MultiLingual
Language | Technology | Business

MultiLingual Computing, Inc. • 319 North First Avenue, Suite 2 • Sandpoint, Idaho 83864-1495 USA • 208-263-8178 • Fax 208-263-6310

Best practices in localization testing

Libor Safar & Jiri Machala

Localization testing has become an indispensable component of just about every software localization endeavor. Just like localization *per se*, it has experienced major developments since it emerged as a recognized standalone testing type more than 20 years ago. It has also faced similar challenges. If trends such as reducing overall costs, automating "wherever possible," shortening time frames (think reducing the number of test passes in testing) and an overall shift to outsourcing sound familiar to you from the software translation side of things, you will feel immediately comfortable in the realm of testing, too. Let's take a closer look at where localization testing stands today, the current issues, touch points and synergies that exist between localization and localization testing.

Localization testing follows after product localization and is performed to ensure that the localized product is fully functional and linguistically accurate and that no issues have been introduced during the localization process. Issues introduced as a direct result of localization may be linguistic or terminological in nature; may be related to the graphical user interface (GUI) and its cosmetic look and feel; and may break or compromise the functionality of the localized product.

Libor Safar is marketing manager at Moravia Worldwide.

Jiri Machala is test development manager at Moravia's testing and engineering unit.

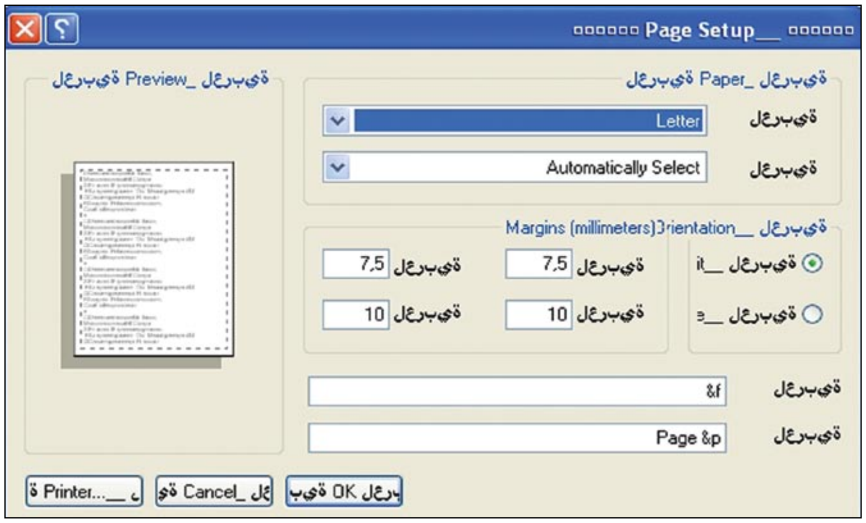
These problems may be introduced simply because software translation normally takes place using tools "outside" of the actual running software application. Individual GUI elements typically pose few problems, but the general string table content may be more challenging, especially with the time pressures frequently imposed on localizers.

Much effort often goes into providing localizers with as much information as reasonably possible about the context in which particular localizable software strings may appear. But the fact of life is the contextual information is typically far from complete, even more so if particular strings appear dynamically in multiple user situations or in different parts of the software interface.

In the past, localization testing has been frequently confused with language or linguistic testing and the assumption was that primarily native speakers or at least advanced-level speakers of the given target language (TL) should be involved. That is no longer the case today. It is now common practice to separate the truly language-specific testing activities, which look at aspects such as in-context language quality or consistency in the running build, from those where only a general knowledge of the characteristics and issues of the particular language is required.

There has always been the dilemma over which is more important: the engineering capabilities of the tester and his or her product knowledge, or his or her linguistic and language background. In practice, it is not easy to get the best of both worlds here and to build, develop and maintain a balanced testing team composed of native TL speakers who also possess excellent engineering capabilities.

