Observations and Recommendations on the Internationalisation of Software

David Barnes
The Computing Laboratory
The University
Canterbury
Kent. CT2 7NF

June 22, 1992

Abstract

As computer programs enter the lives of more and more people worldwide, it is becoming increasingly unacceptable to assume that software with a user interface designed for an indigenous English speaking market will be acceptable outside its country of origin simply by changing the currency symbol. Developers of software who are serious about expanding sales into new markets must consider many issues when giving thought either to the creation of new software or the modification of existing software to work within the linguistic and cultural constraints of these new markets.

The purpose of this paper is to examine the task of preparing software to be used in countries and cultures other than that in which it is created. We do this by reviewing some of the most important localisation issues that have been identified, and some of the tools and practices that are available to the software designer to deal with them. We shall also consider some of the areas of the software development process that are currently less well understood and supported. Our major emphasis is in non-graphical applications targeted at European markets.

Keywords: Internationalisation, I18N, Localising, Enabling, Multi-lingual.

Part I

Cultural Differences

1 Text Strings

For developers intending their software to interact with human users in different cultures, the most obvious obstacle to be overcome is one of language. Ideally, all textual communication between the program and the user should take place in his or her own language.

1.1 Text output by the Program

Text output by a program often constitutes a large proportion of the user interface: command prompts, items in a menu, error messages, help, general information, and so on. Such text that would be presented by a program must be identified, and an equivalent translation prepared that can be used as the output of a localised program. This process is often mistakenly thought of as being both relatively simple and the only significant effort required in the migration process. However, it is our contention that this purely 'string-handling' view of the task is quite mistaken, and will often actually require little effort when compared with that required to deal with some of the more subtle cultural and dynamic aspects of a program's behaviour. Rather, it is the task of maintaining the cultural elements in a form that will enable a qualified translator to create a satisfactorily localised program that is one of the major hurdles to be overcome in the area of internationalisation.

Nevertheless, in later sections we shall discuss various methods for achieving the identification, separation, and translation of program text as a necessary first step.

1.2 Text supplied by the User

Where a user interacts with a program via textual responses, handling of such input must be isolated and appropriate translations found, as with output text. On input, there is a slightly different set of considerations; the layout problems associated with output become issues of handling multi-format input, for instance.

2 Character Set

Migration to most languages other than English requires the use of a character set other than the commonly used 7-bit ASCII. For the most common European languages it is sufficient to employ the 8-bit ISO-8859-1 (ISO Latin-1) but for migration to other markets, most notably Japan, it will be necessary to consider the implications of multi-byte character sets, such as *Kanji*.

Typically, a localised application will be used with only a single interface at a given site, where keyboards and output devices will be appropriate for that locale and in the rest of this paper we shall concern ourselves with this scenario. However, multi-lingual working environments do exist, and here it may be necessary to make adequate provision for the generation of input and output not typical of that locale – such as generating accented characters from a UK keyboard [2].

3 Collating Sequence

Many applications rely upon the underlying values within a character set to impose an ordering on alphabetic data. This will often fail to take account of the distinctive ways in which some countries properly order such data. Del Galdo [4] shows the collating sequences for English, German, and Swedish, noting that 'ä', 'ö', and 'ü' in German are grouped with their unaccented equivalents, whereas Swedish groups 'ü' with 'y' and places 'ä' and 'ö' at the end of the alphabet. Sprung [15] adds that sorting 'o' before 'ö' and vice-versa are both common in German. Some languages sort particular pairs of letters as a singleton, 'ch' and 'll' in Spanish after 'c' and 'l', respectively, for instance.

4 Character Classification and Conversion

Many applications need to classify characters into groups, such as upper case, lower case, digits, white space, and so on. Such classifications need to recognise the place of accented characters within such groupings. Different cultures also have different conventions about whether a character retains it accent when converted to upper case [17, p.124], and it cannot be assumed that simply 'flipping a bit' will convert between upper and lower case.

5 Hyphenation

Writers of text formatters recognise that hyphenation rules are often highly specific to individual languages [10, p.88]. In addition, sometimes changes in spelling are required when a word is broken [15, p.83], [17, p.124].

