INTERNATIONALIZING SOFTWARE TESTING

Andrea Vine

Internationalizing testing? The title is deliberately chosen— *internationalizing testing*. The idea is to internationalize all software testing instead of developing a separate testing process and calling it *internationalization testing*.

Internationalized testing is not the same as localization testing. In internationalized testing, product functionality and usability are the focus. Localization testing is for linguistic relevance and for verification that functionality has not changed as a result of localization.

Does this sound familiar? Some companies avoid internationalization testing and internationalized testing for any number of reasons. Product groups don't think about internationalization at all. Or they think that internationalization testing is performed by "some other group." Or they just don't think international is very important. Many test groups don't know how to test for international and have no resources to help them learn how.

Admittedly, there is not much information specifically geared to internationalization testing. For example, in researching this article, I found that only the most superficial information is publicly available. So, it's no wonder people don't understand the process. Of course, this article can't cover everything you need to know, but I'll try to touch on as many areas as possible.

Some companies do have *internationalization* in place, but often the testing is not adequate to catch enough of the internationalization problems, and it's frequently done too late in the quality assurance (QA) cycle for fixes to be integrated into the product.

I hope to remove most of the FUD — Fear, Uncertainty and Doubt — about internationalizing test suites in the course of this article. It seems that a lot of testers don't think that they can test international specifications and data; they think it's too "hard" or that it will take a long time to learn. The reality is that if the information is presented in a clear, straightforward way, it is very easy to learn. Once testers have done a small amount of internationalized testing, they'll feel like it's a regular part of the testing repertoire.

By internationalizing existing tests, including all testers, and eliminating redundant areas, more of the product internationalization can be covered with the same number of people in the same amount of time. I'll provide some guidance on ways you can get more coverage with the resources you have.

IS ENGLISH A BASE TEST?

Much of the software testing today is conducted as though US English is a sort of base or core test, and other locale formats and language characters are added functionality. But consider how the product is developed. Properly internationalized products are designed and coded with internationalization in mind. The internationalization in the code is everywhere, not in one or two modules. In fact, for most products it would be impossible to have separate internationalization modules installed alongside existing code to "perform internationalization functions." The test design is no different. Test plans and suites must be internationalized the same way program design and code are.

English is written in ASCII, and ASCII is a direct subset of all major charsets. So no matter which charset you're testing, you automatically test ASCII. Therefore, testing English (ASCII) alone is a waste of time. The same is true for the C and US locales. If the product is developed in the United States, then the odds that there will be problems with data in a

US format are very low. Because the risk of problems is far higher for other locale formats, it makes more sense to test in another locale from the beginning.

Another reason for internationalizing testing relates directly to the bottom line. For example, say a product has a group of 10 testers who can't cover all the areas of a given product when simply testing English, ASCII and C locale. That group outsources to one or two people all internationalization testing, which encompasses the entire test suite in several languages, charsets and locales, not to mention machine configurations. However, the market for the non-English, non-ASCII, non-C-locale functionality is over 60% of the total market. This means that the test coverage is grossly inadequate for the size of the market. Test coverage should take into account the entire market and the higher risk areas in relation to the market.

Representative Sampling

One way to look for internationalization bugs efficiently is to test on a variety of locales using several different charsets and formats. To try and touch different aspects of internationalization, I recommend using at least one candidate from each of several categories. The categories in programmatic terms are Western European; non-Latin single-byte languages/locales; Asian; and bidirectional. This covers the locales that make up the major business markets.

In addition to English, the Western European category encompasses French, German, Spanish, Italian, Dutch, Portuguese and in many cases Swedish, Norwegian and Danish. It covers the countries of Western Europe, as the title suggests. Which languages and locales from this group that you choose depends on how much time you have, how much business comes from each market and how many bugs have been found in each locale in prior versions of the product.

By calling the second category non-Latin single-byte, I have focused on the charset used. The charset is the data; if it isn't processed correctly, then nothing else can be. But in this case I am also referring to the locales which use these charsets. Some of the possibilities are Russia and Greece.

Major countries in the Asian category are Japan, China and Korea. These require the written languages Japanese, Simplified Chinese, Traditional Chinese and Korean.

Examples of bidirectional languages are Arabic and Hebrew. While Arabic and Hebrew alone are right-to-left languages, the entire text becomes bidirectional when mixed with left-to-right text or numbers.

Who Is Responsible?

