

# Интернационализация пользовательского интерфейса

Мельников Владимир В.

Изменение языка интерфейса на лету является часто встречающейся задачей. Статья показывает каким образом добиться этого и какие проблемы возникают при этом. Примеры в исходниках совместимы с Java 1.1+ и ориентированы на AWT/Swing, но при небольших изменениях могут быть применены к любой GUI-библиотеке, например SWT. Код был взят из внутренней библиотеки MiniFly (lightweight GUI library подобной Swing) и приспособлен для Swing/AWT.

## 1. Общий интерфейс компонентов

Первое что нужно сделать — это определить единый интерфейс изменения языка, например:

```
interface Localizable
{
    void updateLocale(ResourceBundle bundle)
}
```

Теперь просто пробегаем по дереву компонентов и вызываем метод 'updateLocale' для тех кто реализует интерфейс 'Localizable'.

## 2. Коллекция ресурсов

Но это решение является подходящим только для примитивных случаев.

- Допустим, вы создали необходимые файлы свойств на нескольких языках для первой версии своего приложения и все хорошо. Вот пришло время для второй версии, вам необходимо добавить новые элементы в файлы ресурсов и вы

## Интернационализация пользовательского интерфейса

изменили главные файлы (английский и русский, например) но остались файлы других языков, например португальский и французский, в которых вы не бум-бум. Можно конечно вставлять английские строки, но более удобным было бы считывание данных вначале из файла текущего языка, и если ничего подходящего не было найдено, то попытаться найти это свойство в основном (английском) файле свойств (или классе, если вы предпочитаете такой подход).

- Или вы по каким-либо причинам используете несколько файлов свойств (это очень удобно на этапе разработки, если интерфейс разрабатывается двумя программистами — в конце работы лучше объединить файлы в один для удобства изменений в будущем; или если проект будет использовать какие-то базовые модули, имеющие свои файлы локализации)

Вообще-то стандартные классы имеют возможность построения дерева наборов свойств ('ResourceBundle'), но использовать поле 'parent' класса 'ResourceBundle' на практике очень неудобно (поле 'parent' , как и метод 'setParent' являются защищенными). Как альтернатива, был разработан простой класс коллекции наборов свойств. Первоначально его интерфейс мог бы быть таким.

```
public class ResourceBundles
{
    public void addBundle(ResourceBundle bundle);
    public Object findObject(String key);
    public String findString(String key);
}
```

Но в тестирующем и реальном коде приходилось постоянно использовать код загрузки ресурсов: `ResourceBundle.getBundle (name, locale)`; поэтому он был занесен внутрь класса:

```
public class ResourceBundles
{
    private Vector bundlesNames_ = new Vector();
    private Vector bundles_ = new Vector();

    private Locale locale_ = Locale.US;

    public ResourceBundles(Locale locale)
```

## Интернационализация пользовательского интерфейса

```
{
    locale_ = locale;
}

public Locale getLocale()
{
    return locale_;
}

public synchronized void addBundle(String name, ResourceBundle bundle)
{
    bundlesNames_.addElement(name);
    bundles_.addElement(bundle);
}

public synchronized void addBundle(String name)
{
    addBundle(name, ResourceBundle.getBundle(name, locale_));
}

public synchronized void addBundle(String name, Locale locale)
{
    addBundle(name, ResourceBundle.getBundle(name, locale));
}
...
}
```

Поскольку "набор наборов ресурсов" предназначен для какого-то определенного языка, то было введено поле `locale_`.

### Замечание:

Конечно, лучше было бы вместо использования двух коллекций 'bundlesNames\_', 'bundles\_' использовать одну, но было жалко ресурсов на создание внутреннего класса 'static class Pair {String name\_; ResourceBundle bundle\_};', а по сравнению с другими кривыми способами этот выглядит не так уж и криво.

Также было замечено, что иногда не хватает возможностей класса 'ResourceBundle' для "гибкого" поиска ресурса. Например, имеется файл ресурсов:

```
# settings panel
```

## Интернационализация пользовательского интерфейса

```
msn.settingsPanel.delete=Delete  
msn.settingsPanel.title.delete=Delete  
msn.settingsPanel.tooltip.delete=Delete page
```

В данном случае было бы неплохо использовать одну строку

```
msn.settingsPanel.delete=Delete
```

вместо двух

```
msn.settingsPanel.delete=Delete  
msn.settingsPanel.title.delete=Delete
```

