasible anymore.

3.4.2 Button with BSF4ooRexx Action

After a new frame is created and the title is set like mentioned before, the elements can be added to the frame, because the layout is already a *GroupLayout*. This could be proofed by right clicking the frame and moving the mouse over *Set Layout*. The layout name in bold letters is the actual selected one. In this case it is *Free Design*, which is NetBeans' synonym for *GroupLayout*.

First of all add the labels. Select the *Label* icon and move the mouse to the upper left corner over the frame. There are the same broken lines as in section 3.2.2, but now they are blue and not red. So add five labels among each other and change the text to “First Name”, “Last Name”, “Gender”, “Username” and “Password”. Then enlarge the first one of these labels by clicking on it and moving

the right square to the right until the label is approximately a third of the frame. To make all labels the same size, select them by pressing the mouse and moving it over them. Right click on a label, choosing the menu item *Same Size* and the sub menu item *Same Width*.

Now the input elements can be created. Select the *Text Field* icon and place it next to the first label and a second one beneath. Then go to the *Radio Button* icon and add two of them next to the gender label. Finally place another *Text Field* and a *Password Field* beneath. Enlarge the password field to the right until the blue broken line appears near the border of the frame. Mark the text fields and the password field by clicking on them while the *CTRL* key is pressed and make them the same size just like mentioned before. After that set the text of the radio buttons to “female” and “male” and remove the text for all the other ones. To change the internal names either right click on an element and choose *Change Variable Name ...* or go to the *Inspector* screen and click the element twice with a little time-lag. Do not double click, because this would add an action to the element and switch the screen to the code view. Set the variable names to “firstName”, “lastName”, “female”, “male“, “username” and “password”. Until now the radio buttons are two independent elements – if one is selected the other one will not be deselected. So choose the *Button Group* icon – also located within the *Swing Controls* element group – and click anywhere into the frame. The button group appears in the *Inspector* screen as sub element of *Other Components*. Change the variable name to “gender”. Then select the two radio buttons, go to the *Properties* and set the *buttonGroup* property to *gender.*

If the elements are not yet aligned correctly, this can be done by the icons that are located directly above the frame. To automatically adjust the width of the input fields when the frame is resized, select them and click the second icon from the right, which is called *Change horizontal resizability.*

After the form is finalized, add a button to the right lower corner of the frame and set its text to “OK”. Double click the button and insert the code from section 2.2.2 and section 3.1.2 into the action method.

3.4.3 Menu Bar with Script Actions

Create a new frame and open the *Swing Menus* element group. Select the icon *Menu Bar* and click anywhere into the frame. NetBeans automatically adds two empty menus – *File* and *Edit* – to the new menu bar. Because the name of the second menu of this exercise should be “Scripts”, change the text of the edit menu. Then choose the *Menu Item* icon and click on the file menu. The new

menu item is displayed beneath the file menu. To the left of the menu item name there is a gray square. If this is double clicked a frame pops up where an image could be chosen as icon for this menu item. To the right of the menu item name there is a gray rectangle labeled with *shortcut*. This is the accelerator, which will be handled in the next example. So double click the menu item name to set its text to “Quit”. Add two more menu items to the script menu and call them “JavaScript” and “BSF4ooRexx”.

Now the action will be added. Therefore go to the *Inspector* screen and double click the first menu item. An *actionPerformed* method will be created and the display switches to the *Source* screen. Copy the following line of code into the method.

```
this.dispose();
```

In this case the little method to return the frame is not necessary, because the *actionPerformed* method, that NetBeans generates, is not wrapped into the class definition of the action listener. So the frame can simply be addressed via the term *this*. Now switch back to the *Design* screen and double click the next menu item. Insert the Java code from section 2.2.1 and the JavaScript code from section 3.1.1 into the method. Then do the same for the last menu item and the code from section 2.2.2 and the ooRexx script from section 3.1.3.

