



Multilizer Localization Guide

Multilizer® 6.0 Localization Guide

February 2004

Copyright © 2004 Multilizer Inc. All rights reserved.

Multilizer is a registered trademark of Multilizer Inc. All other trademarks and registered trademarks are the property of their respective owners.

Table of Contents

Introduction

Use of this manual	7
Multilizer products	7
Installation	8
Useful links	8
Multilizer Support and Maintenance	9

Part I – Multilizer localization process

Create Project.....	11
Multilizer project, MPR	11
Arranging the files to localize	11
Project Wizard.....	12
Target type	12
File Target	13
File type	14
Target options	15
Information.....	15
Languages.....	16
Finish	17
Project maintenance.....	18
Introduction	18
Project View	18
Project tree	19
Translation work-place	22
Info page.....	23
Re-scan project.....	25
Categories	25
Pre-translate project.....	25
Translate using Translation Memory	25
Import translations from files and databases	26
Prepare project for translation.....	26
Filtering.....	26
Pseudo languages → QA.....	26
Lock Visual Editors.....	26

Translation control.....	26
Share translation work.....	27
Introduction	27
Default workflow	27
Exchange Wizard.....	28
Creating Localization Kit.....	28
Exchange Wizard steps.....	29
Export Wizard	33
Export Wizard steps	33
Import Wizard.....	37
Import Multilizer Project (MPR)	37
Import from other formats.....	38
Import Wizard steps	38
Options for typical file imports	40
Translate	44
Restrictions	44
Translation work-place	44
Selecting visible columns	45
Row filtering.....	46
Translation grid options.....	48
Visual Editors, Wysiwyg	49
Visual Editor (Wysiwyg) settings	50
Localization of strings.....	51
Localization of accelerators.....	51
Localization of images.....	52
Localization of AVI and other custom resources	53
Software translation specifics.....	53
Characters with a special purpose	54
Maximum length of translations.....	54
Translation Memory maintenance	55
Sending back translations	55
Translation Memory	56
Introduction	56
Ensuring the Translation Memory quality.....	56
Using Translation Memory	56
Finding Translations	58
Installation of Multilizer Translation Memory	58

Create Local Translation Memory	59
Create Server Translation Memory	59
Store translations	60
Save project translations	60
Import documents	61
Segmentation	62
Block words	62
Maintenance	62
Quality assurance	64
Validation Wizard	64
Validation types	65
Working with validation results	66
Pseudo Languages	68
Cover	68
Minimum	68
Pseudo language	68
Informative QA features	68
Cell coloring	69
Display of non-printing characters	69
Statistics panel	69
Control boundary colors	69
Translation Status	69
Set status automatically	70
Set status manually	70
Reports	71
Project reports	71
Validation reports	71
Modifying reports	72
Build localized versions	73
How does build work	73
Part II – Tutorials	
Windows Tutorial	75
VCL Tutorial	92
.NET Tutorial	117
Java Tutorial	140
J2ME Tutorial	148

Database Tutorial	157
XML Tutorial	163
Source Localization Tutorial	167
Data File Localization Tutorial	169

Part III – Appendices

Index.....	174
Table of figures.....	176
Glossary	179
Supported file types.....	181
Localization Walkthrough Quick Reference.....	183

1

Introduction

Use of this manual

Multilizer 6.0 Localization Guide covers the entire Multilizer localization process. It serves as reference for localization engineers, project coordinators, and software engineers, and others using Multilizer for software and content localization.

The translators' tasks are covered in the chapter "Translate," p. 44.



Multilizer products

The manual covers all Multilizer 6.0 products:

- Multilizer Enterprise
- Multilizer for Windows
- Multilizer for .NET
- Multilizer for Visual C++
- Multilizer for VCL
- Multilizer for Java
- Multilizer Translator Edition Pro
- Multilizer Translator Edition

The abovementioned Multilizer editions have all the same basic set of features, including a uniform way of working and user interface.

Multilizer products differ in two aspects:

- **Support for Process features.**
There are different Multilizer editions depending on the major tasks in the localization process. For example, the translator uses Multilizer Translator Edition and the QA person uses Multilizer Translator Edition Pro.

The Multilizer localization process forms **Part I** of this manual.

- **Supported software platforms and contents.**
There are different Multilizer editions with different support for platforms. For example, .NET software can be localized with Multilizer for .NET; and if database localization is also needed, then Multilizer for Windows or Multilizer Enterprise is required.

Software platform and content-specific tutorials form **Part II** of this manual.

The following table summarizes the differentiating features:

Multilizer Enterprise
 Multilizer for Windows
 Multilizer for .NET
 Multilizer for Visual C++
 Multilizer for VCL
 Multilizer for Java
 Multilizer Translator Edition Pro
 Multilizer Translator Edition

Process features								
Create Multilizer localization project	✓	✓	✓	✓	✓	✓		
Scan software/content to include localizable data in Multilizer project	✓	✓	✓	✓	✓	✓		
Translate Multilizer localization project	✓	✓	✓	✓	✓	✓	✓	✓
Build localized versions	✓	✓	✓	✓	✓	✓	✓	
Create localization kits	✓	✓	✓	✓	✓	✓		
Multilizer TM installs on database server	✓							
Multilizer TM installs on desktop database	✓	✓						
Built-in single-user TM database	✓	✓	✓	✓	✓	✓	✓	✓
Supported software platforms and contents								
.NET	✓	✓	✓					
Windows	✓	✓		✓	✓			
Windows CE	✓	✓						
Java	✓	✓				✓		
Desktop databases	✓	✓						
Server databases	✓							
Data files	✓	✓	✓	✓	✓	✓		

Process feature differences are explained in depth in **Part I** of this manual.

More specific info on supported software platforms and contents is available in the tutorials in **Part II** of this manual.

See appendix ‘Support file types’ for a comprehensive list of supported file formats.



Installation

Multilizer is installed either from CD or directly from the Internet. Refer to the instructions of installation software for details.



On installing the commercial version of Multilizer, remember to enter the serial number during installation.

Useful links

The Multilizer website offers useful services both for evaluation version users as well as users of the commercial version.

http://www.multilizer.com/download	Full setups, patches, and updates of all Multilizer products.
http://www.multilizer.com/support	Support pages. Check out latest technical news and notifications of Multilizer products. Customers can register Multilizer product here, in order to get extra services on Multilizer support pages.
http://www.multilizer.com/support/documents	In documents, you can find links to software localization-related documents, Multilizer fact sheets, in-depth articles of Multilizer technology, etc.

Multilizer Support and Maintenance

Multilizer users can subscribe to Multilizer Support and Maintenance Agreement (SUMA). SUMA includes free upgrades and support.

There is always a separately agreed amount of free consulting included in SUMA. This consulting is highly recommended for companies that want to achieve all benefits that Multilizer technology can bring in their particular cases.

For more info, please contact Multilizer sales: sales@multilizer.com.



Part I: Multilizer localization process

This part goes through the typical tasks of the Multilizer localization process.

Each chapter in this part starts with a short description of:

- Multilizer editions that support the specific task.
- Multilizer user's role in the task.
- Wizards that guide the user.

2

Create Project

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java
User's role in process:	Localization engineer, Localization project coordinator, Software developer
Wizards:	Project Wizard

Multilizer project, MPR

Before anything can be localized with Multilizer, a *Multilizer Project* (MPR) must be created. Thus, the first task in localizing software is to create it.

Each Multilizer project can contain 1...N *localization targets*. Each *target* contains 1...N files; a target can be software executable, a .NET solution, a directory with property files, or database, just to name a few.

Multilizer project keeps all information needed in localization, such as list of targets along with *native (source) language*. In addition, target languages are included.

The core idea of working with the Multilizer project is to maintain always the same project file; therefore the project should be created only once. In order to keep the Multilizer project and targets synchronized, you can force Multilizer to scan targets; this is called re-scanning (→ Re-scan project, p. 25). Re-scan checks if there are any changes in targets; any changes are flagged in the project so that the user easily sees the changes.

Arranging the files to localize



NOTE!

Because scanning targets involves synchronizing files to localize with the project file, you should arrange the files to localize so that they are always located in the same folder. Access to the folder must be granted to enable the following Multilizer tasks: create project, scan project (→ Re-scan project, p. 25), and build localized files (→ Build localized versions, p. 73).

Other tasks operate the Multilizer project only.

Localized files are created in folders below the original files. The name of the folder is the same as the locale ID. If the localized file is multilingual (e.g., multilingual EXE), it is put in the folder called 'all.'

The following picture illustrates the files and folders created for a sample 'notepad.exe' localization project. In this example, there is only one target (notepad.exe) and by default, Multilizer creates the project (notepad.mpr) in the same folder.

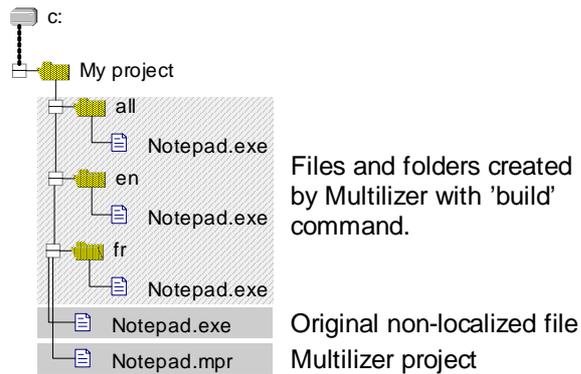


Figure 1: Organizing the files to localize.

Upon building localized versions, Multilizer creates 'all,' 'en,' and 'fr' folders with localized copies of notepad.exe. In this example, Notepad.exe is localized to a multilingual version (in 'all' folder), an English version, and a French version.

Project Wizard

Project Wizard guides the user in creating a new project, in order to ensure that required information is included in the project.



The screenshots in this chapter are taken from a simple Windows software localization project. To know correct settings for a specific platform or database, check out the respective tutorials.

Target type

The first screen prompts the user to choose between file and database localization. You should choose 'Localize a file' if you localize any software or content files, such as XML or INI-files.

