International Swinging - Making Swing Components Locale-Sensitive

Volker Simonis
WSI für Informatik, Universität Tübingen, Germany
email: simonis@informatik.uni-tuebingen.de

June 17, 2002

Abstract

Although Java and its GUI library Swing provide software developers with a highly customizable framework for creating truly "international" applications, the Swing library is not locale-sensitive[1] to locale switches at run time.

The consistent and exclusive use of Unicode together with the builtin libraries for resource files and locales make it easy to create internationalized applications. Taking into account Swings elaborate Model-View-Controller architecture, this paper describes how to create GUI applications which are sensitive to locale changes at runtime, thus increasing their usability and user friendliness considerably.

1 Introduction

Sometimes GUI applications are created with internationalization[2] in mind, but are not immediately fully localized[3] for all target languages. In such a case a user native to an unsupported language would choose the language he is most familiar with from the set of supported languages. But the ability to easily switch the language at run time could still be desirable for him if he knows more than one of the supported languages similarly well.

Other applications like dictionaries or translation programs are inherently multilingual and are used by polyglot users. Such applications would greatly benefit if the user interface language would be customizable at runtime.

Unfortunately, this is not a builtin feature of the Java Swing GUI library. However this article will sketch how it is easily possible to customize Swing such that it supports locale switching at runtime. Therefore a new Look and Feel called the `MLMetalLookandFeel` will be created, where `ML` is an abbreviation for "multi lingual". This new Look and Feel will extend the standard Metal Look and Feel with the ability of being locale-sensitive at runtime.

As an example we will take the `Notepad` application which is present in every JDK distribution in the `demo/jfc/Notepad/` directory. It is localized for French, Swedish and

[1] *locale-sensitive*: A class or method that modifies its behavior based on the locale's specific requirements. (All definitions taken from [JavaInt].)
[2] *internationalization*: The concept of developing software in a generic manner so it can later be localized for different markets without having to modify or recompile source code.
[3] *localization*: The process of adapting an internationalized piece of software for a specific locale.

Chinese, as can be seen from the different resource files located in the `resources/` subdirectory. Depending on the locale of the host the JVM is running on, the application will get all the text resources visible in the GUI from the corresponding resource file. The loading of the resource file is achieved by the following code:

Listing 1: Notepad.java [Line 59 to 65]

```
try {
    resources = ResourceBundle.getBundle("resources.Notepad",
                                         Locale.getDefault());
} catch (MissingResourceException mre) {
    System.err.println("resources/Notepad.properties not found");
    System.exit(1);
}
```

The `ResourceBundle` class will try to load the file `resources/Notepad_XX_YY.properties` where `XX` is the two letter ISO-639 [ISO-639] language code of the current default locale and `YY` the two letter ISO-3166 [ISO-3166] country code, respectively. For more detailed information about locales have a look at the JavaDoc documentation of `java.util.Locale`. The exact resolution mechanism for locales if there is no exact match for the requested one is described at `java.util.ResourceBundle`. In any case, the file `resources/Notepad.properties` is the last fall back if no better match is found.

You can try out all the available resources by setting the default locale at program startup with the two properties `user.language` and `user.country`[4]. To run the `Notepad` application with a Swedish user interface you would therefore type:

```
java -Duser.language=sv Notepad
```

However, a user interface internationalized in this way is only customizable once, at program startup. After the resources for the default locale are loaded, there is no way to switch the locale until the next start of the program. We will call this type of internationalization *static* internationalization. Throughout this paper we will change `Notepad.java` to make it *dynamically* internationalized, i.e. locale-sensitive at run time. We will call this new application `IntNotepad`.

2 The Java Swing architecture

A GUI application is composed out of many UI components like labels, buttons, menus, tool tips and so on. Each of these components has to display some text in order to be useful. Usually, this text is set in the constructor of the component for simple components like labels or buttons. Additionally, and for more complex components like file choosers, the text can be set or queried with `set` and `get` methods.

