DTS 4: Internationalizing DTS

APELON

BECAUSE TERMINOLOGY MATTERS
75 Sgt William B Terry Dr, Suite 2005, Hingham, MA 02043

+1 (203) 431-2530
www.APELON.com
www.ApelonDTS.org

Table of Contents

Introduction

The DTS Knowledgebase has always supported local language terminology data through its use of the UTF-8 format. But UTF-8 doesn't address local language requirements in the user interface. The Version 4 DTS Editor has been extensively revised to facilitate internationalization (I18N) of the user interface. All prompts and messages are retrieved from a Java resource file and layouts have been coded to take advantage of Java internationalization features to support component orientations (left-to-right vs. right-to-left, and top-to-bottom vs. bottom-to-top).

This document has been designed to serve two purposes. First, to assist organizations wishing to internationalize the Editor for their local languages, and second, to help DTS Editor Module developers in internationalizing their own applications/modules. The *Internationalizing the DTS Editor* section addresses the first goal, while the *Developing an Internationalized DTS Module* addresses the second.

Internationalizing the DTS Editor

Resource File Format

All textual elements of the DTS Editor: titles, prompts, messages, etc., are retrieved at run-time from Java standard "resource" files. The base (English) resource file, `dtseditor_en_US.properties`, is provided in the

DTS_Install\lib\conf client directory. (A copy of this file is also included in the distribution jars as `dtseditor.properties`. This instance provides an “ultimate fallback” copy of resources in case the conf directory is compromised.) By creating copies of this file containing local language translations of the standard elements, the DTS Editor presentation can be customized for different languages.

A Java resource file consists of a sequence of key/value pairs as shown below:

```
#DTSEditor
DTSEditor.Name=Apelon DTSEditor
#name version
DTSEditor.Title={0} {1}

DTSEditor.Splash.ReadingConfig=Reading DTSEditor configuration ''{0}'' ...
DTSEditor.Splash.InitializingLog=Initializing log file ''{0}'' ...
DTSEditor.Splash.BuildingMap=Building module map ...
DTSEditor.Splash.ReadingLayout=Reading layout ''{0}'' ...
DTSEditor.Splash.BuildingFrame=Building application frame ...

DTSEditor.Error.LayoutRead=Error reading layout.
DTSEditor.Error.Layout=Unknown layout error. See log file for details.
DTSEditor.Error.Module=Module error detected during layout. See log file for details.
DTSEditor.Error.ModuleTitle=DTSEditor Layout Error
DTSEditor.Error.Config=Error creating module configuration file.
DTSEditor.Error.Title=Error Launching DTS Editor
```

As shown above, most lines consist of an Editor *key*, `DTSEditor.Name`, an equal sign, and the *value* of the key, i.e., the string displayed in the Editor: `Apelon DTSEditor.` Keys are structured to facilitate understanding of their usage (context). The typical pattern is:

```
module_name [. component ] . element
```

where brackets indicate an optional pattern “component”. Common component names are:

- `Button` – for button labels
- `Error` – for error messages
- `Message` – for non-error messages
- `Menu` – for menu and menu item names
- `Popup` – for popup item names
- `Title` – for frame and dialog title text

For example, the value of `DTSEditor.Error.Layout` is the error message displayed at Editor start for a layout error, `Common.Button.Save` is the label for the “Save” button used by all modules, and `Detail.Popup.Defined` is the text displayed in the Details Module popup for the Defined attribute.

Lines beginning with a hash mark are comments and are frequently used to further describe the meaning or use of the following key/value pair(s).

Internationalization policies recommend against excessive run-time “composition” of messages due to differences in tense, gender, plurals, etc. between languages. The DTS Editor does use some composition to limit the number of elements needing translation or to address run-time components. For example:

```
DTSEditor.Splash.ReadingLayout=Reading layout ''{0}'' ...
```

embeds the name of the DTS Editor layout file in the splash message. Programmers will note that the format is that used by the Java `MessageFormat` class. Replacement (parameter) values are designated by the parameter value’s positional index in brackets. (Note that single quote marks must be doubled to prevent recognition as an “escape” character.)

This example shows a confirmation message with two parameters:

```
#Are you sure you want to delete the Concept 'foo'?
Common.Message.Delete=Are you sure you want to delete the {0} ''{1}''?
```

The comment provides an example of the element’s use. Note that the first (zeroth) parameter will itself be a translated element, e.g., `Common.Concept`.