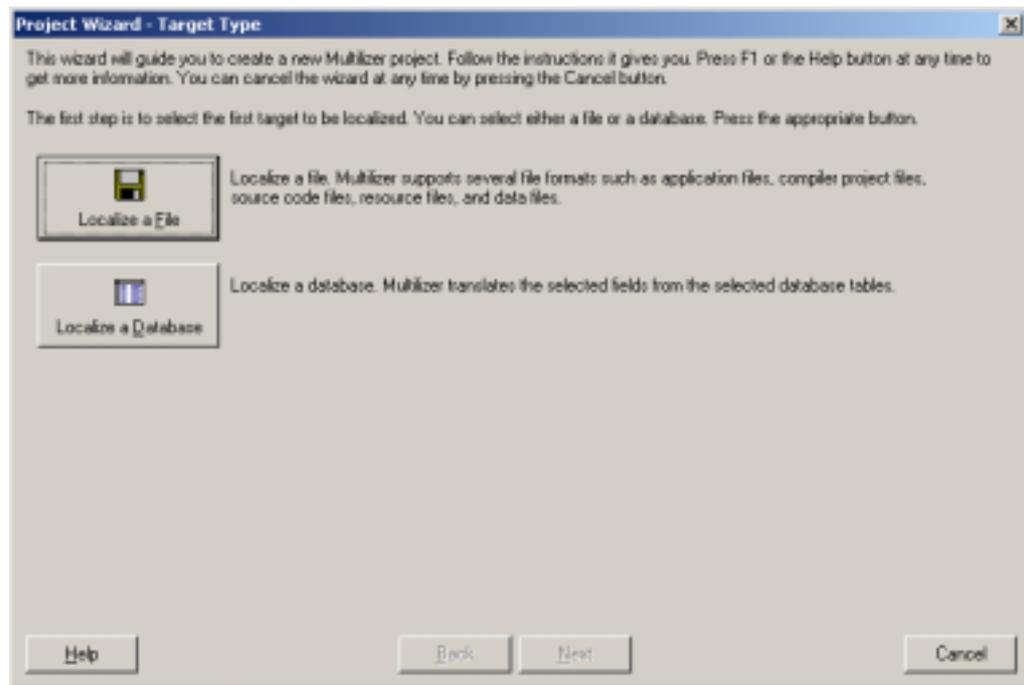


Figure 2: Selection between localizing file and localizing databases.

Regardless of what you choose as the target type, you can always add different targets afterwards. Upon finishing Project Wizard, you can choose **Project**→**Targets...** to see and maintain a list of targets.

File Target

The next screen asks you to specify the file(s) to add in the project.

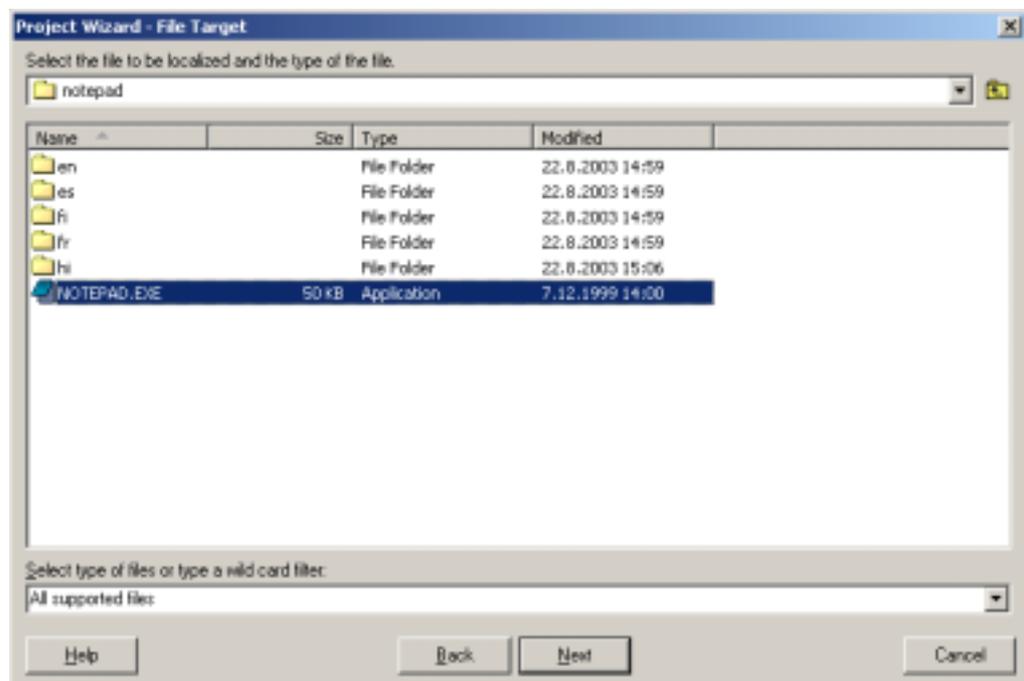


Figure 3: Specifying files to be included in project.



If you select multiple files, we recommend that all files are of the same *platform type* and *target type*. You can ensure this by selecting type of files from the combo box.

You can add different platforms and target types later to the project. This is done either directly in project tree, or from Targets dialog (Project→Targets...).

Select file type

If you localize software/content of a specific platform, select 'Select file types...' from combo box. It activates a dialog that allows user to choose platform and file type.

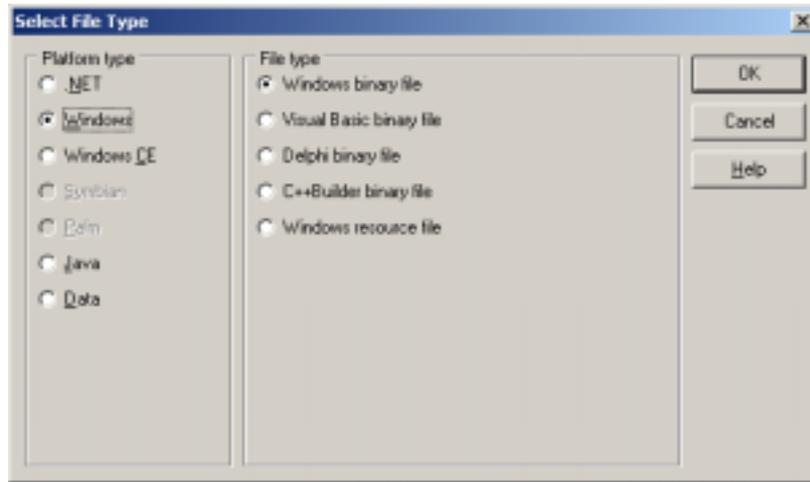


Figure 4: Specifying file type.

File type

If file type was not defined on previous page, certain files require specifying it on this page.

This page is not shown, if file type was already specified or selected file can be localized in only one way.

This is the case with Windows executables for instance; an executable is localized differently, depending on the type. This is due to the fact that for example Delphi binary file differs from standard Windows binary file.

Selecting file type ensures that localization is performed correctly on selected file.

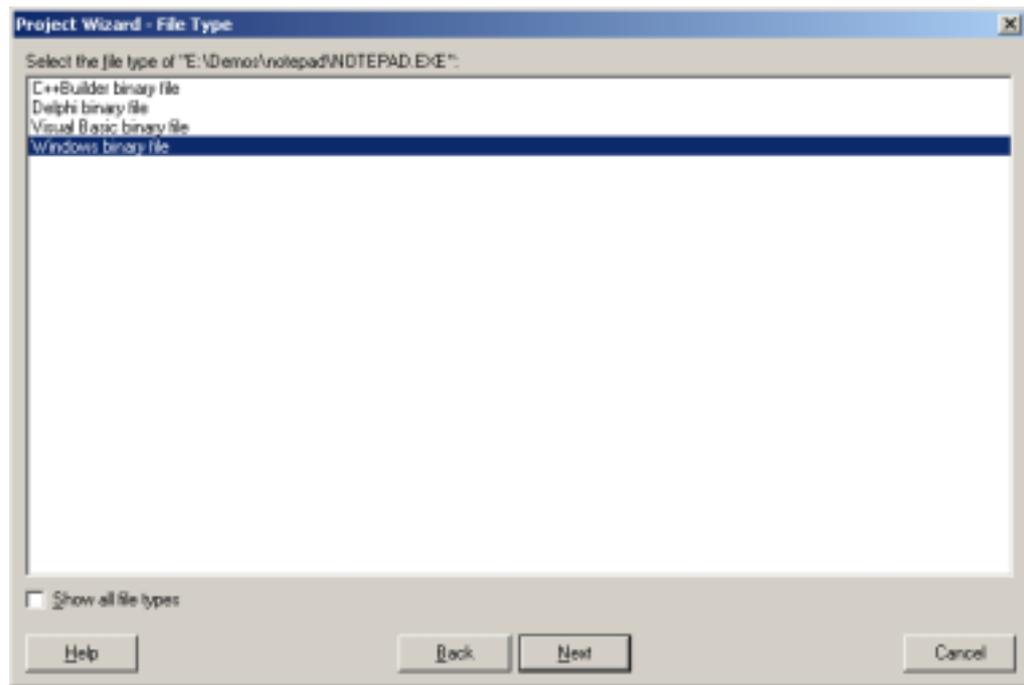


Figure 5: Specifying file type.

Target options

Depending on the selected target platform and type, the next screen allows the user to specify localization-dependent options for the target.



The available options for each target platform and type are discussed in the tutorials (See: Target Options in each tutorial).

Information

The next screen lets you enter information that identifies the project.

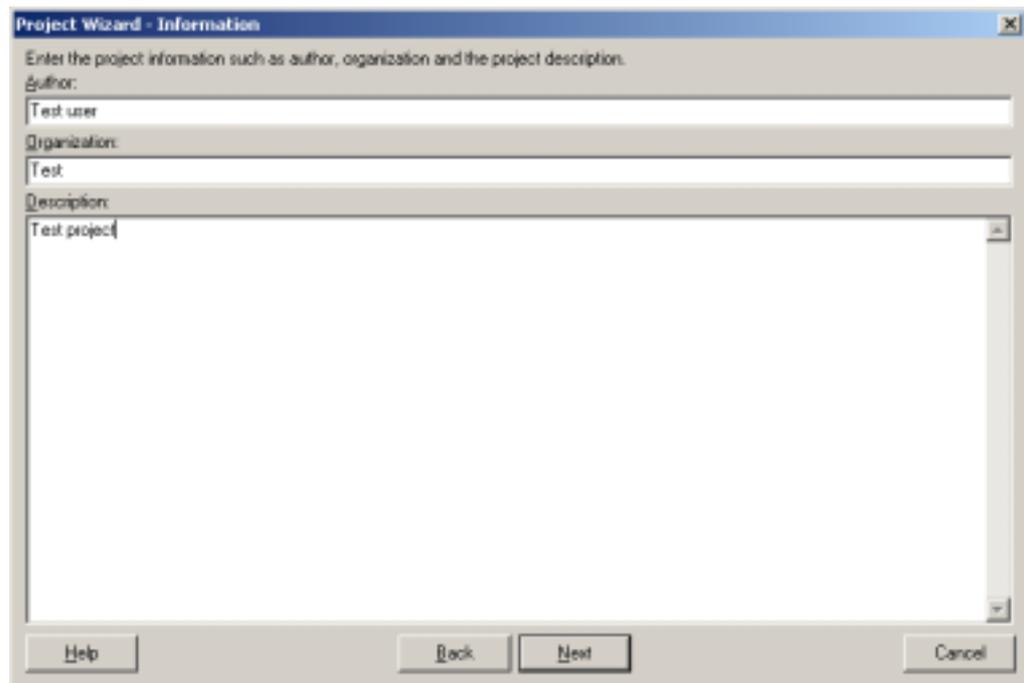


Figure 6: Specifying project information for new project.

