Gaia

Version 2011.1

User Manual

Contents

1 Introduction

Gaia is a free Java library from the BeeSoft company.

It was created as the internal library to support the application development in company. It is available on the web since 2010. the library is not an open-source, but it is free to use.

Here you can find its main features:

- support for building Swing form from XML file including the data-binding
- framework for building Swing client / server applications with the code processing on the server side
- launcher subsystem to run appliciation
- general logging interface
- support for easy XML processing
- utilities for beans, value objects, streams and so on

2 Swing support

Gaia offers the strong support for Swing programming.

There is a few interesting components contained in our library, but especially the building Swing forms from XML file supplemented with automated data-binding is great help for programmers.

With Gaia you can also very easy build client / server application with Swing user interface and bussiness logic concentrated on the server.

2.1 Components

Gaia library comes with a few Swing components to support programming. The greatest are described in the separate chapters, here we name the others:

JZebraTable

Expands capacities of *javax.swing.JTable* component. JZebraTable has the following features:

- each column has a visibility - you can show and hide columns and preserve their order
- right click on the table header invokes a popup menu with names of all columns - user can change visibility of the columns this way
- the table rows can have the alternate background (each even row)
- this component resolves the problem with JTable which becomes uglier with AUTO_RESIZE_OFF
- you should use *createScrollPane(JZebraTable)* method instead of *new JScrollPane (table)* to ensure the pretty look of the table

JZebraTree

The JZebraTree expands the capacities of *javax.swing.JTree* about these features:

- the tree rows can have the alternate background (each even row)
- there are completed methods to expand / collapse all nodes

EmptyIcon

Represents a square icon having no graphical content.

Intended for use with `Action` and `JMenuItem`. Alignment of text is poor when the same menu mixes menu items without an icon with menu items having an icon. In such cases, items without an icon can use an `EmptyIcon` to take up the proper amount of space, and allow for alignment of all text in the menu.

JButtonPanel

This component solves repeated problem with buttons positioning and displaying. Its features:

- the buttons are layouted side by side
- the buttons can be aligned to the left, right or centered
- all buttons have the same dimension
- it is possible to show line separator above the buttons
- the component should be displayed below the form

JCloseableTabbedPane

This component solves a close-buttons for tabs of the tabbed pane.

From JDK 1.6 you can use any component to paint tab, but this is impossible in version 1.5. This component paints a small close button behind the tab title, so you can use this close-tab feature in version 1.5.

The component is closeable (paints the close button) by default. You can change this by calling setCloseable(boolean) method.

The component notifies about the mouse click on te close button all registered action listeners. You can accomplish also automatic close of the tab after the listeners are notified. Set property autoClose to true via method setAutoClose(boolean).

JLink

A Swing component to display some label and to invoke internet browser for requested URL by clicking on it.

2.1.1 JDateField and JCalendar

The *JCalendar* is the component that displays a small table with the 28 - 31 days of one month. User can change the month and / or the year with arrow buttons.

The day and month names are obtained from java.text.DateFormatSymbols class and they depend on the user locale.

The component has methods to set initial date and to get the selected date. All you have to do is install ActionListener to be notified about user selection. ActionEvent you will get to the listener's actionPerformed() method will contain one of the two possible commands:

- JCalendar.SELECTED
- JCalendar.CANCELED

and you can decide the next steps.

JDateField is the component created to edit the date values.

It contains text field to edit the date and small button on its right side. This button popups a *JCalendar* component to choose the date from the popup calendar.

2.1.2 Lazy tree data loading

You can implement tree data lazy loading very easily with two classes:

- *ExplorableTreeNode* - is an abstract descendant of *javax.swing.tree.DefaultMutableTreeNode* with enhanced expansion functionality capabilities. If this node has set the 'explorable' property and tree where this node is displayed has installed *ExplorationTreeListener* as tree expansion listener, then method

exploreImpl() is invoked when this node is the first time displayed. Subclass and override this method to process lazy loading.

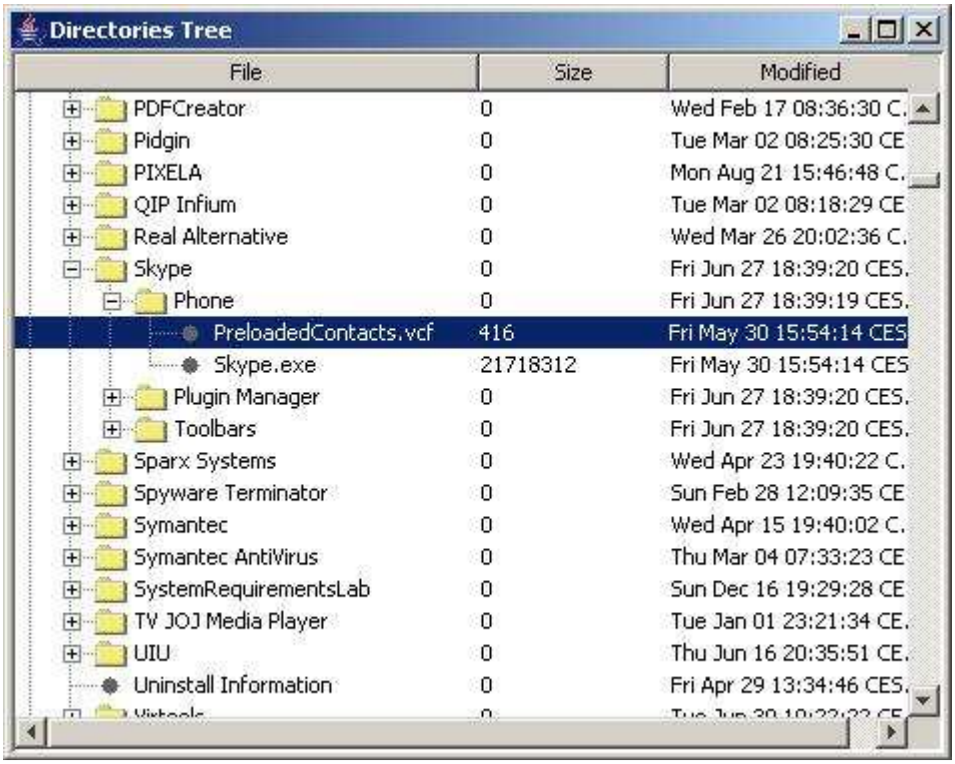
- *ExplorationTreeListener* - is a listener which you can add to any tree - it listenes for expand / collapse events and if expanded node is instance of *ExplorableTreeNode* it invokes its method *explore()*.

