


The Theory and Practice of Pseudo-Translation

Internationalization and Unicode Conference #30

About this Presentation

- 
- ◆ Addison Phillips
 - ◆ *Internationalization Architect, Yahoo!*
 - ◆ *Introduce Pseudo-Translation, its uses and several methods of implementation.*

What's Pseudo-Translation?

- ◆ Convert text (typically English ASCII text) to non-random non-ASCII.
 - Typically to test a product for “localizability” before actual translations are ready.

"key", "display string"

"dialogTitle", "Dialog Title"

"aMessage", "This is a Message."

"key", "öisplay stríng"

"dialogTitle", "Điálòg Títlè"

"aMessage", "Thìß ís â Mésßägê."


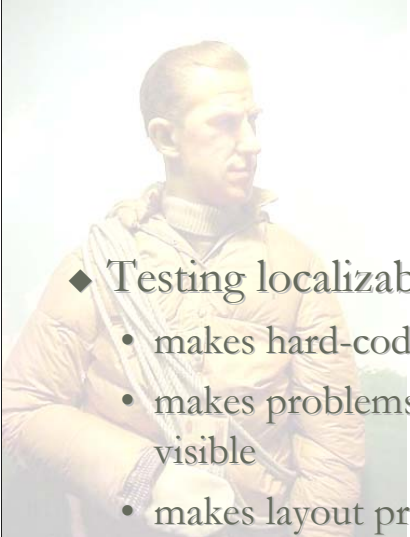
This presentation is about pseudo-translation or, variously, *pseudo-localization*. Sometimes you'll see this abbreviated as "PL". Basically, pseudo-translation starts with a source text, typically an ASCII text, and converts it to some non-ASCII text algorithmically. Because it is automated, it can be applied to a source text much more quickly than a translation can typically be done and it produces results that are predictable following a particular pattern.

Traditionally, pseudo-translation is done to test a product's *localizability*, that is, whether it can be successfully localized or translated.

What is it good for?

- ◆ **Testing localizability**
 - makes hard-coded strings visible
 - makes problems with resources and resource formats visible
 - makes layout problems emanating from length/height/font changes visible

- ◆ **“Does anybody really do that any more?”**



Pseudo-translation allows localizability to be tested in a number of ways. Mainly, because the “target language” is recognizably English, the quality engineers working on the product don’t have to struggle with a foreign language. Yet the kinds of changes that affect a localized product are simulated for test purposes.

For example, any hard coded strings will be visible as unmodified English, while items that are formatted by the runtime, such as numbers and dates, should follow the locale settings on the computer and, of course, the pseudo-translated strings are modified.

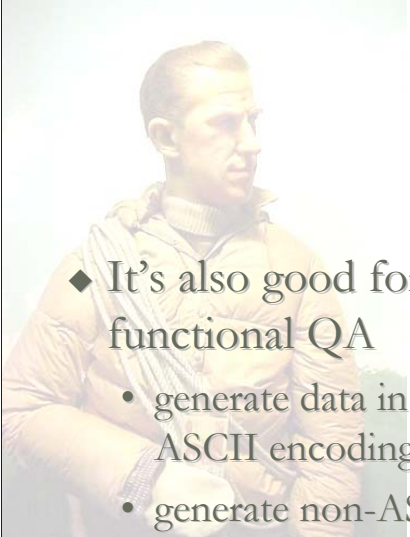
It also makes clear that the resource loader or resource formats work properly. If the program won’t load the pseudo-translated strings when expected, it probably won’t load the translations later.

Finally, the localized strings can have attributes very similar to those of an actual translation: they can be longer, taller, or use fonts or display capabilities that regular ASCII text doesn’t exercise.

When I was chatting with a colleague about this presentation, he very patiently listened to my description and made a couple of helpful remarks... and then he asked the fateful question: “Yes, but does anybody really do that any more?”

It turns out that people do. Let’s see why...

(Note: pictures in the presentation were taken from the associated text using Yahoo! image search. I don’t know why the kitten is associated with that sentence!)



