

A framework for multiple language support

William Hollings

FOR MANY SOFTWARE applications, communicating with the user in a single language (usually English) is sufficient. However, some applications, such as those supporting customers in an urban banking environment, must communicate with users from diverse language backgrounds. These applications must be designed to support dynamically changing languages at the user interface. For example, Figure 1 illustrates a single test window that has been opened in both English and French as determined by the user's language preference selection.

I refer to this concept of dynamic language selection as *multiple language support (MLS)*, to differentiate it from the more common *national language support (NLS)* which assumes the use of only a single language. This article describes parts of the multiple language framework that our team at the Toronto-Dominion Bank is developing using Digitalk's Visual Smalltalk.

THE LANGUAGEMANAGER CLASS

Central to the MLS framework is a subclass of `NationalLanguageSupport` called `LanguageManager`. A singleton instance of `LanguageManager` is plugged into the existing global variable `NationalLanguage` at application start up time.

In addition to the responsibilities it inherits, `LanguageManager` adds the capability to manage *language files*. Each language file contains the information required to translate and format all on-screen text within the application into a particular language. We can incrementally add new languages simply by distributing new language files without redeveloping the application.

The `LanguageManager` singleton manages these language files (with help from `ObjectFiler`) via two public methods:

- `LanguageManager>>getSupportedLanguages`. Answers a collection of language names that are currently supported by the application. During application startup, the `LanguageManager` singleton scans the directory for all language files (e.g., *.lng) to build this collection of language names.
- `LanguageManager>>setLanguage: aLanguageName`. Sets the current language to the one identified by `aLanguageName`. This method loads the contents of the appropriate language file from disk. This message is typically received from a user preferences selection tool.

Each language file contains a dictionary to translate wid-

get labels, menu labels, and other strings. It also contains various data formatting information such as the decimal separator character and date formats. Once loaded from file, this information is maintained in instance variables within the `LanguageManager` singleton. In particular, the instance variable `stringDictionary` is populated with the dictionary of translated strings.

The keys for `stringDictionary` are language-neutral string abbreviations. In the case of widgets and menu items, the label text that was assigned to the widget or menu item at GUI, design time is used as the key (e.g., 'CloseButn'). In the case of other displayable strings, such as those displayed in message boxes, the key is a language-neutral abbreviation such as 'ErrMsgComm073'.

Typically, the language dictionary for each language is created and maintained in a spreadsheet. It is read into the development image and saved into the language object file using additional `LanguageManager` methods.

Some points regarding performance are in order here. If the `stringDictionary` gets too big, the dictionary lookup times may become unacceptably slow. Also, the use of `Strings` as dictionary keys is less efficient than using `Symbols` as keys to an `IdentityDictionary`.

One solution to the first problem is to factor the dictionary into smaller dictionaries. Our production application uses three dictionaries within the `LanguageManager` singleton, one each for widget labels, menu labels, and general strings. Other methods of factoring are possible, however for simplicity, the framework described here has only one language dictionary.

With respect to the second issue, the choice of `Strings` as keys in `stringDictionary` was motivated by the fact that the GUI environment and tools assume the use of `Strings` for the names and labels of widgets and menus. A design using `Symbols` would require hacking the GUI environment and tools. Such a design would also have to avoid using the `String>>asSymbol` method, which performs its own (larger and longer) string-keyed dictionary look up. The `String` keys have proved to be fast enough in our application, though this remains an area for potential performance improvement.

TRANSLATING STRINGS

The primary collaborator with `LanguageManager` is the `String` class itself. `Strings` respond to the `asMLSString` mes-

sage and simply delegate the translation work to the LanguageManager singleton as follows:

```
String>>asMLSString
  "Answers my translation in the current language."
^NationalLanguage translateString: self
```

which is handled by LanguageManager as:

```
LanguageManager>>translateString: keyString
  "Answer the translation of the string keyString."
^self stringDictionary at: keyString ifAbsent: [
  keyString ]
```

Notice that the original key string is returned if a translation string could not be found. This allows the application to work even if some or all translations are missing from the language file. This comes in handy during development when the GUI and language files are in a state of flux.

