

Introduction 2  
Globalization Considerations 2  
Cultural 2  
Technical 3  
Issues of Concern 3  
Workflow 5  
Instructions for Branch Office Testing 9  
Internationalization Checklist 10  
Cultural User Interface 11  
References 13

# Globalizing and Localizing an Application or Web Site

Cynthia Shehata

## Introduction

**Globalization** (aka internationalization) is the process of developing a program core whose feature and code design support other languages; the code source base simplifies the creation of different language editions. It can also refer to code changes that are made to ensure that a product or Web site can be localized so that information is presented in a format to which your users are accustomed.

**Localization** is the process of adapting software for a particular country or region. For example, the software must support the character set of the local language and must be configured to present numbers and other values in the local format. *The Computer Dictionary* (Microsoft Press, 1994) defines localization as “the process of altering a program so that it is appropriate for the area in which it is used.” It also refers to the translation of strings and localization engineering tasks

- A computer must be capable of displaying the user's native language, character set, and notations (such as currency symbols).
- The user interface and documentation must be translated into the user's native language in a way that is understandable and usable.
- A system must match the user's cultural characteristics. It must accommodate the way business is conducted and the way people communicate in various countries.

The goal is to give users a consistent look, feel and functionality across different language editions of a product. Users expect localized software to support the basic features as does the native language edition.

Many projects fail due to lack of acceptance by target cultures.

## Globalization Considerations

### Cultural

- Avoid abbreviations, acronyms, idiomatic expressions, words that have multiple meanings, colloquialisms, and slang.
- Recognize that humor is very culture dependent and typically does not translate well into other languages.
- Avoid making comparative statements that position your product against the competition. In some countries this type of positioning is illegal (it is in Germany).

- Avoid pictures of sports equipment, utensils, national monuments, or any symbols that might be unfamiliar to members of other cultures.
- Recognize that the meanings attached to symbols is often culture dependent.
- Be cautious with representations of animals, religious and mythological symbols, national emblems, colors, people (especially racial, cultural, or gender stereotypes), hand gestures, and body language, which may be misinterpreted or may offend users in other countries.
- Avoid using graphics that represent holidays or seasons, such as Christmas trees, pumpkins, or snow.
- Be culturally sensitive when choosing sounds for use in your program. While some users may find it helpful to hear a beep when they make a mistake, users in Japan may find a beep embarrassing because it calls attention to their mistakes.
- Avoid images in bitmaps and icons that are ethnocentric or offensive in other cultures
- Avoid visual puns (e.g., a picture of a dining table as the icon for a table of numbers)
- Translate fully.
- Avoid maps that include controversial national boundaries as they could be politically offensive.

### Technical

- Recognize that language can affect the length of text. For example, “Edit” becomes “Bearbeiten” in German, and “Sort Ascending” becomes “Lajittele nousevassa järjestyksessä” in Finnish.
- Utilize user preferences for the formatting and display of dates, numbers, time, currency, etc.
- Be consistent in your terminology. In addition to being good interface design, it also makes translation easier.
- Keep bitmaps simple. The more complex they are, the more difficult they are to decipher.
- Avoid the use of letters in bitmaps and toolbar icons. The letters may not exist in the target language, and will often no longer reflect the intended function.
- Avoid run-time concatenation of strings. Because of differences in syntax and sentence construction, concatenated strings often cannot be properly translated into other languages.
- Support international conventions - especially entering and displaying international characters. With the advent of Unicode, it is no longer acceptable to limit valid characters in the ASCII set.
- Use Unicode as your character encoding to represent text.
- Consider multi-lingual user interfaces: launch the application in the default user interface language, and offer to change to other languages.
- Watch for windows messages that indicate changes in the input language, and use that information for spell checking, font selection, etc.

### Issues of Concern

To **choose a language** from a list up to 7, list the name of each language as a word, using each language's own name for itself. (English - Deutsch). For lists of 8 - 21 languages, use visual symbols to supplement the names. For lists of 22 languages or more, use an alphabetical list (in which case non-Latin languages should be listed twice: once in Latin characters in the proper alphabetical order and once in the true character set at the end of the list). The best visual symbol for a lan-

guage is probably a flag. Use flags that match the geographical location of the service and its main intended audience.

