
GNOME アプリケーションの国際化

Tredinnick Malcolm [FAMILY Given]

<malcolm@commsecure.com.au>

Japanese translation: Satoru SATOH

日本 GNOME ユーザー会翻訳チーム

<ss@gnome.gr.jp>

製作著作 © 2002, 2003 Malcolm Tredinnick

Legal Notice

The following copyright notice applies to the text of this article.

Copyright 2002, 2003 Malcolm Tredinnick

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found here or in the file COPYING-DOCS which shipped as part of this package.

Translators are given permission to use the current title of this document in any translated versions.

Many of the names used by companies to distinguish their products and services are claimed as trademarks. Where those names appear in any GNOME documentation, and those trademarks are made aware to the members of the GNOME Documentation Project, the names have been printed in caps or initial caps.

改訂履歴

改訂 0.5	13 November 2003
	Editorial changes sent in by helpful readers.
改訂 0.4	13 October 2003
	Release to web.
改訂 0.3	19 July 2003
	Emphasising the need to carefully choose which strings to mark for translation.
改訂 0.2	7 November 2002
	Small fixes. See ChangeLog appendix for details.
改訂 0.1	3 November 2002

Initial version. This document is maintained in the GNOME CVS repository. It is in the `gnome-devel-docs` module under `gnome-devel-docs/tutorials/i18n`

目次

1. はじめに	2
---------------	---

1.1. フィードバック	2
2. 翻訳のためのソースコードの準備	3
2.1. i18n ヘッダーファイル	3
2.2. 翻訳可能な文字列をマーク	6
3. パッケージのビルドインフラに i18n [の仕組み]を取り込み	11
3.1. autogen.sh の変更	11
3.2. configure.in の変更	13
3.3. Makefile の変更	14
3.4. po/ディレクトリ内での変更	15
4. 次のステップは?	16
5. 秘訣と技	16
A. アプリケーション国際化のチェックリスト	17
B. 変更履歴	18
参考文献	19

概要

GNOME アプリケーションがひとたびある種の“臨界”に達してしまうと、他の言語を話す人たちがアプリケーションを使えるようにするにはどうするか考えることが重要となってきます。このチュートリアルは、GNOME の他の部分と一貫性のある方法でアプリケーションを国際化するために必要なステップについて取り上げています。その方法は主に C 言語で書かれたアプリケーションを対象としていますが、原理やサポートの仕組みはすべてのアプリケーションに適用することができます。

1. はじめに

コンピュータプログラムのほとんどが英語による何らかの形式で書かれていると主張することは、おそらく妥当なことでしょう。確かに、人気のあるプログラム言語の多くが、言語のキーワードを想起するものとして英語を使い、国際的な共同作業による多くのプロジェクトには、(もしかすると暗黙に)コメントやメッセージの表示に英語を使うように指示するガイドラインがあるでしょう。

そうは言っても、この惑星上の大多数(そしておそらく他の惑星でも同様)は英語を理解しないし、あるいは理解するとしても、他の日常生活で使う言語の方を好みます。従って、可能な限り多くの人に使われることを望んでいる GNOME アプリケーションは、他の言語にも翻訳されていなければなりません。この文書の執筆時点の GNOME 安定版リリースでは、少くともいくつかの部分において 55 以上の異なる言語に翻訳されています。

この文書は、internationalisation (国際化; 通常 i18n と略されます) の手順について取り扱っています。アプリケーション開発のこの段階で、開発者は、翻訳者を手助けし、後で任意のメッセージの適切なロカールに対する翻訳版を表示するために必要なパーツを組み込みます。

プログラム内の任意のメッセージを他の言語に翻訳する手順は localisation (地域化; または l10n) と呼ばれ、[kmaras] のような他の場所でカバーされています。

アプリケーションの国際化は二つの段階から成ります。項2.「翻訳のためのソースコードの準備」では、ソースコード全体に渡る翻訳のために文字列をマークアップする手順について順を追って行きます。また、ビルドできるように保つために、ソースコードに加えなければならない変更点についても言及するつもりです。次のパート 項3.「パッケージのビルドインフラに i18n [の仕組み]を取り込み」では、Makefile や configure.in の変更のような、アプリケーションのビルド手順の変更についてふれるつもりです。

1.1. フィードバック

もし、このチュートリアルを使ってアプリケーションを国際化していて、困難に遭遇したら、私に教えて下さい。不明瞭だったり、誤解させやすい、または単純明白に欠けている点について知りたいのです。<malcolm@commsecure.com.au> までメールを下さい。

注意

この翻訳についての間違い、誤解させやすい点、欠けている点などがあれば 訳者(佐藤 暁 <ss@gnome.gr.jp>) まで知らせて下さい。

2. 翻訳のためのソースコードの準備

国際化の手順の真髄は、ユーザーが見るであろうすべての文字列を見つけ出して翻訳できるようにマークするというに基づいています。これは必ずしも簡単な作業ではありません。中〜大規模のアプリケーションは数百の文字列を含んでいることがあります。また文字列はそうでもなければ忘れてしまいそうなファイルに隠されています。幸運なことにこの作業を“離陸させる”のを手助けするツールが利用できます。加えて最初は全部の文字列について処理する必要はありません。部分的にマークアップされたアプリケーションであってもなお普通に動きます。唯一の問題は“隠れている”文字列が翻訳されないことです。

次の数セクションではこの手順の個々のステップについて説明します。

- ・ 必須な関数の宣言
- ・ システムライブラリ (libc または glibc) サポートの初期化
- ・ ソースコード内のすべての文字列を検索、マーク
- ・ サポートファイル内のすべての[ユーザーに]見える文字列をマーク

2.1. i18n ヘッダーファイル

このチュートリアル全体を通してのゴールは、アプリケーションを常にビルド可能なままに保つことです。文字列を何らかの新しい関数でラップしたら、プログラムがこれらの関数を確実に認識しているようにする必要があります。それからアプリケーションの最初の部分で i18n コードを初期化し、コード全体を通して文字列をマークアップして行きます。

