Localización de Software con Gettext

Fran Diéguez
fran.dieguez@glug.es
www.mabishu.com - www.glug.es

19 de decembro de 2008

Índice Xeral

0.1 Introdución

A localización dun programa implica a tradución dos recursos dun programa (cadros de diálogo, menús e mensaxes en xeral), ademais dos procesos de axuste de tamaño e testeo do programa.

A meirande parte dos proxectos de localización incluen documentación de referencia e posiblemente autoxerada. Lamentablemente non hai un formato unificado polo que teremos que traballar con diferentes tipos de arquivos e formatos. Coa axuda e ferramentas de tradución asistida, podemos asegurar coherencias e eficiencia na tradución das distintas partes que compoñen un software completo.

En calquera aplicativo que queiramos "internacionalizar" precisamos un sistema que automaticamente nos traduza a nosa aplicación coa lei do mínimo esforzo. Para isto temos duas opcións:

- Facer-nos o noso propio sistema de internacionalización, ergo, levar un traballo adicional sumada á programación do software en si e sen empregar un sistema estandarizado.

- Ou traballar cun sistema xa probado e implementado.

A idea máis directa é empregar un sistema xa probado e hai que poñer fincapé en que empregando sempre tecnoloxías xa desenvoltas asegurar un nivel de calidade e eficiencia de traballo que se reflexará na calidade final da tradución. Podemos engadir ademais que o emprego de tecnoloxías abertas nos vai a asegurar que o que estamos a empregar estará ben "testeado" por milleiros de persoas en distintas partes do mundo, con situacións lingüísticas diversas.

Loxicamente os sistemas de internacionalización actuais non son a panacea, e como todo produto feito polas persoas ten os seus fallos e carencias, si ben, pode axudarnos moito na tarea de internacionalizar un aplicativo.

0.2 Gettext como sistema de internacionalización

No ámbito do SL, Gettext é o sistema de i18n polo que opta a maioría das persoas desenvolvedoras. O máis habitual é que unha persoa tradutora de software entre en contacto coa tecnoloxía Gettext a traverso dun ficheiro PO, que se describirá máis adiante.

Gettext basease nun sistema de marcado de cadeas que logo automaticamente se poidan extraer e a arquivos de texto plano para poder xestionar de forma sinxela e traducir. Feita a tradución dese arquivo de texto plano convertese a un arquivo binario (mo-file) que vai empregar o programa na execución para adaptar a linguaxe e comportamento do mesmo segundo a linguaxe do sistema no que se emprega e pola elección e configuracións do usuario final.

0.2.1 Como recolle o idioma o sistema Gettext

En sistemas UNIX (GNU/Linux, Solaris, Mac OS, ..) existen unha serie de variables de sistema que podemos ollar co seguinte comando:

```
\$ locale
```

o cal terá unha saída semellante á seguinte:

```
LANG=gl_ES.UTF-8
LC_CTYPE="gl_ES.UTF-8"
LC_NUMERIC="gl_ES.UTF-8"
```

```
LC_TIME="gl_ES.UTF-8"
LC_COLLATE="gl_ES.UTF-8"
LC_MONETARY="gl_ES.UTF-8"
LC_MESSAGES="gl_ES.UTF-8"
LC_PAPER="gl_ES.UTF-8"
LC_NAME="gl_ES.UTF-8"
LC_ADDRESS="gl_ES.UTF-8"
LC_TELEPHONE="gl_ES.UTF-8"
LC_MEASUREMENT="gl_ES.UTF-8"
LC_IDENTIFICATION="gl_ES.UTF-8"
LC_ALL=
```

estas variables de sistema veñen a definir a linguaxe, variante de linguaxe e codificación a empregar dentro dos aplicativos co sistema de internacionalización activado (sexa Gettext ou calquera outro).

Analizando polo miúdo as anteriores variables ollamos que os seus valores teñen tres campos:

- **gl** Parte da variable que nos informa o código ISO 639-1 de idioma. Pódese ollar unha listaxe de estes idiomas en `http://www.gnu.org/software/gettext/manual/gettext.html#Language-Codes`
- **ES** Código do país, neste caso ES, que cumpre a ISO 3166. No caso do idioma español existen múltiples variantes (es_ES para o español de España, es_AR para o español da Arxentina, etc.). Poderío ollar unha listaxe dos códigos de país en `http://www.gnu.org/software/gettext/manual/gettext.html#Country-Codes`
- **UTF-8** O seu uso é opcional e nos vai indicar o sistema de codificación de caracteres.