Testing Software

- ◆ It's also good for functional QA
 - generate data in non-ASCII encodings
 - generate non-ASCII data
 - generate test cases from ASCII

What to Test With

- Test Non-English configurations
 - Non-English locales (lying to your machine)
 - Native configurations
- Test Non-ASCII data
 - Encodings, encodings, everywhere
 - Non-ASCII character values
- Test Across Time Zones
 - Two or more and consider the international date line and DST issues

One thing that Yahoo! uses pseudo-translation for is functional QA. While localizability is useful to test, platform QA or Web properties can also benefit by testing with non-ASCII data and particularly non-ASCII encodings. Using a good pseudo-translation suite, QA engineers can generate data in various encodings or formats to test support for non-English or non-ASCII.

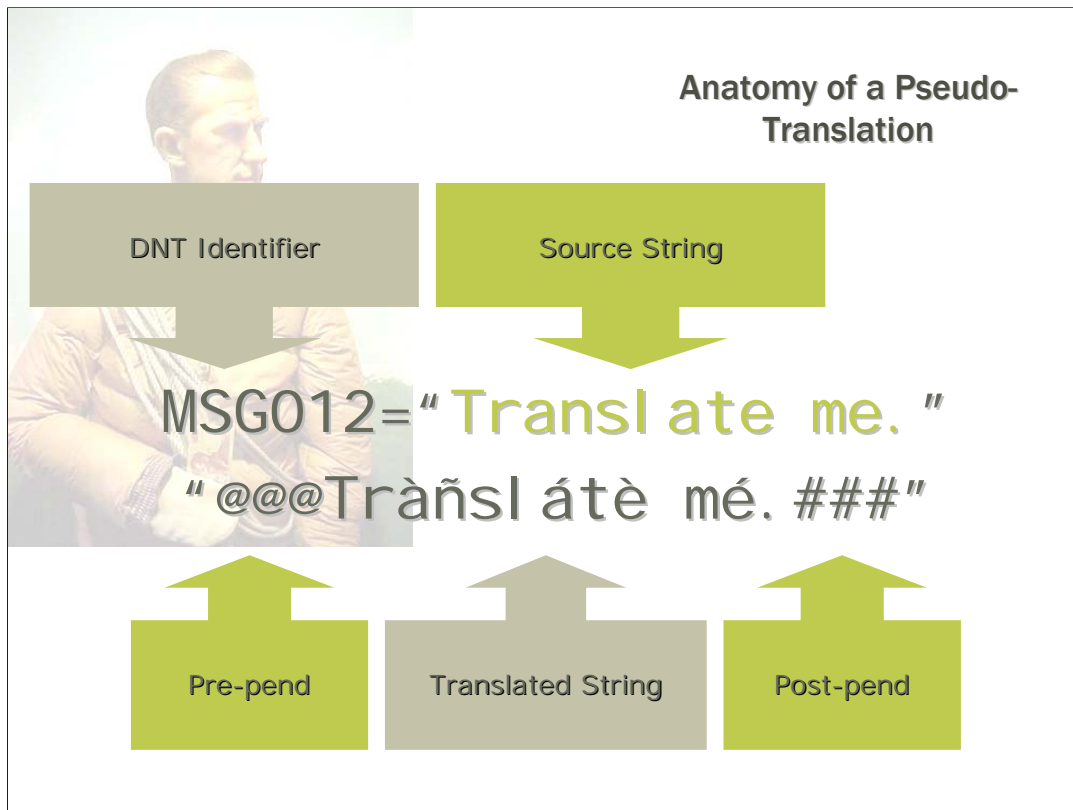


Your Mission...

Pseudo-translate text to support localizability and other test purposes using general purpose code.