TRANSLATING WIDGETS

The translation of widgets and menu labels takes place during the opening of a window. We added the following method to the TopPane class:

```
TopPane>>translateWindow
  "Tell myself, all my widgets, and my menu bar to
  translate themselves."
  self translate.
```

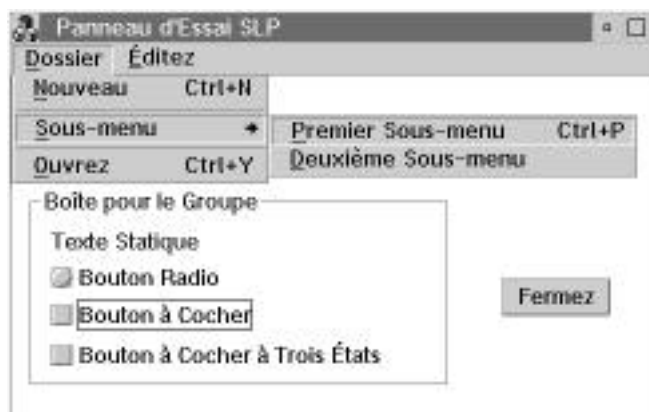
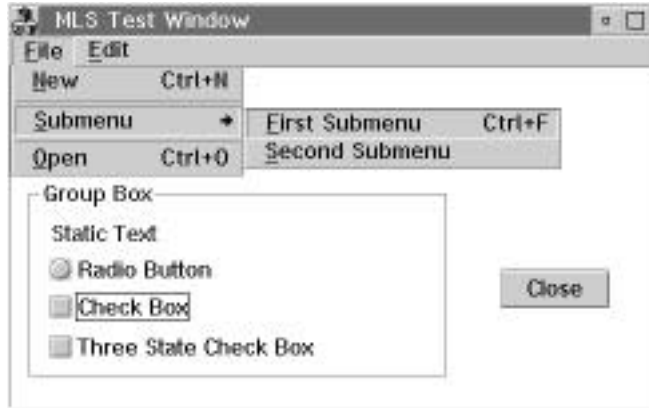


Figure 1. Illustration of a single test window opened in both English and French. Notice that even the menu accelerator keys can sometimes be different.

```
self allChildrenDo: [ :each | each translate ].
self menuWindow translate.
```

which sends the translate message first to itself, then to all of the widgets contained on the window, and then to the window's menu bar.

The TopPane>>translateWindow message is sent after the widgets and menus have been created as objects but before they have been made visible. The question of which object sends this message depends on which GUI builder is being used. Under WindowBuilder, a good place to send this message is in the preInitWindow method of the ViewManager subclasses. Under PARTS, a subclass of PARTSWindowPart can be created to override the open method so that the window translates itself before opening:

```
MLSPARTSWindowPart>>open
  "Translate myself before opening."
  self translateWindow.
  ^super open
```

In our MLS framework, all objects respond to the translate message. The default method (defined in Object) does nothing. All widgets and menus with a displayable label override this default method to specifically translate their label text. For widgets and windows this is done in the ControlPane and TopPane classes, respectively, and is the same for both classes:

```
TopPane>>translate
ControlPane>>translate
  "Translate my label according to the current language."
  self label: (self label asMLSString).
```

In this method, the widget retrieves its existing language-neutral label string and tells it to translate itself. The resulting translated string is then assigned back to the label. To stop nil labels from breaking the system we also added an UndefinedObject>>asMLSString method which simply answers nil.

TRANSLATING MENUS

Translating menu labels is a little more complex because of the need to register menu selection accelerator keys (e.g., - Ctrl+S), which may be different for each language.