Tódalas anteriores partes de forma conxunta nos indicarán con certeza que linguaxe teñen que empregar os aplicativos.

O nivel de prioridade varia de unhas a outras, de xeito que a definición dunha variable cun valor específico prevalece sobre outras que contan con menos nivel. Amoso a continuación as variables segundo a orde de menos a máis prioridade:

- *LANGUAGE*
- *LC_ALL*

- *LC_XXX*, de acordo coa categoría de locale seleccionada: *LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, LC_MONETARY, LC_MESSAGES, ...*
- *LANG*

LANGUAGE é unha variable especial de entorno que configura unha listaxe de idiomas de preferencia polo usuario, de xeito que concatenando distintos idiomas separados por "dous puntos"(:) obtemos a nosa preferencia, dende a primeira e máis prioritaria, ata a última e menos prioritaria. Se calquera dos locales selecionados na variable LANGUAGE non está instalado no sistema, omítese e pasase á seguinte e no caso de que non existan ningún instalado no sistema empregarase como última posibilidade a linguaxe do código fonte, que por norma xeral está en inglés.

LANG é unha variable de entorno para especificar un locale. Como usuario, normalmente configuras esta variable no arquivo de inicialización /etc/profile.

LC_CTYPE, LC_NUMERIC, LC_TIME, LC_COLLATE, LC_MONETARY, LC_MESSAGES, etc., son variables de entorno que sobreescriben a variable LANG e afectan a unha categoría de locale. Por exemplo, pensa que eres un Sueco vivindo na Galiza, e queres que os programa xestionen as datas e os números de acordo coas convencións de España, e só que as mensaxes sexan amosadas en sueco. Entón podes crear un locale chamado "SV_ES" ou "SV_ES.UTF-8" empregando o programa *localedef*. Aínda que a forma máis sinxela de acadar isto é configurando a variable *LANG* a "es_ES.UTF-8" e *LC_MESSAGES* á variable "SV_SE.UTF-8", estes dous locales veñen instalados por defecto non sistema operante.

LC_ALL é unha variable de entorno que sobrescribe a todas estas. Empregase normalmente polos scripts que executan programas Por exemplo, os scripts de configuración xerados por GNU autoconf empregan LC_ALL para asegurarse que os tests de configuración non operan de xeito que sexan dependentes dos locales.

0.2.2 Marcado de cadeas

Gettext emprega, como se comentou anteriormente, un sistema de marcado de cadeas que logo o tradutor terá que traducir ó idioma destino. Co fin de ter un sistema unificado, emprega a seguinte sintaxe a traverso da maioría das linguaxes de programación:

```
_("String to translate")
```

Ou vendo un exemplo práctico escrito en Ruby entón será coma o seguinte

```
puts  _("String to translate")
```

Mantendo esta sintaxe garantimos que as utilidades xenéricas das que conta Gettext extraian correctamente todas e cada unha das cadeas e flexibiliza e uniforma o traballo de tradución.

0.2.3 Arquivos de traballo

Gettext traballa principalmente con dous tipos de arquivos, ademais de un tipo de arquivo adicional derivado de un dos primeiros:

po-files Son arquivos que conteñen texto en plano, co cal se poden editar con calquera editor de texto

```
# hello.po - sample for GetText._()
#
#
# Francisco Diéguez <fran.dieguez@glug.es>, 2008.
#
msgid ""
msgstr ""
"Project-Id-Version: ruby-gettext-package 0.0.1\n"
"POT-Creation-Date: 2004-11-5 10:11:00+0900\n"
"PO-Revision-Date: 2008-12-13 13:57+0100\n"
"Last-Translator: Fran Diéguez Souto <fran.dieguez@glug.es>\n"
"Language-Team: Spanish\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"

#: ../hello.rb:7
msgid "Hello World\n"
msgstr "Ola mundo\n"
```