Languages

On the next screen, you have to specify target languages for the project.

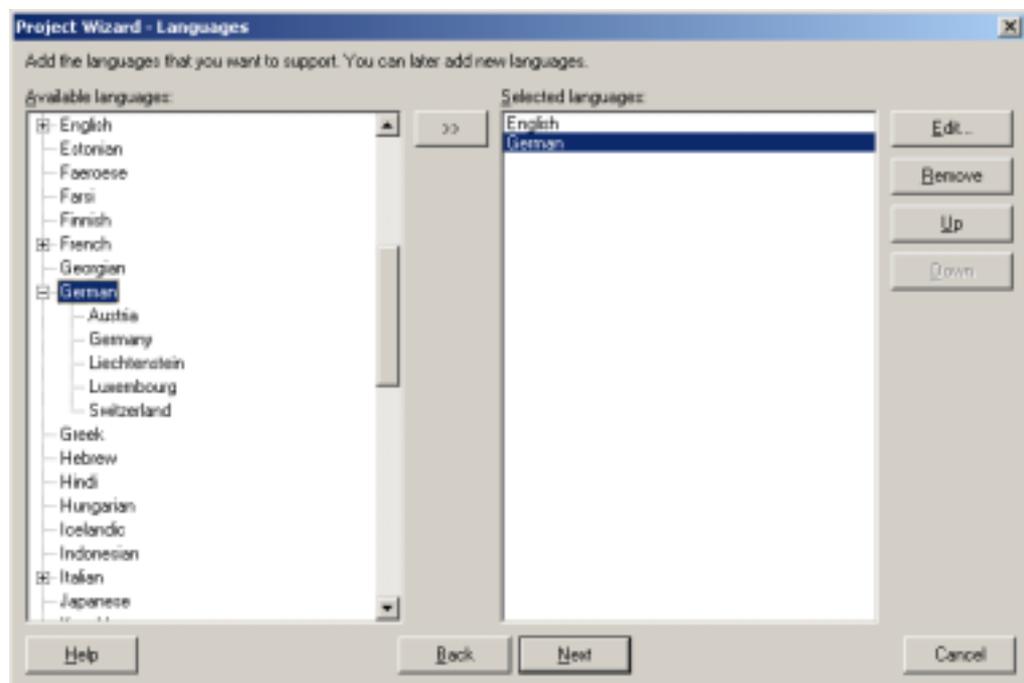


Figure 7: Selecting languages for project.

Drag and drop desired languages to the right-hand side to select them.



It is preferred to add languages instead of sublanguages (language + country). Sublanguages should be added only if there are country-dependent translations for the target language.



You can add default languages for new projects. Choose **Tools→Options→Environment→Default languages...** from the Multilizer main menu to specify languages that should be automatically selected in new projects.

If there are languages that are not included in the list, you should continue and finish the project. Then add custom languages (Tools→Languages and Locales...). After defining custom languages, you can add them in your project (Project→Languages...).

Finish

The last screen lets you go back and modify settings.

Finish button will create the Multilizer project, and scan the files to localize. Scanning will pick the data to localize from the files to localize and store them in the Multilizer project.

3

Project maintenance

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java
User's role in process:	Localization engineer, Localization project coordinator
Wizards:	Scan project Translation Memory

Introduction

There are three main tasks to do in maintaining the Multilizer project:

- Check if software/content to localize has changed; if it has, synchronize it with the project. (→Re-scan project, p. 25)
- Pre-translate project (→ p. 25)
- Prepare the project for translation. (→ p. 26)

Project View

When a Multilizer project is created or an existing project is opened, Multilizer shows it in Project View. All project maintenance tasks are done within Project View.

Project View consists of project tree, translation work-place, and Info page.

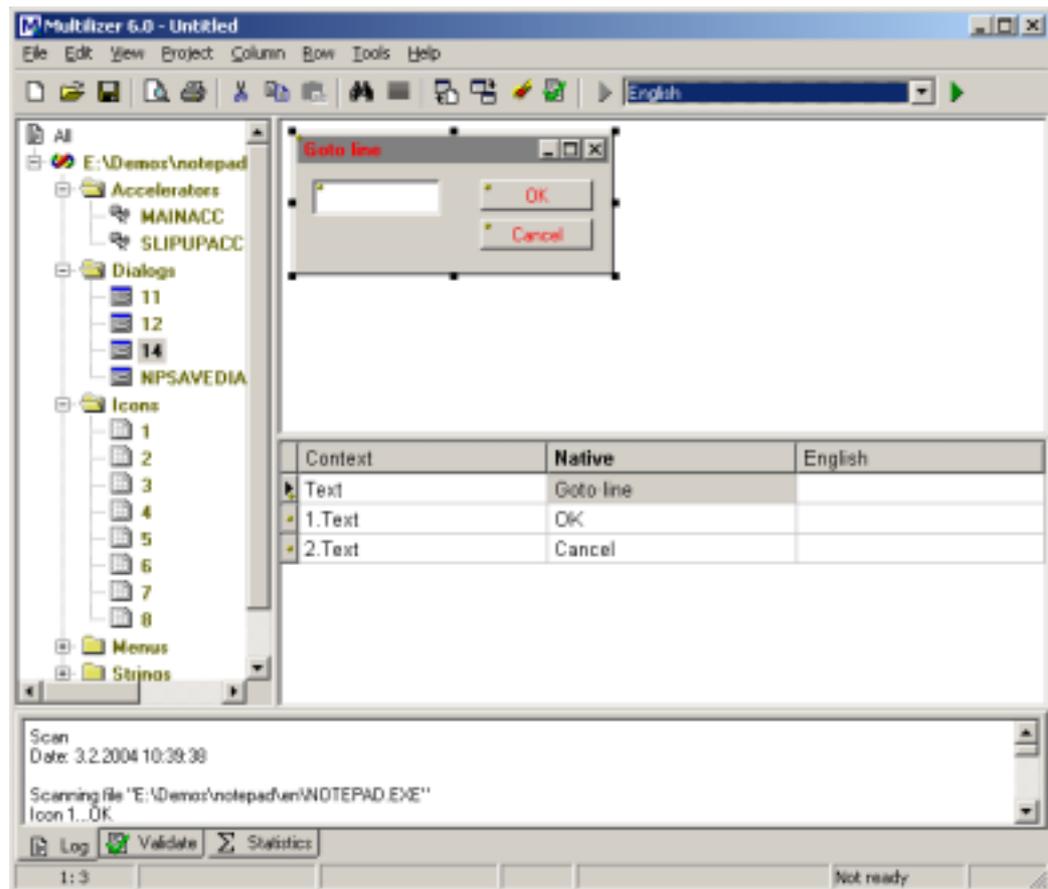


Figure 8: Multilizer project view, with project tree, translation work-place (translation grid and visual editor), and info page.

Project Maintenance tasks are done on three levels:

- **Project level**
Project maintenance is done from Project menu.
- **Target level**
Target level maintenance affects the way of localizing individual targets, such as output of localized files. Maintenance of targets is done either in Project Tree or by choosing Project→Targets... from the main menu (→ Project tree, p. 19). Further platform-dependent options that may affect localization of targets are found in **Tools→Options** in the respective platform settings.
- **Translation level**
Translation level maintenance is done in Translation work-place. Besides editing translations, strings can be locked, hidden, and much more. (→ Translation work-place, p. 22)

Project tree

All targets of the project are shown below the root node (*'all'*) of the project tree.

Translations of each target are grouped by resource type, file name, or other way, depending on the target platform and type. These groups are shown as nodes below each target.

New nodes are shown in **bold**. Nodes that were removed from software are shown in **blue**. (→ Re-scan project, p. 25).

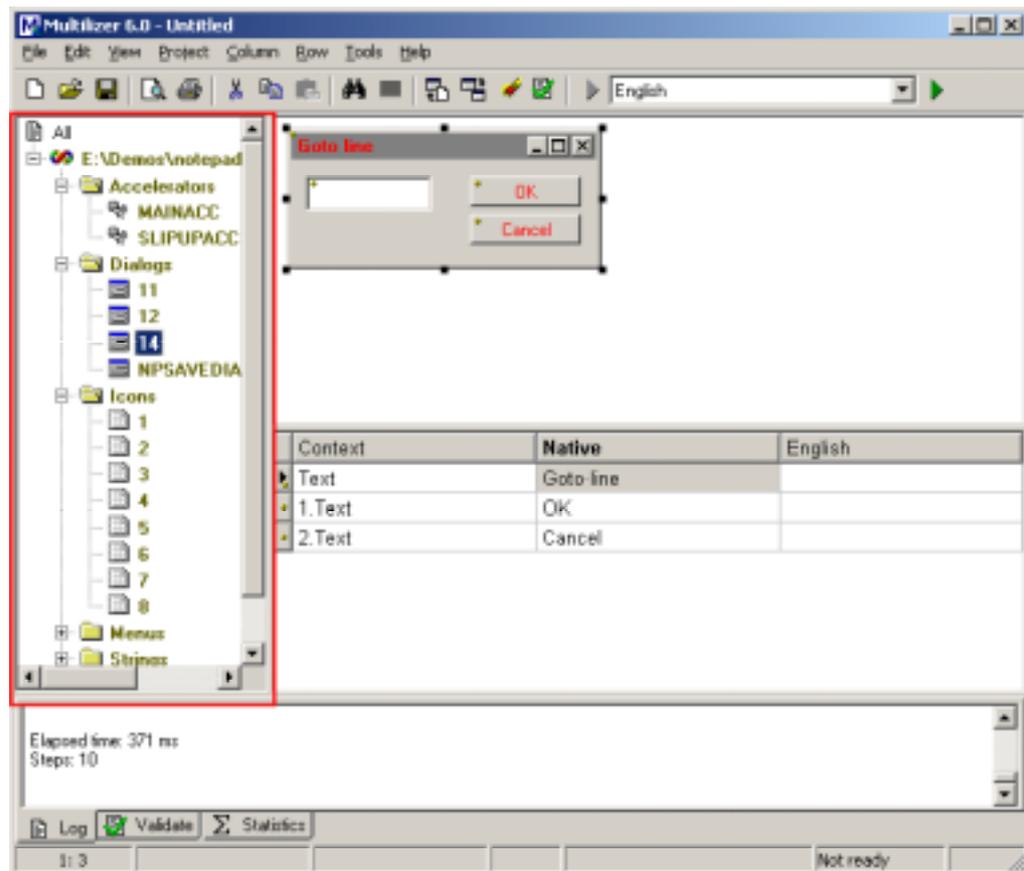


Figure 9: Project tree with mainform resource shown in bold.