In broad terms, everyone in the company is responsible for internationalization. This includes people not involved in engineering, such as product marketing, finance and operations. Everybody who tests the product needs to think about internationalizing the tests. Developers who unit test their modules must use international data in their testing, just as they internationalized their code. QA engineers may assist developers in their unit test design; they can provide information on internationalizing the tests. By the same token, test engineers are responsible for internationalizing the system tests. They may receive help from development in understanding configurations and settings for international use and should apply this information to their configuration design.

If your company has some internationalization experts available, consult with them on internationalizing your tests. Most likely they can provide you with more detailed information. Remember that internationalized testing or even internationalization testing is not the same as localization testing. Just because there is a group doing localization testing doesn't mean that internationalization is covered.

Configurations

Plan which configurations will best cover the internationalization of your product. The Internationalization Requirements/ Taxonomy Document and Checklist Matrix from the Sun Globalization Resources define what it means for a product to be internationalized. The primary function is to help groups assess the internationalization status of their products according to a matrix of interfaces and functions. The document and checklist matrix are available at http://developer.sun .com/dev/gadc/des_dev/i18ntaxonomy

A product group can complete the matrix in the planning and design stage and then revisit it for the development stage. Testing can then take the matrix and verify that areas marked as compliant or partially compliant work as described.

Locale is an important configuration element. Several types of locales, such as client, system and thread, can be used in or by a product. It is essential to find out which type of locale your product will detect or where it can be set. In a client-server product, it is important to understand how the product processes client locales as well as server locales. Programs can set their own locale environment, sometimes per thread. Find out where they get this information.

Try different combinations of locales. Set up a system configuration with a single locale throughout all machines and processes. Then test on a mixed set of locales. For efficiency, you can rotate locale settings throughout the entire testing phase of a product instead of testing on the same locale configuration for each test cycle or build. Keep in mind the sampling categories discussed in the section on representative sampling. Use locales from each of the categories. Combine locales in different categories on a multilocale setup.

Time zones are separate from locales. While a locale can indicate a time zone or a set of time zones, the two are not tied together and must be set separately. Make sure that a combination of different time zones is used in a client-server or server-server setting. In individual configurations, rotate the time zone setting to examine the effect. This easily mimics what real customers do. Even in the continental United States there are four time zones, and it is likely that a client will access a server in a time zone different from his or her own.

Above all, varying charset data must be processed by the system. Use the representative sampling categories as guides to select the most appropriate charsets. Plan the data and create a small data bank.

Data Planning

Planning the data for your test suites is crucial to good internationalized testing. Some data may be handled by the

system and only displayed, and some may be input. Know which data appears in your product and how it is used. If the product is new or heavily revised or has never been localized, include pseudo-localization as part of the test plan.

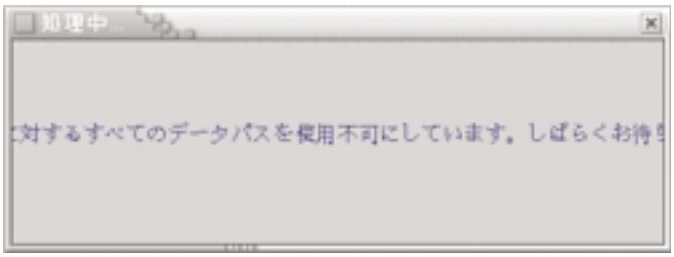
Textual test data needs to cover all aspects of text processing. Include a wide selection of characters, punctuation and symbols which appear in the language. Create long strings to force word wrap, if relevant. Sorting and search should always be included if the product performs these functions. Text is related to language, so consider language parameters in processing. Create a data bank that you can reuse. Reusing data isn't always possible, but even if the data needs minor adjustments with new revisions, it can be much faster than creating fresh data. Textual data should span a range of languages planned with the representative sampling information as a guide.

Numerics are related to locale. Dates and times should be part of the test, along with currency, telephone numbers and units of measure. If there is searching based on numeric data, such as by date or price, include this in the test suite.

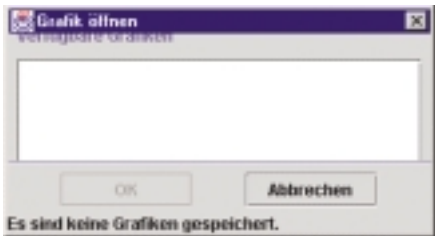
It is important to review graphics early on, since they take longer to create and adjust than text. While they are often not related to product function, they are part of the user interface. Make sure icons, banners, backgrounds, pictures and all other images are part of the test suites for the user interface before it is frozen. Colors and window elements, such as buttons and checkboxes, should also be verified.

