

Using Enable with PowerBuilder to Internationalize Applications

By Gian Luca De Bonis



***Making your applications
useable in multiple countries
and languages***

Gian Luca De Bonis, CEO/CTO of Enable, has considerable international experience in PowerBuilder and database technologies. He can be reached at gianluca.debonis@enable-pb.com.

Today it is becoming increasingly necessary for applications to be available to users across international boundaries, in a variety of languages. Adapting these applications for local use, however, often creates time-consuming transfer and translation problems for application developers.

In this article, we examine how to localize internationally used applications to maintain their usefulness and accessibility. As part of this study, we examine the various approaches available for the localization of software. We discuss the Enable suite of tools, including a review of its architecture and a discussion of its multilingual database. The process followed to transform an application using Enable is then described, with particular emphasis on how meaningful source strings are identified.

Approaches to Software Localization

There are various methodologies for localizing software for international use. These include:

- ◆ Standard approach: use of identifiers
- ◆ Generation of multiple single-language source codes
- ◆ Use of emulators
- ◆ Integration with the application framework

The approach selected has an extensive impact on the final result.

Using Identifiers

The standard approach (used in virtually all programming languages) is to replace strings with identifiers. This irreversibly alters the legibility of the source code,

which now contains mnemonic constants (e.g., `ids_generic_error_msg`, `idw_no_data`) rather than easily readable text. The strings are then defined externally—either as a static resource, to be included when compiling the application, or as a dynamic resource to be read at runtime.

The static approach is more secure, since translations are protected from accidental or deliberate changes, but requires:

- 1) Distribution of an updated application every time the translation is changed.
- 2) Separate executable applications for each language.

Dynamic Approach

The dynamic approach faces the problem of unauthorized changes, which means having to sign the resources and validate them at runtime. Dynamic resources can be standard (e.g., the resource files used by environments such as Visual Studio) or custom. When translations are updated, it is now only necessary to deliver the external resources and the related signature file.

The combination of identifiers and dynamic resources appears to provide a good solution to the problem, but the following issues still need to be considered:

- ◆ The source code is no longer legible.
- ◆ The creation of identifiers is a non-trivial exercise and requires constant maintenance of the translation tables.
- ◆ The translation of resources in a standard format requires access to tools (workbenches) that allow translators (as non-IT experts) to do their job.
- ◆ The translation of resources in a non-standard format requires the development of dedicated tools.

Furthermore, it is sometimes necessary to change language dynamically at runtime, considering either the entire application or just specific parts, e.g., in order to print an invoice in the client's language without changing the user interface. In this case, the identifier approach does not help much, since special algorithms have to be written to allow this (unless restarting the application is acceptable). The same is true when dealing with "right-to-left" languages, which change the layout of the application.

Using Native PowerBuilder Tools

An alternate approach is to use native PowerBuilder tools, such as the Translation Toolkit. This tool extracts all the strings from the application (unfortunately together with various spurious elements, such as Describe, Modify, SQL statements presented as strings, strings representing fonts/icons/images) and allows the developer to specify which should be translated. Following translation, the tool generates a new source code for each language. This approach has a number of drawbacks:

- ◆ Separate installations for each language.
- ◆ New deployment after changes to the translation.
- ◆ No support for right-to-left languages or dynamic language change.
- ◆ Software changes generate major administrative issues, with multiple updates and re-releases of the applications concerned.

Using Emulators

Next, the use of emulators can be considered. These tools hook into the operating system's routines, intercepting write messages and changing the text to reflect the active language. This is a dynamic change approach that avoids all the problems mentioned above (and, in theory at least, does not require access to the source code). However, a number of new problems arise in this scenario:

- ◆ Developers have a low level of control and cannot, for example, choose what to translate and what to leave, while the application is unaware of the changes made at runtime, which can cause conflicts with framework routines for resizing and the changing of text.
- ◆ The use of several languages at the same time is complex and requires considerable interaction with the source code.
- ◆ The dynamic objects normally used by PowerBuilder developers [dynamic DataWindows, instructions such as OpenUserObject, Modify(Create), etc.] are not automatically captured and become rather difficult to handle.