3.4.4 Script Code triggered by Key Event

As already mentioned in the last example, defining an accelerator with NetBeans is very simple. So take the previous frame, open the file menu on the preview screen and double click on the rectangle labeled with *shortcut* next to the quit menu item. The *Accelerator* frame opens. Click into the input field next to *Key Stroke*, type the letter *Q* on the keyboard and select the box next to *Ctrl*. When the form is submitted, the shortcut appears to the right of the menu item name as displayed in the following graphic. Do the same with the key *J* for the *JavaScript* and with *R* for the *BSF4ooRexx* menu item. Now the accelerators are ready to use, run the frame and type the corresponding shortcuts to start the scripts or to close the frame.

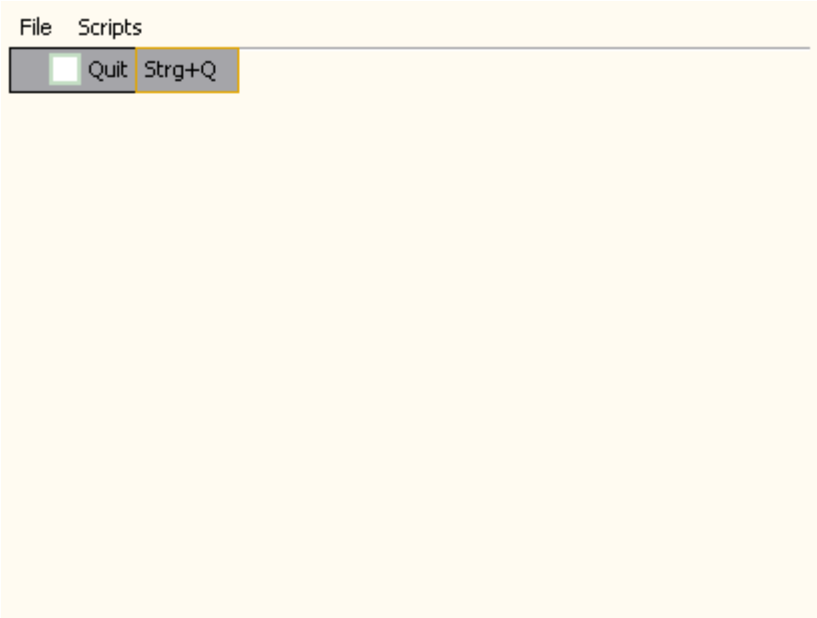

Figure 13: Accelerator combination displayed in the preview frame

3.4.5 Script Code triggered by Mouse Event

For the next example create a new frame and add a label to the left upper corner of the frame. Then press the right lower square of the label and enlarge it to make it about half the size of the frame. Change the text of the label to “Click me!” and open its *Properties*. To make the limits of the label visible check the *opaque* flag and change the property *background* accordingly. When clicking the “...” button next to the *background* field a frame appears, where the color can be chosen in different ways. Set also the *horizontalAlignment* to *CENTER* and the *foreground* color and *font* as desired. Then open the *border* dialog by clicking next to the property of the same name. There are different border types to choose as well as other specifications like the border color and width. Select any of them and the design of the frame is completed. Just like Jigloo, NetBeans also allows to add a container around already created GUI elements. This can be done by right clicking the label and choosing the menu item *Enclose In*. The sub menu items are the available *Swing* containers.

This is the point where the mouse actions have to be created. Hence go to the context menu of the label and select “Events – Mouse – mouseEntered”. The display will toggle to the *Source* screen and the cursor is located within the newly generated method. Put in the code described in section 3.1.5. Then switch back to the *Design* screen and repeat these steps with the sub menu items *mouseClicked* and *mouseExited*. Do not forget to change the text, which is set to the label in the script according to the actual mouse event.