- Easy to deploy and customize
 - Support functional, encoding, and localizability testing

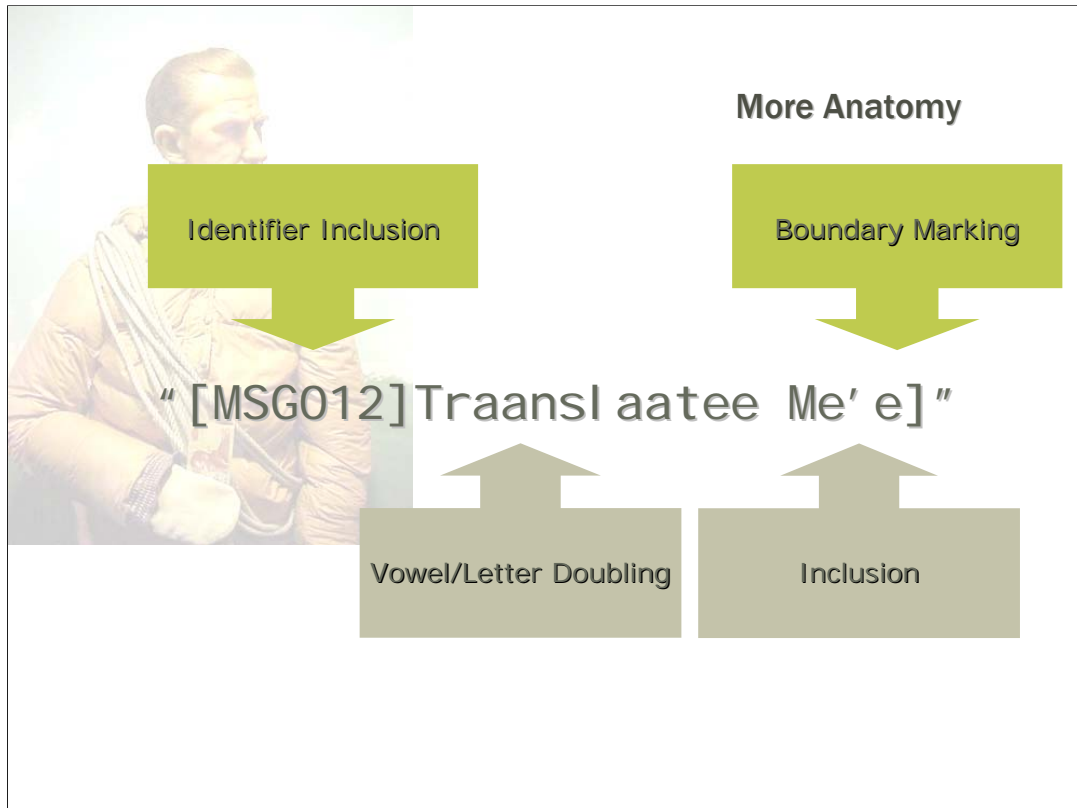
So... your mission is to create a pseudo-translation system that is easy to deploy and customize, which supports both testing and localizability needs.



Let's pause to look at the structure of a pseudo-translation.

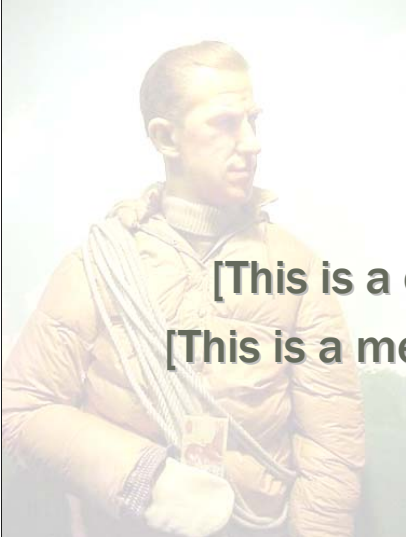
The original text, typically an ASCII text, is called the *source*. Source materials may come in many formats. The source will also include items which must **not** be translated, such as resource identifiers and the like. The localization industry way of referring to such formatting gunk is "do not translate" or *DNT* material.

The *target* string consists of multiple parts. Obviously there is the translated string. It may also be *pre-pended* or *post-pended* with additional character sequences to make the text more visible as well as to make it longer.



The translated string may also have additional letters inserted into the body of the text. A typical example would be vowel doubling (or tripling or ...), as shown here. Some translation schemes use algorithms to variably adjust how much longer to make the pseudo-translated string. For example, a short source string might need to be expanded more than a longer string to simulate the effect of translation.

