Careers in Software Internationalization

Larry G. Childs

Careers in Software Internationalization

Larry G. Childs

Abstract

Anyone who plans to work as a professional translator will almost certainly have the opportunity to be involved with software internationalization. Many large corporations such as IBM, Microsoft, WordPerfect, and Novell have software internationalization departments. It is also a major source of work for freelance translators. This paper presents a summary of what is necessary to be successful in this field. In addition to a knowledge of translation and culture, software translators must understand concepts such as software enabling, retrofitting, and localization. They must also understand how computers store and display text.

Introduction

Today's software vendors, eager to reach a worldwide market with their products, have created unprecedented opportunities for translators and other language experts. If you are a professional translator or plan to become one, you will almost certainly have the chance to be involved with adapting software for international use. You may find full-time employment in the software internationalization department of a large corporation, such as IBM, Microsoft, WordPerfect, Borland, or Novell. Many software companies also use freelance translators and translation houses. If you choose to work as a freelance translator or as a staff translator in a translation house, software internationalization projects can become a major source of work for you. Besides translators, software internationalization projects need language and culture consultants, as well.

To compete in this fast growing segment of the translation industry, you must have some specialized knowledge. Software internationalization is more than just translating the messages of a software program. The software itself must be designed for international use, and you must understand the basics of this design to be an effective translator. Successful software translators are also experts in adapting the non-linguistic cultural aspects of software programs. This paper will deal with the cultural aspects of software internationalization, and then discuss some key issues in international software design.

Terminology

We first need to explain some terminology. Learning the industry jargon will not only facilitate this discussion, it will help you show a prospective employer that you are indeed familiar with the software internationalization industry.

The first term to learn is *software localization*. Briefly defined, localization is the process of adapting a piece of software to a particular locale. It refers to translation and to the adaptation of other culturally specific parts of the software. To be successful in software internationalization, you must be more than a translator, you must be a localizer.

During the localization process, the localizer translates *text strings*. These are the pieces of text in the software that the user sees on the screen. They include error and informational messages, prompts, and menu items.

Another important term is *enabling*. This is the programming design and implementation necessary to allow localization to take place. If a piece of software is enabled properly, it can be adapted to any number of locales with a minimum of programming effort.

Finally, IBM has defined *software internationalization* as enabling plus localization. In other words,

Larry G. Childs has a B.A. in German (1977), and an M.A. in German and Linguistics (1979) from BYU. He is currently employed by Novell, Inc., Provo, Utah, in the field of software internationalization.

the programming effort to enable a piece of software, plus the linguistic effort to translate and otherwise adapt it to a particular locale yield internationalized software.

Cultural Adaptation

Translation forms the largest single localization task in software internationalization, but it is also the part of localization that translators understand best. This paper will focus on the less understood part of localization, namely adapting non-linguistic aspects of the software to different locales. There are two main aspects of cultural adaptation: format conventions, and graphics.

Format Conventions

Many software programs display dates, times, numbers, and currency. The format of each of these varies from locale to locale, as figure 1 shows.

LOCALE	DATE	TIME	NUMBER	CURRENCY
USA	03/15/93	8:37:00 PM	12,345.67	$1.22
UK	15/03/93	20:37:00	12,345.67	£1.22
Germany	15.03.93	20:37:00	12.345,67	DM1,22
Portugal	15-03-93	20:37:00	12.345,67	1$22 Esc.
Sweden	93-03-15	20.37.00	12.345,67	1,22 kr

Figure 1: Format conventions

Ideally, the end user of the software can choose the notational conventions used to format this information. Software must be designed to give the user a choice of format conventions. A good example of well enabled software is Windows, whose International Icon allows the user to set the preferred locale settings from a menu of choices. Windows even allows to the user to customize a locale's default format settings to suit individual preferences.

Software engineers who program multilocale format menus will expect you, the localizer, to be an expert in all format conventions, not just those associated with the target languages into which you translate. An excellent source for this type of information is IBM's four-volume series called *National Language Design Guide* (1991).

Adapting Graphics

One very important localization task is to adapt icons, the little pictures on the screen that depict file names and programs, etc. Icons will be familiar to you if you use a graphical user interface, such as Macintosh or Windows. As a localizer, you are responsible to know what connotations a graphic image has in your target culture. Images that are perfectly understandable and innocuous in one culture may be meaningless, confusing, or even obscene in other cultures.