3.4.6 JavaScript Extension

NetBeans also requires JavaBeans to extend its GUI forms, but in this case it is not necessary that the action class is a subclass of *AbstractAction*. So it is possible to simplify the *JavaScriptAction* class a little bit. Import this class by clicking the *New File...* icon in the tool bar and then choosing the file type *Java Class*. On the next page set the class name to *JavaScriptAction* and replace the code with the Java code shown below.

```
package gui;

import javax.script.*;

public class JavaScriptAction {
	public String javascriptCode;

	public JavaScriptAction() {
		super();
		javascriptCode = "";
	}
	public String getJavascriptCode() {
		return javascriptCode;
	}
	public void setJavascriptCode(String code) {
		javascriptCode = code;
	}
	public void executeJavaScript() {
		ScriptEngineManager factory = new ScriptEngineManager();
		ScriptEngine javascriptEngine =
				factory.getEngineByName("JavaScript");
		try {
			javascriptEngine.eval(javascriptCode);
		} catch (ScriptException e) {
			e.printStackTrace();
		}
	}
}
```

Then take the menu bar frame created in section 3.4.3 and delete the already available actions. This is a little bit more complicated than with the other both tools, because the actions are not displayed

in the *Inspector* screen. So go to the *Properties* of each menu item and select the button *Events* in the top row. Next click the "..." button to the right of the *actionPerformed* event and a new dialog pops up, where the event can be deleted via the *Remove* button.

The next step is to add an action to the JavaScript menu item. Therefore open the *Beans* element group and select the icon *Choose Bean*. Now a frame pops up, where the class name of the JavaBean can be typed in. Type "gui.JavaScriptAction" – if the package name differs from the term *gui*, of course the class name has to be adjusted accordingly. Unfortunately there is no list, where all matching classes are offered, in contrast to Eclipse. Then click anywhere into the frame and the *JavaScriptAction* appears on the *Inspector* screen under *Other Components*. Go to the *Properties* of this action, click the "..." button next to *javascriptCode* and insert some JavaScript code into the appearing window.

Now select the *Connection Mode* icon, which is directly to the left of the *Preview Design* icon and shows two windows pointing at each other. After that click the *JavaScript* menu item within the *Inspector* screen following by the *JavaScriptAction*. The *Connection Wizard* pops up. Select the *actionPerformed* event, which can be found within the *action* folder. On the next page check the radio button next to *Method Call* and choose the *executeJavaScript* method. The display will switch to the *Source* screen and the connection is created.

At last add a button to the frame and change its text. Then create a new *JavaScriptAction*, set the property *javascriptCode* and connect the new button to it, just like mentioned before.

3.4.7 BSF4ooRexx Extension

The last example shows the ooRexx extension for NetBeans. Therefore import the *BSFAction* class, that is displayed below and continue with the recent frame.

```
package gui;

import java.util.Vector;
import org.apache.bsf.*;

public class BSFAction {
	public String ooRexxCode;
	public Object element = null;
	public Vector guiElements = new Vector();

	public BSFAction() {
		super();
		ooRexxCode = "";
	}
	public String getOoRexxCode() {
		return ooRexxCode;
	}
	public void setOoRexxCode(String code) {
		ooRexxCode = code;
	}
	public Object getElement() {
		return element;
	}
	public void setElement(Object o) {
		element = o;
		guiElements.clear();
		guiElements.addElement(element);
	}
	public void addElement(Object o) {
		guiElements.addElement(o);
	}
	public void executeOoRexx() {
		BSFManager mgr = new BSFManager();
		try {
			BSFEngine rexxEngine = mgr.loadScriptingEngine("rexx");
			rexxEngine.apply("", 0, 0, ooRexxCode +
					"::requires BSF.CLS;", null, guiElements);
		} catch (BSFException e) {
			e.printStackTrace();
		}
	}
}
```

Then select the *Choose Bean* icon, provide the term “gui.BSFAction” as class name and add the action to the frame. After that change the *Properties* of the *BSFAction*. Click on the *null* next to *element* and a new dialog pops up. Check the radio button next to *Component* and choose the term *Form*. When this is done set the property *ooRexxCode* to the following script.

```
parse arg f;
frame=bsf.wrap('<O>' || f);
frame~dispose;
```

Afterwards connect the quit menu item with the *BSFAction* by using the *Connection Mode* icon. Choose the *actionPerformed* event of the menu item and the method *executeOoRexx* on the next page. By now it is possible to close the frame via the ooRexx script. There would also be an easier way to quit the frame, when directly connecting the quit menu item with the *JFrame* and selecting *dispose* as *Method Call*. However for demonstrative reasons the longer alternative was picked.

