XML4SWING

William R (http://wrey.free.fr/)
SOURCEFORGE.ORG

April 14, 2011

Contents

Chapter 1

Welcome

1.1 Abstract

XML4SWING is a tool to create swing templates based on a XML file. The developer provides a XML file and the software will create a source code (not a compiled class) with all the stuff necessary to have a graphic user interface in Swing.

The main difference with other tools proposed are:

- Provides a JAVA source code.
- Integrates "out-of-the-box" features (look and feel menu, etc.)
- Some helpful way to manage the AWT-Thread.
- Components organized on a HTML `<table>` style (alignments, grouping, etc.).
- Native SWING components extensions are possible.

1.2 Using native interface

Now, there are many providers giving you good solutions to create your own GUI. The idea behind the XML4SWING is to give a way for writing simple graphic interfaces very fast. Not only the display itself but also programming a SWING application.

The program will help to create plain JAVA source code you have to extend with your own class. You should NEVER use directly the code generated by the program. Creating an GUI using the swing library is a natural choice for creating basic things. If you want beautiful and more than modern graphical interfaces, use JavaFX instead (but be careful, it is not part of the OpenJDK project and subject to licences). We just provide a basic software with an open source code.

You simply have to provide all the things to put together in your GUI. Only give names to components you want to control (as a `JTextField`). Menus can be generated quickly (and XML is the best way to describe them): for each menu, you simply give the method for processing the code behind.

About the licence: you can use this software for any purpose. The generated classes can be distributed in binary mode or with the source code included. When your development provides the `.java` code source, it must be also provide the `.xml` file used to generate it (if possible in the same directory or in the resource directory if you use Maven). This is typically non sense to distribute the source code generated by this software but without the original XML file.

Note some of the functionalities described in this document may be not yet implemented. Send me a request directly if needed.

1.3 How it works

You write an XML file containing the GUI interface you want. Normally, only the graphical part is expressed in this file. Once the XML file is ready, you run the software and it will generate a JAVA code source in the right directory (you can specify the directory in arguments). The class created includes a `main()` method to test the graphical interface but you have to create your own class based on the generated one to add the logic.

The generated class includes:

- A `main()` method for test purposes.
- A `initComponents()` method for initializing the graphical objects.

That's quite simple. Do not try to understand the generated code, I can just say: "It works" but the swing implementation is a confusion addition of multiple things including menus, components, layouts and some other stuff.

1.4 My first application

1.4.1 Create the frame

Usually, all the GUI applications run into a `JFrame` object. It is the base of our XML file. Write the code of the figure 1.1. It will create a very simple frame but with all the stuff needed.

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<JFrame
  title="First frame"
  name="com.oxande.xmlswing.example.AbstractSampleFrame"
  coolsize="300,150"
  icon="./forward16.gif">
</JFrame>
```

Figure 1.1: The minimal application

Of course, in respect of object abstracting, you should not use the frame like this. Please create the following code to support the frame.

```

package com.oxande.xmlswing.example;

public class SampleFrame extends AbstractSampleFrame {
}
```

Figure 1.2: The minimal implementation

If you use the same package than the frame object, it is quite easy to support the graphical part. Now, you have to configure the XML4SWING program to generate the

code. Then you have to run it as follow (we consider you are in the root of your Maven workspace, where is stored the `pom.xml` file):

```
java -cp xml4swing.jar
     -d .
     ./src/main/resources/com/oxande/xmlswing/example/tutorial.xml
```

Then to run the sample created, please run your code as follow (do not forget to add the root of your classes to your class path):

```
java -cp ./target/classes
     com.oxande.xmlswing.example.example.SampleFrame
```

And, the first application is created and display something similar to what you have in figure 1.3. It is the first but quickly (I suppose) created application.

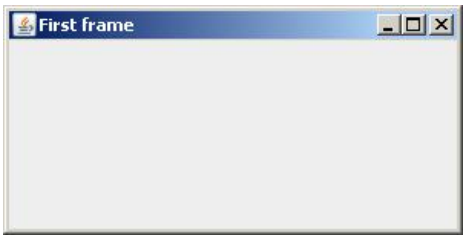

Figure 1.3: My first Application

Then, of course, it is very simple. Notice we didn't provide the `main` method for the application as we use the inherited one. It is a good idea to create its own (figure 1.4). You should first instantiate the class. Then you initialize the components. In the last part, you set the frame to the visible mode.