Product layout is especially important for localizability. User interface test suites should check screens, dialog boxes, pop-up windows and frames for their internal arrangement and relative screen position. This is best done using a pseudo-localization, which will be discussed later in this article.

Edge testing is important to expose deficiencies in code internationalization. Push through minimum and below-minimum values and lengths, exceed maximum values and lengths. Plan to enter incorrect characters and formats, bad or mismatched dates and times.



Truncation resulting from a line wrap problem



Vertical truncation due to insufficient space

Telephone Sort	Dictionary Sort
A-1 Apples	A-1 Apples
A1 Dog Grooming	A-1 Roofers
A-1 Roofers	A1 Dog Grooming
AMD Security	Aaron Tailors
Aaron Tailors	All Gone Pest Control
All Gone Pest Control	Allen Plumbing
Allen Plumbing	Am-Center
Am-Center	Ambrosia Caterers
Ambrosia Caterers	AMD Security
Azalea Planters	Azalea Planters

Two sorting orders in US English

What Do You Look For?

Text. Now that you have your textual data planned and executed, what should you look for? Note that in some cases, data can be constrained by a standard. If you're not sure, write the bug. Most often, standards won't affect these areas. Remember that text can be affected by both the language setting and the locale setting.

An underlying theme in looking at resulting data is that there is not always a right and a wrong behavior for internationalization. Sometimes instead of writing a code bug, writing a documentation bug to make sure the behavior gets documented makes sense. A classic example of this is a sort of multilingual data. Unless your product has a specification for this behavior, the data may be sorted in several different ways. As long as it's sorted in some way, then the claim of sorting is justified. But the behavior of the sort is best documented in case a customer has a preconceived notion of multilingual sort.

Truncation is a frequent occurrence, especially in localized or pseudo-localized user interface elements. For multibyte character data, look for split characters where some of the bytes of a single character have been truncated. The result is often an ASCII character or some other strange character at the end of a string. Also, look for obvious length truncation and vertical truncation.

Provided your system is configured properly for rendering those characters, check that the rendering is valid. Although most products do not do their own rendering, they often do specify fonts and point sizes, which can drastically affect the appearance of characters. Be careful with cutting and pasting text, as it is unreliable from one software product to another, especially when the text is not in ISO-8859-1 (Latin1).

Different languages have different line wrap rules. Since some languages have no spaces between words, line wrap isn't always obvious. If the product can control wrapping, check that lines are broken appropriately for the language. Check the length of the lines as well, since one of the problems can be determining line length by byte rather than by font metrics. This can result in either lines that are too short or some that are so long they get truncated.

Searching text is complex. Different people have different preferences for how data is matched. For example, a user who is looking for the French word *thé* in a body of text with both English and French probably doesn't want all the occurrences of the English word *the*. Conversely, a German might prefer that searching for ander finds both *ander* and *änder*. Another problem is that the search text might be in a different charset from the body of text being searched or that the search text might be encoded differently from the body text (this can happen with Unicode encodings). Know what the results should be, and document anything which isn't found. The best option is to give the user a choice of search style; the

next best option is to be consistent. Make sure that one or both of these options are available and that the search behavior is documented.

Sorting is a little more straightforward than searching, but not much. Every language has at least one sort order, but most have several. Even in US English, the telephone book sorts differently from the dictionary. Understand the type of sort your product is aiming for and verify that it is appropriate for the language. Try sorting multilingual data to see how that is handled and verify that the behavior is documented.

If there is any kind of indexing in the product, make sure it is appropriate for the language. For example, if the product has a screen with buttons based on the letters of the language, be sure that for alphabetic languages all letters are included in the proper order and that for non-alphabetic languages the index headings make sense and are usable. The problem with testing this particular layout is that usually a localization is needed before the problems are detectable. If this is the case, alert development that this sort of layout can be problematic and that they should work with the localization team to find out if the layout is viable. Check other indexing as well, such as on-line documentation index pages. Again, this may be testable only after a localization.

Numerics. While text is sensitive to language settings, numerics are sensitive to locale settings. Keep this in mind when looking at numeric data.