By clicking any node, the translation view is automatically updated so that the corresponding content is shown in the translation grid.

By clicking nodes of a certain resource type, Multilizer shows both the translation grid and a WYSIWYG (what you see is what you get) view of the resource. Resources that are shown in WYSIWYG are:

- Dialog (form) resources
- Bitmap resources, including cursors and icons
- Menus
- Frames (VCL, Delphi & C++Builder)

In addition, Multilizer shows in Wysiwyg the following file types:

- XML
- Source code
- Bitmaps and source code embedded in XML.

Modifying targets

Targets can be modified, added, and removed from project tree. This is done either from a pop-up menu (right-click project tree to show) in project tree or from Project menu (Project→Targets...) in the menu bar.

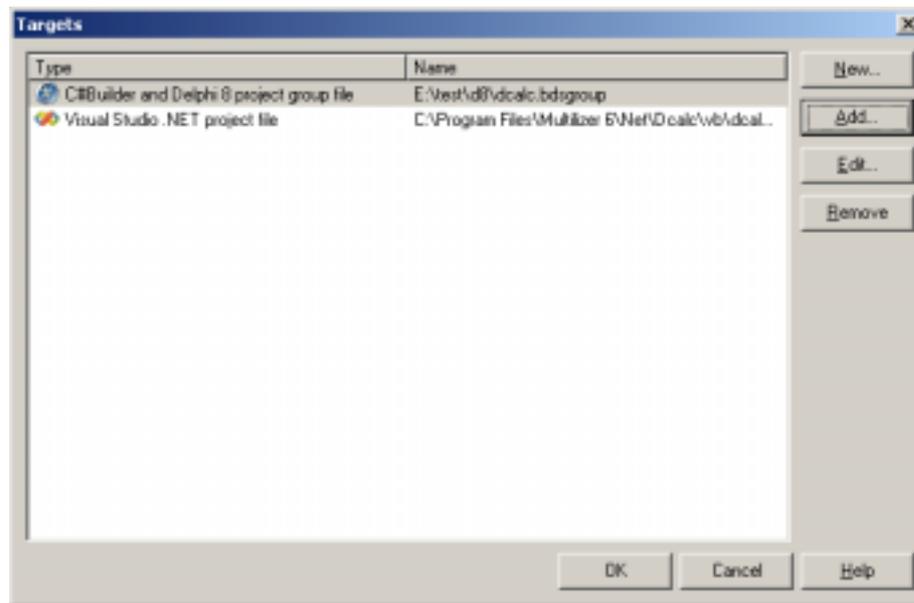


Figure 10: Dialog displaying project targets (*Project*→*Targets...*)

- Choosing **Add...** will add a new target using the Project Wizard (c.f. Project Wizard, p. 12).
- Choosing **New...** will add a new target by allowing the user to select target type from the list (→ See next picture) and specify the location of it.
- Choosing **Edit...** will show target-specific localization options. The settings of these are platform- and type-dependent, and they are discussed in the tutorials.
- Choosing **Remove...** will remove the target from the project. All strings associated to it will be in the project until you explicitly remove them by choosing **Project**→**Remove Unused Strings**.

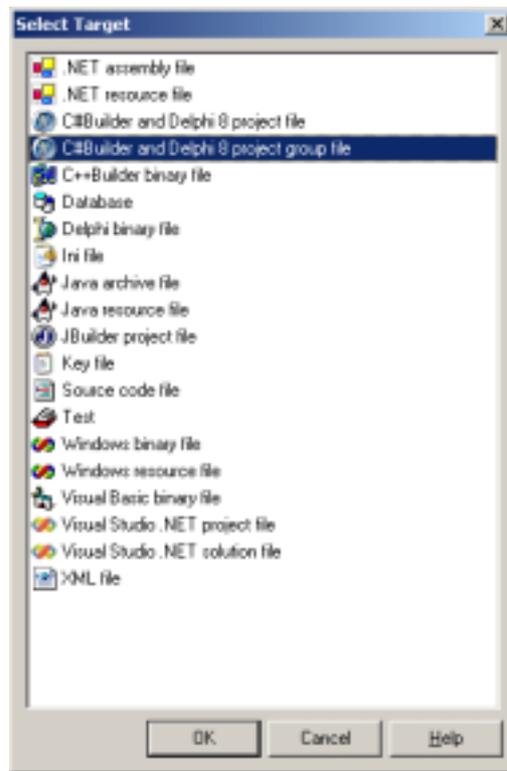


Figure 11: *Adding new target by file type.*

Translation work-place

Translation work-place shows the localizable information associated to the selected node in the project tree.

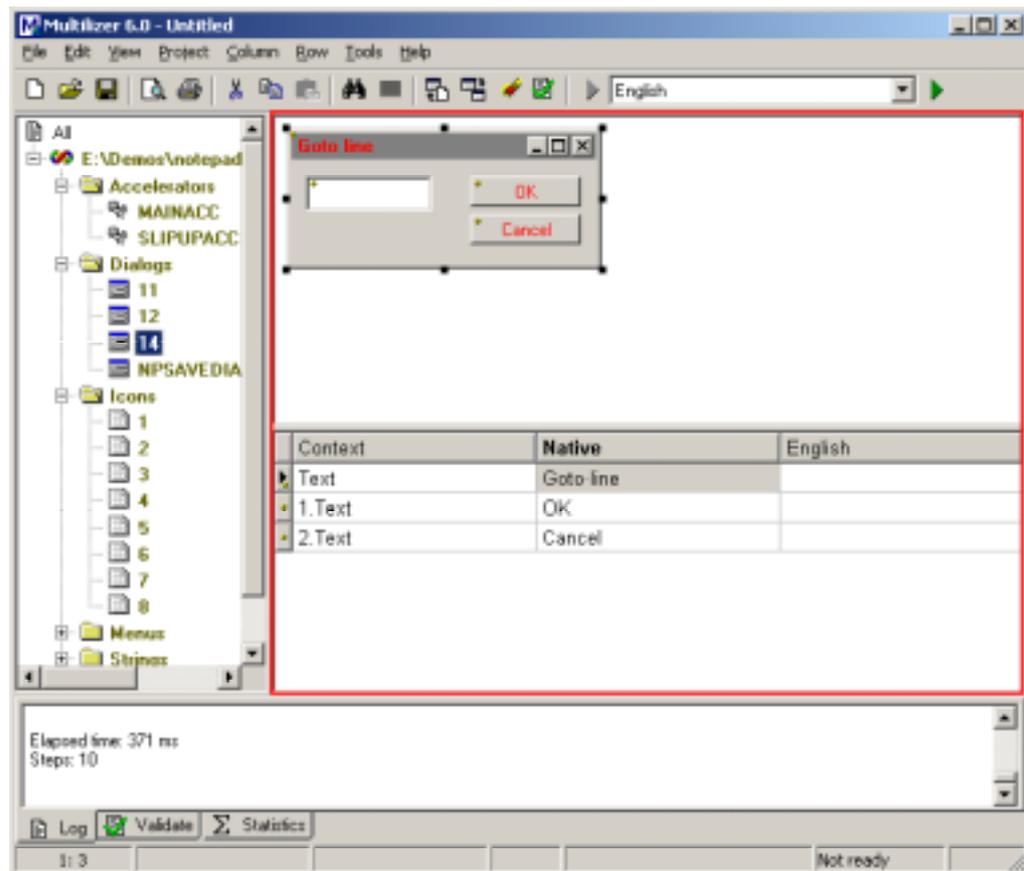


Figure 12: Translation Workplace; translation grid and visual editor.

Translation work-place shows a translation grid for editing properties of localizable data for translating. If a specific resource type is selected in project tree, Multilizer also shows them in WYSIWYG (→ Glossary, p. 176) mode above the translation grid.

Translation grid can contain one or more target languages; language visibility is toggled from **View** menu. One language at a time can be edited; this active language can be set either from toolbar's language drop-down list, or from **View→Edit language...** menu.

Translation work-place is introduced in detail in the chapter 'Translation work-place,' p. 44.

Info page

Info page shows log information of processing the Multilizer project.

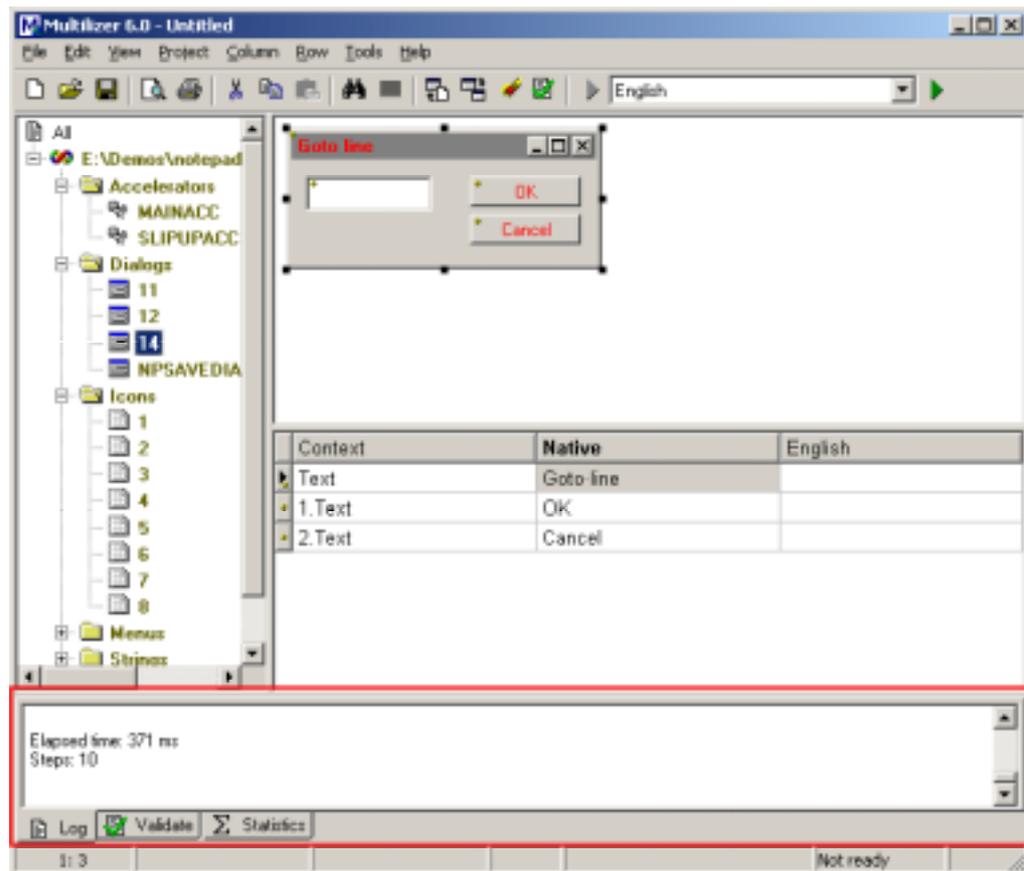


Figure 13: Info Page.