First, such people may be difficult to find, and being a native speaker does not automatically qualify one for becoming a language expert. Next, resource utilization and management of peaks or valleys may be a problem if the emphasis is keeping native speaker testers dedicated to their respective languages. Testing as a profession is notoriously prone to fluctuating resource requirements, even despite the best planning efforts and the general trend toward ongoing product releases



An example of a pseudo-localized dialog box.

and upgrades rather than big "all-at-once" major releases. Last but not least, by aiming to employ native speakers, you may forego the opportunity to utilize any of the current locations where production costs are lower or where the pool of qualified and available engineering resources is larger.

Previously much localization testing was in-sourced — managed internally — as publishers built and maintained their in-house teams, similar to having in-house translation teams. Only specific testing types or peak requirements would get outsourced. Currently, it is safe to say that a large majority of localization testing is sourced to countries and regions such as China, India and Eastern Europe.

This is similar to what has happened with other general product testing and engineering and often complements the location of the core product development and quality assurance, which frequently shifted to these cost-effective locations. Publishers that continue to in-source their localization testing for their specific and good reasons are a minority, and even with these the testing location has often moved eastward.

Language neutrality pays off

Localization testing owes much of its current efficient production to the concept of single worldwide binary, often called also the language-neutral model. With this model, which is increasingly used in software development, the core source code includes all the functionality any users worldwide may use.

This may include, for instance, code for handling input method editor data for East Asian languages if that option is to be available to users of the product. With this approach, there are no language-specific versions of the source code, which would require separate builds for individual languages, or so-called code forks, which would branch on to specific language versions of the software.

This means that the code does not need to be modified after compilation just to provide additional functionality specific to a language or locale, or recompiled. The same applies to content or resources that get displayed as part of the running software, such as GUI, error messages, menus or other general string table content, which may need translation. In this model, the externalized translatable content is retrieved dynamically as and when needed.

These language-specific elements are typically separated from the program code that is in the single binary and are instead located in separate language libraries (.dll files). And remember, "English is just another language" here, as the popular saying goes. This means additional language versions (.dlls) can be easily added later on, which is especially useful when languages are released on a staggered basis, with increasing deltas, rather than simshipped with the original language version.

This physical separation of program code from resources that get translated makes one of the main objectives (or dreams) of internationalization come true, but also has some practical benefits as regards testing. For one, it makes it possible to effectively separate globalization or internationalization testing from localization testing. The core functionality, including international support, support for other languages and locales, or indeed any locale-specific functionality, can be tested just once, during the internationalization testing phase, thus eliminating the need for repetition or modifications for each and every language as part of the subsequent localization testing cycle.

Typically only a sanity check or a few test cases are included in the localization test pass to ensure language-specific or international functionality has been preserved in the product after localization. In practice, the internationalization testing may then be conducted by one team knowledgeable about globalization, rather than a disparate set of testers with required TL knowledge. As a result, few internationalization-related bugs get logged at this late stage. The benefits of removing duplicate efforts are clear and increase with the number of languages.

The lynchpin

Pseudo-localization testing is one of the lynchpins that connects localization with localization testing. Pseudo-localization testing is a type of testing designed to verify the localizability of applications prior to their localization. Executed normally as part of the internationalization (testing) phase, it follows the simple idea of pseudo-translating the localizable GUI elements and other software strings so that standard display characters are replaced by potentially problematic characters or other characteristics of the given target languages, or other international characters. These pseudo-translation characters either replace parts of the source strings or are appended at the beginning or end of strings.

Such a pseudo-localized product may be then tested to verify its general functionality for when it eventually is localized — including issues resulting from over-localization — as well as its international support such as inputting extended characters, fonts and so on. It also helps uncover GUI issues such as truncations, concatenations or hard-coded strings. With this approach, specific aspects of each TL are normally taken into account, such as the typical text expansion. Generally, some 70%-80% of bugs that would be normally logged later on during localization testing are caught at this earlier stage as a result of pseudo-localization.

In addition to the practice of using existing translation memories to provide rough translations as an alternative to random pseudo-localization, a new possibility is now being offered by machine translation. This allows for automatically translating software strings to the selected target language on the fly and so getting very close to what the properly localized application will eventually look like, and how it will function.