No exemplo anterior distinguimos dúas seccións, unha cabeceira, onde definimos o nome do proxecto, tradutor e os seus datos, sistema de codificación, entre outros:

```
"Project-Id-Version: ruby-gettext-package 0.0.1\n"
"POT-Creation-Date: 2004-11-5 10:11:00+0900\n"
"PO-Revision-Date: 2008-12-13 13:57+0100\n"
"Last-Translator: Fran Diéguez Souto <fran.dieguez@glug.es>\n"
"Language-Team: Spanish\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=utf-8\n"
"Content-Transfer-Encoding: 8bit\n"
```

E por outra parte están as entradas a traducir:

```
#: ../hello.rb:7
msgid "Hello World\n"
msgstr ""
```

Como se pode ollar a primeira liña nos di o arquivo de código fonte no que está presente dita tradución, a segunda nos día liña orixinal e finalmente na terceira é onde o tradutor insire a cadea traducida quedando polo tanto esa tradución da seguinte maneira

```
#: ../hello.rb:7
msgid "Hello World\n"
msgstr "Ola Mundo\n"
```

mo-files Son arquivos binarios, isto é compilados e adaptados a cada arquitectura, e polo tanto non son portábeis dunha plataforma a outra. Isto significa que non é posíbel a modificación directa dos ficheiros mo-files, senón que teñen que xerarse sempre dende os po-files. Estes arquivos sóenche chamar sóense chamar PACKAGES (pacotes) como ben o chaman no proxecto GNOME.

Este tipo de arquivos emprégaos Gettext directamente para recoller as traducións no momento de execución do programa. Os motivo da compilación poden ser diversos, dende a optimización dos datos para a súa recollida ata a de compresión e empaquetado do mesmo.

pot-files Existe un tipo adicional de arquivos de traballo en Gettext pero non deixa de ser un arquivo pofile, este novo tipo denominase potfiles. A finalidade de este tipo de arquivos é a de ter un índice de cadeas fonte para poder actualizar coas modificacións do programa, e aplicar nel

as cadeas engadidas, modificadas ou mesmo eliminadas do mesmo. O arquivo consiste por tanto nun arquivo PO onde as súas cadeas *mgstr* están baleiras.

Empréganse ademais para a actualización de pofiles xa traducidos e aplicar así os cambios dunha forma automatizada en forma de parches.

0.2.4 Xestión de traducións con Gettext

Describirei aquí o proceso de tratamento dos anteriores arquivos ademais das distintas utilidades de consola que ten Gettext para extraer, converter, unir, e en definitiva xestionar os arquivos po-files e pot-files para acadar os mo-files finais que empregará o programa.

Creando un arquivo PO Template

Collamos un exemplo simple escrito en Ruby, este é o contido do arquivo hello.rb:

```
require 'rubygems'
require 'gettext'

class OlaMundo
  include GetText

  bindtextdomain("hello")

  def say_hello
    puts _("Hello World")
  end
end

OlaMundo.new.say_hello
```

Este exemplo é o clásico Ola Mundo pero desta vez con soporte para internacionalización. Como se pode ollar inclúe a xema Gettext ademais de ter marcada unha cadea coa que vamos a operar.

O primeiro paso é crear un arquivo PO template que nos servirá de modelo para logo xerar os arquivos onde vamos a traballar (os pofiles). Para isto temos xa comandos específicos que nos simplifican o traballo. Neste caso farase da seguinte maneira:

```
rgettext hello.rb -o hello.pot
```

Nota: no caso de Ruby emprégase *rgettext* pero usualmente se emprega o comando *xgettext*, o uso de rgettext está xustificado por dous motivos, primeiro porque rgettext está optimizado para código Ruby, e en segundo lugar porque xgettext non ten suporte nativo para Ruby, aínda que se poden empregar o outro parser.

O comando anterior nos xerará un novo arquivo hello.pot que será o arquivo modelo co que logo xeraremos os distintos pofiles para cada idioma. Basicamente é un arquivo pofile sen dato algún na tradución das cadeas.

Se modificamos as cadeas dentro de hello.rb teremos que voltar ó comezo para xerar un novo POT Template e actualizar os pofiles, pero diso falaremos un pouco máis adiante.