In some cases, to help quality folks identify which message is causing a problem, the pseudo-translation will pre- or postpend the resource identifier. This is called *identifier inclusion*. Other types of *inclusion* might be *boundary marking* or the inclusion of additional characters in the interior of the string. In the example on the slide above, vowel-doubling is one form of inclusion. The string boundary marker "]" is another. And the embedded apostrophe is a third.



Interesting Boundary-Marking Patterns

[This is a concatenated][string][.]
[This is a message-formatted [string].]


Boundary marking is often done to make string concatenation visible in the user interface.

In the examples above, notice how the first sentence's pattern has the boundary markers arranged in a non-overlapping pattern. The translator is going to have a string such as "." to deal with, as well as a string with a dangling space ("This is a concatenated "). Word order rearrangement will be a problem in this string.

The second pattern shows how the runtime string "string" is inserted into the sentence by a formatting function. This does present word-interdependency problems (it still isn't a fully internationalized string), but this is a better pattern than the first. If the word "string" were a runtime-formatted date, number, or piece of data it might be okay.

Types of Pseudo-Translation

- ◆ Algorithmic Translation
 - Maps code points using “ASCII Math”
- ◆ Mapped Translation
 - Maps code points using a table
- ◆ Random Translation
 - Maps code points at random
- ◆ Simulated Encoding
 - Generate encoding-based test string
- ◆ Auto-magic Translation
 - See graphic
- ◆ More?



So far we've looked at how a pseudo-translation is structured. Now let's look at the different types of pseudo-translation one might consider. The slide above shows a number of basic strategies for creating a pseudo-translation algorithm. Certainly there are other options. Most pseudo-translation schemes can also be tailored to specific applications or requirements.

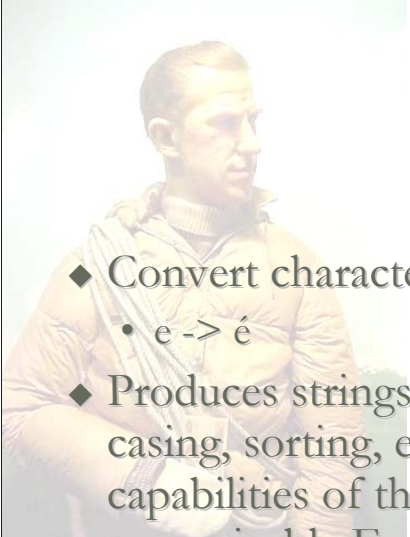


Algorithmic

- ◆ The most common example: ‘zenkaku’ ASCII
 - abcd => **abcd**
 - U+0061 U+0062 U+0063 U+0064
 - U+FF41 U+FF42 U+FF43 U+FF44
 - Add “0xFEE0” to ASCII!
- ◆ Produces strings that are visibly “ASCII” but structurally multi-byte (in SJIS or UTF-8 for example)
- ◆ Produces characters that are kind of chubby, tests Asian font support, and stresses layout/spacing

Algorithmic pseudo-translation is one of the most common types. Basically, this scheme relies on the fact that ASCII is arranged in a nice package and that one can add or subtract from the character number to arrive at a different character. The most common version of this is mapping ASCII to *wide* or *zenkaku* ASCII characters. These is the ASCII set of symbols present in many multibyte character sets for compatibility with older software or presentation. These are **real** multibyte characters... with all of the same attributes, problems, and structure of other multibyte text. It’s just that the characters have a very familiar shape. Quality engineers should expect to be able to read the screen.

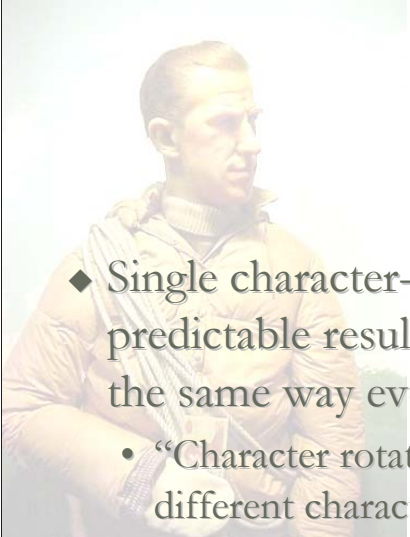
If they can’t, this can represent problems with font selection, presentation, processing, and so forth that the product would encounter with real Asian multibyte data.