**Truncation** happens when the design of an interface does not allow for the word length variations in other languages. Any words that are longer are then truncated in the interface. In the design, consider that all of the words and phrases could expand. The general rule given is to expect terms to expand by 30%. However, this rule works best when applied to large sections of text but doesn't work as well when applied to a single word. A term can expand anywhere from 200% to 400%. Having a flexible design is critical to the success of localizing a Web site.

### **String truncation** Determine

- Whether strings are accessible to localizers (depends on your tools)
- Whether keyboard shortcuts can be localized
- Characters displaying correctly in HTML and on all controls/elements of Web site
- Characters displaying correctly in and out of a database

A **composite string** is an error message or other text that is dynamically generated and presented to the user in sentence form. It is advisable to have multiple error messages that are translated separately, or to find a different way of presenting the same information. For example, here are two ways of telling the user how many mail messages they have.

- MailMsg = [Number of messages in user's mailbox]
- `<P>Inbox: <%=MailMsg%></P>`

The **sort order** is not the same for all languages, particularly for languages that do not use the Western alphabet. If one were to just translate the terms as given, the user would not know how to find the information they were looking for. Find a way to automatically sort the items (this can be a very difficult task) or ensure that the localizers can change the sort order of the list while they are localizing the code.

**Date and time:** use unambiguous dates. It is suggested to use

- a four-digit number for the year.
- words for the months (can be abbreviated)
- for forms, structured dates and pop-up menus.

If the last update will be displayed, calculate based on the user's time so that the Web site's content appears fresh. If giving a local time, make it relative to Greenwich Mean Time.

A special problem is the **search** of multi-lingual information spaces. If all of the information has been replicated in every language, then there is no need to search more than one language. In this case, the search interface should know of the user's preferred language and only display hits in that language. Unfortunately, it is often not possible to translate all documents, so many sites require searches of several languages if the user needs complete coverage of the available information. Currently, multi-lingual search requires the user to manually enter synonyms of the desired search terms in all the requested languages. Users often forget to search for translated terms, even if they understand several languages. It would be better to have the computer automat-

ically perform multi-lingual searches by understanding the meaning of the search terms in several languages. Doing so is easier than the general problem of natural language translation

**Printing** For rich or large hyperspaces, provide a special version that can be downloaded and printed as a single document. Any file that is intended for printing must be able to accommodate the two most common page formats:

- A4
- 8.5x11 (U.S. Letter)

To do so, the width of the page must fit on an A4 sheet and the height of the page must fit on an 8.5x11 sheet, since A4 is the narrowest format and 8.5x11 is the shortest format. It is recommended to leave a margin of at least half an inch (13 mm) for all four sides of the page to ensure that it will print on all printers and to facilitate photocopying. With half-inch margins, the printable area is 7 1/4 inches (18.5 cm) wide by 10 inches (25.4 cm) tall; with one-inch margins (preferred), the printable area would be 6 1/4 inches (15.9 cm) by 9 inches (22.9 cm).

The most common issue with localizing **graphics** is that no proper source files are provided. Most localizable graphics consist of text on top of some sort of structured background. To localize the text, you need to get access to the text only. If you get a GIF or JPEG file, the text and the background are in the same layer so changing the text means touching the background as well. If the background is just a plain color, you can easily replace the text. If the background contains structure, it will be more difficult. The most common format used to hand off localizable graphics is PhotoShop (.psd extension), because it supports layers, which means localizable text can go into a separate layer and the other components of the file don't have to be touched.

## Workflow

1. Prepare the project (globalization)
2. Research the hurdles (normally this would go in a spec)
3. Identify the scope (file list, word count)
4. Identify your audience (technical, non-technical, etc.)
5. Write instructions for each specific group of people working on the project

It is most efficient to create an internationalized core that serves as the foundation for all language editions of a product when development begins. It has been found that the best success is achieved when globalization is done in parallel with core development (development of the base product, usually targeted for the United States market) and before localization has begun. Benefits are it

- Removes errors early in the process
- Ensures that people familiar with the code fix the problem, instead of having someone else fix the errors
- Educates your developers so globalization bugs aren't as likely to be introduced later in the project or in future projects