Internationalized applications like the `Notepad` application do not hard code these text strings into the program file, but read it from resource files. So instead of:

```
JFrame frame = new JFrame();
frame.setTitle("Notepad");
```

they use the following code:

[4]Be aware that setting the default locale on the command line with help of the mentioned properties does not work with all JDK versions on all platforms. Refer to the bugs 4152725, 4179660 and 4127375 in the Java Bug Database [JDB].

```
JFrame frame = new JFrame();
frame.setTitle(resources.getString("Title"));
```

where `resources` denotes the resource bundle opened in Listing 1.

Basically, we could just reset all these strings at run time every time the user chooses a different locale. But for an application which uses tens to hundreds of different components it would not be practicable to manually do this. Even worse, some components like `JFileChooser` do not even offer accessory methods for all the strings they display. So we have to come up with another solution which requires a closer look at the architecture of the Swing GUI library.

The design of the Swing library is based on a simplified Model-View-Controller [MVC] pattern, called Model-Delegate [ModDel]. Compared to the classical MVC pattern, the Model-Delegate pattern combines the View and the Controller into a single object called the Delegate (see figure 1). In Swing, these delegates, which are also called the user interface (UI) of a component, are Look and Feel specific. They are derived from the abstract class `ComponentUI`. By convention have the name of the component they are the delegate for with the `J` in the component class name replaced by the name of the specific Look and Feel and `UI` appended to the class name. So for example the UI delegate for `JLable` in the Metal Look and Feel has the name `MetalLabelUI`.

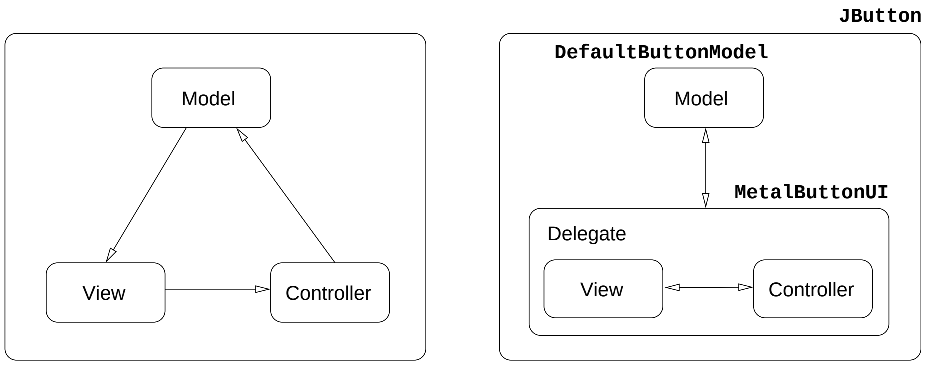


Figure 1: The left side shows the common Model-View-Controller pattern, whereas the right side shows the Model-Delegate pattern used in Swing along with the class realizations for `JButton`.

One of the tasks the UI delegate is responsible for is to paint the component it is tied to. In contrast to the AWT library, in Swing it is not the `paint()` method of every component which does the work of painting itself. Instead, the component's `paint()` method just calls the `paint()` method of its delegate along with a reference to itself.

3 The solution - idea and implementation

After knowing the internals of the Swing architecture, we are ready to make the Swing components aware of locale switches at runtime. To achieve such a behavior, we will introduce one more level of indirection. Instead of just setting a text field of a component to the real string which should be displayed, we set the field to contain a key string instead. Then we override the UI delegate in such a way that instead of just painting the string obtained from its associated component, it will look up the real value of the string to paint depending on the actual locale.

Let us substantiate this in a small example. Listing 2 shows how a `JLabel` is usually created and initialized, followed by a code snippet taken from the `BasicLabelUI.paint()` method which is responsible for rendering the label's text:

Listing 2: Creating a usual `JLabel` and a part of the `BasicLabelUI.paint()` method.

```java
// Create a label.
JLabel label = new JLabel();
label.setText("Hello");

// Taken from javax.swing.plaf.basic.BasicLabelUI.java
public void paint(Graphics g, JComponent c)  {
  JLabel label = (JLabel)c;
  String text = label.getText();

  // Now do the real painting with text.
  ...
}
```

We will now create a new UI delegate for `JLable` called `MLBasicLabelUI` which overrides the `paint()` method such that it not simply queries the text from the `JLable` and renders it. Instead it interprets the string received from its associated `JLable` as a key into a resource file which is of course parameterized by the current Locale. Only if it doesn't find an entry in the resource file for the corresponding key, it will take the key text as the string to render. Thus, the changes in the UI are fully transparent to the component itself.