2.1.3 JTreeTable

A JTreeTable is a combination of a JTree and a JTable - a component capable of both expanding and contracting rows, as well as showing multiple columns of data.

This component always displays a tree in the first column and the additional data in the next columns. You can see its usage in a very simple example of the directory tree.

The code in this class is derived from the source from the JTreeTable article that can be found at: http://java.sun.com/products/jfc/tsc/articles/treetable2/index.html.



This component has the next features:

- it inherits from its parent class ability to hide / show columns
- also inherits possible "*zebra*" look
- the tree is always painted in the first column
- as its model you must use TreeTableModel or subclasse it

- supports data lazy loading
- descendants of the TreeTableNode are only nodes acceptable in the JTreeTable

The *TreeTableModel* which is used as model for JTreeTable, extends *javax.swing.tree.DefaultTreeModel* to meet the requirements for model for JTreeTable.

The *TreeTableNode* is hierarchical node for the JTreeTable. It inherits lazy loading possibility from the *ExplorableTreeNode* and adds the methods for get / set value for required column.

2.1.4 SimpleSwingForm

SimpleSwingForm is a component designed to the simplification of building the Swing forms.

- it uses GridBagLayout to layout the components
- it layouts components automatically, each component on the new row, but you can change this behavior by overriding methods *customizeLabelConstraints(GridBagConstraints, JLabel)* and *customizeComponentConstraints(GridBagConstraints, JComponent)*
- the most simple is building of the two-columns form, where are labels in the first column
- component offers a lot of the *showDialog(...)* methods to show this form in a dialog
- if you subclasses this component, you can use standard OK action and serve it in *okActionPerformed()* method without writting any action

Here is a simple example of SimpleSwingForm usage:

```
SimpleSwingForm form = new SimpleSwingForm ();
form.addLabeledComponent ("Label 1", form.createTextField ());
form.addLabeledComponent ("Label 2", new JTextArea ());
Action okAction = new MyOkAction (); // you must do something on OK
form.showDialog ("My form", okAction,
form.createStandardCancelAction());
```

This builds on the 5 lines of code a two-rows form with labels "Label 1" and "Label 2" in the first column and components text field and text area in the second. Then is the form displayed in a dialog.

2.1.5 Form

The Form is a component to hold and layout components in one form. It uses *java.awt.GridBagLayout* as a default layout and instances of eu.beesoft.gaia.swing.form.CellConstraints as the component constraints (instead of the

GridBagConstraints). Each inserted component has its border replaced by eu.beesoft.gaia.swing.form.EditableComponentBorder to show the user if component is editable. For editable components you can set the background color by method setEditableComponentBackground(Color) and border color by method setEditableComponentBorderColor(Color).

The *Form* instance has the ability to paint the horizontal and / or vertical lines around the component cells, if it is set in CellConstraints instance(s). CellConstraints is derived from GridBagConstraints and contains boolean variables for painting left border, top border, etc. This gives the "*table*" look to the form. The color of the cell borders can be set by method setCellBorderColor(Color).

2.2 XML builder

One of the most attractive features of our Swing part of the Gaia library is the XML support for building Swing forms or applications. Yes, there are many such tools, but as far as we know, no one of them supports also data binding and internationalization. So Gaia offers the possibility to build Swing form (or whole application) from XML file in runtime with minimum programmer effort.

For example, this data class:

```
import java.util.Date;

public class Person {

     private String firstName;
     private String lastName;
     private Date dateOfBirth;
     private Person father;
     private Person mother;


     public Person (String firstName, String lastName, Date dateOfBirth)
{
             this.firstName = firstName;
             this.lastName = lastName;
             this.dateOfBirth = dateOfBirth;
     }


     public String getFirstName () {return firstName;}
     public void setFirstName (String firstName) {
             this.firstName = firstName;}
     public String getLastName () {return lastName;}
     public void setLastName (String lastName) {
```

```
            this.lastName = lastName;}
    public Date getDateOfBirth () {return dateOfBirth;}
    public void setDateOfBirth (Date dateOfBirth) {
            this.dateOfBirth = dateOfBirth;}
    public Person getFather () {return father;}
    public void setFather (Person father) {this.father = father;}
    public Person getMother () {return mother;}
    public void setMother (Person mother) {this.mother = mother;}
}
```

can be displayed in this form:

with this code:

```
import java.awt.Frame;
import java.io.InputStream;
import java.util.Date;
import javax.swing.JDialog;
import javax.swing.JPanel;
import eu.beesoft.gaia.swing.builder.SwingBuilder;
import eu.beesoft.gaia.swing.builder.SwingBuilderFactory;
import eu.beesoft.gaia.util.Streams;

public class ShowPerson {

    public static void main (String[] arg) {
```

```
            // prepare data objects
            Person person = new Person ("John", "Cassidy", new Date
(1980 - 1900, 5, 20));
            Person father = new Person ("Frank", "Censky", null);
            person.setFather (father);
            Person mother = new Person ("Annie", "Abell", null);
            person.setMother (mother);

            // build form
            InputStream is = Streams.getInputStream ("PersonForm.xml");
            SwingBuilderFactory factory = new SwingBuilderFactory ();
            SwingBuilder builder = (SwingBuilder) factory.build (is);
            JPanel panel = (JPanel) builder.getObject ();

            // bind data
            builder.setBoundData (person, null);

            // show form
            JDialog dialog = new JDialog ((Frame) null, "Person", true);
            dialog.getContentPane ().add (panel);
            dialog.pack ();
            dialog.setVisible (true);
    }
}
```