A number of current tools provide good support for pseudo-localization, including visual localization tools such as SDL Passolo and Alchemy CATALYST, and pseudo-localization is naturally included as one of many features in internationalization tools such as Lingoport's Globalyzer.

The most frequent subcategories of localization testing are the build verification test, the smoke test, GUI testing, localization functional testing and linguistic testing. A build verification test or subsequent smoke testing follows a similar objective of eliminating potential bugs before localization testing proper starts. Typical build acceptance test criteria include the questions: is the build installable? Does it run? Is it free of major flaws? Can it be tested further?

Another set of build verification tests verifies that all the required files and folders, whether localized or not, made it into the localized build and are present in the correct version. With thousands or in some cases tens of thousands of files present in today's applications, it's easy to omit some files or include extra files into the localized build. That might cause obvious functionality problems later on. The way to identify such an issue is by performing large-scale files and folders structure comparisons. The possible scenarios used include:

- Compare current localized build vs. current source language build. The purpose is to find any files that might be missing in the localized product but which should be present by design.
- Compare current localized build vs. former localized build. Ideally, they should not differ except for the files in which bug fixes were made.
- Compare differences vs. a Table of Expected Changes for specific language versions of a given product.

The next step after successful build acceptance is a smoke test. Just like in plumbing, where smoke is used to locate any possible leaks in piping after construction or major repairs, smoke tests in software testing are a set of simple verifications done by testers before they accept a new build for actual testing.

A smoke test aims to reveal simple defects that would prevent correct execution of localization testing or would result in too much "noise" — unnecessarily tracked bugs, which would cause a ripple effect and would consume time and resources for testing, bug verification and bug closing, especially in the case of multilingual testing.

The typical sequence of steps in this brief and inexpensive test includes installing the application, starting it, creating a file using the application, saving and closing the file and then restarting the application again. Next, open the file just created, make a few changes to it, save it and close again. The process is rather short and straightforward, and it is wise to automate it since it is repeated with different languages and on different test passes.

GUI testing is the element of localization testing that is completed once a build is accepted. The usual types of defects found include:

- text expansion, resulting in truncated strings
- GUI alterations, resulting in overlaps of GUI elements and controls or their misalignment
- automatic hotkeys assignment, resulting in duplicated hotkeys
- hard-coded strings, resulting in untranslated strings
- unsupported code pages, resulting in garbage čħaŗäςŧə®ß
- missing or extra controls, resulting in missing or broken functionality

Some languages may have significant text expansion compared with English while some may result in text contraction — Japanese is a typical example — but regardless of the language, a missing piece of text rendering the text incomprehensible is normally considered a critical/stop shipment error, not a cosmetic bug.

Most potential cosmetic GUI bugs are normally eliminated during the localization phase, thanks to localization engineering of the GUI before creation of a new localized build and to the constantly improving features and verification checks available in current visual localization tools. Problems may occur with dialog boxes that are generated dynamically or where a UI feature is composed of several dialog boxes.

In a similar vein, duplicated hotkeys are also all too common, a problem that may arise when hotkeys are assigned automatically. They can be eliminated through standard checks in localization tools, but duplications may still occur at runtime when menus or dialog boxes are combined. Hotkey assignment is a small issue with a potential big impact. Similar to the use of shortcut keys, users may get used to a sequence of hotkeys to access specific features quickly and will be flummoxed if a standard set of sequences doesn't work in a new localized version of the product.

Think also about the number of visually-impaired users who use means and tools other than visual control and a mouse to work with applications. These users rely on shortcuts. Ideally, existing shortcuts should not change between releases and should have a high degree of conformity with the operating system or similar and related applications. With automatic hotkeys assignment and the high level of content and UI element recycling available today, it requires a careful localization effort to keep this perspective.

GUI test automation

There are three basic approaches to visual inspection of an application's GUI at runtime:

- manually — by a person sitting at a computer following a script and opening every single dialog box or other piece of the application and examining it by naked eye for any errors.
- semi-automatically — by controlling the localized application manually but running tools that automatically locate and report GUI errors.
- fully automatically — by using automation tools and scripts that control the tested application and find and log errors found. Borland SilkTest or HP QuickTest Pro are examples of such automation tools.