6 Monetary Information

Currency symbols vary from country to country, as do their lengths and positions, for instance: £23.45p, 12FF, US$13, etc. Formatting of monetary items must take this into account.

7 Numbers

Separators for thousands and decimals vary considerably. Del Galdo [4, p.4] suggests allowing for thousands a comma, period, space, apostrophe, and no separator. In addition a period, comma, and centre dot should be allowable as radix characters. These issues affect both the output of information and what the user is allowed to supply as input. It is essential that the conventions are made clear to the user and that sanity checks, feedback, and opportunity for confirmation provide the necessary safeguards against incorrect data.

8 Dates

Numeric day/month ordering is a potential source of error, particularly where the day is within the range 1-12. Differences between the UK day/month/year and the US month/day/year orderings have even provided a fertile ground for clues within detective fiction! A third common form of year/month/day offers an acceptable alternative, as long as four digits are used for the year as after the year 2000 two digit years will render this form indistinguishable from the others. Where ambiguity might arise, an alphabetic month with four digit year is always to be preferred on output, and user supplied numeric dates should be confirmed.

Further considerations are found in Taylor [17, p.124], who notes that the Western Gregorian calendar is by no means ubiquitous, and in Beyls [3, p.258] who describes some of the unique demands of the Arabic lunar calendar.

9 Time

Although the ordering problem associated with dates does not arise with time formats there is a need to clearly distinguish between twelve and twenty-four hour representation; AM and PM strings may be needed, for instance. Where the time displayed is not local time, or the context demands it, reference to a recognised time zone should be included. Del Galdo [4] suggests that typical three letter abbreviations of this information are not appropriate.

10 Units

Not all countries regularly use SI (Système International) units. In the UK, where the metric system has been part of the education system for a long time, the older Imperial system of miles, gallons, pounds, and ounces, still flourishes in every day life. In addition, there are differences to be found between countries with similarly named units – pints and billions between the US and UK, for instance. Failure to agree on the right units could have life threatening consequences, typified by the notorious failure to distinguish between land miles and nautical miles during WWII [9].

11 Telephone Numbers

A great deal of flexibility is required in handling telephone numbers – particularly in recognising what is acceptable as input. Numbers of digits vary, as do the typical separators and lengths of groups within a number. Provision should be made to accept abbreviated local forms, as well as full national and international formats, where appropriate.

12 Names and Addresses

As with telephone numbers, no standard format can be assumed to be available. Some countries order the most local information first, others order it last. Names are also an area where care must be exercised, and can be a source of much personal annoyance [14]. Provision should be made for people with single letter names, hyphenated names, no forenames, and so on.

Nielsen[11, page iv] points out that simply initials and surname are insufficient to identify a person in Denmark since three surnames alone account for nearly 25% of the Danish population.

13 Things that Niggle

A program with an interface developed in one country and then made available in another where the same language is spoken might not be an obvious candidate for the skills of a translator, but seemingly minor cultural differences can create a sense of unease that may significantly affect a user's perception of whether or not an interface really has been tailored to their needs. Examples between the UK and US markets might be the different spelling of common words (such as colour/color), the use of different terms for the same thing (bonnet/hood), as well as the date ordering problems that have already been mentioned.

Part II

Support for Internationalisation

1 Where it happens

The multiplicity of examples within the literature demonstrates that there is no single accepted method for creating and maintaining a piece of software in a form that will enable its adaptation to national variations [2, 5, 6, 12, 13, 17]. In addition, there are different points within the source or binary form of a program where diversification might take place. Taylor [17] identifies all three phases of program building – compile-time, link-time, run-time – as being potential points for this. To illustrate the essential differences between these points we shall use the program fragment in Figure 1, which is written in ANSI C [1].

The purpose of the function is to prompt the user with a question and return '1' or '0' for a 'Yes' or 'No' response. Our goal is to write a French language version that will do the same thing. We shall concentrate on the purely textual aspects of the differences.

1.1 Compile-Time

With this approach, no significant attempt is made to isolate the language specific elements of the program from the rest. A French version might look like that in Figure 2, therefore.