2.1.1. 国際化ヘッダーファイル

様々な i18n マクロを含むことになる新しいヘッダーファイルを作成します。次のリストは追っていくのに適切な例です。

例 1. slice-i18n.h

```
#ifndef __SLICE_INTL_H__
#define __SLICE_INTL_H__

#ifdef HAVE_CONFIG_H
#include "config.h"
#endif /* HAVE_CONFIG_H */

#ifdef ENABLE_NLS
#include <libintl.h>
#define _(String) gettext(String)
#ifdef gettext_noop
#define N_(String) gettext_noop(String)
#else
```

```
#define N_(String) (String)
#endif
#else /* NLS is disabled */
#define _(String) (String)
#define N_(String) (String)
#define textdomain(String) (String)
#define gettext(String) (String)
#define dgettext(Domain, String) (String)
#define dcgettext(Domain, String, Type) (String)
#define bindtextdomain(Domain, Directory) (Domain)
#define bind_textdomain_codeset(Domain, Codeset) (Codeset)
#endif /* ENABLE_NLS */

#endif /* __SLICE_INTL_H__ */
```

注意

このチュートリアルの場合のために Slice-n-Dice という架空のプロジェクトで作業することとします。私はこのアプリケーションが何をやるものか知りませんが、きっと世界中のユーザーが利用できるようになるでしょう。

当然 slice-i18n.h ではない何か別のファイル名にもできます。しかし、ENABLE_NLS 変数名については後で 項3. 「パッケージのビルドインフラに i18n [の仕組み]を取り込み」で見えるように特別な意味を持っているので変えるべきではありません。

もしアプリケーションが libgnome に依存しているなら、slice-i18n.h (または他の名前前のファイル) の代わりに <libgnome/gnome-i18n.h> を単純に include することで、独自の i18n サポートファイルを作成する必要はなくなります。ここでは、メタ法的な“食物連鎖”のより低次のところで作業している開発者が便利のように、上のファイルを読み込むこととします。

ライブラリ開発者は上記ファイルにいくつかの変更を加える必要があるでしょう。以下のセクションはなおも適用可能ですが、項2.2.1.1. 「ライブラリについて特別に必要なこと」まで飛ばすと必要な変更数行をみつけることができます。

これらの関数各々の目的についてはすぐに説明するつもりですが、開発者から見て重要なのはまさに文字列を _() と N_() 呼出しでラップしようとしているということです。これらの関数をソースコード内で用いるのなら、前述のヘッダーファイルがきちんと include されていることを確認する必要があります。もちろん include しない場合の失敗は全く明らかなもので、いくつかの関数が宣言されていないので単純にアプリケーションはビルドできなくなります。

2.1.2. i18n サポートコードの初期化

実際に必要なコード変更を施す前に、GNU gettext アプリケーションについてあまり詳しくない方のために、大まかな状況について簡単に説明します。ここでは、いくつかの詳細については省略して、主に GNOME に関連する部分について集中します。もしすべての詳細について知りたいのなら gettext マニュアル ([gettext] を参照) が非常に参考になるでしょう。

インストールされている翻訳文字列のコレクションは、別々のメッセージドメインに分けられています。各々別々のアプリケーションまたはライブラリは、ほとんど常に異なるドメインを持ち、ドメイン名は通常パッケージにまとめた後に決定されます。与えられたドメインの文字列はインストール時に効率的なバイナリ形式 (各々のロカルについて一つのファイル) にコンパイルされ、パッケージと一緒にインストールされます。当然のことながら、プログラムがこれらファイルが必要なときにどこにあるか見付け出せるように指示する必要があります。

一つのアプリケーションがいつも一つのドメインだけからメッセージを取得しているわけではありません。アプリケーション自身のメッセージはそれ自身のドメインからのものですが、アプリケーショ

ンはまた、翻訳文字列を返す、または表示するライブラリ関数も呼出していて、それら[メッセージ]はそのライブラリ固有のドメインからのものです。そういうわけで[国際化の]仕組みは、(必ず一つだけ存在する) 現在アクティブなドメインを切り替えるために、C ライブラリ内に存在しています。

重要な最後の点は、翻訳された文字列は、順々に回していくときに、何らかの方法でエンコードされていなければならないということです。中国語またはロシア語で使っているなら、英語圏のロカールで一般的な ASCII または ISO8859-1 エンコーディングは適切なものではありません。デフォルトでは、gettext は与えられたロカール設定について最も自然なエンコーディングを知っていて、適切なエンコーディングで文字列を返すでしょう。しかし GNOME 全体ではすべての文字列は UTF-8 でエンコードされているので、結局地域エンコーディングから UTF-8 に再度エンコードし直さなければなりません。幸運なことに、C ライブラリにこのことを伝えることができ、C ライブラリはロカール固有のエンコーディングに悩まされることはなく、各々そして毎回の文字列の検索における多少の時間を節約することができます。

最後の三つの段落は次の三行のコードが何をするか非常に詳細に説明しているだけです。これらのコードを、可能な限り早くユーザーに可視な文字列の処理の準備をするために、アプリケーションの main() 関数のすぐ最初に置きます。もしクライアントアプリケーションが使うライブラリを書きたいのなら、最初の二行をライブラリの初期化関数に入れておく必要があります。

```
bindtextdomain (GETTEXT_PACKAGE, SLICELOCALEDIR);
bind_textdomain_codeset (GETTEXT_PACKAGE, "UTF-8");
textdomain (GETTEXT_PACKAGE);
```

ここで何が起きてるか説明すべきですね。GETTEXT_PACKAGE 変数はビルドプロセスの間に (通常は configure.in または configure.ac) で定義され、このアプリケーションのメッセージドメイン名となります。従ってこの値はプログラム名に設定できます。SLICELOCALEDIR 変数 (この名前は普通はアプリケーション名に基づいていますが、完全に任意でああなたの判断にまかされています) はビルド時に定義され、メッセージカタログを保持するディレクトリ階層のルートとなります。[つまり] コンパイルされたメッセージカタログは SLICELOCALEDIR/locale/LC_MESSAGES/GETTEXT_PACKAGE.moとしてインストールされることになります。

注意

一般に LC_MESSAGES コンポーネントはどのようなカテゴリ名 (詳細については locale(7) マニュアルページの説明を見て下さい) にできます。GNOME では LC_MESSAGES だけについて処理すべきなので他については無視します。

それから上述のコードの断片はアプリケーションに次のように指示します。

- ・ どこでメッセージカタログをみつけるか
- ・ すべての文字列について UTF-8 エンコーディングを使うように
- ・ 最初に、少なくとも、このアプリケーションのドメインの文字列を使って翻訳するように

この最後の項目はライブラリでは用をなさない (クライアントアプリケーションがほとんど常にアクティブドメインを制御します) ので、そうする必要はありません。代わりに、ライブラリは、その固有の文字列を取得する前にメッセージドメインを設定し、その後で元に戻す必要があります (詳細についてはこの後で)。