For large numbers, check that the groupings are appropriate. The amount of numbers per group can change with the locale, as well as the separators between groups. In Germany, for example, number groupings are separated by a period, but in France they're separated by a space, and in Japan by a comma. Make sure the correct character is used as the decimal point. Like the groupings separator, it can be a period, a centered dot, a comma or even a space. The number of digits following the decimal point may need to change based on locale.

Dates are formatted in a myriad of ways. Check the shortest format for the order of day, month and year values. Different locales use different separators. In longer date forms, check to see if the day and month names are appropriate for the locale and if their abbreviations make sense. If the application works with different types of calendars, such as Hebrew and Japanese, verify that the dates are accurate and the conversions function correctly. When looking at time values, check that the hours and minutes separator is appropriate and that the 12- or 24-hour format is used based on standard locale preference. Make sure the time value is correct for the time zone setting.

When verifying currency, be sure not only to verify that the format is correct, but that the currency symbol does not automatically change with a change of locale setting when there is an existing value. The reason is that currency denotes an actual value, which can change significantly with a simple change of currency symbol. Also, check that the field is capable of expanding, and that it does not require a decimal point and digits following the decimal. Consider that, for example, on 2004-05-13, the Turkish lira was valued at 1,541,500 to one US dollar. Obviously, a Turkish lira value needs much more space to express the same value given in US dollars and does not need a decimal value.

Units of measure change with the locale, but like currency, the unit name should not automatically change on existing values. Measurements are real values, and unit changes affect the value expressed. Expansion room is also important for measurements, so make sure that there is available space for larger values.

There is an international standard for telephone numbers, but most people are unaccustomed to seeing their local phone numbers in the international format:

+ 1 416 872 2372
+ 507 441 2345
+ 852 2345 6789
+ 44 121 123 4567

That is, the plus sign, a space, the country code (optional), space, region/area code (optional), space, and groupings of numbers to represent the local telephone number, separated by spaces.

More likely they will see these numbers as:

(416) 872-2372 (US and Canada)
441-2345 (Panama)
2345 6789 (Hong Kong)
(0121) 123 4567 (UK)

Try typing in several phone number formats and see if the product rejects any of them or tries to reformat them for redisplay. Note that current platforms do not have a default format for telephone numbers by locale.

Take a close look at address formats in your product. There is no international standard for address formats, and they vary from country to country. In order to have a workable set of address entry fields, it's important that the following is true:

The name field is labeled clearly. In some places the first name a person uses is the surname, not a given name. Labeling the name fields as "First Name" and "Last Name" can be confusing. This should be called out to the development team.

Address lines are generic, that is, they are set up as "Address Line 1," "Address Line 2" and so on.

The "State/Province" field is not required. In many countries, these aren't used.

The postal code area should be labeled "ZIP/Postal Code" and not simply "ZIP Code" since ZIP codes are specific to the United States. (Did you know that ZIP is an acronym for "Zone Improvement Plan"?)

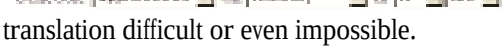
There should be a country field.

Graphics and layout. This area is often overlooked, but it can have some serious effects on product localizability and usability. For more information on graphics and layout, see the presentation "Internationalization in Software Design, Architecture and Implementation" from the 19th International Unicode Conference proceedings or from http://developer.sun.com/dev/gadc/technicalpublications/articles/archi18n.html

Check that there are no images using human figures, body parts, hand signals or animals. There should not be picture representations of English words or visual puns. The orientation of maps or geographic region depicted is often

biased; make sure that maps are appropriate for your entire market. Verify that any object in an image has worldwide meaning, for example, the shape of a telephone. Keep in mind that graphic designers know as much about internationalization as you do, maybe less. Make sure to check if the images are customizable.

Colors should not be used to mean something inherently; this is important for accessibility as well as for internationalization. The use of color should be consistent throughout the product and the documentation. Check to see if the color scheme is customizable or user-selectable.

Translating the product can expand the user interface significantly. Look for expansion room in windows and dialog boxes. If a screen looks crowded, there may be a problem localizing it. Check to see if any of the layout structures force a word order dependency such as  . This makes translation difficult or even impossible.

If you're testing bidirectionality, check the window layout for proper right-to-left formatting. Look for quirky behavior of window objects, particularly horizontal progress bars, image positioning and anything with an arrow. Sometimes looking at a screen design in a mirror can bring out some potential problems with a right-to-left layout.