The position usually taken is that the compile time solution is impractical for all but the simplest of applications. Its use results in having to maintain multiple versions of what is, essentially, the same software. Bug fixes and enhancements in one version must be carried over to all others and it is very easy for even just two versions to eventually diverge to the extent that they become different pieces of software. Nielsen, however, does give an example of a commercial application that used this approach at one stage [12, p.160].

```
int
ask_a_question(const char *question)
{   char *answer;
    /* Keep going until we find a valid answer. */
    int answer_found = 0;
    int result;
    /* Prototypes of functions called to break down the task. */
    char *read_a_reply(void);
    int samestring(const char *,const char *);

    /* Repeatedly ask the question until either Yes or No is given. */
    do{
        /* Ask the question. */
        puts(question);
        /* Accept a reply from the user. */
        answer = read_a_reply();
        if(samestring(answer,"Yes")){
            answer_found = 1;
            result = 1;
        }
        else if(samestring(answer,"No")){
            answer_found = 1;
            result = 0;
        }
        else{
            puts("Please reply Yes or No.");
        }
    } while(!answer_found);
    return result;
}
```

Figure 1: English version of `ask_a_question`

```
int
ask_a_question(const char *question)
{   char *answer;
    /* Keep going until we find a valid answer. */
    int answer_found = 0;
    int result;
    /* Prototypes of functions called to break down the task. */
    char *read_a_reply(void);
    int samestring(const char *,const char *);

    /* Repeatedly ask the question until either Oui or Non is given. */
    do{
        /* Ask the question. */
        puts(question);
        /* Accept a reply from the user. */
        answer = read_a_reply();
        if(samestring(answer,"Oui")){
            answer_found = 1;
            result = 1;
        }
        else if(samestring(answer,"Non")){
            answer_found = 1;
            result = 0;
        }
        else{
            puts("Répondez avec Oui ou Non, s'il vous plaît.");
        }
    } while(!answer_found);
    return result;
}
```

Figure 2: French version of `ask_a_question`

1.2 Link-Time

This approach consists of isolating into distinct modules those cultural elements that require translation. When a program is prepared for a new locale, new versions of only these modules need to be prepared. These are substituted for the original locale-specific modules and linked in with the common modules of the application.

Following this method, our example must be split into two: a shared part (Figure 3), and a locale-specific part (Figure 4).

The locale-specific part is simply a data structure, `question_strings` holding the possible answer strings and the error message.

Access to this information is granted to the shared part via the module interface: `question.h` that is included in both parts (Figure 5).

The French version of the language-dependent module is shown in Figure 6.

1.3 Run-Time

This method involves the complete removal of the cultural elements from the source of the program to external files. Whereas the other two methods produce a different binary version for each interface, this method produces a single binary whose interface will depend upon the external environment in which it is run. The external files containing the environmental information are variously known by designations such as resource files [12], message files [6], and catalogs [5, 17]. In their simplest form they contain the texts of the different user interfaces. Following the conventions of [5] the message files from which the catalogs of our example are created might look like those in Figure 7 and Figure 8.

Catalogs are prepared from these message files, and at run-time, the required catalog is opened for that locale. The language-dependent strings are then retrieved via calls to the `catgets` function, and our `ask_a_question` function would look something like that in Figure 9.

The run-time approach is usually associated with a degradation in execution speed, when compared with the other two approaches, because of the need to access external data. This may be an important factor, especially if an existing base of users would find a significant decrease in speed unacceptable [13, p.105]. However, if it is possible to load a heavily used catalog into memory at runtime then the degradation may not be a factor [2].

2 Existing Support for Internationalisation

As we saw earlier, there is a good deal more to localisation than simply translating text from one language to another. Several vendors have addressed many of these issues and provided support for them. DEC Ultrix NLS, Hewlett-Packard HP-UX NLS, IBM and Microsoft's OS/2, and Microsoft's Windows are all examples that do so to a greater or lesser extent. The areas covered typically include the infra-structure facilities listed below.

```
#include "question.h"

int
ask_a_question(const char *question)
{     char *answer;
      /* Keep going until we find a valid answer. */
      int answer_found = 0;
      int result;
      /* Prototypes of functions called to break down the task. */
      char *read_a_reply(void);
      int samestring(const char *,const char *);

      /* Repeatedly ask the question until an appropriate answer is given. */
      do{
            /* Ask the question. */
            puts(question);
            /* Accept a reply from the user. */
            answer = read_a_reply();
            if(samestring(answer,question_strings[YES_STRING])){
                  answer_found = 1;
                  result = 1;
            }
            else if(samestring(answer,question_strings[NO_STRING])){
                  answer_found = 1;
                  result = 0;
            }
            else{
                  puts(question_strings[ERROR_STRING]);
            }
      } while(!answer_found);
      return result;
}
```

