David B. Kumhyr
Tivoli Systems Division
IBM Corporation

Developing International Products

Most software development organizations now understand the critical importance of producing international ready products. Marketing and sales organizations have made this requirement quite clear to the product managers. What is not so clear is the technical means to achieve this goal; the method is usually left to the development organization.

Most development teams are composed of bright and talented programmers whose engineering education did not include internationalization training or exposure to linguistics[1]. Each globalized product solution developed is different and with multiple releases and many products the resulting control, support and development cost become enormous.

In Tivoli an internationalization team initially developed the first international product[2], which was a rewritten adaptation of the base product. Thus we took the simple brute force approach; rewriting the base product 'fixing' it for a target language and market. This approach is still a favored method for small development companies. The advantages are that the development can be outsourced to a team with internationalization expertise and the original development team can concentrate on the higher priority tasks of defect support and development of the next release.

The difficulty with this approach is that it involves the creation of a separate product for each market, each with its own potential set of new defects for the support organization to fix. Upgrades and fixes become increasingly difficult to manage. What seems a simple and cheap method to support a new market becomes a very costly ongoing effort.

A more supportable method is to develop the base product so that it will support all languages and locales, separating the human language and data formatting operations using accepted internationalization techniques. This approach creates a code base that is functionally the same in all locales, only the language is

[1] Other than computational linguistics.
[2] Tivoli Management Environment 3.1J for Japanese.

different. The languages may be applied during the product install or separately installable.

Two methods commonly used in the handling of the translations of these types of product is to have the development team handle the installation methods and transferring of files between translators or to have a separate team handle these logistics.

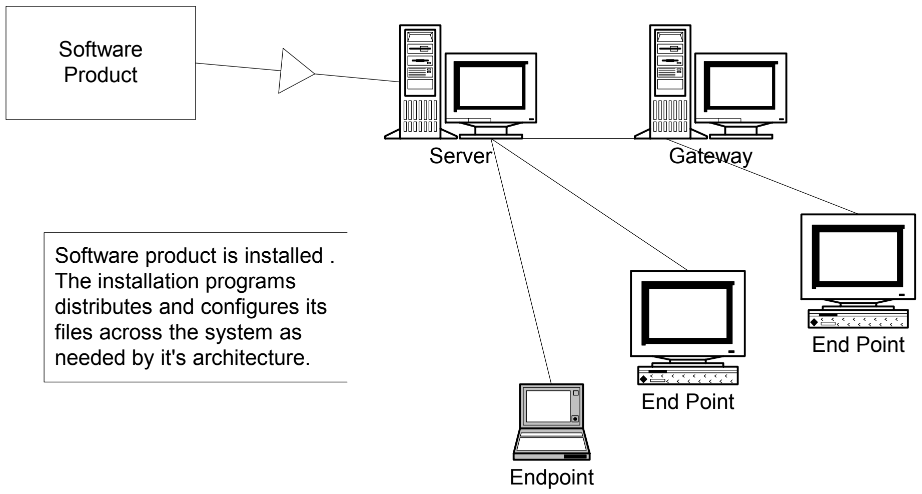
In Tivoli after the initial foreign product release the decision was made to have the internationalization team handle the translation and installation of the translated files. The development teams were unsure of how to do so and did not have the extra personnel to spare for additional development work.

At this time the entire Tivoli product suite was based upon a common programming model. This model allowed external products to be integrated into the Tivoli suite. Using this model for separate development enabled the internationalization team to develop translations and deliver them as products to be added on the base product as a translation. Thus was born the ‘language pack’ for Tivoli.

Language Pack Background

Language packs are certainly not unique to Tivoli it is a common and logical method for separating the translation from the development work. They are implemented in primarily two methods; development built or translation built; within IBM they are mostly development built.

In the development built model the development team assigns some of their personnel to the development of the language pack. The language pack may be integrated into the base product and executed as a choice during product installation or as a supplemental part of the product install. These developers integrate the translated files into the project and deal with the translation team and the translation verification team as well as perform all of the engineering of the installation and configuration of the product for the chosen languages.


Software Product
Server
Gateway
Software product is installed . The installation programs distributes and configures its files across the system as needed by it's architecture.
End Point
End Point
Endpoint

In the translation built model a separate team works with the development team to send language files to translation, create the language installation and coordinate testing. This architecture allowed for translation and translation testing to be decoupled from the base product development. This helps schedules and reduces the load on the development team but requires a service organization to support the translation, packaging and installation.