Make sure that if some windows expand, screen positioning does not become a factor. For example, if a help window pops up next to the screen it refers to so that the user can keep working with the help available, make sure that expansion of the help window will not obscure the primary window. Check to make sure that the rendering does not rely on certain resolutions. This can affect text which is set to be very small, intricate images, and the window size on the screen. Sometimes it helps to use a laptop to investigate this area.

Sounds are very culture specific. If there is language in the sound, it then becomes language specific. If your product has sound, verify that the sounds are more general, such as a beep, tone or buzz. Game show buzzers, sirens, ringing telephones, doorbells and the like are not universal. For some cultures, sounds can be offensive. Verify that there is a way to switch off the sound, and that it is clearly documented. Check if the sounds are customizable.

Are Any Tools Available?

Some tools are available that can help with test planning and code checking. Some are programmatic and others are in document form. For code-checking there are "lint" style programs. LingoPort has a tool called Globalyzer, which includes a development environment as well as filtering and reporting capabilities. See the LingoPort Web site (http://lingoport.com) for more information.

Talk to your internationalization and localization groups or vendors to find out if there are any home-grown tools available. Take a look at the Sun Globalization Resources Web site for periodic updates to tools offerings.

Pseudo-localization is the practice of automatically "translating" all the software resources. The "translation" can be something as simple as adding a few accented characters to the front and back of the string to as complicated as changing every ASCII letter to another character which looks like that ASCII letter. Pseudo-localization helps in several ways:

It tests whether or not the product will pick up the translated resources, providing the pseudo-localization is configured as a real localization will be.

It helps determine whether all relevant resources have been made localizable, as long as the user interface is thoroughly reviewed.

It can reveal problems due to externalized messages which should not have been made localizable.

It checks that locale-sensitive elements such as date formats are changed automatically to match the locale setting, as opposed to being a change that needs to be made manually in a resource file.

It helps verify that a particular charset (or a subset thereof) will display properly.

It helps check the expansion capabilities of the user interface, and may help with some edge testing, since string expansion is a by-product of pseudo-localization.

It can help with layout checking.

Note that it may also be useful to pseudo-localize your test input data.

In preparation for testing, a data bank should be created with data in various charsets based on the representative sampling you have chosen. Make sure to create files that are large enough to cover stress and performance testing.

The Sun Internationalization Requirements Taxonomy is a useful tool for helping design some tests and for recording the results. At test design time, take a look at the matrix checklist form and check the individual fields to provide information on testing the different areas represented by the matrix.

Before deciding on any test automation tools, verify their international capabilities. If you standardize on a tool that can handle only ASCII or Latin1, then it is not sufficient for testing the product. Even if you're used to selecting a tool that has shortcomings, not being able to test over half the data your product is supposed to handle is more than a shortcoming and can cost a great deal for each test cycle used.

When evaluating a tool, simply asking the tool makers about their tool's internationalization capabilities alone is not a good approach. You may ask them, but bear in mind that a) they want to sell the tool to you, and b) they don't know what

For More Information

More information can be found at the Sun Globalization Resources Web site:
http://developer.sun.com/techtopics/global

Other useful Web sites:

http://www.unicode.org/cldr — This site shows the locale-specific data formats from the Common Locale Data Repository, with comparisons of the different platforms

http://www.unicode.org — The Unicode Web site, lots of useful information

http://www.w3.org/International — The W3C I18n Web site, Web standards and recommendations

E-mail lists:

i18n-prog@yahoogroups.com — For general i18n questions, http://groups.yahoo.com

unicode@unicode.org — For Unicode questions, http://www.unicode.org/consortium/distlist.html

www-international@w3.org — For Web i18n questions, http://lists.w3.org/Archives/Public

You must first subscribe to the list before you can send e-mail to it. See the relevant Web site for subscription information.

you mean by internationalization. Instead, ask detailed questions about specific charsets, locales, screen comparisons, throughput and so on. Then try out the tool. If you find tools that have some internationalization capabilities, please tell me about them so I can document them for others.

What Should I Tell My Group?

The number-one point to take back to your group is that existing tests should be internationalized, which is the most time-efficient, cost-effective way to maximize your test coverage. No one knows how to test your product better than your own test team, and with a few incremental changes you can verify the quality for all your customers worldwide. ■

Andrea Vine is an internationalization architect for Java Enterprise System at Sun Microsystems. She can be reached at andrea.vine@sun.com.

The author thanks Bob Silva for his edits.