Estes arquivos ".pot" son os que empregan os tradutores de software e realmente é o único arquivo que eles precisan coñecer.

Creando un arquivo PO novo

Chega o momento crucial, o tradutor quere traducir este programa ó galego. Para iso ten que xerar un arquivo pofile co que el traballará para logo devolverllo ó programador orixinal e integralo posteriormente na aplicación.

Para conseguir o pofile a partir do potfile simplemente se executa o seguinte comando:

```
msginit
```

Pregúntasenos o nome do autor que xera este pofile e como resultado crearase un novo arquivo co nome do locale configurado no sistema, no meu caso gl_ES.po. Se por calquera motivo queremos xerar unha tradución distinta da nosa só teremos que engadirlle a "msginit" a seguinte opción "–locale nome_locale". Loxicamente o comando ten moitas máis opcións pero para este exemplo esta funcionalidade nos cobre perfectamente a nosa tarefa.

Proceso de tradución con programas cliente ou web

Este punto é sen dúbida o que máis laboura ten nos pasos para internacionalizar un software. Consiste en traducir os pofiles con programas específicos para esta tarefa. Se ben, podemos distinguir dous tipos:

- **Programas standalone:** software que executamos no noso propio ordenador e que non dependerá da conexión á rede. Podemos destacar **Poedit**, o único que é multiplataforma, cunha interface simple pero efectiva e con suporte para memorias de tradución.

- **Programas baseados en web:** software que se executa nun servidor centralizado

Cadeas Fuzzy

En moitas ocasións durante o traballo dun tradutor de software chegamos a unha cadea a cal pola súa complicación temos que deixar a medio traducir, e precisamos un sistema de marcar esas cadeas como dubidosas (en inglés "fuzzy").

Gettext conta con esta funcionalidade que se plasma na entrada do pofile

```
#: myapp.rb:7
\textbf{#, fuzzy}
msgid "Hello World"
msgstr "KON-NICHI-WA SEKAI"
```

Lóxicamente cando se edita o arquivo pofile dende un programa de escritorio, p.ex. poedit, existen botóns ou atallos de teclado que facilitan moito a tarefa de marcado.

En cando esa frase se traduza completamente, eliminase a marca de "dubidosa" (fuzzy"

```
#: myapp.rb:7
msgid "Hello World"
msgstr "KON-NICHI-WA SEKAI"
```

Xeración de arquivos binarios MO

Chegados a este punto, xa só nos queda un paso para poder executar a nosa aplicación, a creación do mo-file e dispoñelo na ruta correcta. Neste exemplo, non configuramos a ruta en bindtextdomain(), polo que Gettext procurará os mo-files en /usr/share/locale/*linguaxe*/LC_MESSAGES/ ou en /usr/local/share/locale/linguaxe/LC_MESSAGES/, polo que hai que poñelos arquivos nese lugar. Neste exemplo porémolos en /usr/local/share/locale/*gl*/LC_MESSAGES/.

Por tanto para xerar o arquivo mo empregamos "rmsgmft",

```
rmsgfmt hello.po -o /usr/local/share/locale/gl/LC_MESSAGES/hello.mo
```

pero tamén podemos empregar o xenérico "msgfmt" que ven coas utilidades de Gettext, empregando a variable GETTEXT_PATH, podes elixir a ruta dos mo-files. O exemplo seguinte aplicaríase ó exemplo se no mesmo estivera definido "bintextdomain("locale") así recollería as traducións do cartafol locale na mesma ruta que o script.

```
export GETTEXT_PATH="locale"
rmsgfmt gl_ES.po -o ./locale/gl/LC_MESSAGES/hoge.mo
```

Execución da aplicación traducida

Se todo foi correcto podemos executar o noso script do seguinte xeito:

```
ruby hello.rb
```

para o cal teríamos a seguinte resposta

```
Ola Mundo
```

e se queremos executalo en castelán

```
LC_ALL=es_ES ruby hello.rb
```

para o cal teríamos a estoutra resposta

```
Hola Mundo
```

Se non podes ollar as mensaxes no teu idioma ou aconteceu calquera erro, executa o script coa opción -d.

```
ruby -d hello.rb
```