```
public class SampleFrame extends AbstractSampleFrame {
   public static void main(String[] args)
   {
4     SampleFrame appl = new SampleFrame();
      appl.initComponents();
      appl.setVisible(true);
   }
}
```

Figure 1.4: The main

Be careful, if you do this, when you close the window, the application remains active. One way is to set the default close mode to `EXIT_ON_CLOSE`. But you have not to add new code: simply add a tag to the XML file for the *onClose* event. This event is trigerred when the window is closed (i.e. the frame). See figure 1.5 for the code added. It is quite simple, no? Now, your application closes.

Another way to do is to use an attribute rather than the code directly in the XML file. In this case, simply use the attribute `onClose="exit"` to give a method rather than the code. You just have to implement this functionality by providing a method as shown in figure 1.6. Do not forget to provide the method, if not the program will provide a default one which throws an `UnsupportedOperationException`.

```
<JFrame ...>
  <onClose>
     System.exit(0);
  </onClose>
</JFrame>
```

Figure 1.5: The minimal application

```
public void exit(){
  System.exit(0);
}
```

Figure 1.6: The exit method

Now, you want to add a menu. Very simple, provide one as cascaded menus in the XML file. Of course, we provide not implemented ones. Each time, we provide a menu, the system provide a default implementation showing a message saying it is not implemented.