Figure 3: Locale-independent shared components

```
#include "question.h"

const char *question_strings[ ] = {
      "Yes",
      "No",
      "Please reply Yes or No.",
};
```

Figure 4: Locale-specific data structure for the English version

```
/* Provide access to the language-dependent strings. */
extern const char *question_strings[ ];

/* Define the index for each string in question_strings[]. */
#define YES_STRING 0
#define NO_STRING 1
#define ERROR_MESSAGE 2
```

Figure 5: Module interface to the locale-specific information (`question.h`)

```
#include "question.h"

const char *question_strings[ ] = {
    "Oui",
    "Non",
    "Répondez avec Oui ou Non, s'il vous plaît.",
};
```

Figure 6: Locale-specific data structure for the French version

```
$english_question.msf
$Messages for the English version of the ask_a_question function.
$quote "

$set ASK_QUESTION
YES_ANSWER "Yes"
NO_ANSWER "No"
ERROR_MESSAGE "Please reply Yes or No"
```

Figure 7: English message catalog

```
$french_question.msf
$Messages for the French version of the ask_a_question function.
$quote "

$set ASK_QUESTION
YES_ANSWER "Oui"
NO_ANSWER "Non"
ERROR_MESSAGE "Répondez avec Oui ou Non, s'il vous plaît."
```

Figure 8: French message catalog

```
#include "nl_types.h"
#include "question.h"

int
ask_a_question(const char *question)
{    char *answer;
     /* Keep going until we find a valid answer. */
     int answer_found = 0;
     int result;
     /* Prototypes of functions called to break down the task. */
     char *read_a_reply(void);
     int samestring(const char *,const char *);
     /* A catalog descriptor is needed. */
     nl_catd catd;

     /* Open the catalog for the strings required. */
     catd = catopen("question.cat",0);
     /* Repeatedly ask the question until an appropriate answer is given. */
     do{
          /* Ask the question. */
          puts(question);
          /* Accept a reply from the user. */
          answer = read_a_reply();
          if(samestring(answer,catgets(catd,ASK_QUESTION,YES_STRING,""))){
               answer_found = 1;
               result = 1;
          }
          else if(samestring(answer,catgets(catd,ASK_QUESTION,NO_STRING,""))){
               answer_found = 1;
               result = 0;
          }
          else{
               puts(catgets(catd,ASK_QUESTION,ERROR_STRING,""));
          }
     } while(!answer_found);

     catclose(catd);

     return result;
}
```

Figure 9: Locale-independent version of `ask_a_question`

- run-time access to locale-specific text,
- character sets, including multi-byte characters,
- collating sequence,
- character classification,
- monetary format,
- numeric format,
- date and time format.

These provide a good base upon which to develop the higher-level aspects of internationalisation that we shall be dealing with in Part III.

3 Enabling Tools

Many software developers will want to take an existing program and convert it for use in other cultures. In order to facilitate this, some tools have been developed to eliminate much of the repetitive nature of part of this task. The Ultrix and HP-UX NLS [5, 8] have a great deal in common, both leaning towards the X/Open Portability Guide [18]. Both provide broadly similar facilities to extract text from existing software into text message files, from which the run-time catalogs are then created. Ultrix NLS provides both an interactive program, `extract`, and a batch program `strextract`. These tools make use of directives to determine which strings to match and which to ignore during this process. When a string is matched and placed in the message file, its occurrence in the source can be automatically replaced by the appropriate call to retrieve it from a catalog at run-time. For example the line in Figure 1 above

if(samestring(answer,"Yes")){

would be converted automatically to something of the form

if(samestring(answer,catgets(catd,set_number,item_in_set,"Yes"))){

by the extraction process. The values in the places `set_number` and `item_in_set` are simply numbers allocated by the tool. Their purpose is to allow catalogs to be organised into different sets of messages. The final argument to `catgets` is a default string in case the requested value cannot be retrieved, for some reason. It serves the additional purpose of documenting the code. Peterson's method [13] retains the original message text for this purpose, too, but instead of the set/item principle he uses the original text as an argument to a hashing function in order to retrieve the translated version from an external file.