Mapped

- ◆ Convert characters using a mapping table
 - e -> é
- ◆ Produces strings that are non-ASCII and exercise casing, sorting, encoding, display, and other capabilities of the software, but are still recognizably English
- ◆ Often used with vowel doubling or pre/post-fixing to simulate text swell.

A “mapped” pseudo-translation converts a source character to a specific target character (or even one of a list of target characters). This is usually used to add accents to non-accented data. Like the algorithmic mapping, this produces text that simulates the problems and effects of localizing into a language that uses accents. The words remain recognizably English, just with all sorts of funny “scribble” around them. This method is often used with vowel-doubling, inclusions, and pre/post-pending of strings to simulate text swell common to localization into European languages.



Mapping Rotation

- ◆ Single character->character mappings produce predictable results. Same string pseudo-translates the same way every time.
 - “Character rotation” replaces one character with different characters so that the same string is rarely the same twice.
- ◆ Exercises assumptions in the code about strings having to match.

Note that if you map the same letter (e) to the same target letter (é) all the time, your translations will always be identical. Using a technique called *mapping rotation* or *character rotation* allows you to create strings that are pseudo-translated differently even though the source strings are identical.


This is important in simulating the effect of translation for different « contexts » within your software and for discovering places where the software relies on translations being (artificially) consistent. If the same string is rarely translated the same way twice, this effect can be spotted and corrective actions taken.



Rotation Frequency

- ◆ Using variable length replacement lists helps ensure pseudo-strings aren't the same
- ◆ Do more than just the vowels. Do consonants too!

Of course, for character rotation to be effective, you need to maximize the likelihood that two identical strings will have different translations. This means using variable length replacement lists, mapping a greater number of characters, and injecting a certain amount of randomness into your mapping algorithm. Note that inclusions, such as identifiers, can also help make target strings unique.



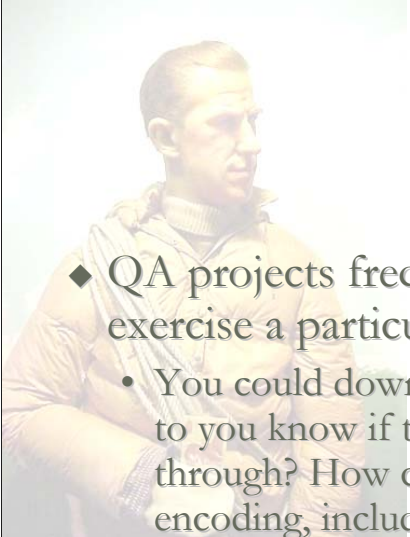
Random

- ◆ Replace one character with a random character
- ◆ Exercises a larger range of Unicode or the target encoding.
- ◆ But... produces strings that are never legible.

Random pseudo-translators replace text with random characters. For example, you might have the translator algorithm replace ASCII characters with random characters from the Cyrillic block of Unicode.

Unlike mapped translators, these algorithms create a distinct pattern each time they are run—mapped pseudo-translators will produce (generally speaking) identical results given the same inputs. This can be a good thing, but might not exercise the full range of character combinations.