Finally the last menu item needs to get an action behind it. So add a label to the frame, call it “Toggle me!” and create a new *BSFAction*. Go the the *Properties* of this action, select the internal name of the new label as *Component* for the *element* property and insert the script from section 3.1.7 as *ooRexxCode*. After that connect the *BSF4ooRexx* menu item to the *BSFAction*, just like described in the last paragraph, run the frame and the label can be toggled via the BSF menu item.

4. Analysis of Tools

After implementing the nutshell examples from the previous chapter, the tree selected GUI builder tools will be evaluated and compared to each other on the basis of some assessment criteria.

4.1 Evaluation Criteria

In this section the criteria, by means of which the tools will be characterized, are defined. These criteria are divided into two main categories – functionality, describes what kind of graphical user interfaces can be created with each tool, and usability, helps the GUI developer to work more quickly and precise.

4.1.1 Functionality

First of all the **amount of available components** is important. This includes all kinds of containers, input elements – like check boxes, radio buttons and password fields –, buttons, menus and so on. Primarily the *Swing* elements are considered, but GUI elements from other packages like *AWT* and *SWT* also will have an impact on the evaluation.

A quite similar criterion is the **availability of different layouts** at which the more flexible ones – like the *GroupLayout* – will be prioritized in comparison with the simpler ones.

The next point is, whether the properties of the elements – like titles, colors, fonts and borders – are editable. These **definitions for components** also imply how attributes, that affect other GUI elements and therefore may not be restricted to one element only, can be changed. Examples for this purpose are the size, position and alignment of components as well as their behavior at window size changes.

Another criterion is the **event handler integration**, which means how actions can be assigned to different events – like mouse, key or action events. The possibility to trigger methods of available components by an event without the need of typing code would have a positive impact on the rating, for example to close a frame with *dispose,* to change the text of a label with *setText* or to read the value of an input field with *getText*.

The last functionality aspect is whether there is already a **script integration** or not. If there is none, it will be verified, how easy it would be to implement an extension that allows script integration.

4.1.2 Usability

One of the most helpful usability features is to provide a preview, that looks like the real window, because many differences between them would force the GUI developer to run the unfinished GUI several times for checking the layout. The **preview reliability** describes for instance whether all visible components are displayed with their actual attributes in the preview.

The more advanced GUI builder tools do not only provide a preview screen and a range of components. There are also some **additional displays**, which help to develop the graphic user interface more quickly, just like a hierarchical structure of the components as well as their properties. Whether these displays are shown on the same screen will have an impact on the estimation too.

The next criterion is the **user interface behavior**, which summarizes all features, that help the GUI developer to perform within the preview window. Instances of these are drag and drop, the usual re-size behavior and graphical assistance. Changing attributes via double click or opening selected menus belong to these features as well.

Another important measurement is the **flexibility of component structure**. It is a very useful feature just to move a component instead of deleting it and creating a new one under the right container, every time a mistake was made or the specifications have changed. If for example a container was forgotten to add before its components are created, these elements can be wrapped into a new container, which is created around them.

Just like the criterion before the next one deals with the correction of mistakes. The **undo behavior** describes whether there is an undo functionality or not as well as if this does exactly what is expected, for instance that only the very last step is revoked. Of course the redo functionality is also part of it.

One more aspect is the ease of running the just created graphical user interface. Therefore an **integrated compiler and executor** is very convenient.

Furthermore there is the question, whether the tool is generating the Java code immediately, so that the developer is able to change it manually if he likes to. Another point of this **coding support** is if there are automated warnings on code errors, for example missing libraries or syntax errors.

4.2 Evaluation of Tools

The following section will show the ratings of the selected tools with the help of the just defined functionality and usability criteria.