но оставить при этом возможность использования двух отличающихся значений на будущее или в другом языке. Т.е. Java-код должен выглядеть таким образом:

```
delDialog_.setTitle(bundles.findString(  
    "msn.settingsPanel.title.delete"));  
delButton_.setText(bundles.findString(  
    "msn.settingsPanel.delete"));  
delButton_.setToolTipText(bundles.findString(  
    "msn.settingsPanel.title.delete"));
```

а файл свойств:

```
msn.settingsPanel.delete=Delete  
msn.settingsPanel.tooltip.delete=Delete page
```

Для этого была добавлена возможность "поиска" по главному ключу, т.е. вызов `bundles.findString("msn.settingsPanel.title.delete")` будет реально искать ресурсы по ключам "msn.settingsPanel.title.delete", "msn.settingsPanel.delete", "msn.delete" и "delete", что является довольно удобным для детальной настройки внешнего вида элементов управления. Поскольку теперь в классе имеется два вида поиска, то в метод `find` был добавлен дополнительный параметр 'policy', принимающий одно из двух значений:

- `BY_LAST_BUNDLE` — предпочтителен поиск по различным подвидам ключа

## Интернационализация пользовательского интерфейса

внутри одного набора ресурсов, потом внутри следующего набора, и т.д.

- `BY_ALL_BUNDLES` — предпочтителен поиск по одному подключу во всех наборах, потом по второму подключу во всех наборах, и т.д.

Основная функциональность находится в методах `'handleFindObject'`, `'handleGetObject'` и `'getPossibleKeys'`, последний находит все подключения для заданного ключа. Остальные методы созданы для возможности гибкого использования класса.

### Замечание:

Метод `handleGetObject (getObject, getString)` в отличие от `handleFindObject (findObject, findString)` НЕ использует подключения при поиске.

Теперь изменение языка будет выглядеть приблизительно так:

```
Locale rusLocale = new Locale("ru", "RU");
ResourceBundles bundles = new ResourceBundles(rusLocale);
// base locale
bundles.addBundle("myapp", new Locale("en", "US"));
// target (custom) locale
bundles.addBundle("myapp", rusLocale);

com.mvv.minifly.FlyUtilities.updateComponentTreeLocale(
com.mvv.minifly.FlyUtilities.getRoot(this), bundles);
// or
// com.mvv.minifly.FlyUtilities.updateComponentTreeLocale(
//     javax.swing.getRoot(this), bundles);
```

Метод `'updateComponentTreeLocale'` оповещает все компоненты об изменении языка. Метод имеет приблизительно следующий код, при этом подразумевается, что компоненты, которые реализуют интерфейс `'Localizable'`, должны, если желают, оповещать дочерние компоненты самостоятельно (просто вызывая `FlyUtilities.updateComponentsLocale(this, bundles)`).

```
public static void updateComponentTreeLocale(
    Component c, ResourceBundles bundles)
{
    updateComponentTreeLocale0(c, bundles);
    c.validate();
}
```

```
}  
  
private static void updateComponentTreeLocale0(  
    Component c, ResourceBundle bundles)  
{  
    if (c instanceof Localizable)  
        ((Localizable) c).updateLocale(bundles);  
    else  
        updateComponentsLocale(c, bundles);  
}  
  
public static void updateComponentsLocale(  
    Component parent, ResourceBundle bundles)  
{  
    if (parent instanceof Container)  
    {  
        Component[] children = ((Container)parent).getComponents();  
        for (int i = 0; i < children.length; i++)  
            updateComponentTreeLocale0(children[i], bundles);  
    }  
}
```

Рекомендуется поместить реализацию 'Localizable' в ваш базовый компонент (если вы давно занимаетесь разработкой настольных приложений/апплетов, то у вас такой наверняка имеется). В исходниках есть примеры реализации базовых компонентов:

- BasePanel — для AWT
- JBasePanel — для Swing

Можно было бы подумать вот и все, НО...

### 3. Инициализация компонентов

Итак, вы изменили язык, и после этого создали (динамически) новую панель. Как она узнает, что нужно сменить язык, откуда она возьмет набор ресурсов?

Напрашивается решение о необходимости контекста приложения, в котором можно было бы сохранять различные характеристики: текущий язык, look&feel (тему), и др. В простейшем случае (возможно в большинстве случаев) для приложения можно использовать глобальную переменную, но для апплетов такой подход является

## Интернационализация пользовательского интерфейса

абсолютно неприменим (даже в Java-плагине 1.4 от Sun два экземпляра одного и того же апплета используют один экземпляр загрузчика классов, и естественно разделяют одни и те же глобальные переменные).