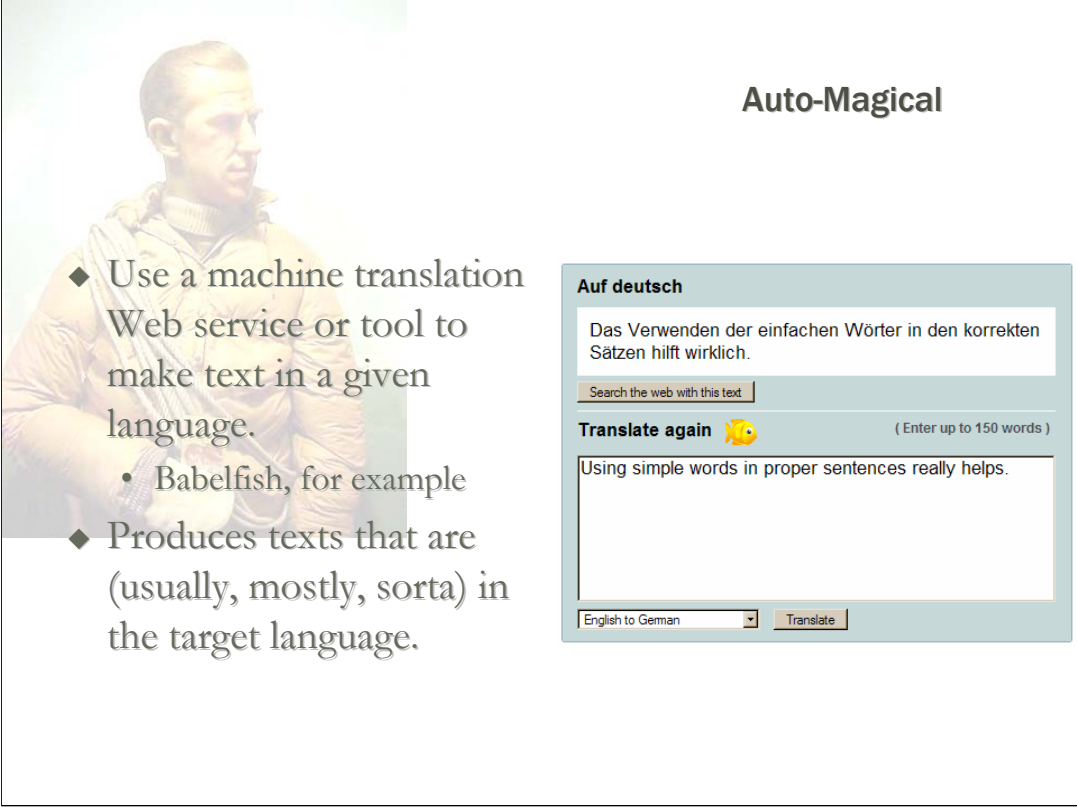
Unlike mapping, though, random pseudo-translators typically produce text that is not recognizably in any language. They can be good for generating targeted test cases.



Simulated Encoding

- ◆ QA projects frequently require globs of text that exercise a particular character encoding.
 - You could download some from the Web... but how to you know if the encoding is correct through-and-through? How can you exercise the full range of the encoding, including rare characters?
- ◆ Replace an ASCII text buffer with a similarly sized and structured text in a given encoding (or using a given character repertoire).

Encoding simulation is a pseudo-translation method that also produces illegible buffers. The main point of this method is to produce text that is reliably in a given character encoding. Searching the Web for content in some character encoding—especially a rare or unusual encoding—can be problematic. How can you tell if the buffer is actually in the given encoding? That it doesn't have any encoding errors in it? That it exercises all of the characters in the encoding? A simulated encoding translator can use the encoding's translation table to or from Unicode to produce a complete set of characters to test the structure of the given encoding. And then you know you have a buffer that should work.



Auto-Magical

- ◆ Use a machine translation Web service or tool to make text in a given language.
 - Babelfish, for example
- ◆ Produces texts that are (usually, mostly, sorta) in the target language.

Auf deutsch

Das Verwenden der einfachen Wörter in den korrekten Sätzen hilft wirklich.

Translate again 🐟 (Enter up to 150 words)

Using simple words in proper sentences really helps.

Another class of pseudo-translator is one that interacts with machine translation tools or computer assisted translation tools to produce a translated user interface. For example, if you have a translation memory from a previous version of the same software, you can import the existing translations and pseudo-translate only the new strings to check if functionality is available.

Another mechanism is to use a machine translation system such as Babelfish. This produces text that is actually in another language (exercising features such as grammar, spelling, sorting, text segmentation, etc.), even if the translations themselves tend to be pretty hilarious. You can even use the two methods together (just don't mix up where the strings came from!)