```
<JFrame ...>
        <menubar>
    <menu>
      _File
                        <item action="openFile">_Open File</item>
                        <item>Close Fil_e</item>
                        <separator />
                        <item perform="exit">Exit</item>
                </menu>
                <menu>
                        Help
                        <lookandfeel />
    </menu>
  </menubar>
</JFrame>
```

Figure 1.7: The menu bar

The menu bar is simple. If you add `actions`, you will have something behing the menu. Note the "_" (underscore) sign to specify what letter will be underlined when the user clicks on Alt button (for keyboard access). You should pay attention to `<separator/>` to simply create separators and `<lookandfeel/>` to have a complete menu to change the look and feel of the application (evrything imported).

Note the `action` attribute which points to the same `exit()` method when the user close the window or select the *Exit* menu item.

1.4.2 Add a status bar

But, it is not the only the remarquable part. Now simply add the tag `<statusbar id="statusBar" property="statusMessage" />` and you have a status bar. Simply use the bean property to set the message. You can write in your code `setStatusMessage(msg)` to change the status message. More than this, the change of the text is made in the AWT-Thread because the text is changed by calling the correct JAVA code to avoid an issue if you try to change the text outside the

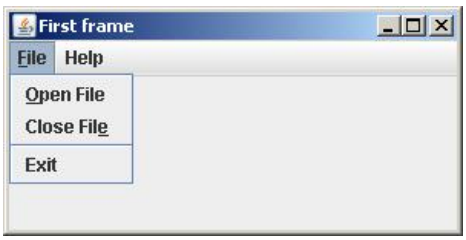

Figure 1.8: My first menu

AWT thread.

That's a king of magic. And the same is valid for the text objects (`JTextArea`, `JTextField`, etc.) where the fact to set the property permit to set it easily. But, please, don't look at the code behind, it is a mess...

1.4.3 Add a toolbar

As simple as for status bar, use the tag `<toolbar>` to add a toolbar (see section 2.6.5 for details).

1.5 Arguments

As for other JAVA programs, you must call the software using the jar file (`xml4swing.jar`) provided as follow: `java -cp xml4swing.jar` or add the jar file in the classpath variable.

The complete command line becomes: `java -cp xml4swing.jar <arguments> <templates>` where `<arguments>` are the arguments shown below and `<templates>` the XML files which describe the different classes to create. Using - (the minus sign) as `<templates>` will use the standard input as XML file.

Please find below the arguments you can pass to XML4SWING:

- `-d <dir>`: the directory for sources. It is the base of all your classes (basically `${project_loc}/src/main/java` if you use Maven.
- `-V`: include the version. If set, a Javadoc tag `@version` followed by the current date and time is added in the source code of the class (do not use if you already have a version control program as CVS or Subversion).

Chapter 2

Inside

2.1 The supported objects

2.1.1 Component

The root of all the supported graphical objects. Do not confuse with the `JComponent` objects (which are the root components for swing objects). There is no dedicated tag for an object: you should always used a defined object.

Nevertheless, all the tags inherits of the following basic attributes:

- **`minsize`** the minimum size of the component expressed as a dimension (e.g. `minsize="50,70"`). Not mandatory.
- **`maxsize`** the minimum size of the component expressed as a dimension (e.g. `minsize="100,130"`). Not mandatory.
- **`coolsize`** the preferred size of the component expressed as a dimension (e.g. `minsize="80,90"`). It refers to the `setPreferredSize()` method. Not mandatory.
- **`size`** the size of the component expressed as a dimension (e.g. `minsize="80,90"`). It refers to the `setSize()` method. Not mandatory.
- **`visible`** you can hide the object by setting the value to `false`. Defaulted.
- **`cursor`** set the cusor type. Can be one of the predefined cursor (see section 2.5.1). Not mandatory

2.1.2 Containers

A container is (as its name says) a component created to store graphical components (preferably `JComponents`). A container should have a layout and you can add several components in it. On the other side, a graphical component is always a container! In fact, I am not sure about the real use of the container as XML4SWING use them internally but can not be created by the XML file itself then you should nevr care about them.

2.1.3 Components

A very generic class for all swing components (the famous `JComponent`, the father of all the swing components). Inherits of the `Container` (see section 2.1.2).

As for the `Component`, it is not possible to instantiate a `JComponent`, you must use a more concrete objet. But a `JComponent` accepts the following attributes inherited by all the other classes:

- **`autoscrolls`** can be set to `true` to have automatic scrollbars if needed. Note this property can conflict with the `hscroll` and `vscroll` of the `JScrollPane` transparent implementation.
- **`background`** set the background color. Any color is accepted.
- **`foreground`** set the foreground color. Any color is accepted.
- **`doubleBuffered`** is a boolean value to set a double buffering to the component. Should never set explicitly.
- **`enabled`** is a boolean value to enable (`true`) or disable (`false`) a component.
- **`opaque`** is a boolean value to call the method `setOpaque()` with the correct value. Should be never used.
- **`opaque`** is a boolean value to call the method `tooltip()` all the components coming from swing can have a tooltip. Simply set this attribute with a text for any component to have a tooltip displayed when the mouse moves under the component.

2.2 Buttons

Buttons are used widely in GUI interfaces. The buttons are also used to define menu items.

2.2.1 AbstractButton

Inherits of `JComponent` (see section 2.1.3). This component is used for generic buttons including menu items (see section 2.6.1). If you set the attribute **`property`**, you will be able to know if the button is selected or not (see figure 2.1). Note the `AbstractButton` is not linked to a concrete object but the properties are used by the <button>, <radiobutton> and <checkbox> tags.