Before starting any localization project, a **localization plan** should be written. Outline

- the various instructions for the localizers
- the engineering changes for a market (if any)
- the overall process to follow

For localizing, it is recommended to do the following:

1. Isolate all user interface elements from the program source code.
2. Use the same resource identifiers throughout the life of the project. Changing identifiers makes it difficult to update localized resources from one build to another.
3. Make multiple copies of the same string if it is used in multiple contexts.
4. Do not place strings in resources that should not be localized.
5. Allocate text buffers dynamically, since text size will probably expand when translated.
6. Keep in mind that dialog boxes may expand when localized.
7. Avoid text in bitmaps and icons, as they are difficult to localize.
8. Do not create a text message dynamically at runtime, either by concatenating multiple strings or by removing characters from static text. Word order varies by language, so dynamic composition of text in this way requires code changes to localize to some languages.
9. Avoid composing text using multiple insertion parameters in a format string, because the order of insertion of the arguments changes when translated to some languages.

The key to success is testing and research. You must know your market --or find someone who does -- to avoid introducing errors in your code, or you must test your product and remove the errors later.

Items that need to be globalized.

- Date format (long and short)
- Time format (Europe uses a 24 hour clock)
- Money and anything relating to it: taxes, etc.
- Number formats (using a “.” instead of a “,” to denote thousands)
- Address formats (having a zip field hard-coded)
- Font names, size, and decoration
- Day turnovers
- Telephone number and address formats
- Character sets
- Collating order sequence
- Reading and writing direction
- Punctuation
- Units of measures and currency
- Spell and grammar check

These items are often hard-coded. It is suggested to either use the system settings when possible or use an automated function to present the information to the user.

Writing a **style guide** is recommended for each of these markets detailing

- the fonts to use
- the font size
- any other font attributes that can make reading some languages difficult.

Write **test cases**. This allows you to focus on smaller and smaller units of a project. As you write a test case for each component, you can start thinking about how others would use it. In addition

to uncovering problems at the globalization stage, you can reuse these test cases at other stages of our project -- particularly when the files have been localized.

Since  $\text{TotalBugs} = \text{CoreBug} * \text{Languages}$ , the more languages you localize in, the higher the bug count. Also,  $\text{TotalBugs} = \text{EngineeringHours}$ , and  $\text{EngineeringHours} = \text{AddToProjectCost}$ . If you are localizing your product in a few languages, running a pilot can save you money. The **pilot project** lets you test the localization process, to make sure all steps outlined make sense and are achievable.

If you are in the design phase of a project, you may want to run a prototype through pseudo-localization to ensure that the design is flexible for all of the terms to be translated. For more complex sites, you could use pseudo-localization to test dynamically generated data or to ensure that your controls can display extended characters correctly. Include some extended characters (used generically to refer to any accented character, such as é or ñ or ö, and sometimes Asian characters) or Asian characters because they increase the length of the terms and paragraphs in the prototype.

It's best to choose a pilot language that will address more issues than a language that won't and one that is of more strategic importance than others. Because a pilot project is ahead of the other languages, it's likely it will be ready for release before other languages. The “best” pilot language will most likely be:

- Japanese
- German

These languages use international characters, more space, and are important markets. Japanese uses Double Byte characters (DBCS), which is an area with very specific issues, and is mostly used as a pilot language together with a non-DBCS language.

To ensure your project is localized properly, provide instructions for the localizers, testers, and engineers. Localizers need information on what to localize, who the audience is and, what not to touch in the file. (Unfortunately, Web-based translation cannot always be done in “safe mode” and localizers often have access to source code.) Static HTML pages are generally easier than ASP or HTML with scripting, because HTML is relatively simple to understand and there are tools that “lock” the HTML tags and allow only plain text or attribute values to be localized. ASP files normally contain a large amount of scripting (VBScript or JavaScript) and localizable strings are often embedded in the scripting. For localizers it can be hard to distinguish localizable strings from functional ones. Testers and engineers also need information on what to do with the files. Providing test cases to the tester as well as an overview of the project will ensure the project is checked thoroughly.

**Quality Assurance** Localized products should be reviewed for quality. There are three common reviews:

1. Testing. With the help of special test cases, the localized product is reviewed for functionality and, to a minor extent, language.
2. Language checks. Localized products will usually be reviewed by a language specialist. This person can be a senior localizer or an editor.