3.1 Getting the localized resource strings

Because this procedure of querying the localized text of a component from a given resource file will be common for all UI delegates which we will create for our Multi Lingual Look and Feel, we put the code into a special static method called `getResourceString()`:

Listing 3: ml/MLUtils.java [Line 35 to 44]

```java
public static String getResourceString(String key) {
  if (key == null || key.equals("")) return key;
  else {
    String mainClass = System.getProperty("MainClassName");
    if (mainClass != null) {
      return getResourceString(key, "resources/" + mainClass);
    }
    return getResourceString(key, "resources/ML");
  }
}
```

This method builds up the name of the resource file which is searched for the localized strings. Therefore it first queries the system properties for an entry called `MainClassName`. If it succeeds, the resource file will be a file with the same name in the `resources/`

subdirectory. If not, it will assume `ML` as the default resource file name. This file name along with the original `key` argument are passed to the second, two parameter version of `getResourceString()`, shown in Listing 4.

Listing 4: ml/MLUtils.java [Line 50 to 76]

```
private static Hashtable resourceBundles = new Hashtable();

public static String getResourceString(String key, String baseName) {
  if (key == null || key.equals("")) return key;
  Locale locale = Locale.getDefault();
  ResourceBundle resource =
    (ResourceBundle)resourceBundles.get(baseName + "_" + locale.toString());
  if (resource == null) {
    try {
      resource = ResourceBundle.getBundle(baseName, locale);
      if (resource != null) {
        resourceBundles.put(baseName + "_" + locale.toString(), resource);
      }
    }
    catch (Exception e) {
      System.out.println(e);
    }
  }
  if (resource != null) {
    try {
      String value = resource.getString(key);
      if (value != null) return value;
    }
    catch (java.util.MissingResourceException mre) {}
  }
  return key;
}
```

This method finally does the job of translating the key text into the appropriate localized value. If it can not find the corresponding value for a certain key it just returns the key itself, consequently not altering the behavior of a component which isn't aware of the multi lingual UI it is rendered with.

Notice that for performance reasons, `getResourceString()` stores resource files in a static map after using them for the first time. Thus, any further access will use this cached version, without the need to reload the file once again.

3.2 Overloading the `paint()` method of the UI delegates

After having understood the way how localized strings can be queried with the functions introduced in Listing 3 and 4, the overloaded version of the `paint()` method in `MLBasicLabelUI` (Listing 5) should be no surprise. Additionally, the label is now initialized to `"MyApplication.HelloString"` which is a key into the possibly localized resource file `resources/MainClassName_XX_YY.properties`.

Listing 5: A locale-sensitive `JLabel` and the `paint()` method of `MLBasicLabelUI`.

```
// Create a locale-sensitive label which has a MLBasicLabelUI delegate.
JLabel label = new JLabel();
label.setText("MyApplication.HelloString");

// Taken from MLBasicLabelUI.java which inherits from BasicLabelUI.
public void paint(Graphics g, JComponent c)  {
  JLabel label = (JLabel)c;
  String text = MLUtils.getResourceString(label.getText());

  // Now do the real painting with text.
  ...
}
```

Notice that a string which will not be found in the resource file will be displayed "as is" in the label. So our example would work perfectly fine even with the usual component UI, it only would not respond to locale changes at run time.

If we want to make the GUI of a whole application locale-sensitive at runtime, we have to create new UI classes for each Swing component we use in our GUI. This sounds like a lot of work to do, but in fact we just have to redefine the methods which query text data from the component they are associated with.

One problem which we may encounter is the fact that in Swing actual Look and Feels like the Metal Look and Feel or the Windows Look and Feel use their own UI classes which are not directly derived from `ComponentUI` (see figure 2). Instead all the different UI classes for a single component inherit from a class called `BasicXXXUI` where `XXX` stands for an arbitrary component name. This is done to factor out all the functionality which is common to all the different Look and Feels into one base class.