from this XML FORM description file:

```
<?xml version="1.0" encoding="UTF-8"?>

<form editableComponentBackground="white">

    <!-- Person section -->
    <label id="personTitle" text="Person" foreground="blue"
           font="arial-bold-18">
           <cell id="personTitleCell" x="0" y="0" width="2" height="1"
                  weightx="1.0" weighty="0.0" anchor="west"
                  border="bottom"/>
    </label>

    <label text="First name: ">
           <cell id="firstColumn" x="0" y="1" width="1" height="1"
                  weightx="0.0"
                  weighty="0.0" anchor="east" border="bottom" />
    </label>
    <textfield id="firstName" binding="firstName" >
           <cell id="secondColumn" as="firstColumn" x="1" weightx="1.0"
                  anchor="west" fill="horizontal" border="bottom" />
    </textfield>

    <label text="Last name: ">
           <cell as="firstColumn" under="firstName" />
    </label>
```

```
    <textfield id="lastName" binding="lastName">
           <cell as="secondColumn" under="firstName" />
    </textfield>

    <label text="Date of birth: ">
           <cell as="firstColumn" under="lastName" />
    </label>
    <datefield id="dateOfBirth" binding="dateOfBirth">
           <cell as="secondColumn" under="lastName" />
    </datefield>


    <!-- Father section -->
    <label id="spaceBeforeFather" text=" ">
           <cell as="firstColumn" under="dateOfBirth" border="none" />
    </label>

    <label id="fatherTitle" as="personTitle" text="Father">
           <cell as="personTitleCell" under="spaceBeforeFather" />
    </label>

    <label text="First name: ">
           <cell as="firstColumn" under="fatherTitle" />
    </label>
    <textfield id="fatherFirstName" binding="father.firstName">
           <cell as="secondColumn" under="fatherTitle"/>
    </textfield>

    <label text="Last name: ">
           <cell as="firstColumn" under="fatherFirstName" />
    </label>
    <textfield id="fatherLastName" binding="father.lastName">
           <cell as="secondColumn" under="fatherFirstName"  />
    </textfield>


    <!-- Mother section -->
    <label id="spaceBeforeMother" text=" ">
           <cell as="firstColumn" under="fatherLastName"
                  border="none" />
    </label>

    <label id="motherTitle" as="personTitle" text="Mother">
           <cell as="personTitleCell" under="spaceBeforeMother" />
    </label>

    <label text="First name: ">
           <cell as="firstColumn" under="motherTitle" />
    </label>
    <textfield id="motherFirstName" binding="mother.firstName">
           <cell as="secondColumn" under="motherTitle" />
    </textfield>

    <label text="Last name: ">
           <cell as="firstColumn" under="motherFirstName"  />
    </label>
    <textfield id="motherLastName" binding="mother.lastName">
```

```
            <cell as="secondColumn" under="motherFirstName"  />
    </textfield>

</form>
```

2.2.1 Step by step

Of course, the first step is to create a XML file with Swing form description. Currently there is no tool to support this work, so you must to do it manually.

You can read about all supported elements and attributes in the next chapter. Please, note, there are supported all important Swing components and layouts, and you can enhance this support with your own classes.

When you have your XML file completed, you need to process it to the Swing form. The basic pattern is here:

```
    // prepare path to form XML
    String formId = "mypackage/myform.form";

    // prepare resource bundle name
    // this is not necessary,
    // you must not always use the resource bundles
    String messagesBundleId = "mypackage.myform";

    // create new builder factory
    SwingBuilderFactory factory = new SwingBuilderFactory ();
    factory.setResourceBundle (messagesBundleId);

    // build component (form, menu, application, ...)
    InputStream is = Streams.getInputStream (formId);
    ComponentBuilder builder = (ComponentBuilder) factory.build (is);

    // obtain a root component
    JComponent component = (JComponent) builder.getObject();

    // use this component
    ...
```

You can see, the SwingBuilderFactory class is playing the key role in our code. This class registers which element tag is processed by which builder class. Each buider instance is responsible for creation one component. And most of our builders are derived from eu.beesoft.gaia.swing.builder.SwingBuilder class.

You can also write your own builder if Gaia built-in support does not meet your requirements, or if you are using the component we don't support. Then you can register it to the factory in the pair with processed element tag with this code:

```
SwingBuilderFactory factory = new SwingBuilderFactory ();
factory.registerBuilderClass ("mytag", MyBuilder.class);
```

A SwingBuilderFactory instance keeps a list of all builders it produced. So you can access to any component in the built form via its **id**:

```
String id = "label_m01";
LabelBuilder builder = (LabelBuilder) factory.getBuilder (id);
JLabel label = builder.getObject ();
```

A special attention is focused on the Swing actions and a ListModels. There are implemented methods to obtain:

```
// returns the action builders as children of the given builder
List actionBuilders = factory.getActionBuilder (rootBuilder);

// returns the actions produced by children of the given builder
List actions = factory.getActions (rootBuilder);

// returns a collection of all list model builders
List listModelBuilders = factory.getListModelBuilders ();

// returns a collection of list model builders
// as children of the given builder
List listModelBuilders = factory.getListModelBuilders (rootBuilder) ;
```

These method are needed to obtain actions supported by the built form and also to initialize list models with data.

2.2.2 XML elements and attributes

As you can see in the example above, XML structure copies the Swing component hierarchy. Each element has tag which is mapped to the corresponding Swing component (more exactly, to its builder). Attributes describes the properties of the Swing component, they names are preserved. And XML inner elements correspond to the child components in Swing component hierarchy.