3. Build Verification Test (BVT). The BVT originates in the software world as a check if the build process of a product has been successful. The BVT consists of a subset (normally 40%) of test cases and will focus on functionality rather than language.

For software products, the focus is on functionality. Products are more often language specific, so,

- The user cares most about functionality
- It might not be cost-effective to localize in different flavors
- There's not much plain text involved

For Web sites functionality is still important but can be fixed much quicker, because it's just a matter of updating some lines of code and re-posting the file. Language quality is much more important because Web pages are mostly text with less functionality than applications. If the language quality is bad, the user will be annoyed. Highest annoyance factors are:

- Text still in English (for localized sites)
- Spelling mistakes
- Grammatical mistakes
- Wrong terminology (can be market specific as well); wrong terminology can sometimes cause an insult

You can localize in-house, or you might outsource the localization completely. In any case, you need to maintain source control. This means that you have to establish a code base that all languages are based on. From this code base, the localization will be done.

To choose a localization company, you can keep the following in mind:

- The company specializes in the type of localization project you provide (i.e., some companies do software only)
- The company can perform engineering and testing tasks as well, which means you can hand off a project and get it back ready to ship
- The company has internal QA procedures

International usability testing is recommended with users from a few countries in different parts of the world. There are four main ways to conduct international user testing:

- Go to the foreign country yourself.
  1. Recruit users who speak your language, even if not fluently. This is often the easiest to implement. Make sure you don't get unrepresentative users who, for example, have spent years at college in your country and thus have been acclimated to the possible linguistic and cultural peculiarities you hope to filter out during the test.
  2. Conduct the test in the local language.
- Run the test remotely (ex. place telephone calls to the users and ask them to think aloud as they navigate the site)
- Hire a local usability consultant to run the test for you
- Have staff from your local branch office run the test, even if not trained in usability. It is cheaper and it brings them into the product-development process and they will likely learn a great deal from the test itself.
- Build additional usability groups in your major markets.

## Instructions for Branch Office Testing

1. Recruit a person who is a typical user for the kind of accounts you expect to be most important in the future. The user should be average for WWW users in that organization and not a super expert or high-level manager (unless they are the only ones using the WWW). The user should have some ability to read written English since the pages will not be translated, but the user should not be uncommonly good at English (and will not need any skills in spoken English).
2. Make sure that you have a room with a workstation reserved for the test and that you will have no interruptions during the test. Make the browser software forget the navigation history so that all link anchors are blue (usually, the ones the user has already seen are purple). At least a day before the test you should translate the test tasks into your local language (the test tasks are described below). Go through the test tasks yourself to make sure they can be done and that the instructions make sense. While doing so, you may observe some aspects of the user interface that seem poorly suited to users from your country, and you should make a note of any such problems.
3. Explain once again to the user that you are doing a usability study to see how easy it is to use the new product. Explain that you are going to be trying a prototype and that not everything has been completed yet, meaning that errors may occur during the test. Ask the user to “think out loud” during the test: the user should keep up a running monologue, saying anything that comes to mind and not hold back anything. Say that you are interested in all the user's thoughts, comments, and interpretations of the design, even the ones that may seem minor at the time. Since thinking aloud is somewhat unnatural, many users stop doing so and need to get prompted to speak. If the user sits quietly for an extended period of time, a good prompt is “what are you thinking about now?” Other than prompting the user to keep talking, you should not say anything yourself during the test. If the user asks about the meaning of some element of the user interface, you should not answer, but instead ask the user “what do you think it might be?” Encourage the user to make a wild guess if necessary. Also, you should neither agree nor disagree with the user's comments. Just make a sound to confirm that you have heard the user's comment without indicating your personal opinion and then write down the user's comment.
4. Ask the user to begin by giving his or her general comments on the interface before starting to click. When the user has finished commenting, tell the user to do anything he or she would normally want to do. Just let the user navigate the system freely for the rest of the exploratory test time. Don't offer help too soon since it is often very valuable to see how users get out of trouble on their own.
5. Give the user tasks, which should be given in writing, one page at a time (you want them to concentrate on one task at a time). If you come to the end of the scheduled test time or if the user starts feeling uncomfortable, you should stop the test.
6. Debrief the user. After the test tasks (and while sitting at the computer), tell the user that this completes the test. Ask the user for his or her general comments about the design. Use two questions: “What did you particularly like about this system?” and “What did you particularly dislike about this system?” If the user does not have enough comments, you can often elicit additional comments by asking the question “If we now compare with other sites you have visited on the Web, what things have you seen that you liked and would want us to do, and what things do you want to warn us against doing?”
7. Report the Results. In writing the report, you should refer to both usability problems experienced by the test participant and to issues you observed yourself while preparing for the test.