There are three tabs on the Info page:

- Log shows progress of Scan, Make, and Build processes.
- Validate shows the results of validation process when run by the user.
- Statistics shows statistics of the project upon the user's request.

Visibility of info page can be toggled from **View** menu.

Log

Log displays information of scan, make, and build processes. It may display warnings or errors, if there were any issues in any of aforementioned processes.

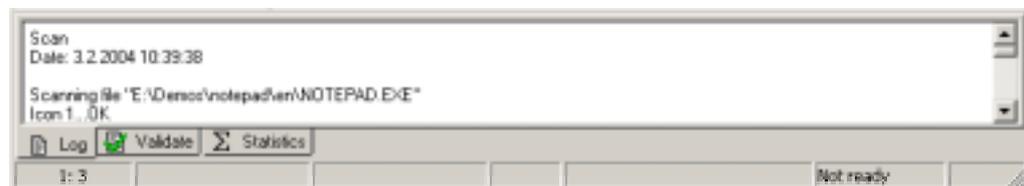


Figure 14: Log view shows information of scan, make, and build processes.

Validation Log

Besides showing the QA issues, navigating in the validation log automatically navigates in the translation grid and focuses the control in the wysiwyg view. This both shows the context better, and simplifies correction of the issue. (→ Validation Wizard, p. 64)

Statistics

Statistics panel displays short information of translation status of native language and currently selected (active) target language.

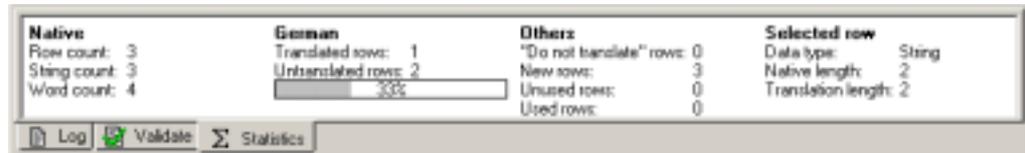


Figure 15: Statistics panel with quick info of translations.

Re-scan project

The idea of re-scanning the project is to check if anything has changed in the localization source (native software or content). Any changes are automatically synchronized with the project contents.

Categories

Synchronization changes row statuses, if any changes are found when re-scanning; the following rules are applied:

- Strings that are not found in the localization source become 'unused strings.'
- New strings get 'new string' flag, and the string is shown in native column.
- If the string of localization source has changed, the old string in the project is marked as 'unused' and the new string is added as 'new string' in the project.

In order to review strings of any of the abovementioned category, the corresponding filter must be applied. (→ Filtering, p. 26)

Pre-translate project

Multilizer is able to use existing terminology to translate the project. There are two ways of doing this.

1. Translate using Multilizer Translation Memory.
2. Import translations from the file.

Translate using Translation Memory

There are two ways of translating project using Translation Memory:

- **Project→Translate→Using Translation Memory...** will translate all languages of the project.
- Right-clicking language column header and choosing **Translate→Using Translation Memory...** translates to current target language.

In both cases Multilizer will look up for translations in default Translation Memory; Multilizer finds matches for Multilizer project's native string, and populates Multilizer project with corresponding translations. In case of several translations user can decide which one to use.



Installation and maintenance of Translation Memory is discussed in the chapter Translation Memory, p. 56.

Import translations from files and databases

Import Wizard can be used for directly importing translations from files and databases supported by Multilizer.

This way of importing translations bypasses Multilizer Translation Memory, and fuzzy matches are not supported. In order to support fuzzy-matching of translations, the files should be imported in Multilizer Translation Memory, and the project should be translated using Translation Memory.



To read more about importing files, databases, and importing translations from other vendors' products (TRADOS, Déjà Vu, SDLX, etc.), c.f.,

Import Wizard, p. 37 .

Prepare project for translation

Before strings and accompanying information can be sent off to the translator (→ Share translation work, p. 27), it is useful to prepare the project for translation.

Filtering

Filtering enables user to show localizable data meeting a specified criteria (translation status, row status, data type). Choose **View→Filter...** to specify the rows to be shown.

For more info on filtering options, c.f., Row filtering, p. 46.



Pseudo languages → QA

Software/content can be localized before translation by using pseudo languages; this enables testing of the software/content localization before any translation work is done. (→ Pseudo Languages, p. 68)

Lock Visual Editors

Sometimes translators shouldn't have the possibility to edit size and location of visual elements. To prevent this from happening, you can make dialogs read-only. Choose from the dialog editor's context menu 'options,' and check 'Read only.'

Marking dialogs read-only allows for translation, but no modifications on dialog size and location can be done.

Translation control

In order to control translations, strings (rows) can be removed manually from the project.

Strings can also be hidden by applying the filter. Filtering shows strings depending on their status.

In order to show strings to the translator but not allowing for translation, strings can be locked.

In addition to the aforementioned features that prevent translation, there is the possibility to limit translation length. Length can be limited to a certain character count or to a certain pixel count.

4

Share translation work

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java
User's role in process:	Localization engineer, Localization project coordinator
Wizards:	Exchange Wizard Export Wizard Import Wizard

Introduction

Multilizer includes built-in support for teamwork. It allows any Multilizer project to be split and shared between team members. This is useful when sharing translation work. There are three powerful Wizards that help in this:

- Exchange Wizard
- Export Wizard
- Import Wizard



In order to streamline the workflow and avoid conversion errors, translations and project information should always be sent in Multilizer format. This is done by using Exchange Wizard for sending off strings for translation, and importing translated strings with Import Wizard.

Default workflow

In default workflow, translation work is shared between software developers/localization engineers and translators. The workflow below shows the tasks in the context of the Multilizer localization process.

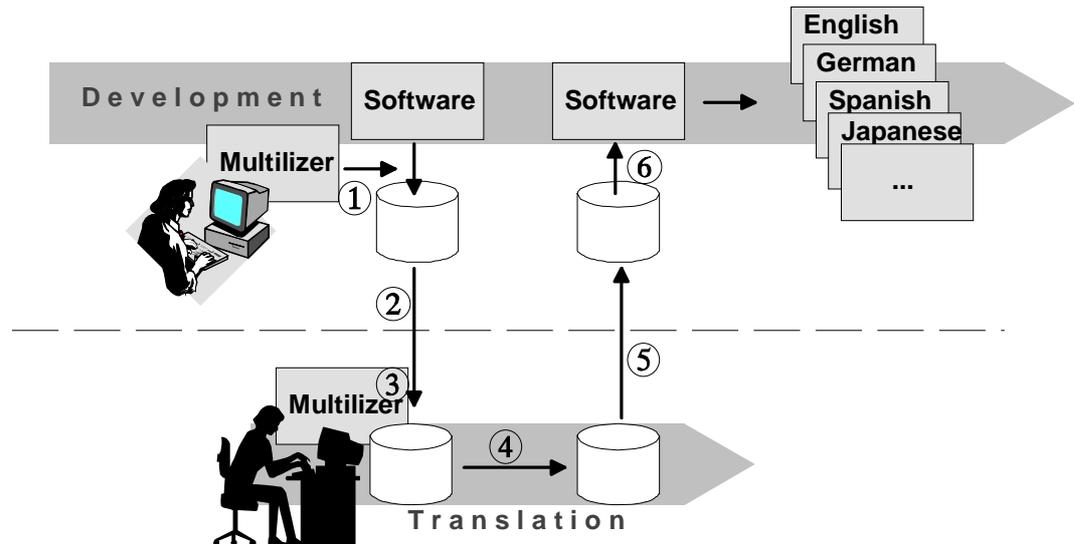


Figure 16: Default work-flow with Multilizer.

① Developer/localization engineer starts a software localization project by scanning the software and content with Multilizer, and creating a Multilizer project file. ② He uses **Exchange Wizard** (→ p. 28) to create a Localization Kit that includes the translation tool and the texts to be translated. The Kit is sent to the translator.

③ The translator opens the Localization Kit. It automatically installs Multilizer Translator Edition and the sub-project with texts to be translated on the translator's computer. ④ The translator uses Multilizer to translate the sub-project. ⑤ When finished, the sub-project is sent back.

⑥ Developer/localization engineer uses **Import Wizard** (→ p. 37) to integrate the translations in the project file.

Using Multilizer throughout has preserved the technical context of the translations, so it's very simple to create localized versions of the software.

Exchange Wizard

Exchange Wizard enables sharing of translation work in an efficient and safe way. Because all linguistic data is sent in the same MPR-format (Unicode®), there is no risk for data loss inherent to format conversions or character set incompatibilities.

Exchange Wizard is used to create a Localization Kit that is sent to the translator. Import Wizard (→ p. 37) is used to import translations back.

Creating Localization Kit

Exchange Wizard creates a Localization Kit that includes the following items:

- Sub-project (always)
- Multilizer Translator Edition (optional)
- User-specified files (optional)

Exchange Wizard assists in creating a sub-project that contains a sub-set of project languages and targets. In addition, filtering can be applied to further control which strings are added in a sub-project.

Multilizer Translator Edition can be added on user-request in the Localization Kit. Multilizer Translator Edition is a freely distributable version of Multilizer, aimed for project translation.

Exchange Wizard lets users also add any additional files in the Localization Kit.

Multilizer compiles the Localization Kit either to a self-installing setup (if Multilizer Translator Edition is included), or a compressed Multilizer package file (MLP). Opening the MLP-file in Multilizer will extract the sub-project and user-specified files in the same directory as the MLP.



In order to ensure the best possible compatibility, ensure that the translator has the same Multilizer build-number as the Multilizer copy used to create the Localization Kit. If build-numbers differ, include Multilizer Translator Edition in the Localization Kit.



Unlike other localization products, users of Multilizer Translator Edition don't need any additional SDK's or libraries (such as, .NET run-time or Symbian SDK, for example) to translate the project. So in order to share the translation work, no other files besides the Localization Kit need to be sent. This minimizes project coordination overhead.