4.2.1 Functionality

When checking the available components of the three tools it was noticed that most of these elements are existing in all of them. The only tool, which lacks a little bit, is the Visual Editor. The most annoying thing is that there is no *ButtonGroup*, so radio buttons cannot be grouped without additional coding. This makes the radio buttons unusable (for unskilled GUI developers), because if one of them will be selected, the already selected radio button will not be automatically deselected. But this is not the only component which is missing within the Visual Editor. There is also neither a *JFileChooser* nor a *JColorChooser*. These two provide a simple interface to let the user select a file on his computer or a self defined color and pass it to the Java application. The *JSpinner*, a kind of select box for numbers, the *JSeparator* and the *JFormattedTextField* are lacking as well. The only *Swing* component that is only available in the Visual Editor is the *TableColumn*. The special element of NetBeans is the *JLayeredPane*, which provides a third dimension by offering five overlapping layers. However there is no *JWindow*. Just like the Visual Editor, NetBeans extends the *Swing* components by the common *AWT* elements, but NetBeans offers some more of them, namely the *MenuBar*, the *PopupMenu* and the *Canvas*. Altogether Jigloo is the best in this category. There is no important component, that is not supported. In addition Jigloo provides the *AbstractAction*. The only *AWT* element, that is offered, is the *Canvas*, but all other common *AWT* components are covered by the *Swing* package anyway. What is more Jigloo enables the creating of *SWT* GUIs and even a so-called *SWT-AWT-Bridge*, which allows the GUI developer to add *SWT* elements to an *AWT* or *Swing* container.

It is a similar situation concerning the layouts. Each of the tools support the most popular layouts, like the *BorderLayout*, *GridLayout* and the *FlowLayout*. Both Jigloo and NetBeans also provide additional layouts, which are neither part of *AWT* nor *Swing*, NetBeans its *AbsoluteLayout* and Jigloo the *AnchorLayout*, *FormLayout*, *MigLayout* and *TableLayout*. Just like before the Visual Editor does not achieve the best results, because the *GroupLayout* is missing, which is the most convenient one, when working with a GUI builder tool.

The properties of the components are available in every tool. In Jigloo the alignment, as well as the adjustment of the elements' size and spaces between them, can be set with the icons to the left of the preview. The behavior on window size changes can be adjusted via the little black triangle at the right upper corner of selected elements. NetBeans provides these settings within the context menu, for alignment and resizability there also exist some icons directly above the preview. Within the

Visual Editor there is a menu item *Customize Layout...* in the context menu, where all the mentioned adjustments can be done, if they are available for the current layout – for example the *null* layout. So the only limitation is the lack of the *GroupLayout*, but that cannot influence the evaluation for this criterion. So all tools fulfill it very well.

To add event handlers to the GUI all tools provides different possibilities. As already mentioned before Jigloo offers the *AbstractAction* class as element, which can easily be placed on buttons and menu items. In addition there is an *Events* tab next to the properties where either inline or as handler method an action can be set for all kinds of events and for every element. When using the Visual Editor event handlers can be created via the context menu for every kind of element and event. There is also the possibility to add an action event for a button or a menu item via the property *action*. However it is necessary to type the code, that is triggered by the event, manually for both tools. Solely NetBeans offers a possibility to define an event action without writing code. With the *Connection Mode* icon two elements can be combined by selecting them. Then an event of the first selected element as well as a property – which should be changed – or a method of the second one can be chosen. For every parameter of the method a value can be stated, which could be again the result of the method or the property of a third element. Of course there is also the alternative to add an event via the context menu and the corresponding action by typing of code.

Although there is no script integration for any of the tools implemented so far, for each of them it is not difficult to write a little extension class that is able to execute scripts, when a button or a menu item is clicked. Therefore the GUI developer does not need to write any line of Java code to trigger a desired script. Because of NetBeans' *Connection Mode* feature the scripts cannot only be triggered by the *actionPerformed* event, but also by mouse or key events. What is more one single JavaBean could handle different script languages by means of different methods. For that reason NetBeans will get a better rating than the other tools.