It is important to note that the service offered by such tools as these is simply a cut-and-paste exercise. The tools only save the programmer from a large amount of hand editing. No attempt

```
const char *menu_strings[ ] = {
     "Edit",
     "List",
     "Print",
     "Quit",
     "Save",
};
```

Figure 10: Command menu data structure

```
const char *menu_strings[ ] = {
     catgets(catd,1,1,"Edit"),
     catgets(catd,1,2,"List"),
     catgets(catd,1,3,"Print"),
     catgets(catd,1,4,"Quit"),
     catgets(catd,1,5,"Save"),
};
```

Figure 11: Incorrectly enabled command data structure

is made to logically group the items into different sets, or provide symbolic names for sets and items, for instance. In addition, the tools contain no understanding of where these replacements are inappropriate. Faced with the data structure in Figure 10 that might represent a menu of commands to be presented to the user, the Ultrix NLS `extract` tool can only convert this into an equivalent of Figure 11.

This is no use because data structures in C may not be initialised in this way. Rather, what is needed is an additional function that performs the run-time initialisation (Figure 12).

The reason that existing tools do not handle this situation properly is that, typically they are little more than pattern-match and replacement utilities; they do not take account of the program context in which the text occurs.

Part III

Exploring the Difficulties

Having examined some of the areas for which there is existing support it is now necessary to look at some of the areas that do not yet have ready made solutions.

```
/* Define how many menu items there are. */
#define NUM_MENU_ITEMS 5
/* Define an array to hold the menu items. */
char *menu_strings[NUM_MENU_ITEMS];

/* Define a function to initialise the menu. */
void
init_menu(void)
{
    int item_number;
    /* A function to make a copy of each string read. */
    char *copy_string(const char *);

    for(item_number = 0; item_number ¡ NUM_MENU_ITEMS; item_number++){
        /* Copy each string read into some new space. */
        menu_strings[item_number] =
                copy_string(catgets(catd,1,item_number+1,“”));
    }
}
```

Figure 12: Function necessary to initialise the command data structure

1 The Goal

The ultimate goal of the software developer who is working towards a localised product must be to have the new users feel as comfortable with it as with a home-grown product. Nielsen says,

> “an interface which is used in another country than the one where it was designed is a *new* interface ...translating the text strings and icons in the software is not enough – one has to translate the *total user interface* including manuals” [11, p.39].

Implementation of such a goal, with a commitment to the design of the total user interface, will require the involvement of a translator who

- is a native speaker of the intended language,
- who has a good knowledge of the target user population, and
- who has a good knowledge of the operation of the program.

Such a combination is likely to be hard to find but it is probable that the first two requirements will be more readily met than the third, particularly if localisation actually takes place outside the development company and/or the country of origin. If this is so then the development and enabling process must provide as much information as possible to support its localisation outwith its basic linguistic and cultural aspects. Unfortunately, this sort of support is completely lacking for the general developer of localised software.

2 Dynamic Context

Much of our previous discussion has focussed on the removal of text in order that it might be translated. It would be simplistic to assume that dealing with such text is analogous to the translation of a technical paper. A good accurate translation of anything depends significantly upon contextual information, and as Nielsen again says,

> "dialogue elements cannot be seen in isolation since they form part of a dynamic interactive context" [11, p.42].

However, the text extraction process that we have looked at does precisely this – removing text to a static external file devoid of all dynamic context.