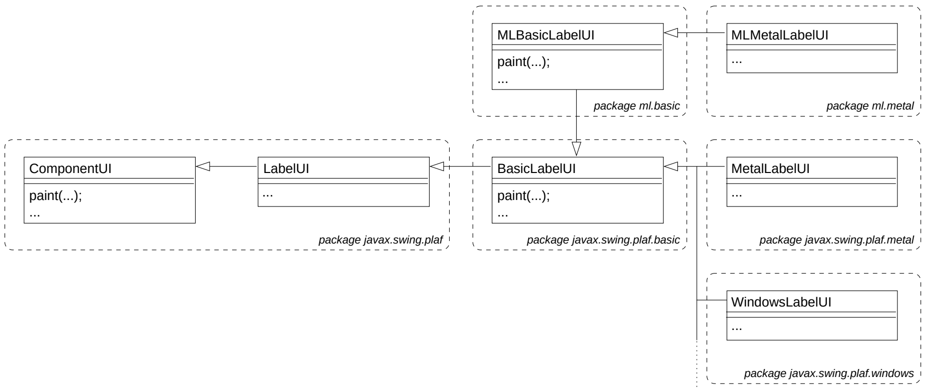


Figure 2: The class hierarchy of the component UI classes of Swing for `JLabel`. In this diagram, `Label` may be substituted by any other Swing component like `Button`, `Tooltip` and so on. The two classes in the upper part of the diagram from the package `ml` are the locale-sensitive UI classes developed in this paper.

This makes our job more difficult, because usually we would like to override the UI's of a distinct Look and Feel, but often the task of querying and painting the actual text is done only or at least in part in the `BasicXXXUI` base classes. Therefore we need to specialize two classes. First we have to specialize the `BasicXXXUI` class for our component and redefine the methods which query the text fields of our component. We will call this class MLBasicXXXUI. Then we have to copy and rename the actual component UI belonging to our desired Look and feel from MetalXXXUI to MLMetalXXXUI and change the base class from which it inherits from BasicXXXUI to MLBasicXXXUI which is the name of our overloaded version of BasicXXXUI. Again, Metal is just an example here. It could be just as well Windows, Motif or any other Look and Feel. Additionally, if necessary, we have to redefine the methods in MLMetalXXXUI which display text attributes from our associated component.

After having implemented all the needed UI delegates, we have to tell our application in some way to use the new delegates instead of the old, default ones. This can be done in two ways. The first one, which is perhaps more simple, is to just register our delegates with the component names at program startup as shown in Listing 6.

Listing 6: Associating Swing components with their UI delegates.

```
  UIManager.put("ToolTipUI",      "ml.mllf.mlmetal.MLMetalToolTipUI");
  UIManager.put("LabelUI",        "ml.mllf.mlmetal.MLMetalLabelUI");
  UIManager.put("MenuUI",         "ml.mllf.mlbasic.MLBasicMenuUI");
  UIManager.put("MenuItemUI",     "ml.mllf.mlbasic.MLBasicMenuItemUI");
  UIManager.put("ButtonUI",       "ml.mllf.mlmetal.MLMetalButtonUI");
  UIManager.put("RadioButtonUI",  "ml.mllf.mlmetal.MLMetalRadioButtonUI");
  UIManager.put("CheckBoxUI",     "ml.mllf.mlmetal.MLMetalCheckBoxUI");
  UIManager.put("FileChooserUI",  "ml.mllf.mlmetal.MLMetalFileChooserUI");
  UIManager.put("ToolBarUI",      "ml.mllf.mlmetal.MLMetalToolBarUI");
```

The second, perhaps more elegant way is to define a new Look and Feel for which the new UI delegates which have been created by us are the default ones. This approach is shown in Listing 7.