Sun использует контекст приложения в своих собственных целях (javax.swing.AppContext в java 1.2, 1.3, sun.awt.AppContext в java 1.4) и естественно этот класс является недокументированным, поэтому придется использовать собственную реализацию.

В нашем каркасе (framework-e) был использован класс контекста, который просто привязывается к корневому компоненту, передаваясь тому в конструкторе. Если вы планируете использовать всего лишь один корневой фрейм (Frame или JFrame) и окна (Window или JWindow) и/или диалоги (Dialog или JDialog), то контекст можно поместить во фрейм, а остальные компоненты всегда могут найти его (просканировав дерево родителей) и получить контекст от него. Если вы используете апплет без каких-либо окошек, то можно просто поместить контекст в него. В исходниках есть примеры базовых классов для фрейма и апплета (BaseApplet, JBaseApplet, BaseFrame, JBaseFrame), которые просто получают контекст приложения в конструкторе или создают его, если получили null.

Итак, с контекстом мы разобрались. Теперь необходимо найти точку, где его можно было бы использовать. Хорошим (а возможно и единственным) вариантом является 'addNotify', т.к. все родители уже созданы и можно легко найти корневой элемент и контекст приложения. Код 'addNotify' будет ориентировочно такой:

```
public void addNotify()
{
    super.addNotify();
    appContext_ = FlyUtilities.getAppContext(this);
    if ((appContext_ != null)
        && (appContext_.getResourceBundles() != null))
        updateLocale(appContext_.getResourceBundles());
}
```

При этом можно сохранить у себя ссылку на 'ResourceBundles' для дальнейших манипуляций, но будьте осторожны во избежание висячих ссылок (и естественно

утечек памяти).

- Использование переключения языков со стандартными AWT компонентами может быть осложнено рядом багов в различных реализациях JVM:
  - в некоторых реализациях (1.4.?) от Sun для Windows вместо русских букв отображается абракадабра
  - в реализациях JVM от Microsoft и Symantec (в браузере Netscape) при динамическом изменении текста меток (после второго-третьего раза) не корректно изменяется (точнее перестает вообще изменяться) 'preferredSize' (рис. 1.2). Для Microsoft JVM это легко исправляется последовательными вызовами 'removeNotify', 'addNotify' для контейнера (см. исходники), но при этом если это окно, то происходит неприятное мерцание. В Symantec JVM (в Netscape) после этого не происходит перерисовка контролов и вызов repaint не помогает (нужно свернуть/развернуть окно).

Рекомендую использовать смену языков со Swing или со своими (custom) элементами управления.

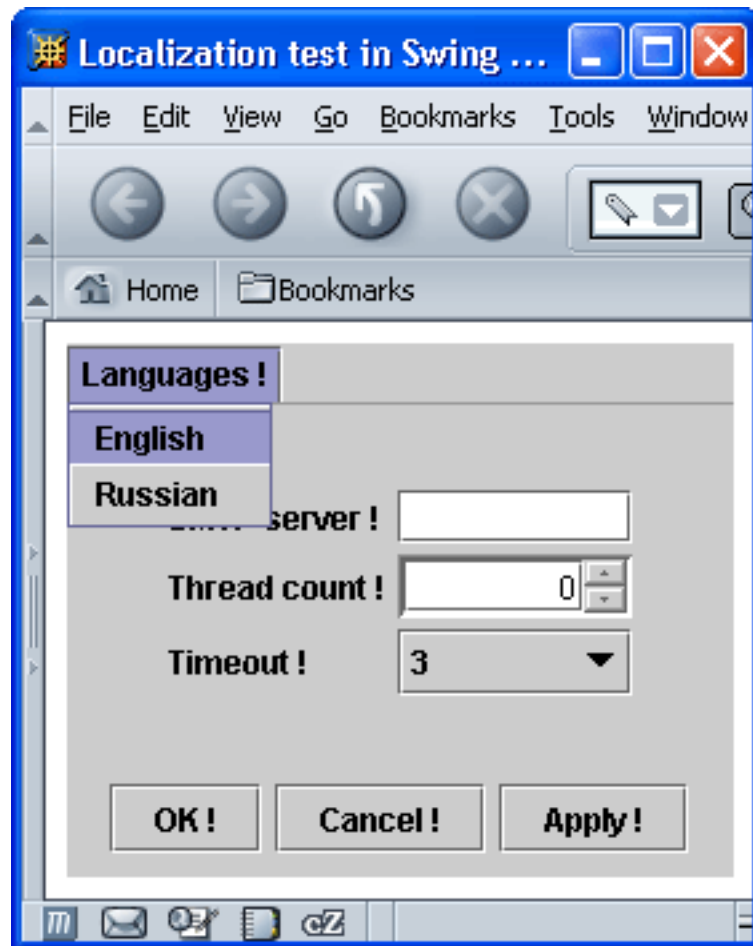


Рисунок 1. Swing-апплет.