As an example, Nielsen's analysis of an initial Danish version of *MacPaint* shows how, when seen in context, the translation of `eject` as `aflever` ('hand-over') caused a user to think the button's purpose was for saving a file, rather than ejecting a disc. A similar problem can exist when translating text representing toggles; without giving the translator the context, it might not be apparent that two isolated texts should be translated as opposites so that the linkage is clear to the user [11, p.41]. Sukaviriya and Moran [16, p.201] point out that direct translation of some words simply does not make sense when placed in context – `box` and `bar` in the Thai version of a word processor, for instance – where the direct translations refer to unrelated physical objects.

A translator, therefore, must have available information that covers the general application area, as well as the static, dynamic, and relational contexts of the text to be translated. This will often have to be directly provided by the enabler.

3 Dynamic Text

It is likely that not all text output by a program is fixed at the point of binding. Error messages typically include portions of the input deemed to have been responsible for causing the associated errors, for instance. Such dynamic text creates two challenges

- providing the translator with enough information to make the static components of a parameterised text translatable,
- providing a mechanism whereby the runtime arguments may be appropriately substituted.

The C programming language [1] uses the '%' character to introduce parameter specifications into text strings. The formatted print statement

```
printf("The %s command requires %u parameter%c",command_name,
    num_parameters,(num_parameters == 1? ' ':'s'));
```

tries to print a grammatically correct message requiring three arguments to be substituted for the parameter specifications: a string ('%s'), an unsigned integer ('%u'), and a single character ('%c'). What the translator is likely to see is simply

```
The %s command requires %u parameter%c
```

Is this sufficient to yield something with the equivalent meaning in the target language? Certainly the translator will need to know what the various parameters mean, and likely values of the runtime arguments.

Furthermore, translations of some parameterised texts will require the order of argument substitutions to be changed in order to make good grammatical sense. ANSI C, by itself, does not provide this capability. Hence the HP-UX NLS environment extends the range of parameter specifications to include those of the form '%`d`$', where `d` is a digit in the range 1-9 that refers to the required argument's position in the argument list. These allow arguments to be reordered when formatted and so provide greater scope for producing a good translation. A similar feature is found in OS/2's *DosGetMessage* API [6, p.4], and INFOFLEX [12, p.173].

4 Format

One piece of advice often quoted is the need to allow for translations producing longer text than the original. The recommendation usually involves simply making sure that array sizes defined within the program are large enough to cover this possibility. How much extra space to allow depends on who you read

- allow up to 50% more when translating English [4]
- at least 60% more [8]
- for strings of 0–10 characters allow 100%-200%, for strings of over 70 characters allow 30% (with percentages for intermediate lengths) [7]

The INFOFLEX project [12] went further. Believing that fixed layouts lead to "absurd and meaningless abbreviations" they incorporated explicit formatting information along with the text held in their resource files. This enabled them to create WYSIWYG translations of screens and reports using a custom built resource editor to design them. Doubtless their task was made easier by the fact that the application was directed to 24x80 character terminals; nevertheless they successfully tackled an important issue not addressed by most other projects.

5 Menus, Macros, and Accelerators

A common feature of menus is that an option may often be selected via an accelerator – typically a keystroke combination involving the first character of its name. This can lead to awkward naming of items, even in the original version, to keep these characters distinct. A translator needs to be made aware of those places where this feature must be present in the new version. Furthermore, where these accelerator characters differ between the versions there is a danger of thwarting users who switch between them [11, p.40]. A related problem with macros in early versions of Lotus 1-2-3 is noted by Sprung [15, p.81] where US macros could not be used by French users because of different key bindings. Support for developers of software to be shared across cultures needs to be provided, therefore.

6 Program Evolution

Successful programs rarely remain static but undergo continuous improvement and development. Established user bases are likely to want to reap the benefits of improvements as soon as a new version is released in the country of origin. This means that the development process must be supported in such a way that the benefits of an original enabling are not lost, and the release of a new version to a wider market only requires the merging of translations of the new features with those that exist already. Tools that support these requirements are few. The ARRIS project built their own [13, p.107], but environments whose enabling tools effectively yield multiple versions of the same program produce potential conflicts: which version to develop – the original or the enabled [8, p.6-55]? The latter, surely, but typically the tools available are most suited for the initial enabling process, and not the long term development and maintenance of a program. Catalogs become out of date as code is changed, yet it may be unsafe to remove redundant messages lest existing translations become unsynchronised. There are no formal mechanisms for tracing the linkage between different translations.