Taxonomy of GUI bugs in localization

Layout	Hotkeys	Text	Graphics
Text truncation	Duplicated hotkeys	Untranslated text	Missing graphics
Control truncation	Missing hotkey	Mistranslated text	Different graphics
Misalignment	Inappropriate hotkey	Unexpected text	Untranslated icons
Overlapping		Inconsistent translation	
Tabbing order		Technical inaccuracy	
Oversized dialogs		Double space after full stop	
Different layout in general		Wrong alphabetical order	
		Wrong date/time format	
		Corrupt characters	

While these test automation tools are not designed specifically for localization GUI testing, they can be used very well for this purpose. Let's look at the following Borland SilkTest script example:

```
PushButton("Help").Click()
// will fail on a localized product
```

When it operates, such a script would simply click on the Help button in a dialog box on the screen. But once the application is localized and `"Help"` is translated to something else, the Help parameter will not work, and this script will fail. The modified SilkTest syntax looks as follows:

```
PushButton("Help|#3|$9").Click()
// will pass on a localized product
```

The unique identifiers after the UI element `Help` allow SilkTest to click on the proper button regardless of whether the word `Help` got translated or not. Such a script will run on any language, including bidirectional or Asian DBCS languages, because it will always look for these unique identifiers and it will simply ignore the caption.

Test automation is *de rigueur* today, and GUI testing is frequently 100% automated. The reduction in efforts, costs and time compared with resource-demanding manual testing can be significant, especially with a higher number of languages. Yet it is not the be-all and the end-all for every testing project. Automation requires preparation, which may offset the savings on smaller projects. Plus, automation tools may be expensive. It always pays off to conduct an analysis to determine the effectiveness of procedures completed as manual or automated processes. The parameters analyzed should include:

- process complexity — the higher the complexity, the more need there is for qualified test engineers to lay their hands on the tested product.
- number of repetitions — a higher number makes the case for automation more compelling.
- human judgment required — automation will not replace human judgment if that is needed to decide on a course of action or to determine if a true nonconformity has been found.

In practice, much efficiency can be obtained from semi-automation of testing. In this case, test engineers follow test scripts and control the application, but are assisted by an automation tool that checks at runtime for GUI defects and logs them automatically. This reduces the time needed to spot a GUI bug visually, reduces the scope for potential errors or omissions (such as truncated texts that seemingly make sense even though they're truncated) and enables testers to make an intelligent judgment about the defect and its severity.

Further testing

Executed concurrently with GUI testing, localization functional testing helps answer several questions. Does the product install correctly? If so, in all installation scenarios? Do all features work as expected after localization? Does the product accept international characters on input? Is it compatible with supported operating systems?

It makes sense to start by testing the installer and to test many different installation possibilities: to test all of the installation options (Standard, Compact, Custom, Full); to try to install to all kinds of file locations — not only the standard location but also to a customized path, preferably with international characters in the path name; and to try to install to an unwritable drive such as a CD-ROM, a USB Flash drive with insufficient space, to a nonexistent drive or to a remote disk with limited write rights.

The purpose of such extended testing of the installer is to ensure that it is possible to install the application under various conditions and to know it won't fail, and to verify the translation of all error messages and alert messages shown to users via the installer.

The testing of actual product features follows next. Few software applications today are truly brand new to the market. Many or most have a history of previous releases, so there are typically two kinds of features in the application: the so-called new features that you have just embedded in the application in the new release and legacy features. A thorough testing of all of the new features is imperative because they have never been tested for localization. It is also important to do a spot-check of the product's legacy features. They have been tested in localized versions before, but re-testing will ensure that all legacy features are working and their functionality has not been broken during the product's update.

This normally requires using manual test cases, not automation testing, because unlike with GUI testing where screenshooting of dialog boxes of an application is a rather simple task, functional testing requires human judgment.