The translation built model is the original method chosen by our internationalization teams. Initially the process worked well, but as products multiplied and product architectures changed it became increasingly troublesome.

Trouble in Paradise

Quite soon each internationalization engineer in charge of developing a language pack found himself working on five or more different products with multiple releases during the year. Each product had different architectures, tools and build methods. The internationalization engineer needs detailed product architectural knowledge to produce a working language pack. They must know how the product locates, loads and uses the translated files. While many internationalization engineers were annoyed by problems in the language pack process most chose simply to live with the problems developing coping strategies for each individual symptom.

Engineers are Revolting

A number of engineers were concerned with the increasing inefficiency and approached product management with a manifesto, not to cause trouble but to point out what had become a poor programming paradigm and propose a radical solution.

Why language packs should be built product development

1. Development knows their product and the architecture; outsiders must reverse engineer and duplicate starting from the current state of the build.
2. Creating a separate langpack product duplicates defects of the original product while creating new potential defects.
3. Changes to the base product necessitate changes in the language pack, however there is no established conduit for notification. The primary notification is a build break due to a change. Example – addition of new java files with a new classpath.
4. Creating language packs for development effectively relieves the development team from considering the implications of NLS[3] concerns in their design and implementation phase. Weakening learning and creation of properly enabled programs. It brings these issues back to development late in the development cycle as defects from globalization verification test (GVT). These built in defects now are mostly deferred since they are intrinsic and require major effort to fix. Removing the NLS education and responsibility from base product development stunts their growth and knowledge and sets enablement back.
5. Overloading of file types (example .xml as a source for html, java and msg cats) means each file must be handled individually in builds based upon explicit development knowledge of the processing intended for each file.

The New Language Pack Manifesto

First I'll explain the points of the manifesto which is a list of the problems faced during development of language packs using the translation build language pack model.

"1. Development knows their product and the architecture, outsiders must reverse engineer and duplicate starting from the current state of the build."

As an engineer building a language pack we must reverse engineer the product build tree and file locations in order to first find out how the product builds the base language files (English), where the files are placed after the installation and how the files are accessed (path, environment variables, etc.)

"2. Creating a separate langpack product duplicates defects of the original product while creating new potential defects."

In addition to duplicating the product (multiplied by the number of languages translated into) we also manipulate product files and environment posing the potential for creating new defects.

[3] National Language Support

"3. Changes to the base product necessitate changes in the language pack, however there is no established conduit for notification. The primary notification is a build break due to a change. Example – addition of new java files with a new classpath."

> During development there are many changes, our language pack design may be predicated upon things that may have changed since our design breaking our language pack. There was no process set up to identify and notify us of these changes.

"4. Creating language packs for development effectively relieves the development team from considering the implications of NLS concerns in their design and implementation phase. Weakening learning and creation of properly enabled programs. It brings these issues back to development late in the development cycle as defects from GVT. These built in defects now are mostly deferred since they are intrinsic and require major effort to fix. Removing the NLS education and responsibility from base product development stunts their growth and knowledge and sets enablement back."

> While the development engineers were happy to be relieved of the need to work on internationalization issues it also kept them in isolation from this important skill. Development engineers were inadequately prepared to solve enablement defects that were reported late in the development cycle – as a consequence of separate language pack development.

"5. Overloading of file types (example .xml as a source for html, java and msg cats) means each file must be handled individually in builds based upon explicit development knowledge of the processing intended for each file."

> As Tivoli moved to greater usage of XML and it's ability to be used as a base for multiple target file types (type overloading) generic build rules were not adequate to process them. Individual product architectural knowledge was required.

Product management was well aware of the problems. They agreed to review our solution.[4]

Callback Language Pack Model

The Callback Language Pack Model (CLPM) is a new approach, which divorces the process of product development and translation delivery from the development architecture. Using object encapsulation techniques we isolate and separate as much of the product development details from the translations details.