File Translation and Deployment

Translation of the resource properties file is typically performed using a translator's favorite (local language) text editor. The deployed version of the file, however, must be in ASCII format. Translated characters not in ASCII, such as those in UTF-8 formats, must be represented by escapes, e.g.,:

```
DTSEditor.Menu.View=\u05ea\u05e6\u05d5\u05d2\u05d4
```

The `native2ascii` utility (http://native2ascii.net/#) can be very helpful to convert files in "native" representations to the required ASCII format.

Once the translated ASCII resource file has been prepared, it must be saved in the *`DTS_Install`*`\lib\conf` client directory with a proscribed file name consisting of the base resource name, `dtseditor`, an underscore ("_") and a two-character language code. It is typical, but not required, that the name be followed by another underscrore and a two-character country code. Finally, the extension of the file must be `properties`.

The language code must be is a valid **ISO Language Code.** These codes are the lower-case, two-letter codes as defined by ISO-639. You can find a full list of these codes at a number of sites, such as:
http://www.loc.gov/standards/iso639-2/englangn.html

If present, the country argument must be a valid **ISO Country Code.** These codes are the upper-case, two-letter codes as defined by ISO-3166. You can find a full list of these codes at a number of sites, such as:
http://www.iso.ch/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html

The following is the name of a resource file for the Hebrew language as used in Israel:
`dtseditor_iw_IL.properties`

The DTS Editor informs Java of the desired language/country selection by setting the system `Locale` in the JVM. The `Locale` object specifies how *locale-sensitive* information is presented in the application. Locale-sensitive information includes text flow (left-to-right vs. right-to-left), and default calendar or currency formats. For further details on *Locale* processing, see the description of the `Locale` class in the Javadoc.

Running an Internationalized DTS Editor

To tell the DTSEditor to use a translation file, the language and country codes must be passed to the `DTSEditorApp` application. These are specified as arguments in the DTS Editor command line:

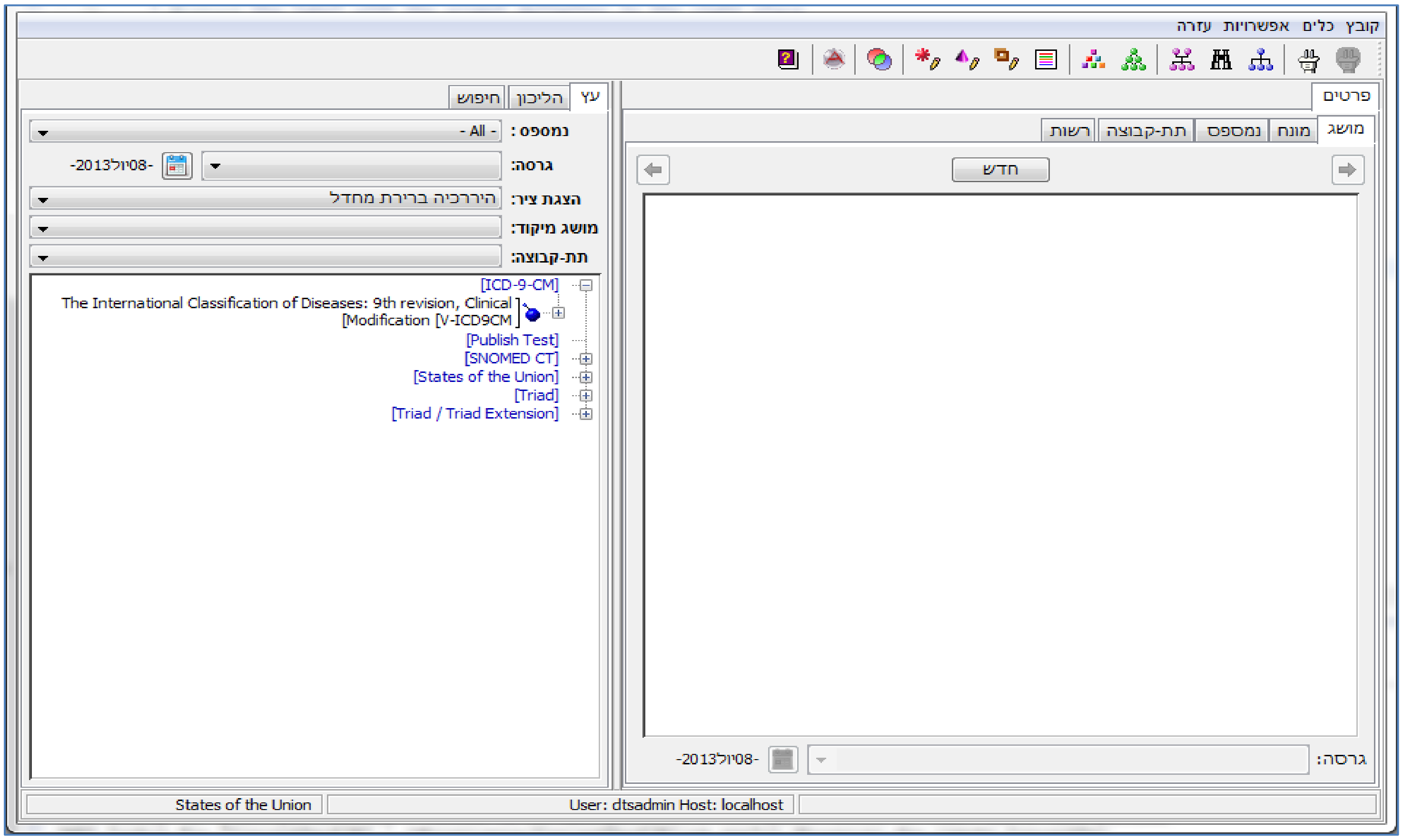
```
"C:\Program Files\Apelon DTS 4.0\bin\runApp_cw.bat" 512 com.apelon.apps.dts.editor.DTSEditorApp –language=iw –country=IL
```