4.2.2 Usability

The first criterion in this category is the preview reliability. The only tool, which really is a “What you see is what you get”-editor, is the Visual Editor. The frame in the preview is rendered exactly the same as the real window, when the Java class is run. Jigloo provides a pretty good preview window, but there might be differences in the design, if the so-called “Look and Feel” differs from the default settings. The “Look and Feel” can be set via the main menu “Window – Preferences”

and the tree menu item “Jigloo GUI Builder – Look and Feel”. Because the GUI developer is forced to check these settings, Jigloo is not graded as good as the Visual Editor. The preview windows of NetBeans have several problems. First of all only the content of the frame is shown, so there is no header where the title can be displayed. Again the “Look and Feel” might not be the same. In the context menu, there is the possibility to open a preview window with any “Look and Feel”, but it is annoying to do this several times. Furthermore when changing the layout to *BorderLayout* the size of the frame will not be shown correct in the preview.

When talking about additional displays, Jigloo provides a very nice arrangement of all necessary screens – preview, code, element structure and properties. So if for example a property was changed, the effect immediately can be seen in the preview and in the code, as well as the other way round. For the Visual Editor the default setting is not as convenient, because the properties and the component structure are not displayed at the same time. However it is possible to arrange the screens as desired by drag and drop. In NetBeans the GUI developer has to toggle between the preview and the code view, so impacts of code changes on the window cannot be seen until he switches back to the preview. Therefore NetBeans is rated slightly worse than the others.

Concerning the user interface behavior, general features like selecting elements via clinking, moving them by drag and drop and re-sizing them with the mouse are supported by each of the tools. In addition there is the possibility to edit text through clicking. In NetBeans a double click on a button automatically adds an event handler and switches the screen to the code view, so the GUI developer can insert the Java code, that should be triggered by the button. Jigloo and NetBeans provide broken lines to assist the positioning of elements in a *GroupLayout*. The Visual Editor offers a real good graphical support for the *BorderLayout* instead by showing the free and the already occupied areas. So the graphical assistance is very well for all of them. What is more NetBeans and Jigloo display the content of a menu, when it is selected in the preview. This enables an easy selection and manipulation of menu items, which is responsible for the better ranking.

In all tools within the hierarchical element structure display every component can be moved into another container by drag and drop. For most elements this is also possible within the preview. The only tool that allows a rearrangement of menu items within the preview window is NetBeans. However if a component is moved into a new container via the structure display, the size of both the new and the old container will be changed and so the arrangement of elements is not the same anymore. A very nice feature of Jigloo and NetBeans is that components can be wrapped into a new container. For example if this container is a *JTabbedPane*, every component is put into its own tab.

The only tool that narrows the flexibility of the component structure is the Visual Editor when setting a parameter that requires an object. In this case only a new element can be created, instead of choosing an existing one. So the already existing one might have to be deleted.

The next criterion is the behavior of the undo functionality. In Eclipse it is available via the *Edit* menu and cancels the last manipulation of the code, which of course also includes any changes within the preview, structure or properties, because they trigger code generations or manipulations. The redo action works accordingly. NetBeans offers undo and redo buttons in the tool bar too, but it differentiates between code manipulation and changes within the *Design* mode. If for instance an action is added to a button through a double click and the screen switches to the *Source* mode, the undo button is deactivated, because there was no change within this mode. To undo the creation of the action, the GUI developer has to toggle back to the *Design* mode where the undo button can be clicked. This might be confusing, especially if someone does not know it. For example a new component was added to the preview and then something was changed within the code. Back to the *Design* mode the developer wants to undo the last change – which should be the code manipulation. So he clicks the undo button and removes the component, that was created just before the code was changed.

Due to the fact that both NetBeans and Eclipse are IDEs, there is of course an integrated compiler and executor available for each of the tools. So they all get the best rating. For the same reason there is also a very good code support with syntax highlighting and hints to code errors and warnings. The only point of critique is that the automatically generated code in NetBeans can only be edited via the *Code Customizer* – which is available through the context menu – and even there not every part of the code can be changed. This might have security reasons, and of course the code can be manipulated when opening the Java file outside of NetBeans, but it restricts the usability of the tool a little bit.

