>>> revista tradumàtica
Traducció i Tecnologies de la Informació i la Comunicació
Número 1 : La Localització : Octubre 2002 : http://www.fti.uab.es/tradumatica/revista

Internationalisation

By Feliciano Donoso
Bowne Global Solutions, Dublin office

Resumen
El autor resalta la internacionalización como factor clave para tener éxito en la localización de un producto. En este sentido el artículo empieza con una introducción al tema a través de un repaso de los diferentes componentes que son objeto de trabajo en el proceso de internacionalització y, destaca también los diferentes niveles de internacionalización a que puede ser sometido un producto.

Palabras clave
Localización, internacionalización, gestión de proyectos.

What is Internationalisation?

Internasionalisation provides the framework and structure in which localisation takes place more easily and efficiently. During development, the software is designed in a way to allow for translation into other languages without the need for re-design or re-compilation. It is often abbreviated to "I18N", where "18" indicates the number of letters betweern the "I" and the "N".

Internasionalisation is important for the following reasons:

- Enable original products to be sold worldwide.
- Faster time to market for a localised software product. Once the product is released all the international requirements are met. Localisation can be started in parallel with the product development cycles.
- Consumer fewer resources, time and money for localisation. Adding international support after the original version is released requires an in-line version. An in-line version requires a re-certification of the original functionality all over again in addition to the international features. This means that the original versions' quality assurance effort cannot be fully leveraged. This means more resources, more time and money are spent to deliver the localised version of the software.

An important aspect of internationalisation is the separation of text and source code. Translatable, user-visible text should be moved to separate strings-only resource files that will prevent translators from translating, or breaking, the program code. Single sourcing of code is possible and thus maintenance is easier and less expensive. If the international requirements are done up front, no in-line version or branching of the source code is necessary. Maintaining a single source code is much cheaper in terms of bug fixing and adding new features.

Example: Localisation of a Java application
Let's see a practical example of how we could localise a Java application and the problem of 'hard-coded resources'

- Java application not internationalised

The application to be localised is a very simple one it only displays three messages, as follows:

```
public class example1 {
        static public void main(String[] args) {
                System.out.println("Hello.");
                System.out.println("How are you?");
                System.out.println("Goodbye.");
                }
}
```

Let's say that we would like to translate this application into Catalan. At this stage the developer or software engineer has two options:

1. The first option will be to send this file as it is, to the translators and ask them to modify the English text, creating a new source code. Since the translators are not programmers the risk to corrupt the code of the new translated applications is very high. In fact this application is only for orientate purposes, what could happen with a real application with hundreds or maybe thousand of messages, menu options, dialog boxes, etc.?
2. The other option is that the SW engineer would make a copy of the source code for Catalan, extracting manually from the code every piece that should be translated, copy into a spreadsheet (using any file editor) and send it to the translators. Once the translators have sent back the spreadsheet with the translated strings, it must be copied back manually to the "new" source code, recompile everything and wait that during all the process no mistake has been done. Of course this process even though is better than the previous one is absolutely slow and inefficient.

It seems that it will be interesting that the messages should be out of the source code and into text files that the translators can edit. Also, the program must be flexible enough so that it can display the messages in other languages, rather than only the ones that we were supposed to translate.

- Java application internationalised

In this case the messages are separate from the source code, making things easier in order to localise the program for different markets.

```
import java.util.*;

public class example2 {

        static public void main(String[] args) {

        String language;
        String country;

                if (args.length != 2) {
                        language = new String("en");
                        country = new String("US");
                } else {
                language = new String(args[0]);
                country = new String(args[1]);
        }
```

```
Locale currentLocale;
ResourceBundle messages;

currentLocale = new Locale(language, country);
 messages = ResourceBundle.getBundle("MessagesBundle",currentLocale);
System.out.println(messages.getString("greetings"));
System.out.println(messages.getString("inquiry"));
System.out.println(messages.getString("farewell"));
}
}
```

To compile and run this program, the following source files are also needed:

MessagesBundle.properties

```
greetings = Hello.
inquiry = How are you?
farewell = Goodbye.
```

MessagesBundle_ca_ES.properties:

```
greetings = Hola.
inquiry = Com estàs?
farewell = Adéu.
```

The internationalised program is flexible; it allows the end user to specify a language and a country on the command line. In the following example the language code is ca (Catalan) and the country code is ES (Spain), so the program displays the messages in Catalan:

```
% java example2 ca ES
Hola.
Com estàs?
Adéu.
```

In the next example the language code is en (English) and the country code is US (United States) so the program displays the messages in English:

```
% java example2 en US
Hello.
How are you?
Goodbye.
```

At this stage, only creating copies of the properties files for every language needed can do the localised versions of these programs. Properties file stores information about the characteristics of a program or environment. A properties file is in plain-text format. That means that the translators only have to modify these properties files.

In the example the properties files store the translatable text of the messages to be displayed. Before the program was internationalised, the English version of this text was hard coded in the System.out.println statements. The default properties file, which is called MessagesBundle.properties, contains the following lines:

```
greetings = Hello
inquiry = How are you?
```

```
farewell = Goodbye
```

Now that the messages are in a properties file, they can be translated into various languages. No changes to the source code are required. A French translator, for instance can create a properties file called MessagesBundle_fr_FR.properties, which contains these lines:

```
greetings = Bonjour.
inquiry = Comment allez-vous?
farewell = Au revoir.
```

Notice that the values to the right side of the equal sign have been translated but that the keys on the left side have not been changed. These keys must not change, because they will be referenced when the program fetches the translated text. Also these keys can be used for the translator as a context reference.

Components of Internationalization

As it has been explained before the key to software internasionalisation is not to hard-code any of the following components:

- User Interface – UI

  The most obvious part of an international environment is the text that end-users see. This includes error messages, help text, menus, prompts and graphics. All the text should appear in the user's native language. This text should be resourced outside the source code. Consideration must be given to the growth of and spacing between items, since translation generally expands text by 20%. Expansions can occur both vertically and horizontally, because Asian fonts can be both horizontally and vertically and larger than their western counterparts. The result is that the translated text may not fit the space originally provided.

  Printed documentation is another critical component of the product. Documentation should be internationalised as much as the software components. Translation is much easier when the documentation does not have base country specific idioms or metaphors.

  The graphical interface should have the same consideration as text. Icons should be culturally neutral. Any text in icons should be resourced separately from the graphic image itself.

- Classification

  Different languages classify characters differently. English classifies only 52 characters as "alphabetic": [A-Z] and [a-z]. Any other characters are not considered alphabetic characters, which poses a problem in Europe where accented characters such é, ä, ô, to name a few, are part of day to day life. In Asian languages such as Japanese and Chinese, almost all of the characters used are phonetic and ideographic, not alphabetic. Not allowing phonetic and ideographic characters in Asia is not an option. The classification (isalpha, islower, issupper, etc.) must be language definable. In addition, there needs to be a mechanism to define additional classifications not present in the English language (for example isdiacritical, isdbcs, isphonetis, isidiogram).

- Transliterations

  On most systems, the C programming language provides the macros toupper(c) and tolower(c) to perfom case conversion from upper to lower and vice versa. However, these macros are often biased toward the English language and ASCII code page. In ASCII, lowercase and uppercase alphabetic characters have the same relative ordering. This is not true for European or Asian code pages.

  The table must be read in at runtime and must be user definable. It must be able to handle Asian languages as well, the array of characters has to be larger than 256 characters.