ついでに、最後のセクションでふれたような i18n ヘッダーファイルを使っているのなら、i18n サポートを利用可能にするかどうかに関らず、上述のコード断片はコンパイルされるということにふれておくべきでしょう。前のセクションのヘッダーファイルのメリットがこれです。一度設定したらあなたも残りのコードも i18n サポートが常に利用可能であるかのように処理できて、もしサポートが存在

していなくても単に空の操作としてビルドされます。

注意

これを書いているときに libgnome/gnome-i18n.h ヘッダーファイルは ENABLE_NLS が未定義の場合に bind_textdomain_codeset() を定義しないことに気づきました。従ってもし libgnome からコードを再利用しているのなら、パッケージは (唯一の) 特別なケースとしてその関数を処理する必要があるでしょう。

2.2. 翻訳可能な文字列をマーク

さて、翻訳のためにすべての適切な文字列をマークする作業を始められるところまで来ました。すべての文字列をマークする必要はないことに注意して下さい。設定ファイルなどはロカール依存とはならないでしょうから、ユーザーが見ることのない文字列は通常はそのまましておくべきです。いくつかのエラーメッセージについても翻訳しないでおきたいと思いかも知れません。ユーザーはそれらのメッセージをコピーし、メールに貼り付けしてあなたに送り、あなたはそれらを理解してコードから grep で見つけ出すことができます。通常名前を翻訳する必要はありませんが、アプリケーションの情報ボックスで何が起きるかについては 項5.「秘訣と技」を参照して下さい。

一般的な法則として、アプリケーションの通常の操作でユーザーに見られる文字列は翻訳するべきです。例えばデバッグ出力と内部エラーメッセージといった、開発者向けの文字列を翻訳するべきではありません。

翻訳のためにマークするメッセージを決める際には注意する必要があります。重要な、[ユーザーに]見える文字列を未翻訳のままにしておくのは、素人くさく、ユーザーに不便を強いてしまいます。無関係の文字列を翻訳のためにマークするのは、無意味であり、GNOME のために作業している多くの翻訳ボランティアが提供しているサービスの誤用となるだけです。

2.2.1. ソースコードファイル

C または C++ ソースコードからそれらしいすべての文字列を探し出す最初の試みとして、ソースコードディレクトリでコマンド `xgettext -a -o my-strings --omit-header *.c *.h` を実行します。

これによって、ディレクトリのすべての .c そして .h ファイルを走査して、すべての文字列をファイル my-strings に出力します。このファイルは翻訳時に翻訳者が作業するものにとってもよく似ていますが、当面注目すべきは各々の行のフォーマットです。次のようになっているはずですが。

```
#: slice-n-dice.c:36
msgid "Sharpening the knives and preparing for action."
msgstr ""
```

これはこの文字列がファイル slice-n-dice.c の行番号 36 にあることを示しています。よって、もしこの文字列を翻訳するべきと判断したのなら、このファイルを開いてその文字列を _() で囲みます。つまり、例えばコード

```
popup_display ("Sharpening the knives and preparing for action.");
```

を次のように変更します。

```
popup_display (_("Sharpening the knives and preparing for action."));
```

マークアップの方法には一つだけ問題があります。_() は関数呼出しなので、すべての状況で使えるわけではありません。例えば、文字列をいくつか含む static 配列を初期化しているなら、そ

の初期化時に関数を呼出すことはできません。次のようなコードについて考えてみます。

```
ShapeData shapes[] = {"circle", "square", "triangle", NULL};
```

このような場合文字列をマークアップするには `N_()` を使うべきです。このマクロは、結局のところ何かするわけでもありませんが、`gettext` ツールは翻訳可能な文字列を探すのにコードを走査してこの種のマークアップをみつけたことを伝え、これらの文字列を抽出します。上の例は次のようになります。

```
ShapeData shapes[] = {N_("circle"), N_("square"), N_("triangle"), NULL};
```

一度 `N_()` で文字列をマークアップしたら、コード内でこれらの文字列が使われている箇所をすべて探し出し、文字列を出現させるコードを `_()` で囲む必要があります。これは `_()` がメッセージを検索し、翻訳する関数だからです。上の例について続けると、(i18n マークアップの後で) 次のようなコードを書けばいいでしょう。

```
for (i = 0; shapes[i] != NULL; ++i)
    show_shape_name (_(shapes[i]));
```

`shapes[]` からの文字列の取得は `_()` の呼出しを通して行なわれ、そうして `show_shape_name()` 関数は元の文字列ではなく、翻訳された文字列を使っていることに注意して下さい。static 文字列が実際に使われている場所を見落とすことは珍しくないで、(文字列を `N_()` で適切にマークアップしていても) コードマークアップのこの部分を正しく行うには、おそらく少し用心しなければなりません。

一度、先につくった `my-strings` ファイル内のすべての文字列についてくまなく調べてみて翻訳すべきかどうか決定したら、そのファイルは削除できます。このファイルは単に手順を先に進めるだけのものでした。この段階で(いつものように)、まだ `i18n` 部分が機能していなくても (`ENABLE_NLS` が未定義なので、すべての特別なマークアップは何もないのと同じになります) コードはなおも通常どおりビルド、動作できるはずで、後で問題が起きるのを避けるためにビルド可能かどうかはテストしておくべきです。

2.2.1.1. ライブラリについて特別に必要なこと

ライブラリを書いているのであっても、スタンドアローンのアプリケーションについて前に書いたほとんどすべてをあてはめることができます。唯一の問題は、ライブラリが文字列を処理する際に、ライブラリ固有のドメインがアクティブであるようにしたくても、通常ライブラリ自身はその時点でアクティブなメッセージドメインを決して知ることがないということです。この問題の最も単純な解決方法は項 2.1. 「i18n ヘッダーファイル」セクションでふれた `i18n` ヘッダーファイル内の定義の一つを変更することです。次のような箇所を探し、

```
#define _(String) gettext(String)
```

これを次のように変更します。

```
#define _(String) dgettext(GETTEXT_PACKAGE, String)
```

このようにして `_()` 関数を介するすべての文字列は常にライブラリのドメインで翻訳されるようになります。

他のもう一つの状況は起こることは非常にまれですが知っておく価値があるでしょう。例えば `libgnomeui`、内には文字列の配列 (ここではメニューアイテム) を処理する関数があります。これらの文字列のいくつかは標準的なもので、ライブラリ自身の中で翻訳されています。他はクライアントアプリケーションが提供し、そのドメイン内で翻訳されます。そこで `libgnomeui` は次の関数と便利