MT systems like Babelfish work best, please note, if you use simple, well-formed, grammatical sentences with simple vocabulary...

Funkadelic

繁體中文版

如果您得到質樸您取得可怕的結果。特別是以複雜專科術語喜歡"blog-worthy" 或"Flickrize"

Search the web with this text

Translate again 🐟

If you get funky you get awful results. Especially with complex jargon like "blog-worthy" or "Flickrize"

English to Chinese-trad

En français

Si vous devenez génial vous obtenez des résultats terribles. Particulièrement avec le jargon complexe aimez le "blog-worthy" ; ou "Flickrize" ;

Search the web with this text

Translate again 🐟 (Enter up to 150 words)

If you get funky you get awful results. Especially with complex jargon like "blog-worthy" or "Flickrize"

English to French

They work less well when you use jargon, technical terms, abbreviations, and the like. You might end up with English stuff sprinkled through you text—or with no translation at all.



An Approach to Pseudo-Translation

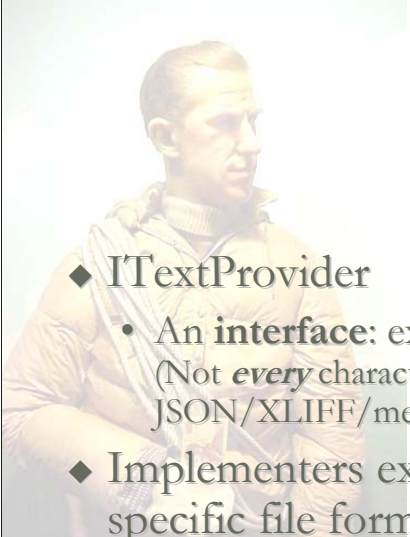
- ◆ I used Java for this.
- ◆ Three main components:
 - Expose text for translation
 - Perform translations
 - Manage translation process
- ◆ Probably: could have used ICU Transliterator classes for this... but probably wouldn't have learned as much...

For my implementation of pseudo-translation, I chose the Java programming language. I could probably have used the ICU4J library's Transliterator classes, but, as noted, I probably wouldn't have learned as much and probably would have shed some flexibility in the results.

I needed confront three basic problems. The first two are ones that all pseudo-translators have:

1. I needed to provide a mechanism for the pseudo-translators to get at the source text without harming the source format.
2. I needed to provide a mechanism that actually did the translation.

The final problem was to make the mechanisms tailorable and scriptable so that different teams could use the translators without my having to become involved in them.



Expose Text

- ◆ **ITextProvider**
 - An **interface**: exposes only the translatable bits of text. (Not *every* character in your properties/JSON/XLIFF/message catalog should be pseudo-translated).
- ◆ Implementers extend this interface for their specific file format or input source.
 - So I don't have to provide a parser for every format.
 - Provides for "segmentation" of text into "translation units"

Exposing the text could take several forms. For example, I could just require that the users create a custom Java application for each file format and pass the strings into the translator themselves. They would manage the process of identifying what to translation, extracting it, getting it translated, and putting it back.

However, it is useful to note that most projects involve many files of the same type or structure. By providing an *interface* that developers can extend, I could allow the engineers using the translation scheme to write code once to expose the source text (while protecting the DNT materials in the file). The details of reading the file format are created and tested once and can then be applied to many files or projects equally. This also allows the same source material to be used with different pseudo-translators.

In many ways this is a dodge around doing work myself: I don't have to build and maintain a processor for every file format out there. And new file formats can be pseudo-translated quickly (since only the file parser has to change).

The "TextProvider" interface basically provides for the *segmentation* of the source into *translation units*.