Severity: Expresses the impact of a bug to the end user, usually scaled 1-4.
1: Crash or hang bugs, loss of functionality, copyright issues, offensive text and so on.
2: Major, mainly functionality (or critical text is not visible).
3: Minor bugs, mainly cosmetic errors.
4: Trivial, very small bug — missing punctuation, for example.

It is a good practice to reuse source test cases. It's also useful to have tiered prioritization of languages for testing because you can't test everything on all languages. Teams should be ideally selected based on specialization in test areas — such as build verification tests or automation testing — or product features/components rather than by languages. Proficiency in testing of a particular component or feature gained through repetition on all TLs being tested is greater than the advantage of having a tester responsible for the whole product in one language.

Having one single defect tracking database for localization testing, functional testing and for all languages is highly advisable. Once you find a functional error in a specific language version of an application, such as an error that causes a crash, you are likely to find the same error in most if not all languages. Similarly, you might encounter the same bug with all Asian languages because whatever the error is, it simply doesn't handle double-byte characters properly.

With one bug database for all languages, you can log a critical error once and then filter for all critical errors regardless of in which language versions they occur. You can then cross-check for these critical errors to see if they reproduce for all languages. This is important since while GUI errors (such as truncations) tend to be language-specific, functional errors may be shared across all languages or their subset (such as all Asian languages or all bidirectional languages).

A clever test strategy is necessary. Imagine you have a product localized into 14 languages, with 100 tests that you need to conduct on every language and that there are five operating platforms that you want to make sure you're compatible with. Then the need to test in three browsers enters the equation. So suddenly you end up with having a high number of tests that you need to conduct if you want to complete an exhaustive evaluation. All of this might leave your localization testing effort over budget and out of time. A clever test strategy calls for choosing the smart way of reducing the number of permutations while still maintaining good test coverage.

Pairwise testing, for instance, is one such method. It is concerned with combinatorial generation of test cases and is based on the fact that most errors are caused by the interactions of not more than two factors. This testing approach therefore covers all combinations of two factors (hence pairs) and so results in a much smaller number of required testing combinations than would be arithmetically calculated. This brings reasonable and feasible test coverage while still offering a high probability of finding the majority of defects.

Linguistic testing is conducted by language-aware or native-language testers on the actual localized product that runs exactly as it will be used by local users in their language. Such testing is required since much of localization takes place out of context, and much of software testing is conducted by test engineering professionals rather than language specialists.

The current approaches are:

■ standard linguistic testing with language resources located onsite or in-house, whether it is in dedicated test centers or via local in-country staff.

■ linguistic testing using screenshots of the localized product, conducted onsite or offsite. In this model, the language aspects of linguistic testing are separated from the engineering ones. Nonlinguistic test engineers prepare screenshots of all the requisite parts of the localized product's GUI and provide them to language specialists.

■ remote access by linguistic testers to a centralized test environment.

Recycling
Recycle, reuse, reduce. Translation, content, cost. Recycling is an old concept in the translation industry, but today's technologies are giving it a new meaning. Use our Machine Translation engines and post-editing capabilities to stay on top of what language technology has to offer. Eliminating waste applies everywhere — at home, at work and with your content.
moraviaworldwide.com/recycling
Moravia worldwide
moraviaworldwide.com
AMERICAS | EUROPE | ASIA

Just as we have the Localization Maturity Model for approaches to localization, developed by Common Sense Advisory as the industry-specific application of the generic Capability Maturity Model, testing professionals work with the Test Maturity Model. Developed at the Illinois Institute of Technology, this defines five stages of maturity from initial to optimized, where each stage is characterized by a level of maturity achieved in individual key process areas. The model helps companies assess their test process maturity and identify weak areas in need of improvement.

Over the past decade, we have seen a major improvement in how organizations approach localization testing and how they stack up against this model. Testing processes have become more mature and a lot leaner. Yet the success of localization testing is highly dependent on how companies approach the whole area of internationalization and localization and how they integrate all of these into one coherent process. It is here that we expect much efficiency yet to be gained in the years to come. **M**