Exchange Wizard steps

Exchange Wizard simplifies sharing of translation work; it creates an Exchange Package (Localization Kit) that can be sent off for translation. Wizard is started by choosing **File**→**Exchange....**

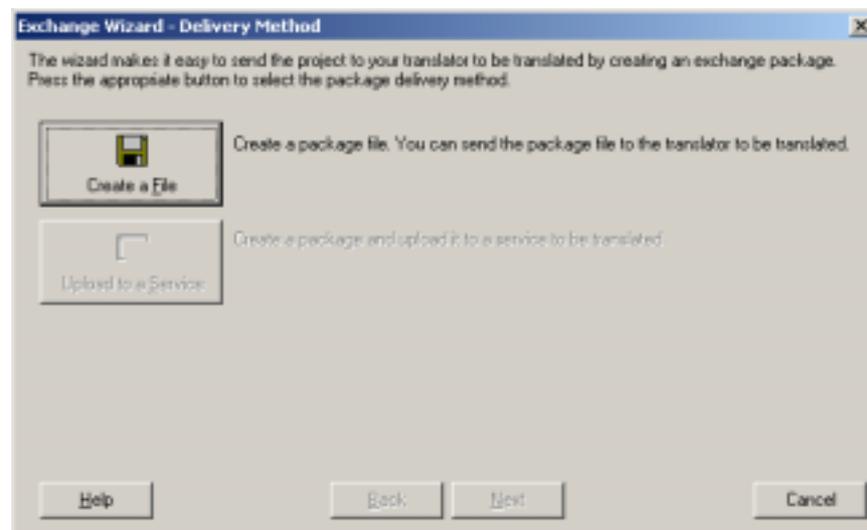


Figure 17: Running Exchange Wizard.

Pressing **Create a File** button take the user to next page.

Languages Sheet

On this page user defines the languages and Items (Project tree nodes) to be included in the package. By default all languages and all items selected.

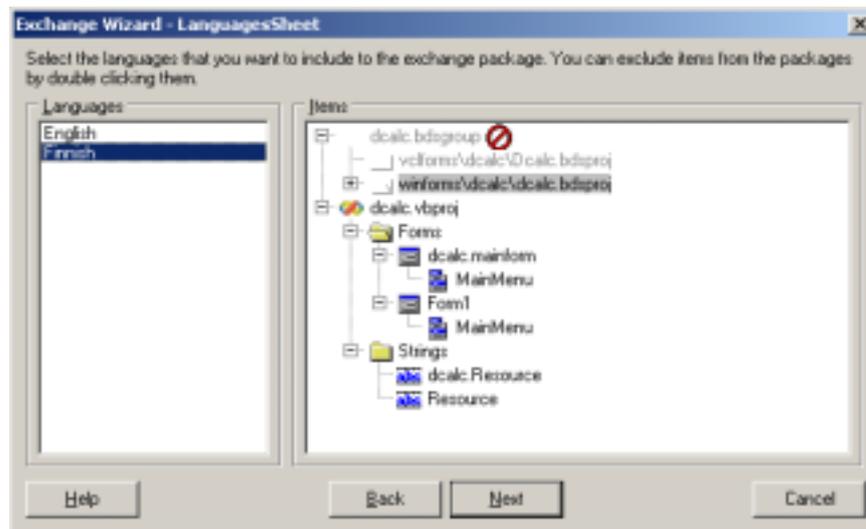


Figure 18: Specifying the targets and resources to be exchanged.

Splitting the project by items makes it possible that multiple translators work on the same language.

Options

On Options page user specifies, which strings are exported.

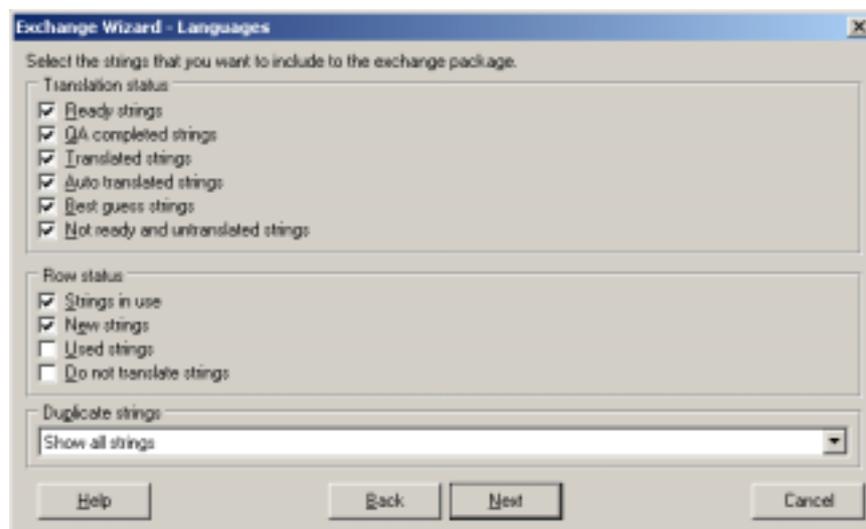


Figure 19: Filtering rows that are exchanged.

Translation status, Row status, and Duplicate strings filter strings in the same way as the filter in string grid (→ Row filtering, p. 46).

Information

Information page lets user specify basic information of the package. In addition package can be password-protected in order to secure its contents.

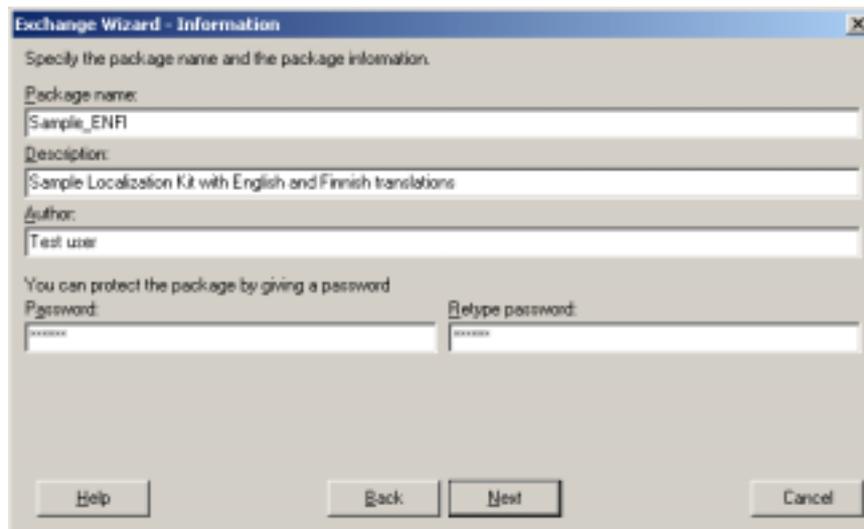


Figure 20: Specifying project info, password protection.

Application

Application page is lets user decide whether to send Multilizer Translator Edition in the package or not. Translator Edition is free, and if translator doesn't have Multilizer, the package should be sent along with it.

Including Multilizer application in package with increase its size with ~3 MB.

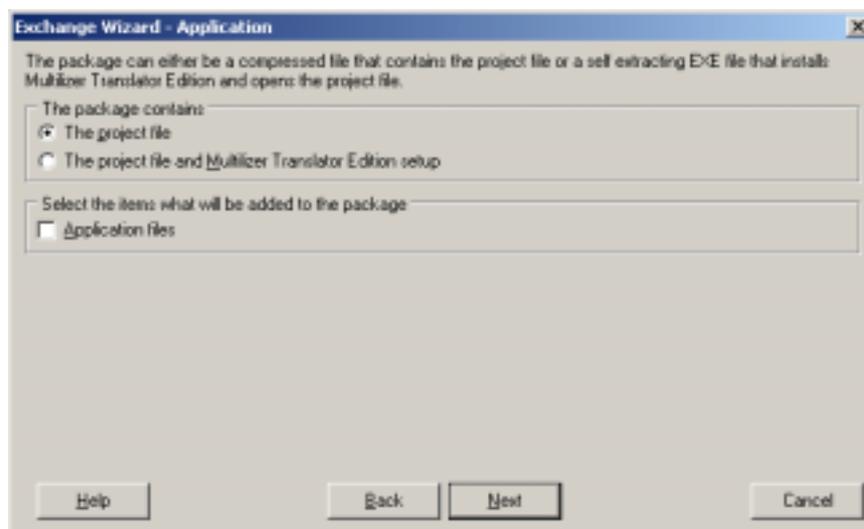


Figure 21: Adding targets in exchange package.

Check 'Application files' to include localization targets in package.

Include Files

This page allows user to include any number of additional files in package. For example, documentation can be included here.

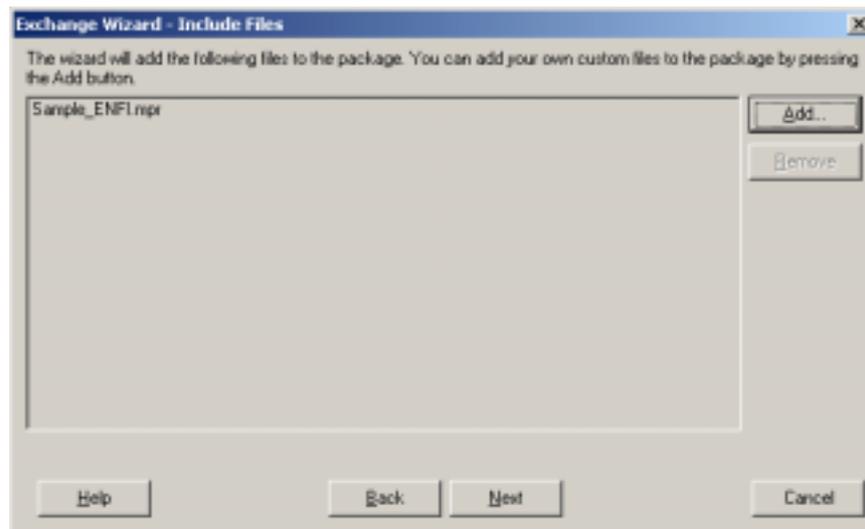


Figure 22: Specifying additional files to be included in exchange package.

File

On this page the filename of package is defined.

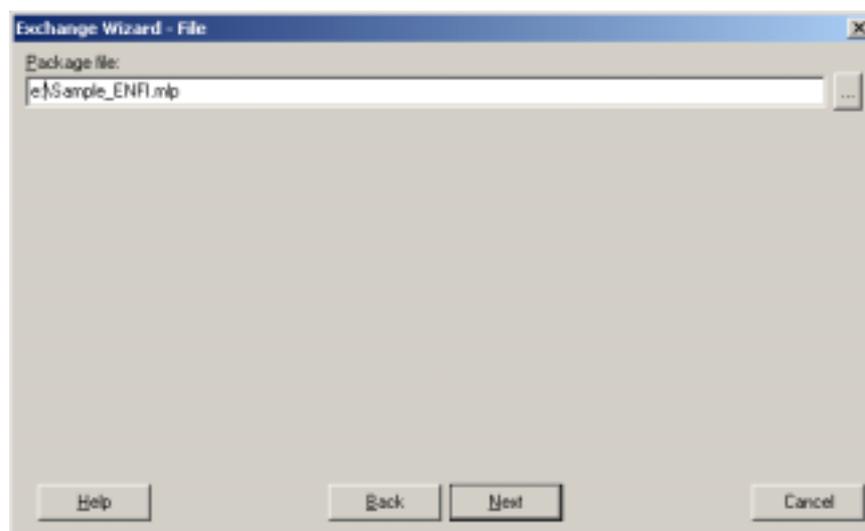


Figure 23: Specifying the name for exchange package.