A classic example of the problems that culture-specific icons can cause is the trashcan icon on Macintosh computers. To delete a file, you drag an icon of the file to the trashcan. This caused considerable confusion when the Macintosh was first introduced to Great Britain because the icon closely resembles the post boxes for the Royal Mail (Taylor 1992, 49).

The meaning of gestures varies widely from culture to culture. For example, an icon depicting any sort of a pointing finger will be obscene in at least one culture. *Gestures* by Roger Axtell (1991) is a good reference work for the cultural meaning of gestures.

A related issue is the choice of colors used to display both graphics and text on a computer screen. For example, showing an error message in red to highlight the warning nature of the message, a common practice in America, sends mixed signals in China, where red depicts happiness (Taylor 1992, 42).

Enabling Issues

As a localizer, you must also understand the enabling design issues that affect the translation and display of text in a software program. First, enabling, or the lack of it, puts constraints on the way you translate. Second, as a localizer, you will probably find yourself in the role of an internationalization expert, giving advice to programmers who are trying to enable their code. Let us look at the major language enabling issues that you need to know to act as a consultant to software engineers who want to write enabled programs.

String Expansion

One of the most important things that you can teach software engineers about translation is that a translated text string is not going to be the same length as the original. When going from English to European languages, the translation is almost always longer. In ideographic languages, such as Japanese, the translation from English may be about the same length as the original, but since enabled software should allow for a wide variety of languages, it must account for the expansion in European languages.

Unless the engineer allows for expansion when designing the screen layout, many translated strings will not fit. Figure 2, taken from Childs (1992), shows an example of a string expansion problem.

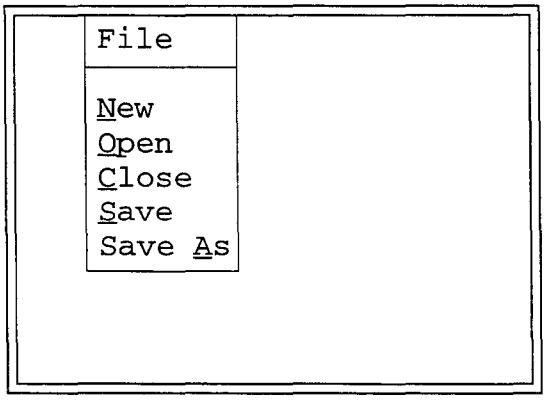

Figure 2: Sample Menu Screen

As you can see, the box around the menu is just wide enough for the longest menu item, *Save As*, which is seven characters long. If the translation of any of the menu items happens to be longer than seven characters, you are in trouble! As a translator, you will be forced to invent confusing abbreviations to make long translations fit into the allotted space.

The enabling solution is to leave extra space on the screen to allow room for translated strings that are longer than the original. Figure 3, adapted from IBM's *National Language Design Guide, Volume 1* (1991, 2-4), shows IBM's general expansion guidelines. They are based on the number of characters in the original English text string.

Characters in String	Maximum Expansion
1 - 10	200%
11 - 20	100%
21 - 30	80%
31 - 50	60%
51 - 70	40%
over 70	30%

Figure 3: IBM String Expansion Guidelines

String Isolation

String isolation is another enabling practice that facilitates software translation. Translatable text strings are removed from the source code and put in a separate file, which makes the translator's job easier. Text strings in unenabled programs are scattered throughout the source code. Figure 4 shows a simple example of strings in a C language program. You don't need to understand the program; just note that the translatable strings are inside double quotes.

```
#include <stdio.h>
extern float get_answer();
void main()     /* Sample program to show text strings */{
    float price, rate;
    char inbuf [130];
    printf("This program computes the sales tax to find\n");
    printf("the total price of an item.\n\n");
    printf("Please enter the amount of purchase: ");
    price = get_answer();
    printf("Please enter the tax rate: ");
    rate = get_answer();
    printf("\n\nPurchase price: %5.2f", price);
    printf("\n       Tax: %5.2f", price * rate);
    printf("\n Total price: %5.2f", price + (price * rate));}
```

Figure 4: Sample C Source Code

An actual program is often thousands of lines long with translatable text strings scattered throughout. Translators not familiar with the programming language may have a hard time finding the strings. Besides, software companies are reluctant to allow translators to modify the source code by changing the strings. Isolating the strings into a separate file solves these problems.