- Numerical formats

  Numerical formats differ from one country to another. The representation of number varies with respect to the symbol indicating the radix character, which is the separator between the integer and the radix character (separating the fractional portion of the number) and the digit grouping symbol. For instance, the US and Germany both represent numbers using decimals and commas, but the symbols are interchanged (5,434.25 versus 5.434,25). The radix and digit-grouping characters are not restricted to just "." and "," - countries like France may use a blank character as the digit-grouping symbol - 5 434,25

- Monetary formats

  Monetary formats also vary from one country to another. The radix and digit-grouping symbols are usually the same as those used in the numerical format. In addition the currency symbol and its position relative to the monetary value changes from one country to another. The currency symbol for the US is "$" and it precedes the amount, as in $5. In France, the currency symbol is "FF" and it follows the amount, as in 5 FF. In countries such as Portugal, the currency symbol is placed in the middle of the amount separating the country´s whole and fractional units. The introduction of the euro "€" it will make easier to approach this problem for the software developers. However it is interesting to think about the huge amount of time that could be saved in the euro-conversion process, if all the software would have been internasionalised properly in the past.

- Date and time formats

  Most Western countries use the Julian calendar. Asian countries often use multiple date formats. Japan uses both the Julian calendar format and a format based on the number of years the emperor has reigned. However, the Julian calendar is fully recognised and used in software applications. Taiwan and China use a date format based on the number or years since the start of the current era, which typically stars and ends with ruling authority during that time frame.

  Even limiting the discussion to the Julian calendar, date formats differ from one country to another. In the US the month precedes the day, in Western Europe the day precedes the month, in Sweden the year precedes the month followed by the day.

- Collations

  Sorting in most applications means sorting in the codepoint order, which is in the ASCII code page order. The ASCII sorting page is not correctfor foreign languages, nor is it sufficient for the American dictionary ordering, as all uppercase letters sort before the lowercase letters. This means that *a* sorts after *B* which is not correct.

The sorting conventions of foreign languages are in general more complex than the sorting convention Americans are accostumed to. Non-English alphabetic sorting includes:

- 1-to-2 character mappings (ß sorts as *ss* in German)
- 2-to-1 character mapping (*ch* sorts between *c* and *d* in Spanish)
- Primary sort, seconday sort and so forth

The above mentioned collation properties apply to alphabetic languages. Asian ideographic characters on the other hand, represent concepts, for which the western notion of sorting is not sufficient. However, Asian languages can be sorted by many methods. For Kanji, the preferred method is phonetic as opposed to sorting by the number or radicals or strokes that make up a character.

- Regular expressions

  Often overlooked but associated with collation are regular expressions, which are patterns used for searching text in data. In some languages there are collating elements that are made up of more than a single character, i.e. *ch* in Spanish. The regular expression [a-ch] will not work, as it means a, b, c or h. Additional regular syntax is needed.

  Regular expressions need to be extended for multi-byte character sets. For multi-byte character sets, pattern matching must be done on a character basis regardless of the character size in bytes.

- Encoding schemes

  An encoding scheme illustrates the one-to-one mapping between a character and the computer's bit representations. The most common code page is ASCII. ASCII is a 7-bit code page capable of representing 128 characters, which is sufficient for the English language. The ISO 8859-1 code page is more commonly used. The ISO 8859-1 is an 8-bit code page capable of representing English and European (western and eastern) character sets. Neither the ASCII nor the various sub-sets of ISO are capable of representing Asian idiographic characters. In order to accurately represent these characters, the Unicode code page is used. Unicode is a true double-byte code page.

- Character versus byte

  Contrary to most developer's belief, they produce code that is byte oriented rather than character oriented. This is because for most developers, a byte and a character are interchangeable, because in ASCII every character is represented by one byte. This does not hold true in the international area, where incrementing the character pointer by one may or may not get one to the next character in the string.

  Multi-byte is associated with problems and considerations that do not exist with single-byte character:

  - Truncation and splitting of characters
  - Wrapping and splitting of characters
  - Window borders and splitting of characters
  - Regular expressions and patter matching by characters, not bytes
  - Memory size versus display size
  - Variable size cursor movements and editing, insertion and deletion based on characters

- Text directionality

  Text directionality is only an issue when dealing with Hebrew and Arabic texts, which are written from right to left. The Western display method (left to right) has been widely accepted in Asia for the input and representation of the Asian idiographic characters.