なマクロを定義しています。

例 2. libgnomeui/gnome-app-helper.cから引用

```
#define L_(x) gnome_app_helper_gettext (x)

const gchar *
gnome_app_helper_gettext (const gchar *str)
{
    char *s;

    s = gettext (str);
    if ( s == str )
        s = dgettext (GETTEXT_PACKAGE, str);

    return s;
}
```

これによって、L_() でマークされた文字列は、可能ならその時アクティブなドメイン、さもなくばライブラリのドメインで翻訳されます。同じような状況にあるのならこのコードを複製してもよいでしょう。

2.2.2. Desktop ファイル

desktop ファイルは GNOME ではパネルメニュー内のどのメニューの下にアプリケーションを表示するか決定するために使われています。また desktop ファイルには、どのようにしてアプリケーションを起動するか、どのアイコンを表示するかという情報と一緒に、パネル上のアプリケーションのツールチップとして表示する説明文を含んでいます。

国際化の観点から .desktop のパーツで興味を引くところは Name と Comment フィールドです。GNOME 2 ではこれら二つのフィールドだけが使われ、これらは desktop ファイル仕様の中で、翻訳可能なものとして定義されています。

これらフィールドを翻訳可能なものとしてマークするには、desktop ファイル slice-n-dice.desktop を slice-n-dice.desktop.in (.in 接尾辞は伝統的なものです) にコピーして、Name と Comment タグの前に下線(_)を加えます。マークアップの後ではファイルは次のようになります。

例 3. slice-n-dice.desktop.in

```
[Desktop Entry]
Encoding=UTF-8
_Name=Slice 'n' Dice
_Comment=Chop things up into shapes
Exec=slice-n-dice
Icon=slice-n-dice.png
Terminal=false
Type=Application
Categories=GNOME;Application;
```

タイプ

このように翻訳のためにファイルを変更する度にファイル名を記録しておきましょう。これはこのセクションとさらに後で述べるいくつかのセクションでふれているファイルタイプの両方にあてはまります。後に 項3.3. 「Makefile の変更」で、これらの新しいファイルを利用するためにビルド変更を施す必要がありますので、リストを持っておけばそれらの内どれについても見落すことがなくなります。

一度 項3. 「パッケージのビルドインフラに i18n [の仕組み]を取り込み」セクションでの変更を施したら、slice-n-dice.desktop.in だけを残して slice-n-dice.desktop は削除すべきです。ビルドプロセスはこのテンプレートに翻訳を統合し、すべての利用可能なロカールについての文字列を含む desktop ファイルを生成します。

2.2.3. サーバーファイル

要求に応じてコンポーネントをアクティブにする bonobo-activation-server が呼ぶアプリケーションは、どのようにアクティブにするかを記述し、[自身についての]説明文を含む .server という拡張子を持つファイルを持っています。そして、これらのファイルについても翻訳のために同様にマークアップする必要があります。

desktop ファイルと同じようにファイルの名前を GNOME_slice.server から GNOME_slice.server.in に変更します。そしてファイルを開き、type="string" 属性を持つ <oaf_attribute> タグを編集します。これらのタグは value="..." 属性を持っているので、これを _value="..." と変更します。こうしてマークアップを終えたら次のような数行があるはずです。

```
<oaf_attribute name="name" type="string" _value="Slicing factory"/>
<oaf_attribute name="description" type="string"
    _value="Factory for slicing and dicing"/>
```

いくつか (本当のところほとんど) のプロジェクトでは、コンポーネントが最終的にどこにインストールされるのかといった、configure スクリプトによって置き換えられるいくつかの変数がサーバーファイルの中に含まれているので、サーバーファイルは既にテンプレートになっているでしょう。よってそのファイルは既に GNOME_slice.server.in という名前になっているでしょう。この場合は単にさらに .in 拡張子を追加した GNOME_slice.server.in.in ファイルを作成します。intltool アプリケーションは configure が置き換えてしまう前に実行されることに注意して下さい。そして 項3. 「パッケージのビルドインフラに i18n [の仕組み]を取り込み」ではただ GNOME_slice.server.in.in は intltool によって GNOME_slice.server.in に変換されるということだけ覚えておいて下さい。

2.2.4. Glade ファイル

ユーザーインターフェースを構築するのに Glade を使っているのなら、glade ファイル内の文字列を翻訳できるものとわかるようにするのにマークアップする必要は特にありません。次のセクションで設定する予定の intltool アプリケーションは、glade 形式のファイルのどの部分 (ウィジェットラベル、メッセージ、アクセシビリティ文字列など) が翻訳可能であるか知っていて、自動的にそれらを抽出します。

唯一つだけ、Glade を使っているときには、オプションボックスの LibGlade タブの中の 翻訳可能な文字列を保存する を設定しないようにしておく必要があります。代わりに Glade が自動的にマークアップ文字列を含むファイルを生成します。

2.2.5. XML ファイル

もしアプリケーションが任意の形式の XML ファイルを伴うのなら、同様にこれらのファイルの一部を翻訳のためにマークアップすることができます。前と同じように slice.xml を slice.xml.in と

名前を変更します。それからファイル全体を通して翻訳すべき要素を下線でマークします。例えば、

```
<type level="safe">
  <shape>blunt triangle</shape>
  <description>Creates a triangle without sharp corners.</description>
</type>
```

を次のようにマークアップします。

```
<type level="safe">
  <shape>blunt triangle</shape>
  <_description>Creates a triangle without sharp corners.</_description>
</type>
```

<description> タグを翻訳のためにどのようにマークアップしているかわかるでしょう。また閉タグも同様に變更され、(おそらく元々の同じ DTD には従っていないけれども) そのファイルがなおも well-formed XML となっていることにも注意して下さい。

注意

- FIXME: 要素が翻訳のためにマークアップされたときそれに含まれる他の要素についてはどうなるか? 内側の要素についてもマークアップされるべきかどうか。
- もし doc-i18n-tool が修正されないのなら、それは奥行かしすぎるので、このセクションは置き換えられるべき。