Gaia currently supports these tags:

- **action** - processed by eu.beesoft.gaia.swing.builder.ActionBuilder, creates an instance of eu.beesoft.gaia.swing.client.SwingClientAction
- **button** - processed by eu.beesoft.gaia.swing.builder.ButtonBuilder, creates an instance of JButton
- **cell** - processed by eu.beesoft.gaia.swing.builder.CellConstraintsBuilder, creates an instance of eu.beesoft.gaia.swing.form.CellConstraints
- **checkBox** - processed by eu.beesoft.gaia.swing.builder.CheckBoxBuilder, creates an instance of JCheckBox
- **checkBoxMenuItem** - processed by eu.beesoft.gaia.swing.builder.CheckBoxMenuItemBuilder, creates an instance of JCheckBoxMenuItem
- **column** - processed by eu.beesoft.gaia.swing.builder.TableColumnBuilder, creates an instance of TableColumn
- **comboBox** - processed by eu.beesoft.gaia.swing.builder.ComboBoxBuilder, creates an instance of JComboBox
- **dateField** - processed by eu.beesoft.gaia.swing.builder.DateFieldBuilder, creates an instance of eu.beesoft.gaia.swing.builder.JDateField
- **dialog** - processed by eu.beesoft.gaia.swing.builder.DialogBuilder, creates an instance of JDialog
- **filler** - processed by eu.beesoft.gaia.swing.builder.FillerBuilder, creates an empty JLabel to fill space
- **form** - processed by eu.beesoft.gaia.swing.builder.FormBuilder, creates an instance of eu.beesoft.gaia.swing.form.Form
- **frame** - processed by eu.beesoft.gaia.swing.builder.FrameBuilder, creates an instance of JFrame
- **gbc** - processed by eu.beesoft.gaia.swing.builder.GridBagConstraintsBuilder, creates an instance of GridBagConstraints
- **label** - processed by eu.beesoft.gaia.swing.builder.LabelBuilder, creates an instance of JLabel
- **link** - processed by eu.beesoft.gaia.swing.builder.LinkBuilder, creates an instance of eu.beesoft.gaia.swing.JLink
- **list** - processed by eu.beesoft.gaia.swing.builder.ListBuilder, creates an instance of JList
- **listModel** - processed by eu.beesoft.gaia.swing.builder.ListModelBuilder, creates an instance of DefaultListModel or DefaultComboBoxModel
- **menu** - processed by eu.beesoft.gaia.swing.builder.MenuBuilder, creates an instance of JMenu
- **menuBar** - processed by eu.beesoft.gaia.swing.builder.MenuBarBuilder, creates an instance of JMenuBar
- **menuItem** - processed by eu.beesoft.gaia.swing.builder.MenuItemBuilder, creates an instance of JMenuItem
- **panel** - processed by eu.beesoft.gaia.swing.builder.PanelBuilder, creates an instance of JPanel
- **passwordField** - processed by eu.beesoft.gaia.swing.builder.PasswordFieldBuilder, creates an instance of JPasswordField

- **radioButtonMenuItem** - processed by eu.beesoft.gaia.swing.builder.RadioButtonMenuItemBuilder, creates an instance of JRadioButtonMenuItem
- **scrollPane** - processed by eu.beesoft.gaia.swing.builder.ScrollPaneBuilder, creates an instance of JScrollPane
- **separator** - processed by eu.beesoft.gaia.swing.builder.SeparatorBuilder, creates an instance of JSeparator
- **splitPane** - processed by eu.beesoft.gaia.swing.builder.SplitPaneBuilder, creates an instance of JSplitPane
- **tabbedPane** - processed by eu.beesoft.gaia.swing.builder.TabbedPaneBuilder, creates an instance of eu.beesoft.gaia.swing.JCloseableTabbedPane
- **table** - processed by eu.beesoft.gaia.swing.builder.TableBuilder, creates an instance of eu.beesoft.gaia.swing.JZebraTable}
- **tableColumn** - processed by eu.beesoft.gaia.swing.builder.TableColumnBuilder, creates an instance of TableColumn
- **textArea** - processed by eu.beesoft.gaia.swing.builder.TextAreaBuilder, creates an instance of JTextArea
- **textField** - processed by eu.beesoft.gaia.swing.builder.TextFieldBuilder, creates an instance of JTextField
- **toolBar** - processed by eu.beesoft.gaia.swing.builder.ToolBarBuilder, creates an instance of JToolBar
- **tree** - processed by eu.beesoft.gaia.swing.builder.TreeBuilder, creates an instance of eu.beesoft.gaia.swing.JZebraTree
- **treeTable** - processed by eu.beesoft.gaia.swing.builder.TreeTableBuilder, creates an instance of eu.beesoft.gaia.swing.JTreeTable

Each XML element has attributes that describe Swing component properties. For java.awt.Component (and all its descendants) are supported these attributes:

- **background** - enabled values are constant names from java.awt.Color(for example: BLACK), or r, g, b (for example: 128,255,15), or one RGB number in hexa format (for example: 504B1C)
- **border** - see description to {@link #initBorder(String)} method
- **enabled** - enabled values are true or false
- **font** - enabled values conform to java.awtFont.decode() format
- **foreground** - enabled values are like for background
- **layoutConstraint** - enabled value is an integer or a name of the constant from the layout manager class of the parent container
- **name** - any string
- **opaque** - enabled values are true or false
- **size** - enabled values have format {width, height} (for example: 80, 51)
- **toolTipText** - any string
- **visible** - enabled values are true or false

In each element type you can use also attributes corresponding to the parent Swing component. So for *label* element you can use all attributes above and also attributes related specially to JLabel:

- **horizontalAlignment** - enabled values are constants from javax.swing.SwingConstants
- **icon** - enabled value is name of the icon file or resource in classpath
- **iconTextGap** - enabled values are integers
- **text** - any string
- **verticalAlignment** - enabled values are constants from javax.swing.SwingConstants

You can find a detailed description of all supported attributes for each supported tag / component in JavaDoc API for corresponding component builder (package eu.beesoft.gaia.

Special attributes

There are some special attributes processed in each element:

- **id** - unique identifier of the builder (or its object) in XML, it is used to reference the builder
- **class** - class of created object (instances of the same builder class can create different objects, but they should be derived one from other)
- **as** - a value of this attribute should be an identifier of the other builder (element); the created builder loads properties from referenced builder and then loads its own properties from XML

You need **id** attribute to reference corresponding component from other component. You can use any string as id.