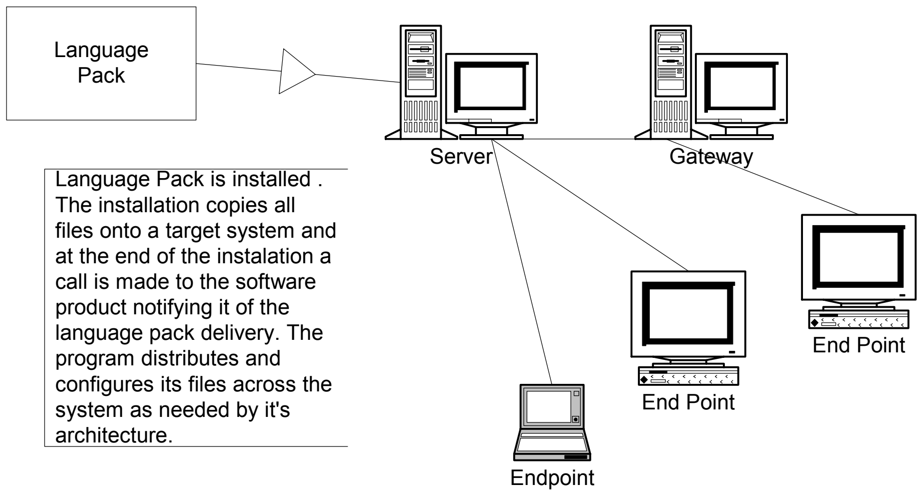
[4] IBM Research Journal, "Product Architecture Independent Method for Language File Delivery", David Kumhyr, Perry Statham, Keiichi Yamamoto.

Our new language pack would consist of the translated files and be delivered on separately installable media with a standardized language pack Installation program. This installer can install the languages for one or more product at a time.

The function of the CLPM installer is limited to unpacking and moving files to a location accessible by the product and to notify the product the delivery has been made.[5] Another attractive point is that this architecture is broadly applicable to diverse software products. The language installer can be standardized and internationalized installer that is uniform in look and feel and operation across the entire IBM product suite.

After it has received the notification the application program processes the language files in the manner it needs. For example the application program may move files to new locations or deploy files to endpoints. The application may also modify paths, classpaths and environments variables.

This encapsulation of function and data allows the translation team to translate and deliver without detailed product knowledge and the product team to ship their product prior to translation being completed. The architecture further allows additional language to be added without product architectural changes, corrections to be made to translations without product impact and patches made by program owners without language impacts.



Language Pack is installed . The installation copies all files onto a target system and at the end of the instalation a call is made to the software product notifying it of the language pack delivery. The program distributes and configures its files across the system as needed by it's architecture.

[5] An easy way to understand the process is to think of a freight delivery; the delivering agent may have no knowledge of the contents nor your intended use or the ultimate location within your house. You do have the knowledge so following the delivery you perform the moving and configuration.

CLPM Components

The CLPM installation consists of the following two components:

- An independent installer to deliver the translated versions of the product files and manifest to the target system.

- A call to a product supplied executable to process the files in the product's preferred manner.

CLPM Method Details

A uniform language pack installation program is used as a container to install the language pack. A simple and consistent interface allowing the installing user to select the products and languages they wish to install is presented. The user selects their choices and executes.

Each language pack needs only a very few stable pieces of data in advance relating to the product it is installing for. These are:

- Product title
- Product version number
- Product AVA code[6]
- Location of product callback
- Execution command for callback
- Authority for execution of callback

When the installation of the language pack is executed the files for the selected languages are copied to the target system. After the files are copied the installation calls a procedure that invokes the target program's callback executable. The product callback uses a standardized naming convention (the AVA code of the product) and passes the version number of the product and the

[6] IBM products each have a unique 3 letter code called AVA code. Other companies may have similar methods for categorizing their products.

fully qualified name of a manifest file listing all of the delivered language pack files.

The callback program is named using a standardized naming convention; the IBM AVA code of the product combined with "_NLS". For example Tivoli's Weblogic PAC product implements it's callback as a UNIX shell script. It is named GWL_NLS.sh. This script parses the list of files and performs the product language configuration (adding classpaths) and then distributes the files to endpoint nodes.

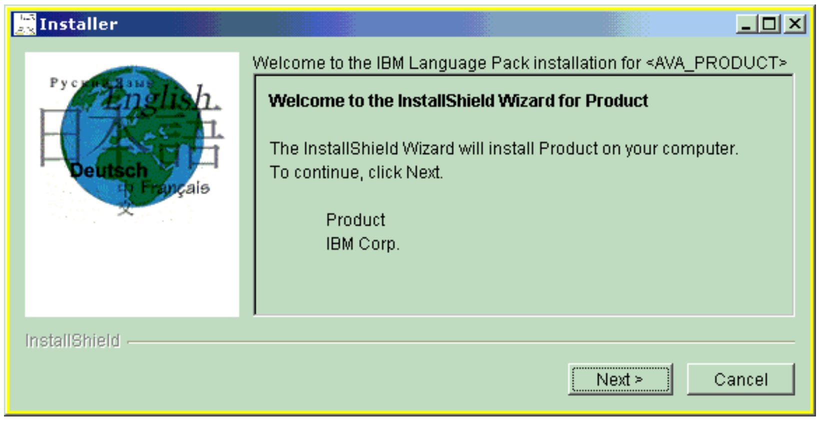
The method of separating the language delivery from the distribution and configuration relieves the internationalization team from modifying the products classpath or other environmental configurations. This frees the product groups from concern that the language packs are making product modifications outside the scope of their control. It also decouples the internationalization team from having to know about product specific manipulations of files. The processing of the language pack files should be done paralleling the processing done by the development team for their base language files.