```
public void setDebugMode( boolean debugMode ){
   button1.setSelected( debugMode );
}

public boolean getDebugMode(){
   return button1.getSelected();
}
```

Figure 2.1: Getter/setter for booleans

Abstract buttons have the following attributes:

- **`selected`** a boolean value specifying if the button is selected or not. useful for toggle buttons like radio buttons and check boxes.
- **`borderPainted`** linked to the `setBorderPainted()` method. Can be set to `true` or `false`.
- **`valign`** set the vertical alignment of the button. Can be `center` (the default), `top` or `bottom`.
- **`align`** set the horizontal alignment of the button. Can be `right` (the default), `left`, `center`, `leading` or `trailing`.

- **vtextalign** set the vertical alignment of the text.
- **htextalign** set the horizontal alignment of the text.
- **gap** set the icon gap. This is the number of pixels between the icon and the text.
- **multiClickThreshold** value for modifying the threshold between 2 cliks of the mouse. This value should be not modified as it is linked to the normal behaviour of the GUI interface.
- **rollover** set to `true` to activate the rollover.

2.2.2 JButton

A tag <button> represents a normal button a user can click (a `JButton`). The class inherits of the `AbstractButton` described above.

The components has the following attributes:

- **default** set to `true` to set the button as the default one. When the user press the enter key, the default button is selected.
- **toggle** set to `true`, the button is a toggle button (`JToogleButton`) rather than an ordinary one[1].

You must provide an attribute **text** or the text directly in the tag to set the text of the button. You can include directly the mnemonic as specified in section 2.5.8.

A toggle button supports the **property** attribute to set and get directly the selected attribute of the button.

2.2.3 Check boxes

Check boxes are buttons which can be selected or not. They inherits of the properties of a `JButton`. The tag <checkbox> must be used.

You can also use the <input> tag with the attribute **type** equals to `checkbox` (this ensure a compatibility with HTML).

2.2.4 Radio buttons

Radio buttons are buttons which can be selected or not. They inherits of the properties of a `JButton`.

The tag <radiobutton> must be used. Note you can also use the synomnym <radio> for the radio buttons. As for check boxes, you can use the <input> tag with the attribute **type** equals to `radio` (this ensure a compatibility with HTML).

Radio buttons should be put in a group because the selection of a radio button unselect the other radio buttons of the same group. You just have to add the **group** attribute with a group name to include the radio button in a group.

2.3 Text components

Inherits of `JComponent` (see section 2.1.3).

If you declare the attribute `property`, you can use the getter and setter property to set or get the text in the component. This is a helper and a good solution to hide the object itself (an anonymous object is de facto declared as *private*). If you declare the property as `zipCode`, you will have 2 methods as you can see in figure 2.2.

[1] It is an hack in the XML notation as the tags are not bivalent for the other swing objects.

```
public void setZipCode( String zipCode ){
   text1.setText( zipCode );
}

public String getZipCode(){
   return text1.getText().trim();
}
```

Figure 2.2: Getter and setter

Please add a "%" (percentage sign) after the variable name to have a getter and a setter using an `int` type (rather than a `String`).

2.3.1 Password fields

The password field inherits of the `JTextField` (see section 2.3.3).

It has the following attributes:

- `echo` the echo character to be displayed instead of the characters. It is a single character.

Note: the `property` attribute can be used and you will be able to get or to set the password as a normal text. If you use the component directly, you must use the `getPassword()()` method to get the password. See the javadocs for more information.

2.3.2 Text areas

The <textarea> tag is declared using the `JtextArea` class and inherits the `JTextComponent` (see section 2.3).

A text area is an area to input several lines of plain text. Supports the **`hscroll`** and **`vscroll`** attributes.

2.3.3 Text fields

The text field is declared using the `JtextField` class and inherits the `JTextComponent` (see section 2.3).

2.3.4 JSlider

The <slider> implements the `JSlider` class inherited from the `JComponent` class.

The attributes are:

- **`property`** to have a getter and a setter for the value. The methods provide an `int` property. It is useful to get and set the value of the slider without manipulating it directly.
- **`orientation`** can be `horizonal` or `vertical`.
- **`minimum`** is the minimum value for the slider.
- **`maximum`** is the maximum value for the slider.
- **`paintLabels`** determines whether labels are painted on the slider (boolean).
- **`paintTicks`** determines whether tick marks are painted on the slider (boolean).
- **`paintTrack`** determines whether track is painted on the slider (boolean).

- **snapToTicks** Specifying `true` makes the knob (and the data value it represents) resolve to the closest tick mark next to where the user positioned the knob.
- **majorTickSpacing** sets the major tick spacing (integer).
- **minorTickSpacing** sets the minor tick spacing (integer).
- **extend** sets the size of the range "covered" by the knob. Most look and feel implementations will change the value by this amount if the user clicks on either side of the knob. This method just forwards the new extent value to the model.

2.4 Java tables

The `JTable` class is useful to display your data. You can display your table with the tag <JTable> (do not confuse with the tag <table> used for formatting the layouts).

Inside the tag, the attribute **model** gives you the opportunity to provide your own table class (must inherit of the `TableModel` interface). You should give the name of the model: you have to provide it through a initialization *before* initializing the GUI.

If you want to do some simple initialization without creating your own table model, you can give the <tr>, <th> and <td> tags as you should do for a HTML table. Note there is no alignment or other stuff: the data is displayed as simple text in the table. This method is an helper to test your template and not intended to be used in a production environment.

The figure 2.3 shows an example to display a table table containing two files with some attributes.