Attribute **class** enables to create instance of other class than is supported by corresponding builder. Of course, given class should be a subclass of the class supported by the builder.

The attribute **as** is supported to minimize effort that is needed to create XML file. If you have element A, which has 5 of 6 attributes identical with element B, it is easy to write element A with attribute as="B" and with 6th attribute only.

2.2.3 Data binding

Data binding assumes that there is one root data object for displayed form. Another objects are referenced from it or from objects referenced by it. So the object hierarchy is requested in bound data.

The binding for builder (and its object / component) is recorded in source XML file with the attribute 'binding'. You can use also so called dot-convention: if your object references an object A via property a, the object A references an object B via property b, and you want to display property c of object B, you can write binding as *a.b.c* - it is easy.

The binding corresponds to the element hierarchy. If element A has set *binding="a"* and element A has sub-element B and this has set *binding="b"*, then the qualified binding in the element B is *a.b* and this is the path how is find data value from root data objet for a component created from the element B.

This algorithm is used to bind a specific data property to component created by the builder:

1. it is looking for getter / setter
2. tries a field access
3. if property container is an instance of eu.beesoft.gaia.util.ValueObject, it uses its get / set method
4. throws exception if property is not accessible

Of course, binding is working in two ways: you can initialize created components from data objects and you can also update data objects by components.

From a programmer point of view, only a few lines of code is necessary to implement data binding:

```
// prepare path to form XML
String formId = "mypackage/myform.form";

// prepare resource bundle name
String messagesId = "mypackage.myform";

// create new builder factory
SwingBuilderFactory factory = new SwingBuilderFactory ();
factory.setResourceBundle (messagesId);

// build component (form, menu, application, ...)
InputStream is = Streams.getInputStream (formId);
ComponentBuilder builder = (ComponentBuilder) factory.build (is);

// bind data object with Swing components
// note: root data object can reference other data objects
Object data = ...
builder.setBoundData (data, null);

// display form
Component component = builder.getObject();
myFrame.getContentPane().add (component);

// process user input
...
```

```
    // create collection to keep changed data objects
    // this collection will be filled in the next method
    Set changedObjects = new HashSet<Object> ();

    // get (modified) data from builders
    // returns root data object
    Object data = builder.getBoundData (changedObjects,null);
```

2.2.4 Internationalization

The internationalization in Gaia is supported in SwingBuilderFactory and in all SwingBuilder classes.

First you have to say to the SwingBuilderFactory where can it find ResourceBundle properties file to use:

```
    String messagesId = "mypackage.myform";
    factory.setResourceBundle (messagesId);
```

Each SwingBuilder overrides initId() method to trying initialize component from resource bundle. It takes given id, appends dot separator and name of the property and tries to load it from resource bundle. If found, invokes corresponding init...() method to initialize property.

So it is possible to write XML element:

```
    <label id="mylabel" />
```

and to create properties in resource bundle:

```
    mylabel.text=This is my label
    mylabel.icon=/dir/my.jpg
```

and your text, icon, etc. is initialized from resource bundle. In the implementation is used class eu.beesoft.gaia.util.Language to get correct values from resource bundle.

Please note, to use this feature each (localized) element must have an **id** attribute.

2.3 Client / server programming

Gaia supports client / server programming with Swing clients.

There are two parts of code on the client and server side to communicate between itself.

Client must be derived from class eu.beesoft.gaia.swing.client.SwingClient. This is an abstract implementation of the swing client for Gaia application (there is the ApplicationServer on the server side). This class has as high level of abstraction as possible: it processes the communication with a server, but has no limits on processing of the incoming instructions.

With this class you get an instrument to build your own client, with your own instruction set and your own Swing presentation.

In the simplest implementation of your subclass you have to:

- create a Swing application (instance of *JFrame* or *JDialog*) - the best place to do it is probably a constructor
- override getCurrentBuilder() method
- override getCurrentWindow() method
- override processApplicationResponse(ApplicationResponse, SwingClientAction) method - here you will process (your own) instructions returned from the server. Please, note, there is the method buildForm(ApplicationResponse, SwingBuilderFactory) prepared to help with this work.

Swing actions should be converted to the SwingClientAction, which forwards actionPerformed() method to the SwingClient.

On the server side there is the class eu.beesoft.gaia.swing.server.ApplicationServer prepared to communicate with a client. It creates the Application and Controller instances and forwards the flow control to them. You can read about programming on the server in the next chapters.

3 Application

Gaia brings its own application framework. It consists from some abstraction level and from a concrete Swing implementation.

In the current version is this framework usable for small applications with not too big load. It has not optimized the resource usage. We will do it in the next releases.

So this release is suitable for small client / server applications. It offers:

- the application server to which can your client (currently Swing client) connect
- class Application to support your appliciation needs
- class Context to hold user data between client requests
- abstract controller to process request, forward it to the next controller, return response
- class FormController to support form-oriented UI with automated data binding

The main advantage of this framework is its simplicity and built-in support for Swing based client / server solutions.

3.1 Application and Context

There are two base classes for the application framework:

- Application - binds the client and server code
- Context - this is a data holder

Application is an abstract superclass for each server application. It is created and managed by the running ApplicationContainer implementation (there is an implementation for Swing client called ApplicationServer).

The main task supported by the *Application* is to receive request from a client, find a controller to process it and to invoke that controller with request and current context as arguments.

Each instance of Application has its own *id*, which is used in talk with client. In your implementation you need override in you subclass only methods getStartControllerClassName() and getFile(String).

Context is a data holder for session and controller. Because all controllers are singletons, this is the only one place where can be stored data between the client requests.

ApplicationRequest is an application request from a client to the application server. It supports these properties:

- **sessionId** - an unique string identifier of the client / server session
- **action** - a name of the requested action
- **parameters** - a free (action-depended) map of parameters
- **data** - (changed) data from client to server

ApplicationResponse is an application response from an application server to a client. It supports these properties:

- **sessionId** - an unique string identifier of the client / server session
- **instruction** - an instruction for a client
- **formId** - a form identifier (a name of the form XML file to display)
- **data** - data from server to display on client
- **parameters** - a free (instruction-depended) map of parameters

Please, note there are some (by methods) supported parameters in Application Response to:

- hide component
- disable component
- transfer validation error

3.2 Controller

Application flow processing is in Gaia realized by different instances of Controller class.