Try to distinguish between these two kinds of data: it is good to know which issues are your personal critiques of the use interface and which issues refer to actual events during the usability test. Also distinguish between cases where the user said that something might be confusing or problematic to other people in your country and where that specific user actually was confused or had problems. (Nielsen, Jakob. "Instructions for Branch Office Testing.")

## **Internationalization Checklist**

(From the Microsoft Win32 Internationalization Checklist)

1. Program specs account for international considerations from the outset.
  - Features important to international markets are included.
  - Icons and bitmaps are generic, are culturally acceptable, and do not contain text.
  - Menu and dialog-box designs leave room for text expansion.
  - Text and messages are devoid of slang and specific cultural references.
  - Strings are documented using comments to provide context for translators.
  - Strings or characters that should not be localized are marked.
  - Shortcut-key combinations are accessible on international keyboards.
  - International laws affecting feature designs are considered.
  - Third-party agreements support international design issues.
  - Consistent English user interface terminology is used in strings.
2. Code is generic enough to work for several languages.
  - Code doesn't concatenate strings to form sentences.
  - Code doesn't use a given string variable in more than one context.
  - Code doesn't contain hard-coded character constants, numeric constants, screen positions, file-names, or pathnames that presume a particular language.
  - Buffers are large enough to handle translated words and phrases.
  - Program allows input of international data.
  - All language editions can read one another's documents.
  - Code contains support for locale-specific hardware, if necessary.
  - Features that don't apply to international markets can be removed easily.
3. Code takes advantage of international functionality offered by product.
  - Code uses international information carried by the system.
  - Code uses system functions for sorting, character typing, and string mapping.
  - Code uses generic text layout functions provided by the Multilingual API.
  - Program responds to changes in Control Panel's international settings.
  - Far East editions support Input Method Editors, vertical text, and line-breaking rules.
4. All international editions of the program are compiled from one set of source files.
  - Mechanisms requiring code to be recompiled for different language editions are weeded out.
  - Localizable items are stored in Windows resource files.
  - All language editions using double-byte character sets are based on a single executable.
  - All language editions using Unicode are based on a single executable.
  - All bidirectional language editions are based on a single executable.
  - All language editions share a common file format.

5. Code is generic enough to handle different character sets.
  - Code properly handles accented characters.
  - Program handles non-homogeneous network environments in which machines are running different code pages.
  - Code for Far East–language applications parses double-byte characters unless the code is based on Unicode.
  - Code supports Unicode or conversion between Unicode and the local code page.
  - Code doesn't assume that all characters are 8-bit or 16-bit.
  - Code uses generic data types and generic function prototypes.
  - Program displays and prints text using the appropriate fonts.
6. Program meets international testing standards.
  - Text is translated and meets the standards of native speakers.
  - Dialog boxes are resized and text is hyphenated appropriately.
  - Translated dialog boxes, status bars, toolbars, and menus fit on the screen at different resolutions.
  - Menu and dialog-box accelerators are unique.
  - User can type accented characters into documents, dialog boxes, and filenames.
  - User can type accented characters into documents, dialog boxes, and filenames.
  - User can successfully cut, paste, save, and print accented characters.
  - Sorting and case conversion are culturally accurate.
  - Application works correctly on localized editions of Windows.

## Cultural User Interface

A Cultural User Interface (CUI) is a user interface that is intuitive to a particular culture. The CUI takes advantage of the common knowledge of a culture defined by country boundaries, language, cultural conventions, race, shared activities or workplace. An application that is CUI-enabled allows the use of many different CUIs. These different CUIs are developed collaboratively with the target cultures, thus problems associated with localization such as misinterpretation of elements in the CUIs, are less likely to occur.