Working at the Framework Level

Developers like maintaining control over their source code, which allows them to change implementation strategy with great freedom while keeping an eye on future requirements, compatibility and performance. The framework-level approach is founded on a function that translates an object and all its children:

```

subroutine of_translate(powerobject apo_object)
    choose case apo_object.typeof()
        case Window!
            Window lw
            lw = apo_object // Explicit cast
            lw.title = of_translate_property(lw.classname()+"title")
        ... (other properties) ...
            of_recurse_children(lw.Control[])
        end choose
    end subroutine

```

where `of_recurse_children` calls `of_translate` for each element indicated. The function `of_translate_property` returns the translation of a specified property in the active language, such as `w_customer.title = Customer list (in EN)`, `w_customer.title = Elenco clienti (in IT)`, and so on. Using an .INI file:

```

[EN]
w_customer.title=Customer list
[IT]
w_customer.title=Elenco clienti

```

This is very simple, but it still leaves a host of issues unresolved:

- ◆ We don't know the current value of the properties, so keeping the list of translations synchronized with the source code won't be easy.
- ◆ How do we analyze the phrases to be translated? Dynamic situations (DataWindows created dynamically, controls added dynamically to windows)?
- ◆ How do we deal with parametric phrases?
- ◆ When do we read the list of translations? Dynamically? On start-up?
- ◆ How do we handle computed fields in DataWindows?
- ◆ What happens when we rename a window (e.g., from `w_customer` to `w_customer_gd`)?
- ◆ When we upgrade to another platform (PocketBuilder) or to another version, we also need to update this part of the framework to take account of the new controls and properties.

Despite these issues, which just need management, the framework-level approach gives us great flexibility when deciding when and how to apply translations. For example, we could decide to translate a tabcontrol page, but only the first time it is displayed. Or, if we are reading data from a database and it needs to be translated, we could do the translation as a one-time exercise before displaying the information in the controls. In addition, the ability to define the way translations are to be applied in ancestor classes or sub-classes means that code can be more readily recycled, which fits nicely with normal development requirements.

Finally, the decision to work at a framework level ensures that developers have considerable control over what goes on, but it is also subject to a number of drawbacks:

- ◆ Special algorithms are needed to handle a non-trivial number of strings, parametric phrases, on-the-fly changes to translations, and the dynamic change of language and layout in open windows.
- ◆ Access to the translation database can be extremely inefficient when the database is remote (and when working with PDAs), depending on the platform and drivers available.
- ◆ The management of translations requires special tools, such as for synchronizing the work of translators.

Architecture Overview

Enable has been developed to take all these issues into account. See Figure 1 for an overview of Enable's architecture. The major components include:

Enable Author: Tool for managing multilingual databases, including the application's languages, source phrases, contexts, translation rules, and translations. Developers can also import/export translations, analyze the temporary file, and produce various reports.

Multilingual database (*.ENA): Managed by a DBMS engine embedded within both Enable Engine and Enable Author. This database contains tables, indexes, multivalued fields, and variable-length fields and records. There is automatic support for structure changes (version upgrades) and no installation or maintenance is required. The temporary file (*.ENT) is managed by the same engine, but has a sequential structure.

Enable Extractor: Fast parser based on PowerBuilder grammar that extracts significant strings from the source code and generates a temporary file. Sophisticated context-sensitive filters ensure that only relevant strings are imported using Enable Author.

Enable Engine: Runtime engine added to the application to read the multilingual database and apply the translations. A series of APIs allow detailed control over the results. Enable Engine becomes part of the application and part of the developer's own framework.

Enable Explorer: An innovative component of Enable Engine that facilitates management of the transformation process at runtime. Enable Explorer helps developers to alter the layout of screens, change translations, understand the visual structure of windows and take notes.

In order to install the application, the following elements must be added to its components: Enable Engine's libraries (PBDs and DLL) and the multilingual database (*.ENA).

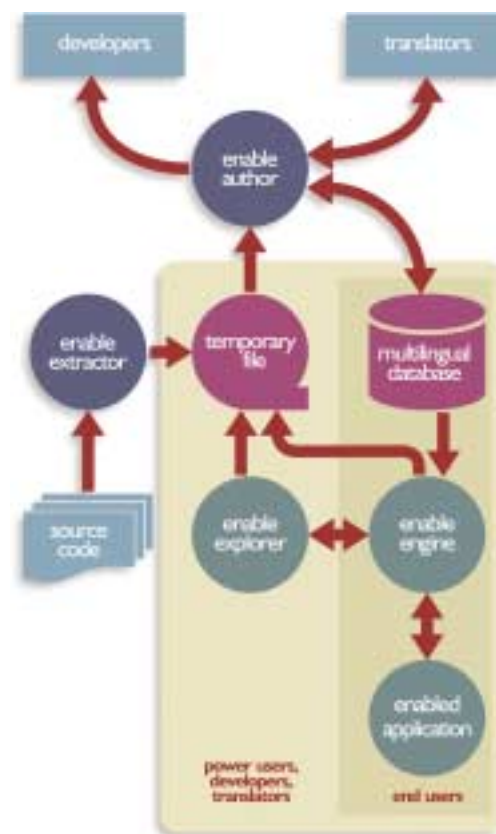


Figure 1: Enable Architecture Overview

As can be seen from Figure 1, there are three deployment environments:

- ◆ Runtime (end users): Comprises the application, Enable Engine, and the multilingual database.
- ◆ Advanced (power users, developers, translators): Includes the above, plus Enable Explorer and the temporary file.
- ◆ Development: Includes all tools.