Listing 7: ml/mllf/mlmetal/MLMetalLookAndFeel.java [Line 22 to 44]

```
public class MLMetalLookAndFeel extends MetalLookAndFeel {

  public String getDescription() {
    return super.getDescription() + " (ML Version)";
  }

  protected void initClassDefaults(UIDefaults table) {
    super.initClassDefaults(table); // Install the metal delegates.

    Object[] classes = {
      "MenuUI",        "ml.mllf.mlbasic.MLBasicMenuUI",
      "MenuItemUI",    "ml.mllf.mlbasic.MLBasicMenuItemUI",
      "ToolTipUI",     "ml.mllf.mlmetal.MLMetalToolTipUI",
      "LabelUI",       "ml.mllf.mlmetal.MLMetalLabelUI",
      "ButtonUI",      "ml.mllf.mlmetal.MLMetalButtonUI",
      "RadioButtonUI", "ml.mllf.mlmetal.MLMetalRadioButtonUI",
      "CheckBoxUI",    "ml.mllf.mlmetal.MLMetalCheckBoxUI",
      "FileChooserUI", "ml.mllf.mlmetal.MLMetalFileChooserUI",
```

➥

Listing 7: ml/mllf/mlmetal/MLMetalLookAndFeel.java [Line 22 to 44] (continued)

```
      "ToolBarUI",     "ml.mllf.mlmetal.MLMetalToolBarUI",
    };
    table.putDefaults(classes);
  }
}
```

Finally, after each locale switch we just have to trigger a repaint of the dynamically internationalized components. This can be achieved by a little helper function as presented in Listing 8 which takes a root window as argument and simply invalidates all the necessary child components.

Listing 8: ml/MLUtils.java [Line 106 to 112]

```
public static void repaintMLJComponents(Container root) {
  Vector validate = recursiveFindMLJComponents(root);
  for (Enumeration e = validate.elements(); e.hasMoreElements();) {
    JComponent jcomp = (JComponent)e.nextElement();
    jcomp.revalidate();
  }
}
```

It uses another method named `recursiveFindMLJComponents` which recursively finds all the child components of a given container. In the form presented in Listing 9, the method returns all components which are instances of `JComponent`, but a more sophisticated version could be implemented which returns only dynamically internationalized components.

Listing 9: ml/MLUtils.java [Line 154 to 173]

```
private static Vector recursiveFindMLJComponents(Container root) {
  // java.awt.Container.getComponents() doesn't return null!
  Component[] tmp = root.getComponents();
  Vector v = new Vector();
  for (int i = 0; i < tmp.length; i++) {
    if (tmp[i] instanceof JComponent) {
      JComponent jcomp = (JComponent)tmp[i];
      if (jcomp.getComponentCount() == 0) {
        v.add(jcomp);
      }
      else {
        v.addAll(recursiveFindMLJComponents(jcomp));
      }
    }
    else if (tmp[i] instanceof Container) {
      v.addAll(recursiveFindMLJComponents((Container)tmp[i]));
    }
  }
```

➥

Listing 9: ml/MLUtils.java [Line 154 to 173] (continued)

```
  return v;
}
```

Notice that the version of `repaintMLJComponents` shown in Listing 8 only works for applications with a single root window. If an application consists of more than one root window or if it uses non-modal dialogs, they also have to be repainted. This can be done by defining a static method `registerForRepaint` (Listing 10) for registering the additional windows and dialogs and by extending `repaintMLJComponents` in a way to take into account these registered components.

Listing 10: ml/MLUtils.java [Line 142 to 146]

```
private static Vector repaintWindows = new Vector();

public static void registerForRepaint(Container dialog) {
  repaintWindows.add(dialog);
}
```

The new version of `repaintMLJComponents()` is shown in Listing 11:

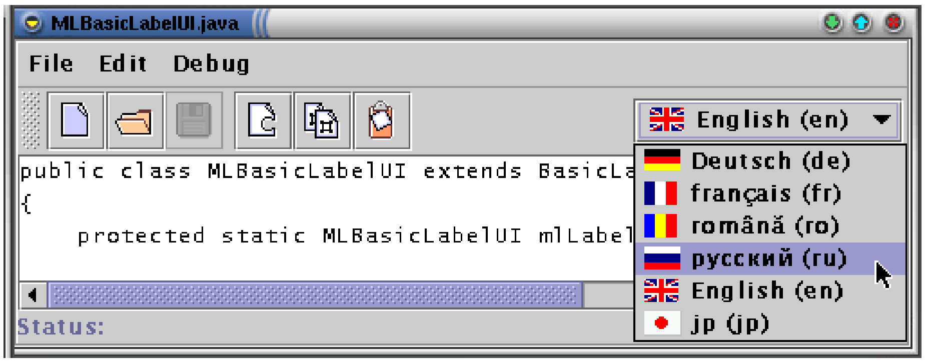
Listing 11: ml/MLUtils.java [Line 116 to 138]

```
public static void repaintMLJComponents(Container root) {
  Vector validate = recursiveFindMLJComponents(root);
  Iterator it = repaintWindows.iterator();
  while (it.hasNext()) {
    Container cont = (Container)it.next();
    validate.addAll(recursiveFindMLJComponents(cont));
    // Also add the Dialog or top level window itself.
    validate.add(cont);
  }
  for (Enumeration e = validate.elements(); e.hasMoreElements(); ) {
    Object obj = e.nextElement();
    if (obj instanceof JComponent) {
      JComponent jcomp = (JComponent)obj;
      jcomp.revalidate();
    }
    else if (obj instanceof Window) {
      // This part is for the Dialogs and top level windows added with the
      // 'registerForRepaint()' method.
      Window cont = (Window)obj;
      cont.pack();
    }
  }
}
```

3.3 The Locale Chooser

After we discussed in detail the techniques necessary to make Swing components aware of locale switches at runtime there remains as last step the presentation of a widget which displays all the available locales to the user and allows him to choose from this list a new default locale.

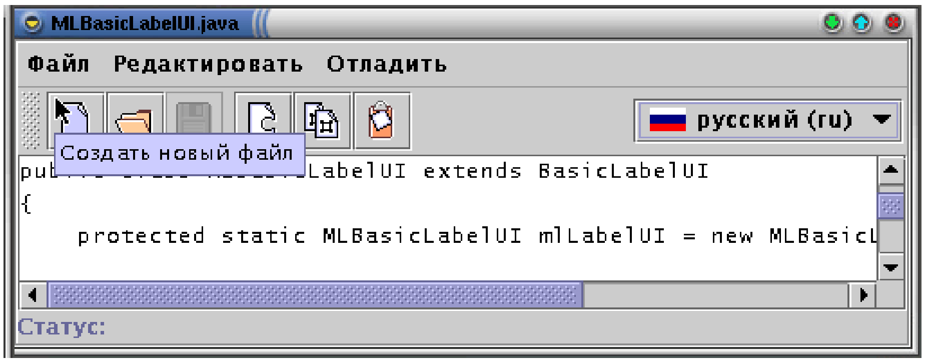
Figure 3 and 4 show the new `IntNotepad` application with the builtin locale chooser. Additionally, the original `Notepad` was extended by a permanent status bar to demonstrate locale switches for labels. The first figure shows the application with the English default locale while the user is just switching it to Russian.



```
public class MLBasicLabelUI extends BasicLa
{
    protected static MLBasicLabelUI mlLabel
```

Figure 3: A screen shot of the `IntNotepad` application. The user just selects Russian as the default locale with the new locale chooser, which is located on the right side of the tool bar.

Figure 4 shows the application after the switch to Russian. Menus, labels, buttons and even tool tips are now displayed with Cyrillic letters in Russian language. Notice that the size of the menus has been resized automatically in order to hold the longer Russian menu names.



```
pub                LabelUI extends BasicLabelUI
{
    protected static MLBasicLabelUI mlLabelUI = new MLBasicL
```

Figure 4: This screen shot shows the `IntNotepad` application after the default locale has been switched to Russian. Labels, menus and even tool tips appear in Russian now.

The class `LocaleChooser` is a small extension of a `JComboBox` with a custom renderer which displays each available Locale with a flag and the name of the corresponding language. The language name is displayed in its own language if available and in English otherwise. Please notice that there is no one to one mapping between languages and country flags, as many languages are spoken in more than one country and there are countries in which more then one language is spoken. Therefore one must be careful when choosing a flag as representation for a language to not hurt the feelings of people who speak that language in a different country. After all, the flags should be just visual hints to simplify the selection of a particular language.

The `LocaleChooser` constructor expects as parameters a `String` which denotes the resource directory of the application and a `Container` which will be the root component passed to the `repaintMLJComponentes()` method presented in Listing 8 when it comes to a repaint of the application caused by a locale switch.