Recall that the TopPane>>translateWindow method sends the translate message to the menu bar. In a non-PARTS application, the menu bar is an instance of MenuWindow and it simply passes the message on to its component menus:

```
MenuWindow>>translate
  "Tell each of my menus to translate itself."
  self menus do: [ :each | each translate ].
```

The Menu>>translate method first translates its own title (e.g., File, Edit, etc.) and then cycles through each of its

INFO@SIGS

SIGS Publications, Inc., 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

ARTICLE SUBMISSION

To submit articles for publication, please contact: John Pugh & Paul White, Editors, 885 Meadowlands Dr. #509, Ottawa, Ontario, K2C 3N2 Canada; email: streport@objectpeople.on.ca

PRODUCT REVIEWS AND ANNOUNCEMENTS

To submit product reviews or product announcements, please contact the Editors at the address above.

CUSTOMER SERVICE

For customer service in the US, please contact PO Box 5050, Brentwood, TN 37024-5050; 800.361.1279; Fax: 615.370.4845; in the UK, please contact Subscriptions Department, Tower Publishing Services, Tower House, Sovereign Park, Market Harborough, Leicestershire, LE16 9EF, UK; +44.(0)1858.435302; Fax: +44.(0)1858.434958

SIGS BOOKS

For information on any SIGS book, contact: Don Jackson, Director of Books, SIGS Books, Inc., 71 West 23rd Street, New York, NY 10010; 212.242.7447; Fax: 212.242.7574; email: donald_jackson@sigs.com

SIGS CONFERENCES

For information on all SIGS Conferences, please contact: SIGS Conferences, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7515; Fax: 212.242.7578; email: info@sigs.com

BACK ISSUES

To order back issues, please contact: Back Issue Order Department, SIGS Publications, 71 West 23rd Street, 3rd Floor, New York, NY 10010; 212.242.7447; Fax: 212.242.7574

REPRINTS

For information on ordering reprints, please contact: Reprint Management Services, 505 East Airport Road, Box 5363, Lancaster, PA 17601; 717.560.2001; Fax: 717.560.2063

ADVERTISING

For ad information for any SIGS publication, please contact:

East Coast/Europe: Gary Portie
Central US: Elisa Marcus
Recruitment: Michael Peck
Exhibit Sales, West Coast: Kristin Viksnins
Exhibit Sales, East Coast: Sarah Olszewski
212.242.7447; Fax: 212.242.7574
email: sales@sigs.com
West Coast: Diane Fuller
408.255.2991; Fax: 408.255.2992
email: dhfsigs@hooked.net

SIGS HOME PAGE

Access the SIGS Home Page at: <http://www.sigs.com>.

menu items telling each menu item to translate itself before setting the accelerator key for the menu item:

```
Menu>>translate
```

```
"Translate my title, then translate my menu items."
```

```
self title: ( self title asMLString ).
```

```
self translateItems.
```

```
Menu>>translateItems
```

```
"Tell each menuItem to translate itself and then set its  
accelerator key."
```

```
self items do: [ :each |  
    each translate.  
    self setAccelKeyOf
```

From here, the MenuItem>>translate method translates its own label and then tells any submenu (which would be an instance of Menu) to translate itself.

```
MenuItem>>translate
```

```
"Translate my label text and if I have a submenu,  
translate it."
```

```
self label: ( self label asMLString ).
```

```
self submenu translate.
```

Note that because the default implementation of the translate method in Object does nothing, this method will work correctly even if the submenu is nil (i.e., there is no submenu).