The runtime elements of Enable (including Enable Explorer) do not require configuration, activation keys, or changes to the registry database. Enable is designed specifically for PowerBuilder: client/server, distributed, and PocketBuilder. All current versions are supported, including PowerBuilder 10.5. This compatibility encompasses Unicode characters and “right-to-left” languages.

Enable’s Proprietary Database

Given the key decision to take a dynamic approach, the problem is where to store the translations so that they are readily available. We could use resources (such as text files, XML, .INI), read the translations on-the-fly from a database, or read them into a datastore upon startup. We need to worry not only about performance, but also about the maintenance and distribution of the translations.

If we use a resource file (ignoring the signature issues), we might end up with something immense (1,000 objects with an average of 20 properties make 20,000 strings, to be translated into multiple languages), and perhaps segmented by language (but remember that we need to be able to change language dynamically and use two or more languages at the same time). The powerful datastore technology available offers us sophisticated cache mechanisms to store the translation tables in RAM; but populating and searching datastores is a non-trivial task when we are concerned about performance.

If we store the translations in a database, we need to support a suitable format optimized for each DBMS used by our applications (and sometimes we might not have a local database, as with distributed applications). What happens when we want to update the data structure to support new characteristics? We also need to update the related translation structures, which is the classic problem within a problem.

Furthermore, when we need to synchronize translations or send them to clients, the process needs to be as painless as possible for the end user. Of course this is possible, but we need to bring many tools to bear.

The proprietary (royalty-free) database developed for Enable has the following characteristics:

- ◆ Same format across all platforms
- ◆ No need for maintenance, installation or registration:
 - It is part of the runtime engine
- ◆ Signed, to protect against unauthorized changes
- ◆ Supports the automatic update of the data structure (each version of the runtime engine can read all existing versions of the database)
- ◆ Supports tables, indexes, multivalued fields

- ◆ Has specific tools for importing/exporting translations and for synchronizing with translators
- ◆ New translations are implemented by sending the new database file (very small) to the end users concerned.

“Enabling” an Application

The following steps are necessary to make a PowerBuilder application multilingual:

- 1) Perform the enabling process:
 - a) Add Enable to the application
 - b) Link Enable to the application
 - c) Capture phrases
- 2) Complete the transformation cycle:
 - a) Run the application in Authoring mode
 - b) Translate phrases
 - c) Run the application with Enable Explorer
- 3) Finalize the transformation:
 - a) Deal with special cases
 - b) Deploy

These steps are considered below, omitting certain details for space considerations but focusing on the essentials.

The “Enabling” Process

Add the Enable initialization code to the open event of the Application Object, as well as the code for selecting the start-up language; initialization must occur as early as possible, ideally after having displayed a start-up splash. The following is typical code:

```
// Enable
n_enable = CREATE n_enable
integer li
li = n_enable.init("<project>.ena", n_enable.ci_mode_coverage, &
"<project>.ent")
if li <> 0 then
    MessageBox("Enable Initialization Error", "Error code: "+string(li))
    HALT CLOSE
end if
n_enable.setlanguage("<startup-language>")
```

Link Enable to the application, so that Enable Engine can cover and therefore translate a window or an object and all its contents. Normally, the following code is included in one of the open events of the windows ancestor class:

```
n_enable.translate(THIS)
```

while this code is included in the event called at the end of the opening, usually in post mode:

```
n_enable.translate_untranslated(THIS)
```

These rules cover the application in general. Specific methods are applied to complete the coverage in cases where, for example, certain parts of the application are changed dynamically after the window has been opened.

As a final step, the system functions responsible for visual aspects, such as MessageBox and SetMicroHelp, must be linked. Applications frequently use wrapper methods for improved control over these functions. In this case, it is sufficient to change the call to the system function as follows:

```
// code in n_cst_message::of_messagebox(string as_title, string
// as_message)
//MessageBox(as_title, as_message)// old
MessageBox(n_enable.translate(as_title, "<message>"), &
n_enable.translate(as_message, "<message>"))// new
```

Enable's documentation explains what to do if a new window ancestor class needs to be created and, if necessary, how to implement wrapper methods (highly recommended) to improve control over system functions.