String isolation also allows a piece of software to become multilingual. Leaving the messages in separate files makes it possible for a user to switch the language of a program simply by switching the language file. The software company does not have to produce separate programs for German, French, and Italian, for example. Rather, the company can produce one enabled, language-independent program, plus language files for each supported language. Programs like this are much easier for a software company to produce and maintain. They also allow for greater flexibility. Adding a Spanish version, for example, is simply a matter of creating a Spanish language file. No additional programming is involved.

No String Concatenation

Software engineers also need to be taught to write each text string as a complete, stand-alone thought. When dealing with a number of similar messages, programmers, for whom economy of space is a virtue, often store pieces of messages as strings in the source code, which they then concatenate to form complete messages to be seen by the user. A simple example is a typical greeting message that is displayed on the screen when a user logs onto a computer. The user sees either *Good morning*, *Good afternoon*, or *Good evening* depending on the time of day that he or she logs on. But to save space, the programmer only stores the four words *Good*, *morning*, *afternoon*, and *evening* in the program, and then concatenates the appropriate pair on the screen as needed.

Combining sentence fragments works for English, but there is no way to ensure proper word order, or agreement for gender and case in other languages. This greeting time concatenation does not work in Italian, for example, where the word for *good* must be inflected to match the gender of the following noun. See Figure 5 (Childs 1992).

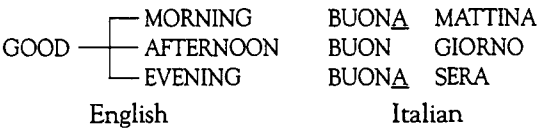

Figure 5: Sentence Concatenation

Easily Translatable Strings

You will also probably need to teach software engineers how to write text strings that lend themselves to translation. You must teach them to avoid overly technical jargon and nonstandard abbreviations when they write messages; these are difficult for the translator, as well the typical user, to understand. Teach them not to be afraid of words like *the*, *a*, and *an*; complete sentences are much easier to understand than cryptic technical messages, such as this error message in Novell's NetWare program:

Volume Mount Problem List Overflow.

They must also learn to avoid slang and humor, which only ends up confusing, rather than amusing foreign readers. Making messages easily translatable tends to make them more readable in the source language, as well.

Other String Considerations

Besides translation, programs must be enabled to handle other aspects of text strings, as well. I will mention two common ones here: *sorting* and *accelerators*.

Sorting is a general term for alphabetization, but includes the ordering of nonalphabetic languages, as well. Different languages have different sorting schemes. In Swedish, for example, the umlauted vowels *ä* and *ö* are alphabetized after the letter *z*, whereas in German they appear next to the non-umlauted versions of the respective vowels. If an enabled software program sorts lists of strings, it must be programmed to sort according to language. Again, you as the localizer will be viewed as the expert to determine the sort sequences for various locales.

We saw an example of accelerators in the underlined letters in the menu example in figure 2. This is the name applied to the practice of allowing a user to select a menu item by typing a single letter from that item that uniquely identifies it. In the example in figure 2, these uniquely identifying letters are marked with underlines.

After a menu is translated, the accelerator letters need to be changed to fit the new language. You need to ensure that the software engineer has allowed for this. In an enabled program, the translator is given some convention for marking the accelerator letter of each translated item. The enabled program must be written to look for whatever letters were marked as accelerators, rather than simply associating *S* with the *Save* option, for example.

Character Sets

Let me conclude with a semitechnical discussion on how character sets are used to store and display text strings on a computer. A character set is the collection of all the letters (both lower and upper case), digits, punctuation marks, mathematical symbols, graphics and other miscellaneous characters that a computer has at its disposal to store and display. (Graphics characters are used to draw lines and boxes on the screen, and come in a wide variety of vertical and horizontal line segments, corners, and junctions: thin and thick; solid and gray; single and double, etc.) As a localizer, you will have little control over the process of storing and displaying characters, but knowing the issues will help you understand the common problems associated with text display in internationalized versions of software programs.

IBM code pages, used in DOS computers, are a common example of character sets. Because of long-standing design limitations, each code page only contains 256 characters. This is enough to hold all of the letters and other characters and symbols for one or even several alphabetic languages, but it is not big enough to hold all the characters needed for all of them. Therefore, if you want to switch from English to Danish, for example, you need to switch code pages. Code page 437 holds all of the English characters, plus a wide range of symbols and graphics characters. Code page 850 includes all of the letters (including letters with diacritics) for most of the European languages including Danish, but leaves out many of the graphics characters in code page 437.