The order of the `DTSEditorApp` arguments is immaterial.

The DTS Editor uses the language and (optional) country arguments to locate the correct translation resource file. The search is performed through a "family" of resource files (or `ResourceBundles` in Java terminology). DTS will first look for the fully-specified resource file `dtseditor_iw_IL.properties.` If this file is not found, DTS looks for a "parent" resource: `dtseditor_iw.properties`. And if this file is not found, the default, or base, file, `dtseditor.properties` is used.

Once a resource file has been located, it is used to populate all the textual elements in the application. It is not necessary that a specific "child" resource have translation entries for all key/value pairs. Elements not found in `dtseditor_iw_IL.properties`, for example, will be "looked up" in `dtseditor_iw.properties,` and finally `dtseditor.properties` for resolution. These defaulted elements will not, of course, have been translated into the local language, but the defaulting behavior can simplify the translation process. It is possible, for example, to only translate common ”surface” terms and leave less frequently appearing error messages in the base language.

The effective (resolved) resource file also determines other attributes of the DTS Editor presentation, such as date/time formats and presentation order. The Hebrew/Israel pair, for example, specifies that DTS element presentation should be right-to-left, as opposed to the English left-to-right. The figure below shows this case:


קובץ כלים אפשרויות עזרה
פרטים
מושג מונח נמספס תת-קבוצה רשות
חדש
גרסה:
-08יול-2013
עץ הליכון חיפוש
נמספס: - All -
גרסה:
-08יול-2013
הצגת ציר: היררכיה ברירת מחדל
מושג מיקוד:
תת-קבוצה:
[ICD-9-CM]
The International Classification of Diseases: 9th revision, Clinical Modification [V-ICD9CM]
[Publish Test]
[SNOMED CT]
[States of the Union]
[Triad]
[Triad / Triad Extension]
States of the Union
User: dtsadmin Host: localhost

Developing an Internationalized DTS Module

Developing an internationalizable module for DTS is not difficult, but does require that a few specific coding procedures be followed. This section outlines the considerations that must be followed. Throughout the section, we will refer to code snippets from the sample DTSMonitor module. This module is described more completely in the ***DTS Editor Module Guide***.

Textual Elements

The most obvious I18N requirement is the ability to easily translate the various titles, prompts, and messages used in the module. As described in the first part of this document, the Java ResourceBundle class provides a convenient way for module classes to abstract their textual elements to an appropriate properties file. From the application standpoint, the requirement is to eliminate literal strings from the line-by-line code and replace them with logical references to entries in the ResourceBundle.

Here is the beginning of `SetCurrentLocalNamespacePanel`, the panel called when setting the value of the Current Local Namespace.

```
private static ResourceBundle resourceBundle =
                     ResourceBundle.getBundle(DTSEditorApp.EDITOR_RESOURCE);
private static String CHOOSE_MESS = resourceBundle.getString("CLN.Message.Choose");
private static String NO_SERVER_ERROR = resourceBundle.getString("CLN.Error.NoServer");
private static String NAMESPACE_ERROR = resourceBundle.getString("Namespace.Error.Load");

//these are the actual names for the UI
private final static String NAMESPACE = resourceBundle.getString("Common.Namespace");
private final static String OK = resourceBundle.getString("Common.Button.OK");
private final static String CANCEL = resourceBundle.getString("Common.Button.Cancel");
```