4.2.3 Comparison

The table below visualizes the results of the evaluation. The graduation ranges from ++ for very well performing tools over ~ for average ones to – – for the weakest ones.

	Jigloo	**Visual Editor**	**NetBeans**
Functionality			
Amount of Available Components	++	–	+
Availability of Different Layers	++	–	++
Definitions for Components	++	++	++
Event Handler Integration	~	~	++
Script Integration	~	~	+
Usability			
Preview Reliability	+	++	–
Additional Displays	++	++	+
User Interface Behavior	++	+	++
Flexibility of Component Structure	++	~	++
Undo Behavior	++	++	~
Integrated Compiler and Executor	++	++	++
Coding Support	++	++	+

Figure 14: Results of the evaluation

5. Conclusions

Generally speaking the three tools Jigloo, Visual Editor and NetBeans are all qualified to assist developers to generate useful graphical user interfaces with Java. Likewise each of them can be used to create a kind of script integration with simple extension classes.

However when thinking about a seamless integration of scripting languages into the GUI generating process – which means that the GUI developer do not need to write any line of Java code – NetBeans would be the best solution. In some cases it is not even necessary to write any code, for instance if only some text or other attributes of an element should be changed or the window should be closed. In NetBeans this can be achieved easily with the aid of the *Connection Mode*. The more event handlers are needed and the less code should be written manually, the better is NetBeans suited.

On the other hand Jigloo would be an excellent decision for developers, who like to change the code quite often manually to increase the programming performance. In this case the usability of Jigloo is much better than NetBeans', because of the split screen and the unlimited code manipulation. Furthermore Jigloo is one of the both that also enable the creation of *SWT* windows, if this was required.

The most prominent characteristic of the Visual Editor is its preview reliability. In all other categories Jigloo performs a little bit better or is equal, because of their common IDE Eclipse.

Altogether it depends on the personal preferences and expectations, which of these top level GUI builders is the most adequate to realize the development of a Java application.

References

[Bea03] Beausoleil, Francois. Java Gui Builder Project Home.
http://jgb.sourceforge.net/

[CBS11a] CBS Interactive GmbH. Java Gui Builder 1.0.
http://www.zdnet.de/java_software_entwickler_unter_windows_java_gui_builder_download-39002345-7473-1.htm

[CBS11b] CBS Interactive GmbH. JFrameBuilder download.
http://download.cnet.com/JFrameBuilder/3000-2213_4-10327454.html

[Clo10] Cloud Garden. Jigloo SWT/Swing GUI Builder for Eclipse and WebSphere.
http://www.cloudgarden.com/jigloo/

[Ecl11a] The Eclipse Foundation. Eclipse Homepage.
http://www.eclipse.org/

[Ecl11b] The Eclipse Foundation. Visual Editor Project.
http://www.eclipse.org/vep/WebContent/main.php

[Gee09] Geeknet, Inc. BSF4ooRexx.
http://bsf4oorexx.sourceforge.net/

[Gee11] Geeknet, Inc. Sourceforge Homepage.
http://sourceforge.net/

[IST11] IST Limited. Visaj – The Visual Application Builder for Java.
http://www.ist.co.uk/visaj/

[OCo06] O'Conner, John. Scripting for the Java Platform.
http://java.sun.com/developer/technicalArticles/J2SE/Desktop/scripting/

[Ora11a] Oracle Corporation. NetBeans Homepage.
http://netbeans.org/

[Ora11b] Oracle Corporation. Java Platform, Standard Edition 6 – API Specification.
http://download.oracle.com/javase/6/docs/api/

[Rex09] Rexx Language Association. Open Object Rexx Homepage.
http://www.oorexx.org/

[Woe11] Wöhrmann Softwareentwicklung. SpeedJG – XML based Java Swing GUI Builder.
http://www.wsoftware.de/SpeedJG/index.html

All references have been accessed on 11th June 2011 for the last time.