Actualización de pofiles

Todo desenrrolo no seu momento se modifica, sexa para engadir funcionalidades ou para eliminar fallas no mesmo. Sexa por onde sexa estas modificacións poden modificar as cadeas que queremos internacionalizar, polo que a pregunta sería: Teño que voltar a traducir tódalas cadeas que xa están traducidas? A resposta é non. Para iso Gettext proporciona unha serie de funcións para actualizar os nosos arquivos pot e po para logo xerar a nova tradución nos mo-files. Ollemos:

```
require 'rubygems'
require 'gettext'

class OlaMundo
  include GetText

  bindtextdomain("hello")

  def say_hello
    puts _("Hello World")
    puts _("Goodbye World")
  end
end

OlaMundo.new.say_hello
```

Neste programa engadiuse unha nova cadea que internacionalizar, polo que o noso potfile xerado anteriormente non está actualizado. Para solventar isto executaremos de volta a xeración do potfile:

```
rgettext hello.rb -o hello-novo.pot
```

Agora coa nova versión do ".pot" imos actualizar o pofile que xa traducimos completamente, para iso temos o comando "msgmerge" que o que fai é unir as cadeas xa traducidas

```
rmsgmerge hello.rb hello-novo.pot > hello-novo.po
```

Editamos o arquivo "hello-novo.po" co editor de pofiles favorito, por exemplo poedit, e ollamos que agora conta con dúas entradas, unha xa traducida anteriormente e outra nova que teremos que traducir.

Coa tradución completada só temos que voltar a compilar o hello-novo.po para obter o mofile actualizado e polo tanto ter o programa traducido totalmente.

```
rmsgfmt hello-novo.po -o /usr/local/share/locale/gl/LC_MESSAGES/hello.mo
```

0.3 API de Gettext en Ruby

Ata o de agora vimos un enfoque externo á programación e que é totalmente aplicable ó tradutor, abordarei agora o enfoque directo do programador e os pasos que ten que dar para internacionalizar o seu aplicativo. Veremos que con moi pouco esforzo podemos engadir dita funcionalidade ó noso traballo.

0.3.1 Funcións principais

En Gettext definimos dominio como pacotes de traduccións que de calquera xeito teñen unha relación e que ó ser así queremos que o tradutor as traduza todas xuntas.

Na maioría dos casos a definición de un dominio implica a creación dun arquivo pofile con nome idéntico. Isto é, se definimos un dominio "xestor-configuracions" entón ó extraelo obteremos un arquivo chamado xestor-configuracions.po.

No seguinte exemplo definimos un dominio "xestor.configuracions" a un módulo en Ruby:

```
require 'rubygems'
require 'gettext'

module Configurador-condensador
  include GetText

  bindtextdomain("xestor-configuracions")

  def arrinque
    puts _("Arrincando o xestor de configuracións")
  end
end

Configurador.new.arrinque
```

Agora ben, este dominio non ten porque restrinxirse a ese arquivo só, senón que se pode aplicar a outro arquivo distinto e incluso dentro do mesmo arquivo a outra clase/módulo, como se pode ollar no seguinte exemplo.

```
require 'rubygems'
require 'gettext'

module Configurador-fluzo
  include GetText

  textdomain("xestor-configuracions")

  def peche
    puts _("Mantando o xestor de configuracións")
```

```
  end
end

Configurador.new.peche
```

Se o lector é suficientemente ollado puido observar que **bindtextdomain** non aparece no segundo arquivo senón que se escribeu **textdomain**, isto é porque bindtextdomain serve para vincular cun dominio de tradución e además creao. Pola contra textdomain vincula unha clase/módulo a un dominio de tradución xa existen, e polo tanto nos serve de *extensor* do dominio de tradución cara outros arquivos e/ou clases/módulos.