In the first line, the program gets a reference to the DTS Editor ResourceBundle (through a `DTSEditorApp` constant). The JVM will select the resource bundle based on the currently active (default) `Locale`. Subsequent lines reference keys in the Bundle and assign the values to String constants. These constants can then be used as shown in the examples below:

```
lblMessage.setText(CHOOSE_MESS);


bSubmit.setText(OK);
```

String Composition

It is usually not feasible, or even desirable, to require translation of all textual data in an application. Consider the following example message:

```
Error deleting the Concept 'Test'.
```

"Test" is the name of a Concept in the Knowledgebase. This value should not be translated and is only known at run-time. There should, therefore, be a way for an application to construct a message from its pieces. Sometimes these pieces will be run-time values, like "Test", while other elements will be from "natural language" elements which should be translated. Luckily, there is a standard Java way of addressing this problem: the Java MessageFormat class.

MessageFormat has lots of interesting capabilities, but for I18N purposes, we will focus on string substitution. In the MessageFormat expression below:

```
MessageFormat.format(CONCEPT_DELETE_ERROR, con.getName());
```

the first argument is a reference string and the second argument, `con.getName()`, is substituted into the reference string. The resource properties file value for CONCEPT_DELETE_ERROR would typically look like:

```
Common.Error.ConceptDelete=Error deleting the Concept ''{0}''.
```

The "{0}" notation refers to the position for substitution of the first (zeroth) argument after the reference string in the MessageFormat.format call. (Remember from the discussion above that single quote marks must be doubled to prevent recognition as an "escape" characters.) So the name of the Concept will be substituted for the "{0}" notation.

MessageFormat makes substitution of run-time values very easy. But it can also be used to provide additional reduction of translation effort. The example above is great for Concepts. But what about Terms? Do we need a different properties entry:

```
Common.Error.TermDelete=Error deleting the Term ''{0}''.
```

And what about Synonyms, Properties, Associations, etc.? Should we have separate entries, with associated translation overhead, for each of these strings? One alternative would be to abstract out the DTS element name:

```
MessageFormat.format(DELETE_ERROR, CONCEPT, con.getName());

Common.Concept=Concept

Common.Error.Delete=Error deleting the {0} ''{1}''.
```

Now we can translate the DTS element name "Concept" once and use it in the error message (and presumably in prompts, other messages, etc.).

It would potentially be possible to carry this composition example one more level and use the following constructs:

```
MessageFormat.format(ACTION_ERROR, DELETING, CONCEPT, con.getName());

Common.Concept=Concept

Common.Deleting=deleting

Common.Error.Action=Error {0} the {1} ''{2}''.
```

Now we can create elements for DELETING, ADDING, MODIFYING, etc., and all would use the same reference message, greatly reducing the translation burden. While this works (perhaps) for English, other languages have different linguistic characteristics, e.g. tenses, declensions, cases, and genders. It is unlikely that such a strategy would permit accurate translation.

DTS has chosen to use the intermediate model. It defines common DTS Object name keys like `Common.Concept`, `Common.Property`, etc., but does not abstract out actions/verbs.

Application Layout

In addition to string substitution, an internationalized Module must adapt to other language-specific issues such as the direction of text flow (panel layout) and data formats. Internally, the `Locale` object provides the JVM with locale-sensitive direction. Some of these dependencies, such as Calendar formats, are generally available without programmer intervention. For other capabilities, the application developer must take explicit steps.

A thorough description of all aspects of application internationalization is beyond the scope of this document. As one example of the principles, consider the `BorderLayout` class commonly used in `JPanel` construction. `BorderLayout` supports two types of positioning constants, *absolute* and *relative*. Absolute positioning constants are `NORTH`, `SOUTH`, `WEST`, and `EAST`. These constants refer to the top, bottom, left and right `BorderLayout` regions. To accommodate language-specific layout, on the other hand, `BorderLayout` supports the relative positioning constants, `PAGE_START`, `PAGE_END`, `LINE_START`, and `LINE_END`. In a container whose `ComponentOrientation` is set to `ComponentOrientation.LEFT_TO_RIGHT`, these constants map to `NORTH`, `SOUTH`, `WEST`, and `EAST`, respectively. If `ComponentOrientation` is set to `ComponentOrientation.RIGHT_TO_LEFT`, on the other hand, these constants map to `NORTH`, `SOUTH`, `EAST`, and `WEST`. `ComponentOrientation` is usually set automatically by the current JVM `Locale`, but can be modified programmatically to address specific layout requirements. Other Layout Managers such as `GridBagLayout` have similar options.

See the ***Internationalization Trail*** in the Java Tutorials for a complete discussion of internationalization in Java.