For every language or language/country combination the resource directory passed to the `LocaleChooser` constructor should contain a subdirectory named by the two letter language code or the two letter language code plus an underscore plus the two letter country code, respectively. Each of this subdirectories should contain a file `flag.gif` which will be the image icon displayed by the `LocaleChooser` for the corresponding language.

Thus, adding more locales to the list of locales displayed by `LocaleChooser` is merely a fact of adding the corresponding directories and files to the resource directory and does not require a recompilation of `LocaleChooser`. Remember however that for a locale switch to show any effects a resource file with the localized component strings has to be available as well.

3.4 Putting it all together

Finally, after the discussion of all the details involved in making Swing components aware of locale switches at runtime, we will summarize the important steps and show how they fit into the big picture of a real application.

First of all the new component UI delegates have to be created for all the components which should be dynamically internationalizable. These UI delegates should be packed together into a new Look and Feel which is derived from an already existing Look and Feel. This way we don't have to create UI delegates for the full set of Swing components at the very beginning, but we have the possibility to stepwise extend our new Look and Feel for new components. Creating the UI delegates has been extensively described in section 3.

Once our new Look and Feel is available, we can start to modify our application to make it locale-sensitive at run time. The first step is to set the system property `MainClassName` to the name of our application. This information will be needed by the `getResourceString()` method (see Listing 3) presented in section 3.1. Then we have to set our new Look and Feel as the standard Look and Feel for our application. These two steps can be achieved by the following two lines of code:

```
System.setProperty("MainClassName", "IntNotepad");
UIManager.setLookAndFeel(new MLMetalLookAndFeel());
```

As a third step, we have to install an instance of the `LocaleChooser` presented in section 3.3 somewhere in our application. Usually this will be the tool bar, but it can also be installed in a menu or in a special options window along with other configuration options. The `LocaleChooser` has to be instantiated with a reference to the main application window, in order for the repaint method shown in Listing 8 to work properly.

That's all. From now on, whenever we create a new Swing component, we have the choice of setting its string attributes to either a concrete string or just to a key value. If the string attribute is available in the applications resource file as a key, its value will be displayed instead, according to the current default locale. Otherwise, the string attribute itself will be displayed.

4 Conclusion

This paper presented a technique to make Swing components locale-sensitive at run time. It works by simply creating a new Look and Feel, without changing any code in the components themselves. As example the `IntNotepad` application was derived from the `Notepad` example application available in every JDK distribution. `IntNotepad` is aware of local changes and rebuilds the whole user interface every time such a change occurs at run time. Together with all the other source code presented in this paper it is available for download at [IntNotepad].

Notice that by using the techniques presented here, it would be possible to lift the entire Swing library and make it locale-sensitive for run time locale switches without any compatibility problems with older library versions.

Finally I want to thank Roland Weiss and Dieter Bühler for their assistance and for reviewing this paper.

References

[IntNotepad] Volker Simonis *"The source code for the MLMetal Look and Feel and* `IntNotepad`*"*, available at: http://www-ca.informatik.uni-tuebingen.de/people/simonis/papers/intSwing/IntNotepad.jar

[ISO-639] ISO *"The ISO-639 two letter language codes"*, available at: http://www.unicode.org/unicode/onlinedata/languages.html

[ISO-3166] ISO *"The ISO-3166 two letter country codes"*, available at: http://www.unicode.org/unicode/onlinedata/countries.html

[JavaInt] Andrew Deitsch and David Czarnecki *"Java internationalization"*, O'Reilly & Associates, 2001

[JDB] Sun Microsystems, Inc. *"The Java Bug Database"*, available at: http://developer.java.sun.com/developer/bugParade

[JILT] Sun Microsystems, Inc. *"Java Internationalization and Localization Toolkit 2.0"*, available at: http://java.sun.com/products/jilkit

[MVC] E. Gamma, R.Helm, R. Johnson and J. Vlissides *"Design Patterns: Elements of Reusable Object-Oriented Software"*, Reading, MA, Addison-Wesley, 1995

[ModDel] John Zukowski and Scott Stanchfield *"Fundamentals of JFC/Swing, Part II"*, MageLang Institute, available at: http://developer.java.sun.com/developer/onlineTraining/GUI/Swing2