```
<JTable>
  <tr>
    <th>File Name</th>
    <th>Size</th>
    <th>Creation Date</th>
    <th>Modification Date</th>
  <tr>
  <tr>
    <td>xml4swing.properties</td>
    <td>1,5K</td>
    <td>18 Oct 2010</td>
    <td>7 Nov 2010</td>
  <tr>
  <tr>
    <td>Othello (Orson Wells)</td>
    <td>700Mo</td>
    <td>13 Oct 2003</td>
    <td>8 Jun 2005</td>
  <tr>
</JTable>
```

Figure 2.3: Example of JTable

2.4.1 Applets

An applet (<applet>) inherits of the `Container` class (section 2.1.2).

An applet can contain a menu bar (as for a frame) and a main component which is the *content pane* as defined by the JAVA API (see method `setContentPane()`). Note a `JFrame` is a valid container for an applet.

This component is very simple as it supports only the `status` attribute enabling to give the status of the applet. If you need an applet with a complete support, you can add a `JFrame` (tag <frame>) as the main component of the applet.

2.5 Special attributes

2.5.1 Cursors

The attribute **`curosr`** available for all components (including those from the `java.awt.*` package) can have the following values:

- `crosshair` CROSSHAIR_CURSOR
- `default` DEFAULT_CURSOR
- `hand` HAND_CURSOR
- `move` MOVE_CURSOR
- `text` TEXT_CURSOR
- `wait` WAIT_CURSOR

2.5.2 Icons

Some graphical components (especially buttons and menu items) accept icons. Currently, an icon can be declared as a classic attribute. The icon is an URL. You should always gives a icon which is inside your code or available at a WEB address. You can not specifiy a file on the disk. To do so, you must do this manually though you own class.

If you specify an property value containing a "**`:`**" (semi colon), you will load the icon directly. If you specifiy a path, the system will use the `getResource()` method to load the icon.

2.5.3 Names and References

When you provide a **`id`** attribute to a component, the component becames visible for the class which extends this one. This is useful to access the object and to manipulate it (buttons, menus, etc.) even if it is not nessary in many cases if you do not change the normal behaviour.

Another interesting property is the **`reference`** one. If a object has been previously declared in the XML, you can resuse it by giving only the **`reference`** property. All other properties will be ignored.

2.5.4 Color attributes

Several attributes accepts to receive colors. The colors can be given in two different ways.

The first easiest way is to give the RGB values in a hexadecimal form prefixed by the sign "`#`". This is strictly the same notation than for HTML and CSS color parameters. For example, the color red will be "`#ff0000`", the blue is "`#0000ff`" and so on.

The second method works fine with several basic colors like white, red, green, cyan, etc. You simply write the color you want to use (it is not case sensitive). The color set you can give is the color set available in JAVA (red, blue, cyan, red, orange, yellow, gray, black and white).

2.5.5 Insets attributes

Insets are used to give some margins. The term *insets* refers to the `Insets` class in JAVA. Basically, it is simply four values to give the space (expressed in number of pixels) on the top border, the left border, the right border and the bottom border.

You have to give the 4 values separated by commas (as for dimensions). For example: "`2,6,8,3`" will give you 2 pixels on top, 6 on the left, 8 on the bottom and 3 on the right.

To simplify, if you give only one value, this value applies to the 4 parts (top, right, bottom and left). This is the easiest way to give the margins. If you give 2 values, the first is affected to the top and the bottom and the second one to the left and the right.

2.5.6 Property attribute

To set the value of a field (text field), you can use the **`property`** attribute. This avoid to use a object identifier and to use the `setText()`/`getText()` methods directly. Rather than this, use the property attribute to define a getter and a setter you can use as bean property.

The value is replicated in the object. But note the setter is used in conjunction with `SwingUtilities.invokeLater()` to ensure the system is not corrupted because you try to set the value from a thread which is not the EDT thread (dedicated to swing management). Then the property is set *after*. If you do try to get the value just after set it, you can have the old value (as the getter does not rely on the EDT thread management).

This is a good thing if you modify the text of the status bar through the property rather than directly. In addition, you can expect better performances if you often modify the value.

2.5.7 Scrollable components