Controller is an abstract superclass for all controllers. It supports the basic infrastructure for a controller lifecycle and communication.

Each controller is a singleton. The requested instance can ba accessed by the static method getController(String). Hence there is no space in the controller variables to store user data. For this purpose exists the Context class instances. You can imagine it as a map of {name:value} entries for your free usage.

Each client has just one server Application at the time and just one current Context instance.

When the controller is requested from the client, the Application finds (or creates) its singleton instance via method getController(String) method. Then *Application* invokes its method process(ApplicationRequest, Context) to process the client request. Controller prepares the server response (for example data to display in some form) and returns it as the return value from this method.

With the next request (for example, user pressed some button on the form), the Application invokes method forward(ApplicationRequest, Context) on it. This is the standard, unchangeable behavior. The runnning controller should process data from the client before it processes requested action (they are delivered together in one ApplicationRequest instance). The request can be in *forward()* method forwarded to the other controller, or can be processed in a local method - it is a matter of the implementation or configuration.

If the incoming request is to be forwarded to the other controller, the *forward()* method first invokes createContextForForward(ApplicationRequest, Controller, Context) to create a new context. Override it to initialize the new context with data from the current context. Then is invoked process(ApplicationRequest, Context) method on the requested controller.

There are two ways to return control to the calling controller:

1. invoke method returnSuccess(ApplicationResponse, Context) - this invokes method success(ApplicationResponse, Context) on the calling controller, which closes current context and prepares its parent context to be current and itself to be a running controller
2. invoke method returnFailure(ApplicationResponse, Context) - this invokes method failure(ApplicationResponse, Context) on the calling controller with the same effects as described above

In the *Controller* class is implementation of methods *success()* and *failure()* the same. But it will differ in subclasses. The first indicates that called controller does its work successfully and data are possibly changed. The calling controller should probably update its data. The second method should be used when something goes wrong (for example, an user pressed CANCEL button) and there is no change in data.

3.3 Form and data binding

Each client / server application must interract with a user. Gaia supports this requirement with the *FormController* class.

FormController is an abstract superclass for all form-based controllers. It supports data binding - it extracts from given data objects data requested by the form, prepares it for ApplicationResponse and also processes returned changed data from a client.

Form controller works with a form description in XML format. So you can use the same form XML file for the Swing client form and also on server side. FormController extracts *binding* attributes from that XML file and creates its internal bind-mapping. Then - when constructing response for client - it mines property values defined in binding from given data object and sends it to the client to show in UI.

There are some methods you have to implement in your subclass:

- getInstruction(Context) - to get an instruction for client (such as "display form", etc.)
- getFormId(Context) - to get a form id (form name)
- getObject(Context) - to get a processed data object
- getObjectsForListModel(String, Context) - to get the data objects for given list model

There is the method *displayForm(Context)* which creates completed application response with extracted and converted values from the context's data object and with the data for list models. This method invokes all of above methods.

The data extraction and data binding for displaying data on the client is processed in method buildFormObject(Object, String, Context) in these steps:

1. the form XML file is parsed and loaded to a FormDescriptor instance
2. it traverses its FormItem instances to find the *binding* property
3. it gets the requested bound property value from given data object
4. it converts this value (if necessary) and stores it to the internally created instance of ValueObject
5. if the bound property value is not a null, primitive value or enumeration constant, it takes this value as a new data object and repeats the step 2 for the children of currently processed instance of *FormItem*

If you want to use this built-in mechanism but you need to modify it, you can override method getMiner() to change the tool to access data properties or getPropertyValueFromDataObject(Object, String, Context) to do the same without changing the miner.

There is also the support to process the modified data from the client. It is started with method acceptFormObject(ApplicationRequest, Context) (you have to invoke it from your code yourself). It takes the server data object (via getObject(Context) method) and incoming data from the client request, walks in client form object(s) property by property and searches the appropriate property in data object(s). When it is found, invokes method acceptFormPropertyValue(Object, Object, Object, Object, String, Context) to set changed value to the data object. Override this last method to:

- ignore derived properties (this is a term from UML, such property has a getter only, but no setter or field)
- accept the changes in the references to the other objects, this is not supported and you have to do it in subclasses (for example, you can change on the client the reference to the other object in combo box, but in client request you obtain on the server just value object with (persistent) id of newly referenced object - to get it from database and set as reference to processed object is your job in overriden method acceptFormPropertyValue(Object, Object, Object, Object, String, Context).

4 Validation

Package *eu.beesoft.gaia.validation* offers a few classes to build your own validation system.

There is a class ValidationError which instance covers one error from validation.

All ValidationError instances from one validation process are held by the instance of ValidationResult class.

The abstract superclass for all validators is class Validator. This class is initialized with message that should be made public when validation fails. And in subclasses is necessary to process method *validate (Object).*

There is currently just one validator implementation - NotNullValidator - which reports validation error when validated value is null.

5 XML processing

Gaia supports XML processing in minimalistic but very efficient form. In this library you can find:

- static helper class to support XML DOM parsing and processing
- class to support XML SAX reading with hierarchical memory structures
- easy creating a XML file via writer

5.1 DOM support

XML DOM support in Gaia is implemented in the static utility class eu.beesoft.gaia.xml.Xml.

There is number of parse() methods implemented in this class to parse given input (file, input stream, reader, ...) to the Document instance. You need no DocumentBuilderFactory etc. to parse XML (it is invoked on background).

Some methods look like DOM4J methods to make programming easier:

- getElementIterator (Element) returns iterator of sub-elements
- getAttributes (Element) returns Map
- getAttributeValue (String, Element) returns attribute value from the element

and there is a number getAttributeValue() methods with default values.