File-extension is MLP (Multilizer package), if package doesn't contain Multilizer Translator Edition.

If Multilizer Translator Edition is included, then package will be a Windows executable. Running the executable will install Multilizer Translator Edition, and open the Multilizer sub-project contained in the package.

Pressing Next button will take user to the next page, and create the package.

Package File

This page shows the process of creating the package. If no issues are encountered then the Wizard will instruct user to send the package to the translator.

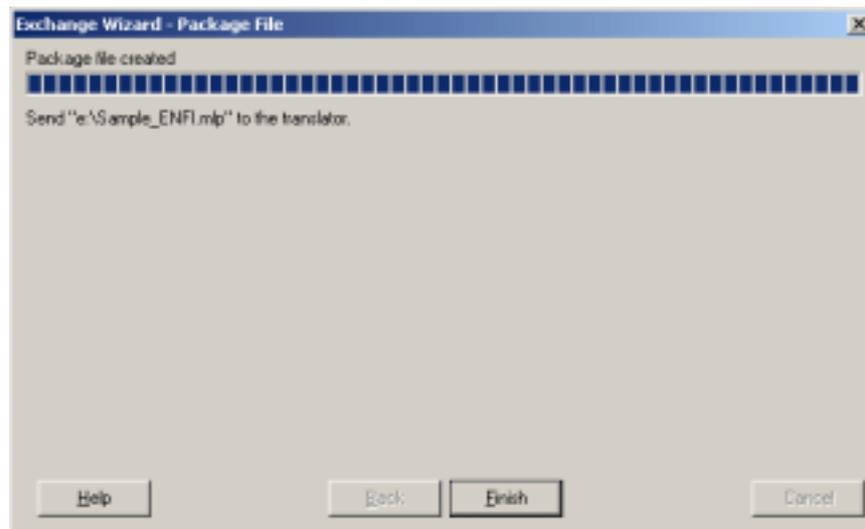


Figure 24: Building the exchange package.

Export Wizard

Export Wizard exports project strings either into TMX or text file. The main idea of using Export Wizard is to exchange translations and terminology – and not localization information – with other products.

Export Wizard enables sharing of translations with other products, such as TRADOS for example.



Although Export Wizard can be used for sharing translation work, it should not be used for that purpose. Export always requires format conversions, which may cause loss of data. Exchange Wizard should be used instead.

Export Wizard steps

Export Wizard simplifies exporting of Multilizer Project (MPR) information to other file formats. The Wizard is started by choosing **File→Export...**

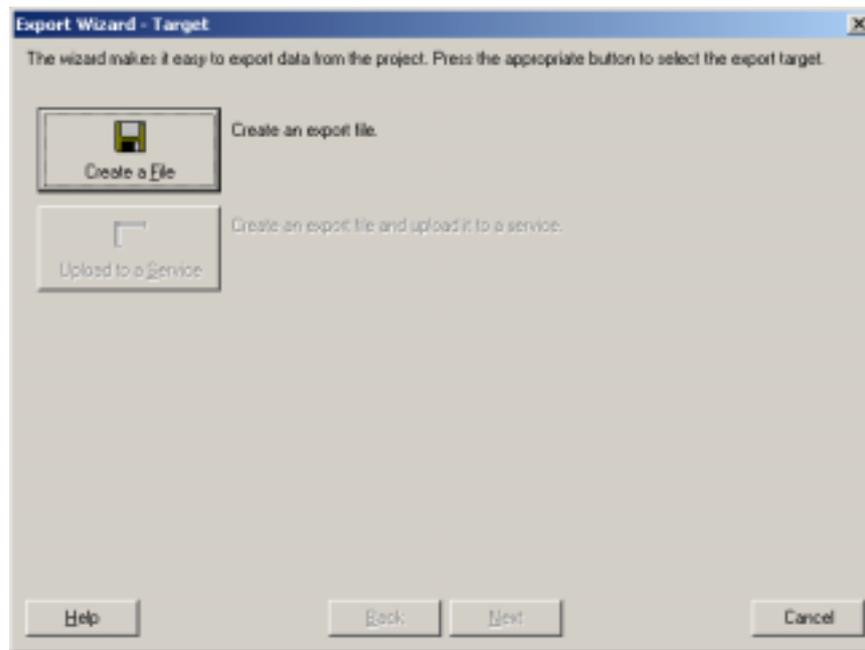


Figure 25: Running Export Wizard.

Pressing **Create a File** button takes to next page in Wizard.

File

File-page is the part of Wizard, where the output format is specified. Export Wizard supports TMX (Continue here → 34), and TXT (Continue here → 35) files.

Translation Memory Exchange file (TMX)

TMX-file export makes it possible to export Multilizer project translations to a format that is supported by a wide range of Translation Memory products.

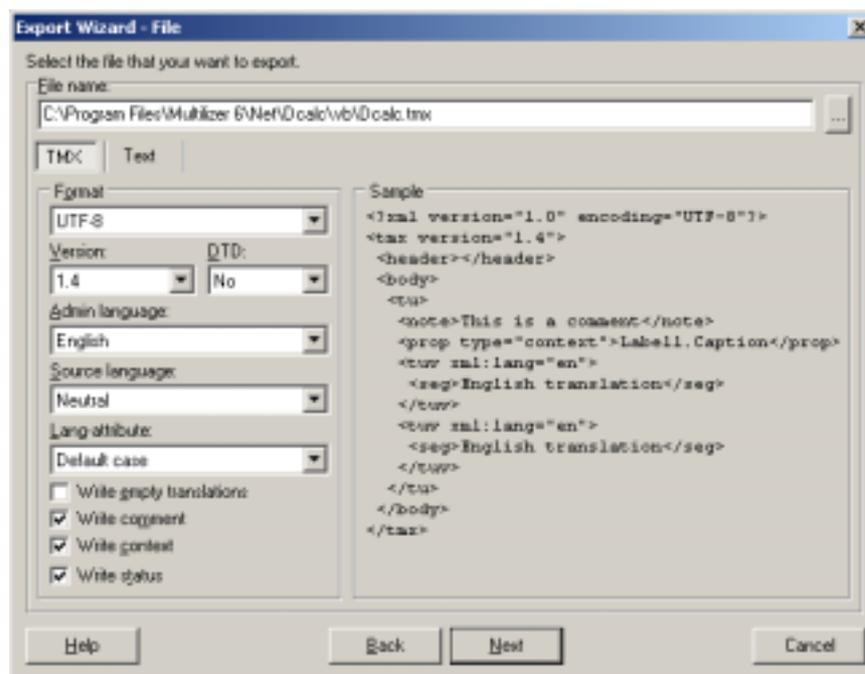


Figure 26: Specifying export file and TMX file format.

There are many variations – and vendor-specific implementations – of TMX. In order to support the original idea of exchanging translations, Multilizer allows users to specify the most important TMX-file format parameters. These are explained in Multilizer help file.

TRADOS. Following screenshot shows the configuration that create a TMX-file that is compatible with TRADOS® Translator's Workbench™: 2.3, 3.0, 5.0.1 (Build 217 or newer), and 5.5 (build 247 or newer)

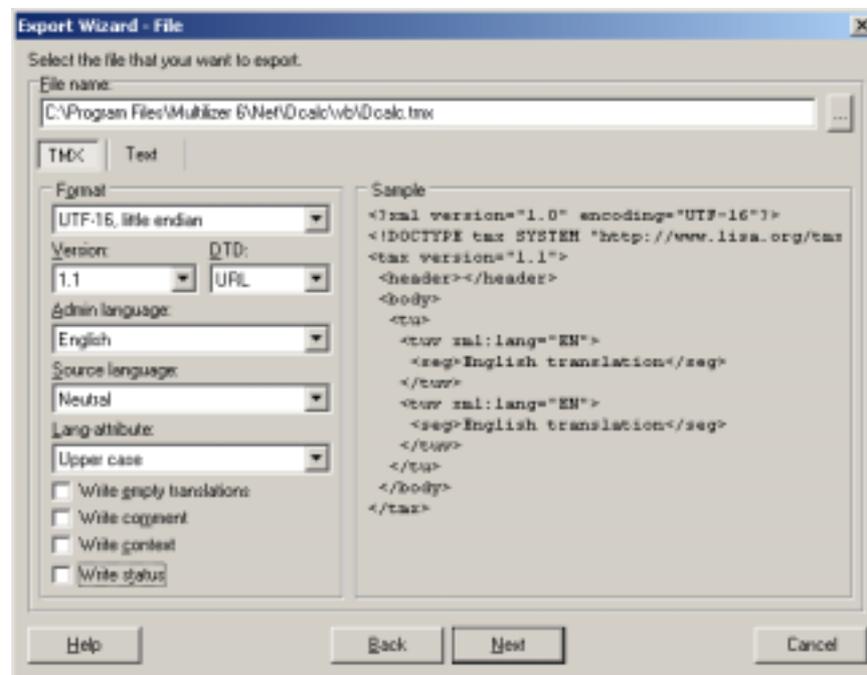


Figure 27: Export settings for TRADOS-compatible TMX.



In TRADOS 5.0 and 5.5 'Source language' must match with the 'Native' language in Multilizer, otherwise TRADOS will not be able to import the resulting file. TRADOS 2.3 and 3.0 require 'Neutral' as Source language.

Text file (TXT)

Exporting to text-file is the most generic way of exchanging translation data. Text-file export creates plain text files, where each string with corresponding translations are written to one line.