Some components supports the `Scrollable` interface (`JEditorPane`, `JFormattedTextField`, `JList`, `JPasswordField`, `JTable`, `JTextArea`, `JTextComponent`, `JTextField`, `JTextPane`, and `JTree`). In this case you put the attributes `vscroll` and `hscroll` to put the component in an anonymous `JScrollPane`.

The attributes can have the following values:

- `always` the scrollbar is always displayed.
- `never` the scrollbar is never displayed.
- `auto` the scrollbar is displayed only when needed.

The component is automatically encapsulated in a `JScrollPane` if you give one of the property.

2.5.8 Text

Normally the text of a button (including menu items) can have a mnemonic. The text can be specified as an attribute `text` or as a text contents for the tag itself. It means `<button text="OK" />` is equivalent to `<button>OK</button>`. If an attribute is given, it has a priority to the contents of the tag.

You can specifiy a mnemonic for the text. It is very easy by preceding the mnemonic by an underscore. For example `<button>_OK</button>` will permit to the user to type `ALT` + `O` to do the same than clicking on the button.

Note the mnemonic is not case sensitive as we provide a key event rather than a character. Only characters and digits should be used as mnemonics, other characters will be ignored.

NOTES:

- The "_" (underscore character) has been choosen rather than the "`&`" (ampersand) because it is a special XML entity and needs to be escaped which is less easier when the XML is written without a dedicated editor[2].
- Do not mix accelerators and mnemonics. A mnemonic can help to access the menu by typing the character when the menu is open. An accelerator is a combination of keys which gives the opportunity to have a direct access to the menu even if the menu is closed.
- The text is trimmed: the spaces at the beginning and the end of the text is deleted *except* if given as an attribute.

2.6 Menus

Each item of a menu (<item>, <checkboxitem> or <radioitem> can have a **`action`** task. The perform task should be mandatory (at least for <item>s): ff not present, a `UsupportedOperationException` is thrown when the user clicks the menu.

If **`action`** is given as an attribute, it calls the specified method: the method is declared as a *protected* one and inherits the code specified as "Not implemented". This is necessary to keep the class independant for testing its behaviour (see section 2.9).

The specifications request a `ActionListener` class but the software will provide the encapsulation and will call directly your method having no return and no input parameter.

If you provide the action as a tag (<action>), the text inside the tag is considered as JAVA code and will be implemented. This option is permitted to put simple logic directly part of the GUI. That is a good solution to add a program exit or some similar code. One limitation is you can not use instance variables (as you can not provide any veriable in the GUI class).

2.6.1 Menu items

The tags <item>s gives you the capacity to add actions in your menu. Menu items are based on abstract buttons (section 2.2.1). When you define a menu item, it creates an associated `Action` interface: this will help you for creating the toolbars for your application.

Note the text of the item can be provided as text in the tag itself or through the **`text`** attribute. The text can include the mnemonic information.

2.6.2 Check box items

Very useful to set or unset an option. Same as menu <item>s but do not generate a "Not implemented" message when clicked.

[2]The ampersand character is used under the Windows SDK starting the beginning to create a menu keyboard short cut in the resources file

2.6.3 Radio buttons items

Very useful to set or unset an option in a group. Same as menu <item>s but do not generate a "Not implemented" message when clicked. You should provide the **`group`** attribute to group in the same group all the buttons.

2.6.4 Look & Feel

I don't think it is the best idea, but you can push in any menu the list of the available *Look and Feels* available in the system by adding the tag <lookandfeel>. The entries are automatically created and the selection is triggerred.

Note this should be done in the menu of the main frame to authorize the system to refresh the GUI display. By default, selecting a Look and Feel option will automatically updates the GUI by refreshing the frame container (found by calling the `getRootPane()` method). You can override this behaviour by giving the object to refresh in the **`refresh`** attribute.

For you information, the look and feel is, by default, the "Metal" one. But, you can find the L&F of your platform and update the GUI accordingly. In this case, you must do this at the beginning of your program because the L&F must not be changed programmatically after the `init()` method of the frame has been called.