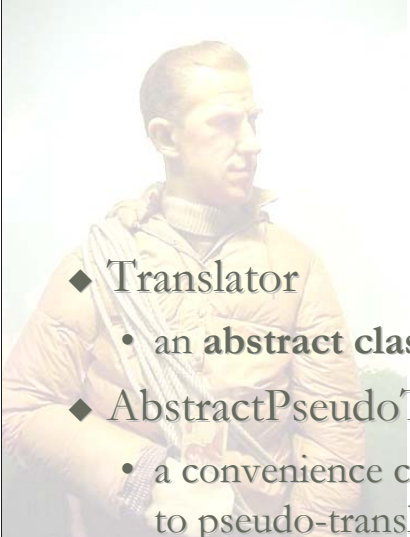
Manage Translation

◆ TranslatorProvider

- Implemented as a “Service Provider Interface” (SPI)
- Loads the various translation classes, including those developed by users separate from the pseudo-translation library
- Provides a consistent interface to the translation process
- ... and I don't have to provide every possible “Translator” class

Before we proceed to the actual pseudo-translator, let's take care of the “management” part of the requirements. In Java it is possible to implement a type of class structure called a “service provider interface” or SPI. SPIs allow a developer to write classes that are instantiated by a factory class. Unlike a regular factory class, though, the classes don't have to be known when the factory class is compiled. They can be added later. This means that any developer can write a new pseudo-translator, as long as it extends the Translator class in my package, and have it be available to any program that knows how to call the factory. When you see the demo programs, note that the selector combo box for the translator itself is populated at runtime by calling a method in the TranslatorProvider that enumerates what it available.

Ultimately, this means that anyone can extend the functionality of the translator subsystem without my having to change any code. And anyone else can make use of the translator without having to merge the two together.



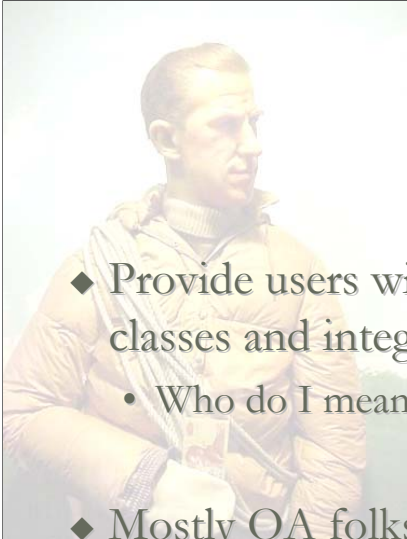
Translate Stuff

- ◆ **Translator**
 - an **abstract class** that defines the translation process.
- ◆ **AbstractPseudoTranslator**
 - a convenience class that adds specific features unique to pseudo-translation, such as pre- and postpending strings.

Finally there is the actual class that does the “translation” process. In my case, I defined an abstract class that does “translation”, called, logically enough, “Translator”. Note that someone could actually implement a translation system around this class (leveraging the fact that we have `ContentProvider` classes that know how to read and write localizable content!)

To help `PseudoTranslator` implementors, I also created an “`AbstractPseudoTranslator`” class. This is another abstract class that implements the common pseudo-translation features described earlier: pre- and postpending, inclusions, mapping rotation, and the like. Developers can implement the specific features that they want and use the default implementation for other things.

Note that the `Translator` classes share an interesting feature: they can store and retrieve their specific configuration as a `Properties` object. This allows a user to configure a specific `Translator` class object exactly right and then store the configuration. They can then restore the translator object’s state when they want to use it again. Combined with ‘ant’ scripts, JUnits or other testing setups, this is very powerful, allowing a wide range of automatically generated tests to be programmed and executed consistently.



Why SPIs?

- ◆ Provide users with the ability to create their own classes and integrate without a re-compile.
 - Who do I mean by “users”?
- ◆ Mostly QA folks. Integration of Pseudo-translation with automated test harnesses.

Summary

- ◆ And a couple of Live Demos, since I don't learn quickly...



<http://images.search.yahoo.com/search/images?p=live+demos&ei=UTF-8&fr=sfp&x=wrt>

Live demos are always fun. On the succeeding slides I've include some screen shots of a couple of pseudo-translation demo programs shown at IUC30. Actually, they worked quite well at the conference....



PseudoTranslator Demo



Q&A

Presentation available (soon) at:

<http://www.inter-locale.com>

Internationalization and Unicode Conference #30