- IMEs

  Kanji characters sets include thousands of characters, which cannot be accommodated by a keyboard. Asian operating systems as well as many third-party companies provide software-driven input method editors (IMEs). The software-driven IME provides the user with the capability to enter thousands of different characters. Most of these IMEs are based on phonetic conversion.

How to achieve a real internationalized software

One key ingredient in such strategy is the initial preparation of the source materials. Making the proper investment up-front in internationalising the software will result in the reduction of time-to-market and the cost of the localization.

Some guidelines in order to achieve this goal can be:

- Define the target languages
- Double-byte enable the software
- Establish software development guidelines with internationalization in mind
- User driver settings form the operating system to facilitate local time & date formats, keyboard layout, etc.
- Allow for the input of international data.
- Support mixed formats for addresses, measurements, numbers, time and dates
- Change case properly
- Take sorting into account
- Put all localizable items in external resource files
- Allow for text expansion; make buffer and UI large enough to hold translated text
- Define hot keys and accelerator keys
- Use comments with code to provide context for translators
- Test the English version´s internationalization features
- Confirm that any third-party components are internationalized.

On the other hand some aspects that never must be done in order to get a relative good quality in localised software would be the following:

- Don´t make assumptions about standard fonts
- Concatenate strings to form sentences
- Concatenate sub-screens to form whole screens
- Put text in icons; make them generic enough for an international software
- Hardcode keyboard commands
- Hardcode screen placement or the size of items on the screen

Internasionalisation / Localisation Levels

A term that is often used in the context of internationalization is *enablement*. Enablement is the process of adjusting software to make it usable in certain countries or regions. For instance, a software product can be *enabled* to process text used in Far-Eastern countries. Therefore, depending on the target markets different localisation levels are developed.

- English product only

  One option is not to enable or localise the software product at all. The result is that the software is an English product only, that can be sold in the US, UK, Ireland, Australia, New Zealand and possibily Singapore. The European market is only accesible via US subsidiaries in which the US parent company chooses for an English language product to be used. The product is however not only in English in language, but in functionality, i.e. it is 7-bit, shows US date formats, currency and sorting collations, does not handle European characters such as é, ç, etc.

- English product handling European data

  At this level, the product is not localised, i.e. not translated and the screens and user interface remain in English. However the product is enabled to handle most single-byte character sets used in European countries. Additionally the product is enabled to handle the cultural conventions used in European countries (date formats, currencies, sorting collations).

- English product handling Far-Eastern data

  Enabling software to process text used in Far-Eastern countries like Japan, China, Taiwan and Korea is more complicated than enabling it for use in Europe. It requires changing the logic of the source (generally US) product and could require re-architecting the product altogether. Enabling for the Far-East might require architecting the software for hardware independence as well.

- English product handling bidirectional data

  In addition to enabling the software to handle different languages, there is also the need to translate the user-interface and printed documentation. Most end-user oriented products require full translation of the user-interface and documentation. Business application type products are end-user oriented. In the ideal world, the end-user will never suspect that the product was developed in another country.

- Full localisation plus local market features

  The ideal product for a local market is not only fully localised, but also has local market-specific features. Specifically in the case of Financial and HR applications, an integral part of the localisation process is ensuring that the local legal reporting requirements are met. In addition to specific functional requirements, additional technical features may also be required, specifically in the Far-East.

Conclusion

Internationalisation is a key factor in localisation. A fully internationalised application makes translators' work faster, more effective and accurate, as they don't need to take care of any development related issues. SW engineers would be only concentrated in the technical issues while translators would be in the linguistic ones. Therefore, the overall localisation process becomes easier and of a higher quality. The ability in delivering localised products of a high quality is a critical factor in the success or failure of a software product worldwide

Bibliography

Esselink, B. (1998): A practical guide to software localization. Amsterdam, Benjamin Publishing Company

Internationalisation tutorial http://java.sun.com/docs/books/tutorial/i18n