```
public static void main( String[] args ){
   String name = UIManager.getSystemLookAndFeelClassName();
   UIManager.setLookAndFeel( name );
   // Put your code here...
}
```

Figure 2.4: How to configure the Look and feel.

2.6.5 Tool bars

You can define tool bars. Toolbars are made of `JComponents`. Then you can encapsulate *any* components. In addition, you can add separator (same as for menus).

You can also add <action> tags. An action tag is a simple way to say the button in the toolbar is the same than the menu item (or any abstract button) existing in a menu. Then you just add the reference to your menu item (of course, you must give a name to your menu item first). In this case, the toolbar inherits of the action linked to the menu item.

When you use actions (the prefered method), it is better to associate an icon to your menu item.

In addition, the «toolbar» tag has the following attributes:

- `floatable` boolean property to set to true for the user to move the tool bar.
- `orientation` orientation for the toolbar. can be "horizontal" or "vertical".
- `rollover` sets the rollover state of this toolbar. If the rollover state is true then the border of the toolbar buttons will be drawn only when the mouse pointer hovers over them. The default value of this property is false.
- `margins` set the margins. Must be 4 numbers seperated by commas.
- `borderPainted` set the property to true if the border should be painted. The default value for this property is true.

2.7 Panels and Layouts

The main objective of this project is to remove the layout implementation. This is why you don't hav to bother with the layouts. In the generated code, you will find only 2 layout: the `FlowLayout` used to add components in a simple way and the `GridBagLayout`.

2.7.1 Table layout

The table layout is based on the `GridBagLayout`. It is an easier (and more HTML friendly) way to declare forms. You can simply use the <table> for the table, <tr> for each row and <td> for the cells.

As for HTML tag, the <td> can accepts the `rowspan` and `colspan` attributes to span a cell on mutiple rows or cells. You can use the `align` attribute to align the component (`left`, `right` or `center`). For vertical alignment, use the `valign` attribute (possible values are: `top`, `bottom` or `middle`). You can put one component only per cell (but can be a panel).

Note you can give a `height` and `width` parameter for cell and this value will be transmitted to the component and will change its preferred size accordingly (TO BE IMPLEMENTED).

2.7.2 Splits

In order to create a splitted view, you can use a <split> tag. This tag can split horizontally or vertically depending of your needs. You can put in it any component (including container).