Switching code pages is generally an awkward process. On DOS computers, it entails rebooting the computer, and restarting all the programs that you want to run. The philosophical assumption behind this process is that code page switching will be very infrequent. Users will only want to configure their computers to their particular locale, and will only want to work in one language. This assumption is not necessarily valid in many countries, but regardless of its validity, code page switching will take place sometimes, and enabled programs must account for it as much as possible.

There are many enabling problems associated with code page switching. For example, the lack of graphics characters in code pages other than 437 causes problems when drawing boxes on the screen, as seen in figure 6 (Childs 1992). An unenabled program, assuming only code page 437, might draw a box as seen at the top of figure 6. However, when that program runs on a computer using code page 850, the box appears with garbage characters in it

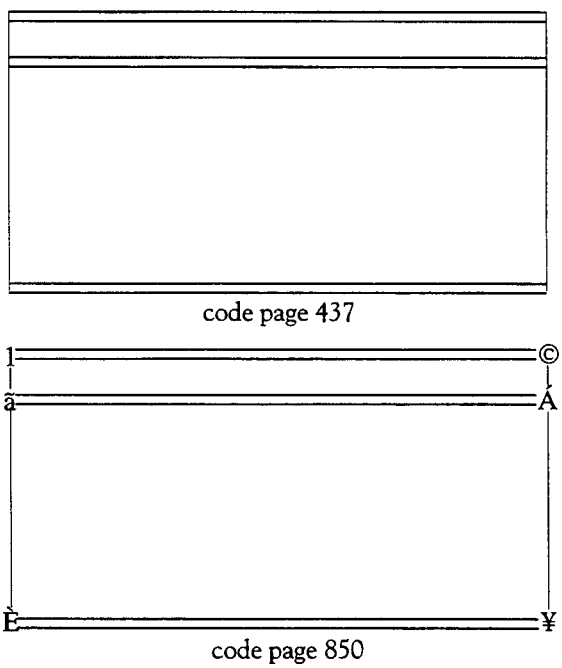

Figure 6: Graphics Characters

because code page 850 does not have graphics characters for some types of corners and junctions. An enabled program would draw boxes using only the characters available in the current code page.

Accounting for ideographic writing systems is even more difficult. The tens of thousands of characters in languages like Japanese or Chinese do not even begin to fit into a 256-character code page. The traditional solution to this problem is to introduce double-byte characters. Double-byte character is a technical way of saying that a combination of two characters from a character set is used to represent one character. Since the total number of possible character combinations is 256 x 256, there are more than enough double-byte (i.e., double-character) combinations to represent all the thousands of characters of an ideographic writing system. (In practice, only certain characters are valid in double-character combinations, so the total number of possible characters is less than 256 x 256.) The problem is that in Japanese, for example, text strings typically contain a mixture of single and double-byte characters. It is a tremendous chore for the computer hardware and the software to sort out whether an individual character should represent itself, or whether it is part of a double-character combination that represents quite a different character altogether. Needless to say, designing software to run in both Oriental and Western locales is a difficult task.

Today, new standards are emerging for large character sets that hold all characters. A character set standard called Unicode seems to be emerging as the de facto world standard among hardware manufacturers and software developers. The Unicode character set contains over 65,000 characters, which means it can represent the characters of all the world's writing systems without character set switching. When such a standard finally becomes universal, the current character display problems in internationalized software will be a thing of the past.

Conclusion

The field of software internationalization is growing by leaps and bounds. Software localizers are in ever increasing demand. If you are interested in enabling or localization, now is the time to start preparing. This paper should provide a road map for further study.

References

Axtell, Roger E. 1991. *Gestures: The Do's and Taboos of Body Language Around the World*. New York: John Wiley & Sons.

Childs, Larry. 1992. "Software Internationalization: A Programming Perspective for the Non-programmer." In *Proceedings of the 33rd Annual Conference of the American Translators Association*. Medford, NJ: Learned Information.

National Language Design Guide. 1991. 4 vols. North York, Ontario, Canada: IBM National Language Technical Center. IBM part numbers for volumes 1-4: SE09-8001-01, SE09-8002-01, SE09-8003-00, SE09-8004-00. Ordering information: "Requests for IBM publications should be made to your IBM representative or to the IBM branch office serving your locality."

Taylor, Dave. 1992. *Global Software: Developing Applications for the International Market*. New York: Springer-Verlag.