2.2.6. Gconf schema ファイル

Gconf が使う schema ファイルにはキーの長短[様々な]説明が含まれていることがあり、これらの文字列は明らかに翻訳の候補となります。また schema ファイル中のキーも、より適切なものにローカライズすべきという意味で、翻訳可能なデフォルト値を持ち得ます。例えば、金融[関係の]アプリケーションは C ロカールで Stock Exchange キーのデフォルト値として "New York" を持っているかもしれませんが、de [ドイツ] ロカールではこのキーのデフォルト値は "Frankfurt" である方がよいはずです。

schema ファイルから引き出す適切なキーを持つための方法は、それらを <locale> 要素でくくみ、オリジナルファイルのロカールを C と指定することです。上述の金融アプリケーションの場合例えば次のようになります (いくつかの行は冗長なので省略しています)。

例 4. financial.schemas

```
<schema>
  <key>/schemas/apps/finance/preferences/current_market</key>
  ...
  <locale name="C">
    <default>New York</default>
    <short>Current stock exchange</short>
    <long>This key holds the name of the stock exchange that
      we are currently querying. It can have any string value
      that corresponds to a world market.</long>
  </locale>
```

</schema>

<locale name="C"> 要素内の説明文やデフォルト値を持たないキーは翻訳されません。普通は長短[様々な]説明文についてはすべて翻訳したいと思うのですが、デフォルト値についてはほんのたまにそうしたいと思うだけです。

2.2.7. ユーザー/API ドキュメント

残念ながら今のところ (gtk-doc によって生成されるような) ユーザーまたは API ドキュメントを翻訳者のために一貫した形式に変換するための簡単な方法はありません。基本的にユーザードキュメントはソースドキュメントを通じて手作業で翻訳しなければなりません。API ドキュメントは今のところ国際化をサポートしていません。

3. パッケージのビルドインフラに i18n [の仕組み]を取り込み

さて、前のセクションでのすべての作業の後で、すべての国際化にまつわる変更をビルドシステムに取り込むように変更するべきところまで来ました。このビルドシステムへの変更によっておおまかに三つのカテゴリに落としこむことのできる機能を得られます。

- ・ 設定、ビルド時に必要なツールがあるかどうかを検出
- ・ 最小限の手間で翻訳者が作業できるように翻訳文字列を抽出
- ・ 翻訳をビルドプロダクトに統合し、インストールできるように準備

パッケージが `aclocal`, `autoheader`, `autoconf` そして `automake` を走らせるために `autogen.sh` スクリプトを実行する、標準的なビルド方法を利用していることを前提とします。これ以降のセクションでする必要のあるすべての変更は `configure.in` ファイルといくつかの `Makefile.am` ファイルに対するものです。

GNOME での i18n サポートは Darin Adler, Maciej Stachowiak そして Kenneth Christiansen の書いた `intltool` ツールに強く頼っています。基本的には `intltool` は GNU `gettext` の機能を `desktop` ファイル、`Glade` や `Gconf` のサポートファイル、`XML` ファイル、そして他の二三のものはや一般的には使用されていない種類のものを含めるために拡張したものです。`autogen.sh` レベルからパッケージを構築する際に必要なのは `intltool` だけです。配布するアーカイブには対応するスクリプトのコピーが含まれているので、ユーザーのマシン上に `intltool` がある必要はありません。

注意

`intltool` の特別な機能を必要としないようなとても低レベルのライブラリを作っているのなら、パッケージではただ `gettext` だけを使うことも可能でしょう。けれども `intltool` のオーバーヘッドはごくわずかであり簡単に使えるので、今のところこのチュートリアルではこのような場合についてはふれていません。

3.1. `autogen.sh` の変更

最初に `glib-gettextize` スクリプトの位置を指示することで、`autogen.sh` スクリプトがすべてのプロセスをブーストラップし、実行できることを確認します。[ただし] GNOME の CVS リポジトリ内の `gnome-common` モジュールの `autogen.sh` スクリプトを利用する場合は、これは自動的になされるので次のセクションにスキップできます。

標準的な GNOME autogen.sh スクリプトを使っていないのなら、適切なスクリプトのチェックを追加し、見つかったものは何でも実行する必要があるでしょう。次はこれ(gettextize と intltool のチェック)を行うサンプルコードです。

注意

autogen.sh スクリプトの構造に依存しているので、このコードをそのまま使うことはできないかもしれません。これらの例は、gnome-common モジュールと Gconf や metacity 内に含まれるようないくつかのスタンドアロンスクリプトのように、GNOME で共通に使われる autogen.sh の形式を前提としています。

また glib または gtk+ の autogen.sh (と対応する acinclude.m4) も見たいと思うかもしれませんが、これらのパッケージはよりずっと自己充足し、glib-gettextize の利用を強制しています。しかしこれらは GNOME の他の部分で使われている標準スクリプトとはずっと異なっています。

例 5. autogen.sh に追加

```
if grep "^AM_[A-Z0-9_]\{1,\}_GETTEXT" "$CONFIGURE" >/dev/null; then
if grep "sed.*POTFILES" "$CONFIGURE" >/dev/null; then
  GETTEXTIZE=""
else
  if grep "^AM_GLIB_GNU_GETTEXT" "$CONFIGURE" >/dev/null; then
    GETTEXTIZE="glib-gettextize"
  else
    GETTEXTIZE="gettextize"
  fi
fi

$GETTEXTIZE --version < /dev/null > /dev/null 2>&1
if test $? -ne 0; then
  echo
  echo "**Error**: You must have ¥`$GETTEXTIZE` installed" ¥
  echo "to compile $PKG_NAME."
  DIE=1
fi
fi
fi
(grep "^AC_PROG_INTLTOOL" "$CONFIGURE" >/dev/null) && {
(intltoolize --version) < /dev/null > /dev/null 2>&1 || {
  echo
  echo "**Error**: You must have ¥`intltoolize` installed" ¥
  echo "to compile $PKG_NAME."
  DIE=1
}
}
```

autogen.sh 内の以降の部分では、libtoolize または他のスクリプトを実行する前に、glib-gettextize と intltoolize を次のように実行して intl プロセスをブーストラップさせるべきです:

例 6. さらに autogen.sh に追加

```

if test "$GETTEXTIZE"; then
  echo "Creating $dr/aclocal.m4 ..."
  test -r aclocal.m4 || touch aclocal.m4
  echo "Running $GETTEXTIZE... Ignore non-fatal messages."
  echo "no" | $GETTEXTIZE --force --copy
  echo "Making aclocal.m4 writable ..."
  test -r aclocal.m4 && chmod u+w aclocal.m4
fi
if grep "^AC_PROG_INTLTOOL" $bn >/dev/null; then
  echo "Running intltoolize..."
  intltoolize --copy --force --automake
fi