The key changes needed to the application's source code are described above. Now it is necessary to capture phrases for translation, in order to complete the process.

All languages, especially 4GL and PowerBuilder, use strings as properties and subsystem commands, and not just as text to be translated. For example:

- ◆ Properties: tags, icons, pictures
- ◆ Methods: describe, modify, createsyntax
- ◆ Statements: create using, open window (specifying the class name as a string)
- ◆ Dynamic SQL instructions
- ◆ Embedded SQL containing strings for the database

Even using sophisticated filters with regular expressions (which are time-consuming to implement), lexical analysis inevitably captures many strings that do not need to be translated. There are no particular dangers with this approach, except for the amount of "rubbish" that is accumulated, to be thrown out, and awkward, without-context questions from translators, such as: What do you mean by select count from customer?

By contrast, Enable Extractor is a rapid parser based on PowerBuilder grammar which captures meaningful source

code strings (phrases) and ignores the rest. In particular, Enable Extractor uses rules (Figure 2) that can be personalized to analyze and capture:

- ◆ Desired properties (e.g., exclude TAG from windows, but not from menus and datawindows)
- ◆ Desired multi-value properties (e.g., code tables, menu descriptions that include tooltips, text, microhelp)
- ◆ Desired method parameters (e.g., just the first and second messageboxes)
- ◆ Desired expressions in datawindows (validation, expression, etc.)

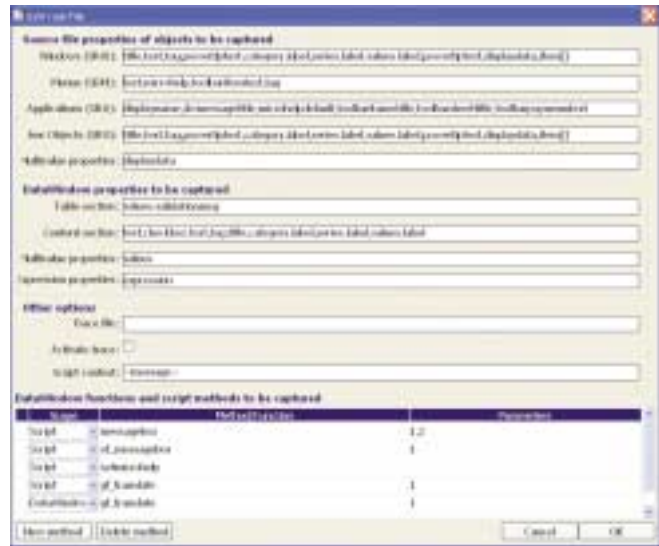


Figure 2

Enable Extractor analyzes: targets (*.pbt), libraries (*.pbl), source code (*.sr*) and text files, generating a temporary file (*.ENT). This output can be analyzed using Exclude If filters, based on regular expressions and DataWindow functions, which can be saved for further use. The contents of the filtered file are then imported, using Enable Author, into the multilingual database as source phrases.

The Transformation Cycle

Enable Extractor uses state-of-the-art parsing technology but, as with all static analyzers, it is unable to capture the dynamic strings that are generated when:

- ◆ A phrase comprises a combination of other phrases, some of which are obtained from functions.
- ◆ A static part of the string is chained together with another part read from the database or from an .INI file.
- ◆ A datawindow is created dynamically.
- ◆ Decoding algorithms are used to convert a code into a detailed description (e.g. an amount expressed in numbers is converted into words).

These strings are therefore captured by the dynamic analyzer available in Authoring mode. Enable Engine searches the database for each phrase presented for translation and, if required, adds any missing phrases after applying sophisticated filters.

Authoring mode captures any new strings found (including invisible objects) every time a window is opened, as well as the dynamic strings mentioned above. It is a good idea to run the application in Authoring mode during the development process, even while translation work is in progress. New information can be passed to translators without confusion.

Translations can be administered using the “Translations” function. The translation for each source phrase can be specified in general (general context), or more specifically, context by context. It is best to use this function for small changes, while the Export/Translation/Import cycle should be used for more extensive changes: MS Excel, XML, and Clipboard formats are supported.