Polo tanto e como modo resumo a estas dúas funcións obtemos a seguinte referencia

bindtextdomain(nome_dominio , opcións =)

Vincula un dominio de texto ó teu programa, hai que ter en conta que os dominio de traducción só se pon aplicar a clases e módulos. Polo que precisas chamar a esta función en cada un dos teus scripts en Ruby. As opcións son as seguintes:

- **nome_dominio:** o nome de dominio
- **opcións:** un diccionario de opcións
  - **:path** a ruta ó arquivo mofile. Cando a ruta é nula procúrase nos directorios por defecto, como por exemplo en /usr/share/locale e /usr/local/share/locale
  - **:locale** será o locale ó que representará. Normalmente emprégase Gettext.set_locale no sitio.
  - **:charset** o conxunto de caracteres de saída. Isto afecta só ó dominio de traducción actual.

textdomain(nome_dominio)

Vincula con un dominio de texto xa existente. Esta función é a mesma que Gettext.bindtextdomain pero máis simple (e máis rápida). Olla que precisas chamar antes Gettext.bindtextdomain antes que esta. Se o dominio non foi definido anteriormente lanzarase unha excepción do tipo Gettext::NoboundTextDomainError As opcións son as seguintes:

- **nome_dominio:** o nome de dominio

_(), n_()

Podemos decir que é a función máis importante, polo seu reiterativo uso vai ser _(), esta función chama a gettext para que xestione a traducción segundo un locale definido. Podemos chamala o corazón de gettext xa que ela segundo unha cadea de texto vai a recoller a tradución do arquivo mofile correspondente e voltará a cadea traducida.

Ten varias formas de definición segundo a frase teña só unha forma singular ou máis, por exemplo

```
puts _("Isto é unha frase que sempre está en singular")

puts n_("Isto é unha mazán",
        "Estas son varias mazáns", función_calculo)
```

Esta función_calculo terá que devoler verdadeiro se queremos unha cadea en plural e falso se queremos a cadea en singular.

locale()

Esta función nos devolverá o locale actual, se non se defineu ningún no script recollerá o locale do sistema segundo as variables que se definiron anteriormente neste documento.

set_locale(locale, so_aqui = false

Configura o locale que se vai a empregar segundo o parámetro que se lle pasa, a variable so_aqui emprégase para restrinxir o cambio ó ámbito ó que se abscribe.

Isto implica, por exemplo que se efectuamos a seguinte chamada:

```
set_locale("gl_ES'')
```

a partires de se intre todo o programa imprimirá as mensaxes en **Galego** aínda que o usuario teña o sistema configurado para outro idioma.

set_locale_all(locale)

Configura o locale que se vai a empregar segundo o parámetro que se lle pasa, en contra con set_locale efectúa o cambio e tódolos dominios de texto.

a variable so_aqui empregase para restrinxir o cambio ó ámbito ó que se abscribe.

0.3.2 Exemplos práctico

O exemplo máis sinxelo para localizar un script de Ruby é o seguinte, coido que son bastante aclaratorios e cos comentarios están totalmente explicados.

```
#!/usr/bin/ruby
# -*- coding: utf-8 -*-
# hello.rb - sample for _() and class.
#
# Copyright (C) 2001-2006 Masao Mutoh
# This file is distributed under the same license as Ruby-GetText-Package.

# importamos o preciso para internacionalizar
require 'rubygems'
require 'gettext'

class HelloWorld
  include GetText

  # decimoslle a gettext en que "pacote" está
  # e además onde estan os mo-files
  bindtextdomain('hello', 'locale')

  def hello
    print _('Hello World\n')
  end
end

if __FILE__ == $0
  a = HelloWorld.new

  a.hello # Amosao no teu locale

  old = GetText.locale
  p old.to_s # Amosa o locale actual

  # Cambia o locale a inglés "en".
  GetText.set_locale_all("en")
  p GetText.locale.to_s
  a.hello # Amosa en inglés
```

```
  # Mostra
  GetText.set_locale_all(old)
  a.hello # Amosao no teu locale
end
```

Podemos ollar o seguinte como un exemplo de uso de Gettext e o seu soporte para formas plurais:

```
#!/usr/bin/ruby
# hello_plural.po - sample for n_() and class.
#
# Copyright (C) 2002-2006 Masao Mutoh
# This file is distributed under the same license as Ruby-GetText-Package.
require 'rubygems'
require 'gettext'

class HelloPlural
  include GetText

  def initialize
    bindtextdomain("hello_plural", "locale")
  end

  def hello
    (0..2).each do |v|
      puts n_( "There is an apple.\n",
               "There are %{num} apples.\n", v) % {:num => v}
    end
  end
end

hello = HelloPlural.new

hello.hello
```

Aledo ó lector a que olle os exemplos adxuntos que están baseados nos exemplos que trae a xema *Gettext* de Ruby.