```

例えば `aclocal.m4` が現在のディレクトリにない場合は、ここである程度のカスタマイズが必要となるかもしれません。Gconf のような既存のアプリケーションから例をコピーするのがコツです。

注意

普通の `gettextize` の代わりに `glib-gettextize` を使うのは GNU `gettext` のメンテナと GNOME 開発者の間に哲学の違いがあるからです。`gettext` メンテナは、複数のバージョンの `gettext` を統合するか `gettext` を使うプロジェクトを開始するときのみ、`gettextize` のようなスクリプトを実行するべきであると言っています。しかし GNOME では (`gettextize` が生成するものと基本的には同じ) `glib-gettextize` が生成するファイルを CVS に保存しておかないで、ちょうど `autogen` 時に生成するのを好みます。二つの作業方法を単一のスクリプトにはできないので、GNOME プロジェクトではオリジナルの `gettext` スクリプトをほんの少し変更したバージョンを使っています。

`gettext v0.11` では `autopoint` という `glib-gettextize` と同じ役割を果たすスクリプトが紹介されています。しかし、GNOME は今のところ `gettext v0.11` を要求していないので、そのスクリプトに依存するのは時期尚早です。

3.2. `configure.in` の変更

`configure.in` には多数の変更が必要となります。`intltool` の存在をテストし、ビルド時に使ういくつかの定義をセットアップし、翻訳されている言語をリストする必要があります。これらすべてを行うコードを次の例で示しています。一つのブロックにすべてまとめて示していますが、`intltool` の検出はもともと `configure.in` の前の方に、テストの残りはファイル内のより後ろの方にあります。

例 7. `configure.in` への追加部分

```

AC_PROG_INTLTOOL([0.23])

GETTEXT_PACKAGE=slice-n-dice
AC_SUBST(GETTEXT_PACKAGE)
AC_DEFINE_UNQUOTED(GETTEXT_PACKAGE, "$GETTEXT_PACKAGE")

ALL_LINGUAS=""
AM_GLIB_GNU_GETTEXT

slicelocaldir='${prefix}/${DATADIRNAME}/locale'
AC_SUBST(slicelocaldir)

```

ここでいくつものアイテムについて説明する必要があるでしょう。第一に、GETTEXT_PACKAGE を処理している三行は、このアプリケーションのメッセージドメインを定義し、その定義が生成される Makefile とまた (AC_DEFINE_UNQUOTED() マクロによって) ソースコード内の #define で利用されるようにします 前に戻って GETTEXT_PACKAGE を名前として使う bindtextdomain() の呼出しで i18n サポートが初期化されます。

第二に、まだ翻訳がありませんので空の ALL_LINGUAS 文字列を追加しました。いくつか翻訳が利用できるのならそれら翻訳の言語コードを ALL_LINGUAS に追加します。この変数はスペースで区切られた言語コード群からなる文字列で、二三言語を追加すると次のようになるでしょう:

```
ALL_LINGUAS="de en_AU no sv"
```

第三に、AC_PROG_INTLTOOL マクロは必要な最低バージョンを指定するオプション引数をとりませんが、このコード断片では Gconf schema 内のデフォルトキーを正確に抽出するようになった最初のバージョンである 0.23 を指定しています。この機能を必要としないのなら、広く利用可能になりほとんど正確に動作するようになったバージョン 0.21 を指定するのが安全でしょう。

最後に、どこにロカールファイルをインストールするか指示する slicelocaledir という変数 (名前は何でも好きなものにできます) を定義しています。次のセクションの Makefile.am の変更で、この変数は生成される各々のファイルに渡され i18n サポートを初期化する bindtextdomain() 呼出しで使われます。これは自動的になされるべきであると思うかもしれませんが、メッセージファイルが GNU gettext の想定する場所とは若干異なる場所にインストールされる Solaris でもシームレスにビルドできるようにするためにこの構文が必要となります。

configure.in で最後に必要な変更は、ファイルの終わりまで行ってファイル po/Makefile.in を AC_OUTPUT マクロによって出力されるファイルリストに追加することです。

この時点でもなお問題なくビルドできるはずですが、i18n サポートは言語がないのと AM_GLIB_GNU_GETTEXT が呼ばれていないためにまだ無効になっています。さらにも言語が利用可能であったとしても Makefile 内で必要不可欠なビルドの変更を行っていないので、intltool の管理する多数のファイルに適用されません。これが次のステップです。

3.3. Makefile の変更

まず最上位の Makefile.am を編集して po を降下していくサブディレクトリのリストに追加します。次のようになるでしょう。

```
SUBDIRS = src po
```

また intltool-extract.in、intltool-merge.in そして intltool-update.in を最上位の Makefile.am 内の EXTRA_DIST ルールに追加します。これら三つのファイル名から .in 拡張子を除いた名前をそのディレクトリ内の .cvsignore に追加します。

それから main() 関数またはライブラリ初期化コードを持つファイルを含むディレクトリへ移動します。そのディレクトリの Makefile.am 内の include フラグを編集し、メッセージファイルを保存しているディレクトリが追加されるように設定する必要があります。

```
INCLUDES =
  -I$(top_srcdir)
  ...
  -DSLICELOCALEDIR="¥"$(slicelocaledir)"¥"
```

ここで SLICELOCALEDIR は bindtextdomain() 呼び出しに渡す変数名にマッチし (項 2.1.2. 「i18n サポートコードの初期化」を参照)、slicelocaledir は前のセクションで configure.in を編集し

実際に選んだ変数名にマッチします。

必要な Makefile 編集の最後のピースは intltool が desktop、schema、server、XML ファイルの完全な翻訳バージョンをビルドできるようにすることです。項2.2.「翻訳可能な文字列をマーク」に戻ると、.in 拡張子をつけた多数の新しいファイルを作成し、翻訳するフィールドを示す特別なマークアップを施したでしょう。(.in 拡張子なしの)これらファイルのすべてが現在いくつかの Makefile.am 中に存在しています。次のような行があるはずで

```
desktop_DATA = slice-n-dice.desktop
```

intltool の手助けでテンプレートからこのファイルが生成されるように、この行を編集する必要があります。上の例では Makefile.am に次のような変更を施します。

```
desktop_in_files = slice-n-dice.desktop.in
desktop_DATA = $(slice_in_files:.desktop.in=.desktop)
@INTLTOOL_DESKTOP_RULE@
```

ここには二つの変更が含まれています。最初にすべての desktop ファイル (この場合は一つだけですが、複数持てます) を(Automake について特別ではない)別の名前の元で並べます。それからそれを含む、リストの前の方で変換パターンを適用してできたファイルを desktop_DATA ルールが算出します。