Part IV

The Way Ahead

We have seen that there is far more to the internationalisation of a program than simply finding all embedded text strings and replacing them with alternatives in a new language. Many of the issues to be considered have to do with the infrastructure of the locale – character set, date and numeric formats, collating sequence, etc. The importance of dealing with these at the base language and operating system level has now been recognised by several operating systems manufacturers, and reasonably well catered for. However, there are at least two major higher-level areas still lacking

- The provision of a framework in which sufficient contextual information is made available to a linguistically competent translator to allow him or her to create an accurate and complete localisation. (It must be borne in mind that such a person is unlikely to be fully versed in the technical implementation of the program.)
- Support of evolution and maintenance of a program that has already been enabled.

The implementation of suitable mechanisms in support of these would permit the removal of the enabling and localising processes out of the *ad hoc* home-grown arena of current practice into one that is rigorous, professional, and generally applicable. It is our belief that both of these areas can be dealt with by the development of appropriate software tools that build upon the existing infrastructure support to create a programming environment for the internationalisation of software.

References

[1] **American National Standards Institute**, 'American National Standard for Information Systems – Programming Language – C', ANSI X3.159-1989.

[2] **David Barnes and Tim Hopkins**, 'Experience with Adapting Software for use in a Multi-Lingual Workplace', submitted to Information and Software Technology, 1992.

[3] **Pascal Beyls**, 'The Arabisation of UNIX', EUUG Conference Proceedings, Autumn 1988, pp 253-264.

[4] **Elisa del Galdo**, 'Internationalization and Translation' in Designing User Interfaces for International Use Ed. Jakob Nielsen, Elsevier 1990, pp 1-10.

[5] **Digital Equipment Corporation**, 'Guide to Developing International Software', Ultrix Software Development Manual, Volume 1, June 1990.

[6] **Asmus Freytag and Michael Leu**, 'Using the OS/2 National Language Support Services to Write International Programs', Microsoft Systems Journal, 5:2, Mar 1990, pp 1-26.

[7] **William S. Hall**, 'Adapt Your Program for Worldwide Use with Windows Internationalization Support', Microsoft Systems Journal, 6:6, Nov-Dec 1991, pp 29-58.

[8] **Hewlett-Packard Company**, 'Native Language Support: User's Guide. HP 9000 Computers', Jan. 1991.

[9] **J.E. Johnson**, 'The Story of Air Fighting', Arrow, London, 1990, page 164-165.

[10] **Leslie Lamport**, 'LaTeX User's Guide and Reference Manual', Addison-Wesley, 1986.

[11] **Jakob Nielsen**, 'Usability Testing of International Interfaces' in Designing User Interfaces for International Use, Ed. Jakob Nielsen, Elsevier, 1990, pp 39-44.

[12] **Jakob Peter Nielsen**, 'International User Interface for Infoflex' in Designing User Interfaces for International Use, Ed. Jakob Nielsen, Elsevier 1990, pp 159-187.

[13] **M.C. Peterson**, 'ARRIS: Redesigning a User Interface for International Use' in Designing User Interfaces for International Use, Ed. Jakob Nielsen, Elsevier, 1990, pp 103-109.

[14] **Gene Spafford**, 'O, Oh, what a difficult name', in USENET comp.risks, Volume 12, Issue 19, 28 August 1991.

[15] **Robert C. Sprung**, 'Two Faces of America: Polyglot and Tongue-Tied' in Designing User Interfaces for International Use, Ed. Jakob Nielsen, Elsevier, 1990, pp 71-102.

[16] **Piyawadee Sukaviriya and Lucy Moran**, 'User Interface for Asia' in Designing User Interfaces for International Use, Ed. Jakob Nielsen, Elsevier, 1990, pp 189–218.

[17] **Dave Taylor**, 'Creating International Applications, A Hands-On Approach using the Hewlett-Packard NLS Package' in Designing User Interfaces for International Use, Ed. Jakob Nielsen, Elsevier, 1990, pp 123-158.

[18] **X/Open Consortium**, 'The X/Open Portability Guide', release 3, April 1989.