5.2 Reader

Class eu.beesoft.gaia.xml.XmlReader is usable to read XML document in SAX style.

It suppresses the interactivity with W3C (SAX) structures and creates its own structure instead. This structure is called {@link XmlElement} and offers these advantages:

- each element is capable to return the reference to its parent element
- keeps its tag and attributes (as a simple map of strings)
- can hold an user object, which is any object you need to pair with this element

All of this is done to make the XML parsing and processing easier. You don't need to create your own structures for each application you write - the *XmlElement* meets your needs.

There are 3 methods you can / should override in your subclass:

- startElement(XmlElement)
- endElement(XmlElement)
- characters(XmlElement, char[], int, int)

Of course, you can override any of the methods of org.xml.sax.helpers.DefaultHandler, they are accessible, but you will not see it necessary, probably.

Let's look at some example. Here is a part of the XML file, with two nested elements and we want to load it and store data from XML to our instances of Person and City class (they can be some plain value objects):

```
<person name="Johny" >
        <city name="London" />
</person >
```

Here is a our subclass of XmlReader:

```
public class MyXmlProcessor extends XmlReader {

    private List<Person> persons = new ArrayList<Person> ();

    public List<Person> getPersons () {
        return persons;
    }

    public void startElement (XmlElement element) {
        String tag = element.getTag ();
        if ("person".equals (tag)) {
            Person p = new Person ();
            String name = element.getElementAttributes ().
                get ("name");
            p.setName (name);
            persons.add (p);
            // store new Person instance to element
            // as an user object
            element.setUserObject (p);
        }
        else if ("city".equals (tag)) {
            City c = new City ();
            String name = element.getElementAttributes ().
                get ("name");
            c.setName (name);

            // here you can see the usage of the user object:
```

```
                    Person p = (Person) element.getParentElement ().
                            getUserObject ();
                    p.setCity (c);
            }
    }

    public void endElement (XmlElement element) {
            // empty
    }

    public void characters (XmlElement element, char[] characters,
            int offset, int length) {
            // empty
    }
}
```

And here you can see its usage:

```
File xmlFile = new File (...);
MyXmlProcessor processor = new MyXmlProcessor ();
processor.read (file);
List<Person> persons = processor.getPersons ();
```

5.3 XML Writer

This class is designed to write XML files. The reserved XML characters are replaced by standard entities. Class supports the automatic elements nesting.

Usage:

```
            PrintWriter pw = ...
            XmlWriter writer = new XmlWriter (pw);
            writer.addElement ("first");
            writer.addAttribute ("value", 50);
            writer.addElement ("second");
            writer.addAttribute ("next", "www");
            writer.closeElement ("second");
            writer.closeElement ("first");
            writer.close ();
```

The code above builds this XML:

```
<?xml version="1.0" encoding="UTF-8" ?>
<first value="50">
        <second next="www" />
</first>
```

6 Logging

Our logging package was created long in the past to ensure the independence of our applications from logging subsystem. It is just a facade before real logging system.

Programmer will work with an implementation of the interface eu.beesoft.gaoa.log.Log.

There are four logging levels:

- DEBUG
- INFO
- WARNING
- ERROR

For each level exist these methods:

- isLevelEnabled()
- level(String)
- level(String, Throwable)
- level(String, Object...)

where 'level' is used for one of 'debug', 'info', 'warn', 'error'.

Please note the last method: you can use it to merge any number of parameters to logged message. In these methods is each character pair {} replaced by one of the given parameters. For example you can log:

```
info ("Current array index is {} of {}", 2, 5);
```

and logged message is *Current array index is 2 of 5*.

On the DEBUG level you can also use methods for logging of entering and exiting method.

The concrete implementation of Log interface can be obtained from LogFactory. It is an abstract superclass for all log factories. It serves as a factory of logger instances for different logging systems. Current factory can be obtained by LogFactory.getInstance() method. The typical usage scenario is like this:

```
LogFactory factory = LogFactory.getInstance ();
Log log = factory.getLog ("mypackage.MyClass");
if (log.isDebugEnabled ()) {
        log.debug ("This is logged message");
}
```

This class also offers methods setThreadBoundIdentifier(String) and getThreadBoundIdentifier() to create link between current thread and an identifier defined by programmer. You can use them to log records with user name, for example. This is useful in server applications, where is difficult to pair a thread to the active client.

To customize this class to the requested logging system, you have to override createLog(String) method for your needs and to subclass the AbstractLog class. There are prepared implementations for:

- Java (*java.util.logging*) system
- Log4J (*org.apache.log4j*) system
- Apache logging (*org.apache.commons.logging*) system

To use some of these logging systems you have to configure the underlaying real subsystem and create a corresponding factory by invoking its constructor (for example, for **Log4J** subsystem you need eu.beesoft.gaia.log.log4j.Log4jFactory instance). Then you can use the pattern from the introduction in your application.

7 Utilities

There is a set utility classes in Gaia library. They are implemented in eu.beesoft.gaia.util package:

AbstractBean

This class was developed to replace java.beans.PropertyChangeSupport class functionality. If you use it as parent class for your beans

- you don't need to write addPropertyChangeListener and removePropertyChangeListener methods or accessors to the PropertyChangeSupport
- your memory requirements are noticeably lower, especially if you have a lot of beans in memory

Language and LanguageListener

Encapsulates the work with resource bundles and language dependent texts. LanguageListener is the listener interface to receive language changed events.

Miner

Miner is a class dedicated to support the data binding. From external perspective it has two simple methods:

- getValue(propertyName, object) - gets value from the property of the object
- setValue(value, propertyName, object)} - sets value to the property of the object

Property name can be a simple property name, or you can use the dot-convention to chain objects referenced from properties. For example, if object A has property a that references object B, and object B has property b that references object C, and object C has property c you want to get, you can use as property name text **a.b.c** to get / set this property.

Miner prioritizes a method access (via getters / setters), but if the method for the property cannot be found, it uses the field access.