二番目の変更は、slice-n-dice.desktop を生成する slice-n-dice.desktop.in への利用可能なすべての翻訳の統合を引き受ける、いくつかの特別な Makefile ルールを取り込むことです。これらのルールは configure.in 内の AC_PROG_INTLTOOL マクロの一部で置き換えられる @INTLTOOL_DESKTOP_RULE@ 変数を使って取り込まれています。

@INTLTOOL_DESKTOP_RULE@ の代わりにそれぞれ @INTLTOOL_SERVER_RULE@、@INTLTOOL_SCHEMAS_RULE@、@INTLTOOL_XML_RULE@ とすることを除けば server、schema そして一般的な XML ファイルについてもすべて同じです。またファイル名を保持するターゲット (上で desktop_DATA と呼ばれているもの) も同様に変更する必要があるでしょう。desktop、schema、server そして XML ファイルについてそれぞれ別のターゲットを持つことも可能です。intltool の配布に同梱される README ファイル内にこれらの変更についてのさらなる例が載っています。

Makefile.am についてすべての変更を終えたら、.in 拡張子を持たないオリジナルの desktop などのファイルについてはこれらは今では自動生成されるようになっているので削除できます。また cvs が新しく生成されたこれらの不明なファイルについて文句を言わないように、おそらくまた新しく生成物の名前を .cvsignore に追加したいと思うことでしょう。

3.4. po/ディレクトリ内での変更

ビルドプロセスがすべての翻訳可能な文字列を抽出するのを助けるためには、プロジェクトの po サブディレクトリ内に重要なファイル POTFILES.in (と場合によっては POTFILES.skip) を置く必要があります。

POTFILES.in ファイルにはプロジェクト内の翻訳可能な文字列を含むすべてのファイルのリストが含まれています。このリストにはソースコードファイル、server ファイル、desktop ファイルなどが含まれています。コマンド intltool-update -m を実行するとこのファイルを生成できます。これによってすべての可能性のあるソースファイルが走査され、翻訳可能な文字列を含んでいるすべてのファイルの、長大なリストのエラーメッセージを吐き出されます。

エラーメッセージの各々のファイルについては、本当にそれらが翻訳可能な文字列を含んでいる(ほとんどはそうでしょう)か確かめて、そうであれば POTFILES.in に追加する必要があります。しかし、intltool-update が認識するけれども翻訳可能な文字列を含まないファイルについては、POTFILES.skip にファイル名を書いておきます。すべてのファイルについて確認を終えたら再度 intltool-update -m を実行すると All files containing translations are present in

POTFILES.in というようなメッセージが表示されるはずです。

うっかり重要なファイルが POTFILES.in から抜け落ちていないことを簡単に検証できるので、アプリケーションの保守作業として定期的にこのコマンドを実行することが推奨されます。

ビルドプロセスの間に po ディレクトリ内に多数のファイルが生成されるでしょう。プロジェクトが cvs リポジトリで管理されているとすると、cvs がこれらの新しいファイルについては知らないために cvs コマンドの発する非常に多くのノイズに悩まされることとなります。cvs を静かにするには po ディレクトリ内に次のような内容の .cvsignore ファイルを作成するだけで十分なはずです。

```
*.gmo
*.mo
*.pot
Makefile
Makefile.in
Makefile.in.in
POTFILES
cat-id-tbl.c
messages
missing
po2tbl.sed
po2tbl.sed.in
stamp-cat-id
```

4. 次のステップは?

一度ここまで来たら作業は本質的には完了です! 翻訳者はもう作業し、すべてのマークアップされた文字列を彼ら自身の言語に翻訳し、成果をアプリケーションに戻すことができます。

ときどき、アプリケーションがそれぞれの言語についてどれだけ翻訳されたか調べるのに、intltool-update -r を実行したいと思うかもしれません。調べた結果についてできることは本当に少いので、これは純粹に情報を得る目的です。

一般的に、開発者と翻訳者の間の協調作業は次のようなプロセスとなるでしょう。

- ・ 翻訳者は彼らの言語について新しいファイルを作成します。これは XX.po という名前で、XX はロカール名(よく知られた言語と言語コードのリストがあります)で置き換えられます。
- ・ 新しいファイル内のいくつか、またはすべてを翻訳したら、翻訳者はそのファイルを po ディレクトリに commit し、そのディレクトリ内の ChangeLog を更新します。また翻訳者は configure.in の中の ALL_LINGUAS 変数に新しい言語コードを追加します。
- ・ リリースの前に開発者は、すべての翻訳者に彼らが翻訳を更新し、commit できるように知らせるべきです。通常一週間あれば十分でしょう。
- ・ CVS ツリーにリリースタグを打つ直前に、開発者は、ビルドプロセスに変更を加えるすべての言語(.po) ファイルを commit するべきです。これは翻訳者に対して無用の競合をつくってしまうことを避けるために、リリースのときだけ行うべきです。一般には言語ファイルはそのままにしておくべきですが、リリース時には CVS ツリーとコード内のものを同期します。

リリース手順の他に国際化を尊重する開発者が責任を持つべきことは、追加した新しい文字列を確実にマークアップしておくことです。時折、様々な状態を通して、[新しい語句について未訳であっても]英語のメッセージが表示されるのを見るためだけに、アプリケーションを、(もしあるなら)完全に翻訳された[英語以外の]言語の内の一つで動作させるのはやっておく価値があります。

5. 秘訣と技

i18n の手順については、様々なメーリングリストや他の場所で蓄積された沢山のちょっとした技があります。いくつかのあまり明白ではない落とし穴についてここで列記しておきます。

これら列記されたものに加えて、GNOME 翻訳プロジェクト (GNOME Translation Project, GTP) のコーディネータの一人 Christian Rose が書いた文書 [menthos] を注意深く読んだ方がよいでしょう。この文書には多数の状況の例と簡単に翻訳可能なテキストの書き方 (慣用句を避けるといった明かなものだけでなく、翻訳のテキストマークアップとはなっていないようなより微妙な問題についても) についてのアドバイスが含まれています。実際、私がこのチュートリアルで彼の素晴らしい働きの重複を避けることができたように、読者も Christian の文書を読むべきです。

- 決して空文字列をマークアップしないように。これによって空文字列を返すのではなく、相当する言語ファイルのヘッダーを取得してしまう副作用があります。[訳注: 多分 po ファイル冒頭の msgid "" フィールドのこと。このフィールドには翻訳ではなく po ファイル自体のメタ情報が含まれている]