0.4 Perspectivas de Futuro

Aínda que Gettext supón un adianto substancial para a axuda do desenvolvedor dun software asemade do tradutor, este sistema se queda bastante estancado e supón certas limitacións respecto a novos sistemas que se están a consolidar actualmente.

Algúns dos problemas que xorden logo de traballar certo tempo con Gettext pódense resumir a continuación:

- Gettext extrae texto dos arquivos de código fonte, pero que acontece co resto de arquivos, isto é, documentos de texto, Open Office, LaTeX ou DocBook XML; que acontece pois cos arquivos de axuda man, ou dos info. **Gettext non se inventou para traducir documentación**, e este é un punto ben importante xa que na actualidade calquera proxecto serio conta polo menos coa mesma cantidade de información de documentación como de código fonte.

- O emprego de memorias de tradución é necesario pero agora presentase o problema da presenza de parágrafos longos, con varias liñas de texto, que non podemos segmentar xa que as memorias de tradución baséanse na **segmentación** de un mesmo segmento. Se a unidade de segmentación é o parágrafo esas coincidencias resultarán case imposíbeis. Podemos pensar en segmentar con "." pero o que nunha linguaxe podemos escribir nunha simple frase, noutra precisamos máis de unha frase separada polo tanto por ".".

- Das **coincidencias parciais** como obtemos as coincidencias dubidosas?, e o máis importante como cuantificamos o nivel de exactitude de ditas coincidencias parciais?

- Se queremos rizar o rizo podemos pensar agora nos ambientes multilingüaxe, isto é, contextos nos que precisamos ter dúas linguaxes simultaneamente. PO creouse para traducir dunha linguaxe a outra. Que acontece se temos distintas codificacións ó longo dun texto ou por exemplo empregamos citas en latín?

- En moitas ocasións unha tradución pode resultar complicada debido a que Gettext non xestiona ben o contexto. Unha mesma expresión pode significar unha cousa ou outra segundo o contexto e por tanto elixir unha tradución diferente pode levar a enganos segundo o contexto. O noso problema reside que unha vez traduzamos un segmento a ún idioma (e as memorias de tradución sen contexto) é que a mesma

expresión darase por traducida nas seguintes aparicións. Gettext conta cun sistema de contextualización que segue o esquema que se olla de seguido,

```
msgctxt "contexto"
msgid "original"
msgstr "tradución"
```

Aínda que en moitas ocasións o contexto proporcionado dista moito de ser o necesario para levar a cabo unha tradución correcta.

- Con Gettext o **traballo colaborativo é imposíbel** xa que Gettext non está preparado para traballar máis de unha persoa nun arquivo PO, por poñer un exemplo. Faise imposíbel polo tanto axilizar a tradución dun software repartindo o traballo entre distintas persoas. Esta descrición está lonxe de ser adecuada para un fluxo real de traballo dun equipo de tradución. O problema podémolo restrinxir a dous sentidos. Primeiro, porque para que unha tradución teña un mínimo de calidade as terminoloxías utilizadas deben ser coherentes (traduciranse os mesmos termos de forma igual por parte de tódolos tradutores do grupo), deberanse compartir as memorias de tradución, etc.
- E finalmente podemos salientar o pouco **nivel de interoperatividade** dos arquivos PO, xa que a meirande parte dos tradutores profesionais e estudantes de tradución empregan ferramentas comerciais que non aceptan o formato PO. Podemos destacar que neste senso o problema agravase se existe un formato aberto e aceptado pola industria para a mesma función, XLIFF, e outra para o intercambio de memorias de tradución, TMX.

Instase ó lector á lectura e información sobre a distintos proxectos coma OmegaT, XLIFF e TMX que son os que van supor un cambio importante sobre todo no eido da interoperatividade de sistemas e proxectos de internacionalización de software, documentación ou calquera tipo de arquivo.