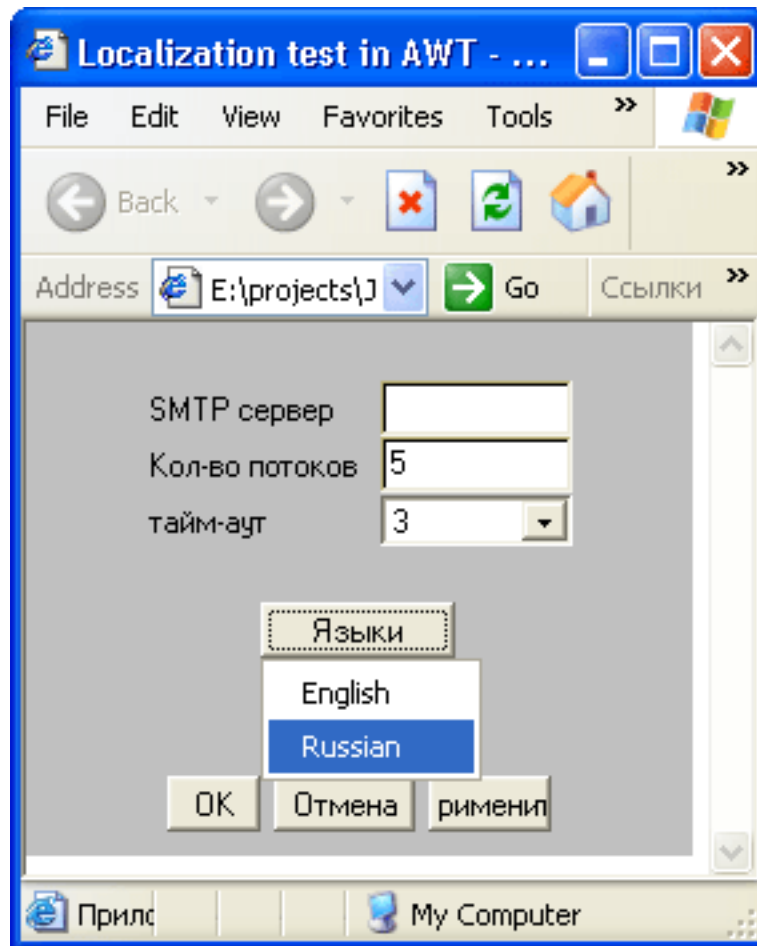


Рисунок 2. AWT-апплет.

## 4. Подготовка ресурсов

Наиболее простым форматом является файл свойств, т.к. при его изменении нет необходимости в перекомпилировании кода, т.е. эту задачу может легко выполнить обычный пользователь. НО! Для хорошей переносимости все не ANSI символы рекомендуется "заискейпить" в следующем виде `\uXXXX`, где XXXX – это шестнадцатеричный код символа в UTF-16 (16-битовом юникоде), это можно легко сделать с помощью командного файла `ansiToProps.bat`, который выполняет преобразование национальных символов в коды UTF-16. Утилита имеет следующий формат:

## Интернационализация пользовательского интерфейса

```
ansiToProps ansiFile propsFile sourceCharEncoding
```

Пример:

```
ansiToProps myapp_ru_RU_ansi.properties myapp_ru_RU.properties  
Cp1251
```

Если кодировка опущена, то будет использована установленная в системе (для Java) кодировка.

Как образец, смотрите файлы `rusToProps.bat` и `createLocBundles.bat`. Также для редактирования вы можете воспользоваться WYSIWYG редактором 'Some Editor', находящийся в архиве вместе с исходными кодами.