_("") というようなコードを書くのを避けるのは比較的簡単ですが、動的に生成される文字列については注意する必要があります。空になるかもしれない可能性があるなら、まずチェックして翻訳すべき内容がないなら _() を呼ばないようにします。

- 国際化されたアプリケーションをテストしようとしているのなら、翻訳の結果を見るのに make install するのを忘れないようにしなければなりません。なぜならライブラリ内の i18n サポートは、メッセージファイルがある場所への明示的なパス (bindtextdomain() の第二引数) を共になって初期化され、プログラムはそのパスを見に行くからです。もし指定ロカールディレクトリに何もみつからなかったとしても、現在のディレクトリを探すようなフォールバックの仕組みはありません。
- アプリケーションの About (情報) ボックスをつくるときには、フィールドの内の一つに翻訳者名を渡すことができます。実行時のロカールの翻訳者を表示するために、ほとんどの GNOME アプリケーションでは次のようなコードを持っています:

```
gchar *translator_credits = _("translator_credits");
...
about_box = gnome_about_new(...
    strcmp (translator_credits, "translator_credits") != 0 ? ¥
    translator_credits : NULL,
    ...
```

それから翻訳者は translator_credits 文字列を彼/彼女達自身の名前に翻訳します。もし実行時のロカールについて翻訳者がいない (すなわち _() が原文を返す) なら、About (情報) ボックスコンストラクタには翻訳者クレジットが渡されず、タブは表示されません。

翻訳者は translator_credits を見て、正しいコンテキストではどう訳せばよいか知っているので、これをこの目的のために使うことを推奨します。

アプリケーション国際化のチェックリスト

次のリストは前のページまでのすべてをまとめです。全部完了させたと思った時点で最終的なチェックとして役に立つでしょう。

- 必要なら _()、N_() そして他のサポート関数を定義する i18n ヘッダーファイルを作成し、

- include します。(項2.1.「i18n ヘッダーファイル」)
- ・ (アプリケーションなら `main()`、ライブラリならその初期化関数の中で) 可能な限り最初の方で `bindtextdomain()` とその仲間の関数を呼出します。(項2.1.2.「i18n サポートコードの初期化」)
- ・ 翻訳すべきソースコード内のすべての文字列をマークアップします。関数呼出しが可能なら `_()` で、さもなくば `N_()` を使います。後者の場合ではその文字列が使われている箇所を `_()` 呼出しで囲みます。(項2.2.1.「ソースコードファイル」)
- ・ “特別な”ファイルでのマークアップ
 - ・ desktop ファイル (項2.2.2.「Desktop ファイル」)
 - ・ bonobo-activation-server のサーバーファイル (項2.2.3.「サーバーファイル」)
 - ・ Glade ファイル (項2.2.4.「Glade ファイル」)
 - ・ XML ファイル (項2.2.5.「XML ファイル」)
 - ・ Gconf スキーマファイル (項2.2.6.「Gconf schema ファイル」)
- ・ 必要なら `autogen.sh` に `glib-gettextize` の検出と実行、そして `intltool` を検出するためのコードを追加します。(項3.1.「autogen.sh の変更」)
- ・ `configure.in` を編集して `AC_PROG_INTLTOOL` を呼び、`GETTEXT_PACKAGE` を設定、`ALL_LINGUAS` を定義、`AM_GLIB_GNU_GETTEXT` を呼び、そしてロカールファイルの位置を `foolocaldir` 変数を使って置き換えます。(項3.2.「configure.in の変更」)
- ・ 最上部の `Makefile.am` を編集して `SUBDIRS` 行に `po` を含めます。また `intltool` を使っているのなら `intltool-extract.in` `intltool-merge.in` `intltool-update.in` をこの `Makefile` の `EXTRA_DIST` に追加します。(項3.3.「Makefile の変更」)
- ・ 最上部ディレクトリの `.cvsignore` を調整して `intltool-*` ファイルを無視するようにします。(項3.3.「Makefile の変更」)
- ・ `main()` 関数かライブラリ初期化コードを含むディレクトリ内の `Makefile.am` を編集してロカール位置を定義します。つまり `-DFOOLOCALEDIR` を設定します。(項3.3.「Makefile の変更」)
- ・ `intltool` が処理するための特別なファイルをマークアップしたすべてのディレクトリで `Makefile.am` 内の適切なルールを変更し、一つまたは複数の `@INTLTOOL_XXX_RULE@` テンプレートを追加します。またこれらのディレクトリ内の `.cvsignore` ファイルを調整して生成されるこれらのファイルを無視するようにします。(項3.3.「Makefile の変更」)
- ・ `po` ディレクトリ内に `POTFILES.in` と必要なら `POTFILES.skip` を作成し、適切なファイルを `.cvsignore` に追加します。(項3.4.「po/ディレクトリ内での変更」)
- ・ 通常のイベントでパッケージについて作業しているときは `*.po` ファイルの変更は `commit` しないようにします。しかしリリースの直前には皆がリリースに同期させられるようにこれらのファイルを `commit` するべきでしょう。(項4.「次のステップは?」)
- ・ 文字列が翻訳者に親切なものになるように Christian Rose の文書を読んで下さい。(〔menthos〕)

変更履歴

0.4 から 0.5 までの変更点

- ・ いくつかの URL の間違いを修正 (Priit Laes と Anand Subramanian、ありがとう)
- ・ タイポ修正(Telsa Gwynne).

0.3 から 0.4 までの変更点

- ・ HTML 版用のスタイルシートを追加
- ・ 間違った場所を指していたリンクを修正

0.2 から 0.3 までの変更点

- ・ デバッグ文字列の翻訳は権限の誤用である (明らかに率直に言うことは望まれていない)ことを説明

0.1 から 0.2 までの変更点

- ・ Christian Rose の文書への参照を追加
- ・ Pdraig O'Briain による多数の説明と修正をとりこみ
- ・ 付録 A. アプリケーション国際化のチェックリスト にいくつかの内容を追加

参考文献

オンラインの情報源

[kmaras] Translation of the GNOME desktop environment.
(<http://developer.gnome.org/projects/gtp/translate-gnome/index.html>).

[gettext] Gettext manual.
(http://www.gnu.org/software/gettext/manual/html_chapter/gettext_toc.html).

[menthos] L10N Guidelines for Developers.
(<http://developer.gnome.org/doc/tutorials/gnome-i18n/developer.html>).