Covert factors in localization deal with the elements that depend on culture or “special knowledge”:

- Graphics/visuals
- Colors
- Functionality
- Sound
- Metaphors
- Mental Models

Much of the literature has advised caution in using such things as metaphors and graphics.

Examples of misinterpretation:

- The “trash can” icon in the Macintosh user interface is not global. In Thailand the “trash can” is a wicker basket

- In the US, the owl is a symbol of knowledge but in Central America, the owl is a symbol of witchcraft and black magic.
- A black cat is considered bad luck in the US but good luck in the UK.
- The OK hand gesture in France means “zero”, “nothing” or “worthless”. In some Mediterranean countries, the gesture implies a man is a homosexual. In Germany, it means someone is a jerk.
- For a Catholic, some commands, such as abort or kill, may be offensive.

Covert factors will only work if the message intended in those covert factors is comprehended in the target culture. The software developers need to ensure the correct information is passed by validating these factors with the users in the target cultures.

Covert symbols usually have the same meaning to members of a particular culture. Thus, communication within these cultures using artifacts and symbols would be possible. There is less likelihood of misinterpretation of covert factors in a culture. This shared “meaning” among members of a culture forms the basis of a Cultural User Interface (CUI). Members of the same culture are likely to have the same “knowledge” of certain things, and would think and act similarly in certain situations. A CUI could be created for each culture to take advantage of the “knowledge” of a target culture. These CUIs should be developed in collaboration with members of the target culture to ensure that the CUIs are acceptable to the target community. These CUIs will cover all the covert and overt elements of the user interface.

Before CUIs can be used, an application must be separated into

- a functionality component
- a user interface component

Different CUIs can be developed, and each of these different CUIs can be used with the same functionality component.

The application could also accept different CUIs. It would be modified to contain tokens, each corresponding to a unique interface element of the application. A file would contain all the localized tokens for a particular culture. When the application is running, different files (different CUIs) could be used to provide the user interface.

- Identify all the elements that need to be localized to a particular culture.
- Identify the target culture as well the level of breakdown of cultures - where to set the cultural boundaries (national level, language and national level or at an even deeper level).
- Select experts of the target culture to work with user interface designers. These experts are preferably denizen of that culture, adept in the language, and have experience with computers. Two or three experts need to be consulted as the expert chosen may be “contaminated” by different cultures.
- Conduct usability tests on the CUI, which will be evaluated and modified (if required) until the target culture is satisfied with it.

## References

Duerst, Martin. "Internationalization/Localization. Dates and Time." <http://www.w3.org/international/O-time.html>

Ebben, Sjoert and Gwyneth Marshall. The Localization Process: Globalizing Your Code and Localizing Your Site. <http://www.microsoft.com/TechNet/Analpln/glolocal.asp>

Interface Hall of Shame. "Globalization Tips." <http://www.iarchitect.com/htglobal.htm>

Microsoft. "Globalization & Localization: Best Practices for Windows 2000 Compliant Software." <http://www.microsoft.com/TechNet/win2000/glbstp.asp>

Microsoft. "Internationalization." [http://msdn.microsoft.com/library/books/devintl/S24B0\\_b.HTM](http://msdn.microsoft.com/library/books/devintl/S24B0_b.HTM)

Microsoft. "Microsoft Win32 Internationalization Checklist." <http://msdn.microsoft.com/library/books/devintl/S24AC.HTM>

Nielsen, Jakob. "Flag Problems." <http://www.useit.com/alertbox/flagproblem.html>

Nielsen, Jakob. "Instructions for Branch Office Testing." <http://www.useit.com/alertbox/testinstructions.html>

Nielsen, Jakob. "International Usability Testing". [http://www.useit.com/papers/international\\_usetest.html](http://www.useit.com/papers/international_usetest.html)

Nielsen, Jakob. "International Web Usability." <http://www.useit.com/alertbox/9608.html>

Yeo, Alvin . "World-Wide CHI: Cultural User Interfaces, A Silver Lining in Cultural Diversity." <http://www.cwi.nl/~steven/sigchi/bulletin/1996.3/international.html>