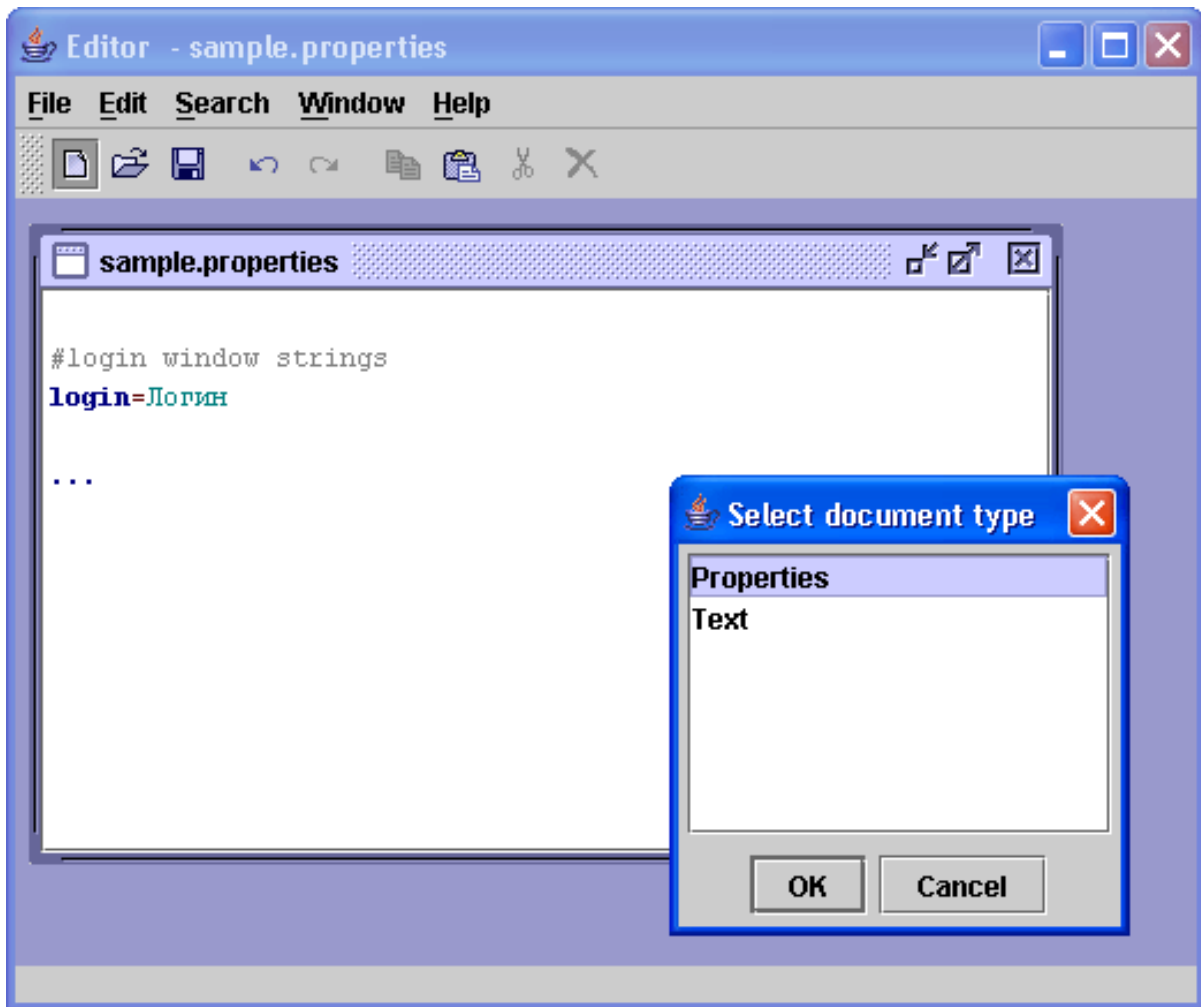


Рисунок 3. Some Editor.

**Замечание:**

Эти инструменты имеют простую реализацию и не полностью поддерживают .properties формат, но для работы (кодирования) с простыми строками, их более чем достаточно.

## 5. Выводы

В этой статье был продемонстрирован метод локализации пользовательского интерфейса "на лету", проанализированы возможные проблемы и их решение.

## **6. Ресурсы**

- [Исходные коды](#)
- [Языковая поддержка для веб-сайтов с использованием JSP](#)
- [Developing Multilingual Web Applications Using JavaServer Pages Technology](#)
- [Java: Русские буквы и не только](#)
- [Java Internationalization: An Overview](#)
- [Internationalization and Localization](#)
- [List of valid language codes \(ISO-639\)](#)
- [List of valid country codes \(ISO-3166\)](#)
- [List of Java-supported encodings](#)
- Платформа программирования J2ME для портативных устройств, Вартан Пирумян, Кудиц-Образ
- Wireless J2ME Platform Programming, Vartan Piroumian, Sun Microsystems Press, Java TM Series