Finally, the MenuItem>>setAccelKeyOf: method is where things get a little complicated and algorithmic. This method extracts the accelerator substring from the menu item label (e.g., 'Ctrl+S') and then parses the components of this substring to convert them into a key code and bit flags. These are in turn inserted into an array of accelerators that is maintained by the Menu instance. This code is as follows:

```
MenuItem>>setAccelKeyOf: mlItem
```

```
| lbl tabIdx accelString itemIdx bits bitsString key  
keyString |
```

```
( lbl := mlItem label ) isNil ifTrue: [ ^self ].
```

```
( tabIdx := lbl indexOf: Tab ) > 0 iffFalse: [ ^self ].
```

```
accelString := ReadStream on: ( lbl copyFrom: ( tabIdx  
+ 1 ) to: lbl size ).
```

```
accelString isEmpty ifTrue: [ ^self ].
```

```
itemIdx := self items indexOf: mlItem ifAbsent: [ ^self ].
```

```
bitsString := accelString upTo: $+.
```

```
keyString := accelString upTo: $+.
```

```
keyString first isDigit
```

```
ifTrue: [
```

```
    key := 0.
```

```
    1 to: keyString size do: [:i |
```

```
        key := key * 10 + ( keyString at: i ) digitValue ].
```

```
    bits := AfVirtualkey ]
```

```
iffalse: [
```

```
    key := keyString first.
```

```
    bits := AfChar ].
```

```
( bitsString includes: $C ) ifTrue: [ bits := bits |
    AfControl ].
( bitsString includes: $A ) ifTrue: [ bits := bits | AfAlt ].
( bitsString includes: $S ) ifTrue: [ bits := bits | AfShift ].
accel at: itemIdx put: (self accelArray: key accelBits: bits).
```

Although all of the Menu and MenuItem methods described apply to both PARTS and non-PARTS development, the structure of the menu bar is slightly different under PARTS. PARTS keeps the menu titles separate from the actual menus, and the Menu instances are not attached to the PARTSMenuBar (a subclass of MenuWindow) instance until immediately before the window is opened. As a result, PARTSMenuBar requires a different translate method:

```
PARTSMenuBar>>translate
    "Tell each menu title and Menu to translate itself."
    self children do: [ :each | each translate ].
    self partApplication componentDictionary
        do: [ :each |
            each isPARTSMenuItem
                ifTrue: [ each menuItem translateItems ] ].
```

This method first translates the menu titles (accessed via self children). It then locates each instance of Menu in the PARTS application controlling the window and tells each of them to translate their menu items.

Incidentally, don't use PARTSMenuBar>>translate as an example of good programming practice. In a production application, we should add methods to both PARTSApplication and PARTSMenuItem to reduce the coupling in the PARTSMenuBar>>translate method. Currently, this method must know that the PARTSApplication has a componentDictionary that contains instances of PARTSMenuItem, which in turn holds on to instances of Menu. I cheated a bit here to reduce the amount of code required for this article.

TRANSLATING MESSAGE BOX STRINGS

Because any string can be translated, we created a new message box that accepts language-neutral abbreviation strings instead of raw text. These are then translated into the current user language. Therefore, instead of coding the following:

```
MessageBox warning: 'This action will destroy the
    known universe.'
```

we would code:

```
MLSMsgBox warning: 'WarnUniverseByeBye'
```


which would be translated via String>>asMLSString before the message box was displayed.

Unfortunately, it is not sufficient to simply create MLSMsgBox as a subclass of MessageBox because that class relies on native OS message boxes, which use their own text for the 'Yes', 'No', 'OK' and 'Cancel' buttons. We built MLSMsgBox (and other utility windows such as MLSPrompter) from scratch.

OTHER ISSUES

The format of numbers displayed as on-screen text or in entry fields varies from language to language (e.g., \$1,000.00 is displayed in some languages as 1.000,00\$). To handle this, we have added methods such as Number>>asMLSString which formats the number with the appropriate "thousands" and decimal separators.

For Help files, we maintain a separate Help file for each supported language. When a new language is selected the LanguageManager singleton renames the Help files so that the one associated with the newly selected language will be used by the Help system.

Finally, I have certainly not exhausted the issues surrounding full MLS support in this brief article. The framework described does not translate the text on any window that is already open. This would involve tagging all menu components with a name and rebuilding the menu accelerator key tables on the fly. I also did not address the formidable challenge of supporting text *input* in multiple languages, which touches on issues as diverse as physical keyboards and database storage. These framework extensions are left as an exercise for the reader. 

William Hollings is a Smalltalk architect and consultant in Toronto. He is currently helping the Toronto-Dominion Bank develop brokerage and banking applications. He can be reached at hollings@inforamp.net.