CLPM Installer

This component is responsible for copying the products translatable files to the target machine. How the files are delivered to the target system is not essential to the architecture independent language pack process. The features that should be implemented by a CLPM installer are:

- The user interface for language pack installers is uniform for all products.
- The installer should execute culturally correctly in all locales.
- The language presented the user is selectable.
- The user may select **any** or **all** of the languages to be installed and that they may be installed at **any** time after the product is installed. (install can also be used to refresh translations)

The installer must also create a manifest file of all of the files that are installed and a final installation action that calls the correct <AVA>_NLS executable following the successful installation of the files.



Welcome to the IBM Language Pack installation for <AVA_PRODUCT>

Welcome to the InstallShield Wizard for Product

The InstallShield Wizard will install Product on your computer.
To continue, click Next.

Product
IBM Corp.

E-Language Pack Installation

In a web based or e-business environment the language pack may be installed as a jar file that is transferred to the client upon a specific request (as selected by the user) or as a result of a browser setting change (like a lang or local attribute changing). The deliver mechanism in this instance would be a jar file delivered for the registered applications though the installation process would be similar utilizing a call-back process to execute the installation and configuration for the language files.

Product Installation Actions

Following the successful installation of the files by the CLPM installer a process needs to be executed that will perform product specific actions on the delivered files. The mechanism to perform these actions can be any type of executable entity. This entity is executed and passed the version number, path location and name of the manifest file.

Product Installation Executable

This executable performs the product specific actions. The default naming for this executable is created using the product AVA code; <AVA_NLS a UNIX shell script. Some of these actions are checking that the correct version level is met and parsing the manifest file. Each file is then processed into the format need by the product. For instance individual .class files can be added into the products base language jar file or moved into class directories. The executable also moves these files to their destination for the programs use for example in distributed applications moving files to endpoint nodes. If early language binding is done the files are moved to the binding location.

In Tivoli NLS team created a sample template <AVA>_NLS executable. This shell script prototypes functions to parse the manifest file, move files into other locations, create/modify dependency sets, update jar files, etc. This example is a Tivoli PAC specific script, products are not limited in the type of executable entity that they choose to use for their application.

Example pseudo code for a product installation:

```
// Generic actions
Check version level
 if !ok – error exit
Check manifest file exists
 if !ok – error exit
Check files specified in manifest
 if !ok – error exit
// Application specific actions
Process files in manifest list that need further (e.g. jarring into specific archives)
Move files to server locations
Move files to other locations (e.g. nodes, clients)
Update paths, classpaths other environment variables
Exit
```

Automatic Generation of Language Pack

In Tivoli the distribution, source control and receipt of files for translation is handled through an automated process. This process is encoded in a tool called WebFM (web file management), it is possible to launch a process as a last step of the file processing in WebFM.

This automated process could build a language pack from the translated files. For example all .msg files could be compiled into .cats and then be compiled into an InstallShield image. Using InstallShield Multi-Platform to create a template installer .XML file the values that need to be altered between different products can be easily identified and substituted. This would allow the automatic generation of language packs as a last step in the file management system.

Major Advantages

A major advantage of this architecture comes from the fact that it is now possible to separate the current entanglement of the language pack development from the product development.

Potential advantages are that products can dynamically create classpaths without impacting international enabling. Development regains control over all of their product files and locations. Defects won't be inadvertently introduced by localization.

- Allows translations to be updated in the field at any time
- Prevents product functional defect introduction by translation
- Enables decoupled development and translation
- Creates language packs that are uniform across products
- Creates a universal language pack

Delivery of translations for software products is a major planning, coordination and design task for development and translation teams. The document describes a method for delivering translations that is independent of development architecture or tools. This architecture allows translation and translation testing to be decoupled from product development. Eventually the production of language packs may even become fully automated.