```
<split orientation="horizontal">
   <pane>...<panel>
   <pane>...<panel>
<split>
```

Figure 2.5: Splitting panels

2.8 High Level objects

High level objects are object you can create. Currently, you can creates `JFrame`s and `JDialog`s. Also, menus can be create to help the context menus to enhance your application.

2.8.1 Frames

A frame can be created with the <JFrame> tag. Inside a `Jframe`, the primary layout is always a `BorderLayout` where the top part is taken by the toolbar if you declare one[3]. The bottom part is for the status bar (declared with the <statusbar> tag).

The elements of your frame goes directly in the center of the panel (the content panel). This behaviour sinplifies the development of your interface. Of cours, you can create your own panels if needed.

[3] depending of your toolbar placement, the toolbar can go to the left or right side

2.8.2 Dialog boxes

Not yet implemented.

2.9 Unit testing

In addition, the class generated includes a `main` method to give you the opportunity to test the generated code. The idea behind the test is to open the frame (or the dialog) and to see the results.

The unit testing of a generated class disable the capability to the class to be declared as an *abstract* class. This is why all the methods are declared *protected* and provides a minimal behaviour: they throw a `UnsupportedOperationException`. The developer is responsible to provide the correct code in the extended class.

Chapter 3

Examples

3.1 File Explorer

You know this under the name of *Finder* if you are familiar with the Apple environment. I named the program "File Explorer" as I coded under a Windows box. Nevertheless, the code is not very complex.

The code is based on a simple frame. This frame contains 2 things: a tree (`JTree`) on the left to display the directory tree and a plain list (`JList`) for the list of the files found in the directory. The program contains about 200 lines of code for the mechanism. You can add the same amount of code for the GUI but generated automatically.

The main difficulty is to drive the tree: each time the user clicks on a directory, this directory and its content is loaded in memory[1]. This part is the tricky part of the code.

When you create the `Explorer`, you can use directly the `pathTree` instance created by XML4SWING and `fileList` also created automatically. Then, you have just to implement the logic.

As you can see, the simple «statusbar» creates a status bar you can access by its property name. The border is created to enhance the visibility of the status bar. Also the font use is always a *plain* one including when the default look and feel is selected (the behaviour is to use a bold font for labels). Use the **`property`** method to set the text of the status bar rather than relying directly to the button because of the use of `SwingUtilities.invokeLater()` called at this time rather than updating immediatly (if you come from a different thread than the EDT one, you could have some issues).

One step is the creation and the management of buttons on the toolbar. If you simply need actions also available in the menus, it is much more simple and no code has to be written (except the XML declarations).

[1] once the directory loaded, changes on the disk are not visible on the screen, this is a demonstration, not a real program.

Chapter 4

Links

Here some usefule links about the swing package and other stuff related to graphic user interfaces.

- http://mbaron.ftp-developpez.com/javase/javavisu.pdf a document about the classes `JTable` and `JTree`. Includes also a tutorial on the `JGraph` class.
- http://gfx.developpez.com/tutoriel/java/swing/swing-threading/ a cool document (in french).
- http://java.sun.com/products/jlf/ed2/book/ A tutorial about how to design dialogs and frames.
- http://download.oracle.com/javase/tutorial/uiswing/layout/visual.html It is a good place to learn about the layouts (even if this XML format has been conceived to avoid using the layouts directly).

Here some concurrent projects:

- http://sourceforge.net/projects/jgb/ The project JDB on SourceForge.
- http://sourceforge.net/projects/xml2swing/ Another project (but very simple, I think it has been cancelled).
- http://netbeans.org/project_downloads/www/flashdemo/matisse.html The Matisse project seems a good concurrent to this project giving you the capability to create the GUI with the mouse.

4.1 Questions and Answers

Question: Is it possible to generate a class using only the `java.awt` package?
Answer: Basically, yes. But the package does not provide the XML tags for this. In addition, the XML focuses on the easy implementation of Swing components. Nevertheless, you can request the addition of the basic objects in this software.

Question: Why generating a JAVA code source rather than providing a dynamic code?
Answer: This is the specificity of this tool: other good programs creates GUI interfaces on the fly. This is not the work of this software.

Question: Can I create applets or dialog boxes?
Answer: This will be possible very soon. Currently, only the frames are available.