You can use this class "as is". But you have to subclass it if

- some property you request is virtual (it is not a name of property, it is derived)
- somewhere in the objects chain can be null object and you need to set value at the end of the chain - you need override the method createObject(String, Object)}

ObjectBuilder and ObjectBuilderFactory

This factory is designed to reading a XML stream of descriptors and to create an appropriate ObjectBuilder instance for each parsed element. Then it manages initializing of builders and creating and initializing objects in builders. There is one or collection of objects created and initialized at the end of this process.

This implementation is abstract enough to let you free to design the form of XML (tags for elements, attributes, ...) and to program your own ObjectBuilder implementations. There is one implementation for building Swing components from XML file in package *eu.beesoft.gaia.swing.builder*.

Regardless of the high abstract level of the basic implementation, there are a few XML element attributes supported by *ObjectBuilderFactory* and *ObjectBuilder*:

- **id** - unique identifier of the builder (or its object) in XML, it is used to reference the builder
- **class** - class of created object (instances of the same builder class can create different objects, but they should be derived one from other)
- **as** - a value of this attribute should be an identifier of the other builder (element); the created builder loads properties from referenced builder and then loads its own properties from XML

ObjectBuilder class is an abstract superclass of all builders. A builder serves as a factory for the creation and initialization of the created object. The building process is managed by ObjectBuilderFactory. This implementation does not dictate relationships between builder, created object and XML element - it is your responsibility.

Each builder is managed by ObjectBuilderFactory instance that created it. *ObjectBuilderFactory* loads a XML description and for each element creates a new appropriate builder. This is done in these steps:

1. this builder's instance is created
2. references to instance of *ObjectBuilderFactory* and builder's parent are stored to this builder
3. attributes from XML element are stored to this builder as properties
4. builder invokes method createObject() or createObject(String), if the *class* property was found

When *ObjectBuilderFactory* processed all XML elements and for each executed steps described above, it walks each builder again and invokes method initObjectProperties() on it. Builder for each property invokes method initObjectProperty(String, String). This method in current implementation searches for initialization method for given property and invokes it.

When all builders were requested to initiaze their properties, the *ObjectBuilderFactory* notifies each builder about its hierarchy position. On each builder with children is invoked method addChild(ObjectBuilder). This is done from bottom to top in the builder hierarchy, so parent is notified about its children when each its child was notified about its children.

After that is builder and its object fully initialized and object can be used in application.

Reflection

Utility class for Java reflection. Implements a set of the methods to get field or method from the class (and its superclasses), get or set value to a field, invoke methods, etc.

Streams

Utility class for the stream I/O operations. Returns an input stream or reader for given name, it is looking for it in classpath or in filesystem, closes streams.

Exceptions are wrapped by RuntimException, so there is no need for try-catch blocks in your code.

This is an example to copy file with this class:

```
InputStream is = Streams.getInputStream ("..."); // resource name
OutputStream os = Streams.getOutputStream ("..."); // target name
Streams.copy (is, os);
Streams.close(os);
Streams.close(is);
```

ValueObject

General value object that holds its values as properties in the internal map. This object has no getters / setters for individual properties, there is one getProperty(String)} method and one setProperty(String, Object)} method. The *setProperty()* method generates property change event (in parent class AbstractBean).

SystemProcess

It encapsulates functionality of the java.lang.Process, java.lang.ProcessBuilder and some methods from java.lang.Runtime classes. It supports to:

- create a new system process
- complete command and or arguments in more steps
- change system environment variables
- change working directory
- merge input and error stream to the one
- execute process and wait while it is not finished
- obtain process exit value
- obtain input, error and output stream for the process

There is also a set of the static `exec()` methods to simplify starting of the system process.

This implementation solves the problem of the Process.waitFor() method, which hangs when are not read the process streams.

8 Launcher

Launcher is an utility subsystem to launch Java application. It supports processing of the instruction file, building a splash screen, building the class paths and invoking *main* method on another class.

The main advantage of this launcher is the ability to run some parts of application in sequences and in different classloaders - so it is easy to process the application autoupdate and then start the application within this launcher, all covered by one splash screen.

You can use this subsystem as-is (just one JAR file), or you can subclass some classes from it and create a new JAR from Launcher classes and your subclasses to launch your application.

The main class of subsystem is *eu.beesoft.gaia.launcher.Launcher*. Launcher processes a *launch* file. It is a file with the instructions. Ih an instruction has arguments, they are appended at the same line and separated by space from instruction name. Enabled instructions are:

- **property** - adds property (given as an argument of this method, in form name=value) to system properties
- **splash** - builds a progress mediator of SplashScreen. An image name is an argument of the call.
- **mediator** - builds a progress mediator (its class is an argument of the call)
- **path** - adds given argument to classpath
- **scan** - adds to classpath each JAR file from directory (given as argument) and all its subdirectories
- **run** - invokes *main()* method on instance of the class, which name and arguments for invocation are in given argument
- **clear** - clears built class paths - used to create a new classpath

Here is an example of the standard launch file:

```
path c:/myapp/lib/first.jar
path c:/myapp/lib/second.jar
path c:/myapp/classes
run mypackage.MyClass firstArgument "second argument"
```

It creates a classpath from two JAR files and one directory and then invokes method main() on MyClass with two arguments.

Launcher can mediate its state by *ProgressMediator* implementation. ProgressMediator is an interface that describes a behavior of the UI component, which displays a progress of some long-term operation. The described methods are:

- getProgressRange() - returns maximum progress value
- setProgressRange() - sets maximum progress value
- getProgressStep() - returns a current progress step
- setProgressStep() - sets current progress step
- getProgressText() - returns the currently displayed progress text
- setProgressText() - returns true if user (in UI) interrupted operation for which is this progress monitor running
- isInterrupted () - closes a progress mediator instance
- close () - sets the progress text to dislay

There are two implementations of ProgressMediator:

- ConsoleOutput - to output to the console (Unix-like without installed GUI). It does not serve a progress steps, just displays texts.
- SplashScreen - a basic implementation of splash screen with ProgressMediator. Creates a main window, displays an image and a progress bar. You can override method build() to change the look of this spash screen.