Figure 3

Translated strings may require more screen space than the source strings, but the graphic layout can be altered easily using Enable Explorer (Figure 3). This tool allows:

- 1) The resizing and movement of controls and DWOs, considering the actual space requirements evident at runtime (e.g., the size of DataWindow controls in relation to the scroll bars and the number of records displayed).
- 2) “On-the-fly” translations of the application, proceeding control by control and considering all the phrases concerned, for the revision of translations in context and last minute changes (Figure 4).
- 3) Analysis of the visual structure of windows, considering the current situation (recognizing therefore any objects that have been created dynamically).
- 4) Notes to be made, in context, about real objects (controls, DWOs).

All the information gathered by Enable Explorer is written to the temporary file, which is analyzed and imported by Enable Author, with the generation of relevant PSR reports.



Figure 4

Since Enable Explorer is embedded in Enable Engine, it can be distributed royalty-free together with the application. This means that it can be used by work groups and translators during beta testing and afterwards as an aid to technical support.

Finalizing the Transformation

Parametric phrases—These are actually very common and must be handled carefully at the translation stage. Enable deals with them automatically, due to a detailed understanding of PowerBuilder grammar. The CPU-intensive pattern matching is pre-processed when writing to the database so that, on reading, all the elements are ready for the application.

The following example illustrates the nature of the issues. Imagine the following source code:

```
n_cst_msg.of_MessageBox("Status Notification", &
"Order "+string(li_num)+" for customer "+ls_cust+ &
"has been "+string(ls_status)+" today,"+f_localized_date(today()))
```

Where **ls_status** can take the values: sent, opened, closed, suspended and **f_localized_date** returns the date in the format of the active language.

Sometime we find that developers translate word by word, which is actually the wrong decision:

```
n_cst_msg_of_MessageBox(f_translate("Status Notification"), &
f_translate("Order")+string(li_num)+&
f_translate(" for customer")+ls_cust+ &
f_translate(" has been")+string(ls_status)+&
" today,")+f_localized_date(today))
```

Apart from the confusion in the source code, this means that the parameters will always be presented in the original order, ignoring the syntax of the active language which could well be different.

The correct approach is to leave the source code untouched, and to capture the second phrase in the following manner:

```
"Order @1 for customer @2 has been @A today, @3"
```

A match is then applied at runtime to replace the literal parameters (@1..@3) with the corresponding parts of the string to be replaced. Note parameter @A: This is a translatable parameter which represents a phrase. The various possible states are also phrases which are translated dynamically during the substitution process. An Italian translator might therefore translate the phrase as follows:

```
"Oggi @3 l'ordine numero @1, per il cliente @2, è stato @A"
```

The source code remains unchanged. In English, the end user sees: "Order 342 for customer Michael Devlin has been sent today, June 22, 2006"; while in Italian this becomes: "Oggi 22 giugno 2006 l'ordine numero 342, per il cliente Michael Devlin, è stato spedito". It is important therefore to substitute the proper parameters into each phrase, which is ideally the job of the parser, and then the pattern matching can be done at runtime.



Changes after windows are opened—Calls to methods that change the appearance of windows after they have been opened (such as a change of title, dataobject, validation message) cannot be covered automatically. The principal technique for dealing with this exception is to call the `n_enable.translate_untranslated(apo_object)` method at the end of the script; `apo_object` is the object changed by the script (could even be the entire window). For example:

```
// code in wf_setcurrent(), called by a cb_select button within w_customer
long ll_row
ll_row = dw_cust.GetRow()
THIS.Title = "Customer maintenance, customer: "+&
dw_cust.Object.cust_id(ll_row)
n_enable.translate_untranslated(THIS) // new
```

Changing the active language—There are two situations in which developers can change the active language:

- 1) On opening the application, using the method `n_enable.setLanguage()`
- 2) When windows are already open, using the method `n_enable.changeLanguage()`

Languages are codified, so developers can control which are to be made available to users.

Translation rules—Enable Engine normally applies the translations to all controls (not to data); it is possible, however, to define which specific controls must be translated using the "Translation rules" function within Enable Author. These rules can specify if a system or developer-defined class should be translated or not, both in general and at individual context level.

Non-displayed elements—Enable supports the translation of reports, text files, and all styles of DataWindow (including composite and nested reports).

Conclusion

Enable allows users to make PowerBuilder software multilingual with a minimal impact on existing source code and rapid results, while retaining full developer control. Enable users therefore avoid the problems of maintaining multiple source codes and the side effects of using emulators. Further development is planned for Enable Explorer to make it a useful runtime tool for all PowerBuilder developers, regardless of whether or not their applications need to be multilingual. ■