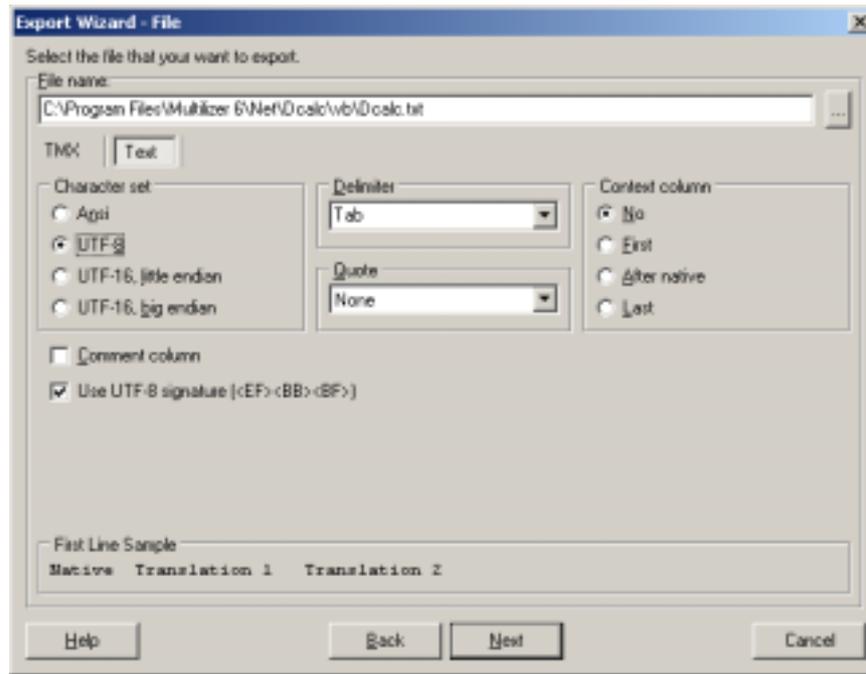


Figure 28: Specifying export file and text file format.

Export to text-file is very suitable for exchanging translations, because most products support importing of it. Highest compatibility is achieved by choosing settings as shown in picture above, and by exporting translations to one language only.

SDLX. The settings above create a text-file that can be imported in SDLX (version 4.0) as such. User needs to import in SDLX 'delimited files' and specify source and target language correctly.

Options

After specifying the file format, Next button takes user to the Options page. On this page user specifies, which strings are exported.

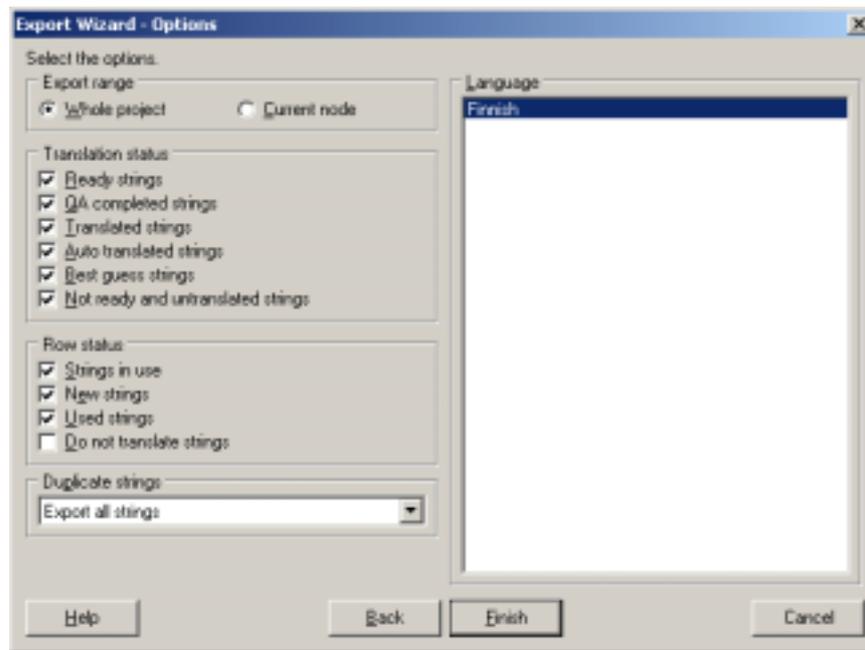


Figure 29: Filtering of rows that are exported.

Export Range can be either the entire project or the current node. Current node corresponds to the strings shown in the translation grid when Export Wizard was started.

Translation status, Row status, and Duplicate strings filter strings in the same way as the filter in string grid (→ Row filtering, p. 46).

Exporting of TXT-files and TMX-files allows users to choose any combination of languages to be written in the file.

Import Wizard

Import Wizard is used to import translations from Multilizer project (MPR) files, and from other formats.

Import Multilizer Project (MPR)

There are two reasons for importing Multilizer Projects.

1. It provides the means for leveraging translations from other Multilizer projects. This is useful when creating a new Multilizer project and translations should be the same as in existing projects. In order to ensure consistent terminology, Translation Memory should be used.
2. When sharing a project with Exchange Wizard (→ Exchange Wizard, p. 28), translations of sub-project(s) are imported back with Import Wizard.

Because the Multilizer project (MPR) is Multilizer propriety format, it is the safest way of exchanging translation data.

When using Exchange Wizard and Import Wizard together, correct import options play a big role in the localization process. Typical import options for a Multilizer project are discussed later in this chapter (→ Import Options, p. 39).

Import from other formats

Multilizer also supports 3rd-party formats for importing translations. This enables translations of other systems (e.g., TRADOS, SDLX, Déjà Vu, etc.) to be imported and used in Multilizer.



If importing of translation data fails, check out the documentation of the product that created the file. Also try out importing with a different set of import options. Because Import Wizard relies on formats (and not on 3rd-party products), Multilizer can't guarantee compatibility with any particular product.

Import Wizard steps

Multilizer imports both files and database content. When starting Import Wizard, the user has to choose the translation source.

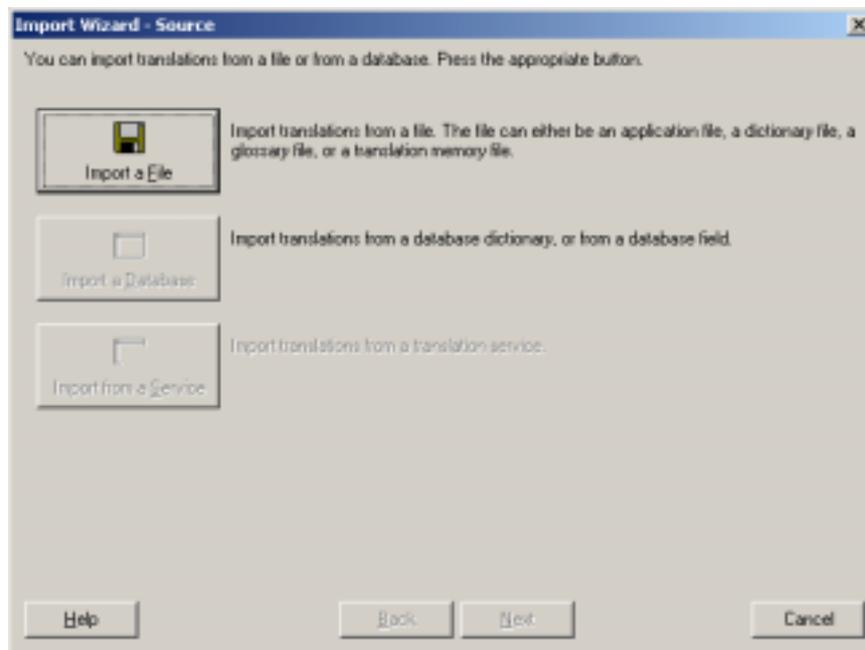


Figure 30: Running Import Wizard.

File import

File import requires that the user specifies the location of the file that is supported by Multilizer.

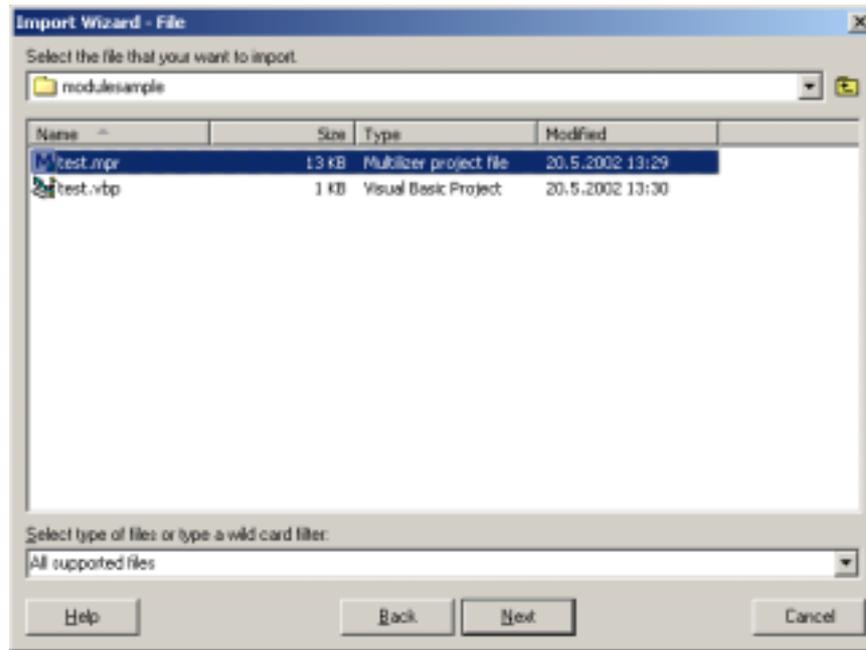


Figure 31: Specifying the file to be imported.

The user can filter the files shown in the file list by choosing the appropriate file format from the drop-down list.

If the user chooses 'select file types...' Multilizer opens the Select File Type assistant that allows the user to specify platform and file type. Upon closing this dialog, Multilizer will choose the correct filter.

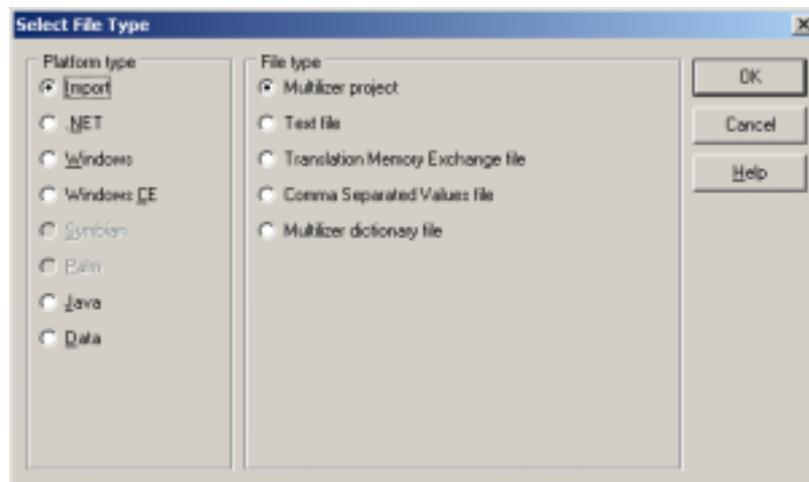


Figure 32: Selecting file type for the file to be imported.

Import Options

Import Options is the most important part of the Import Wizard. The options specified here affect the way that translations are imported.

For importing a Multilizer Project file, the options are as shown in the picture below.

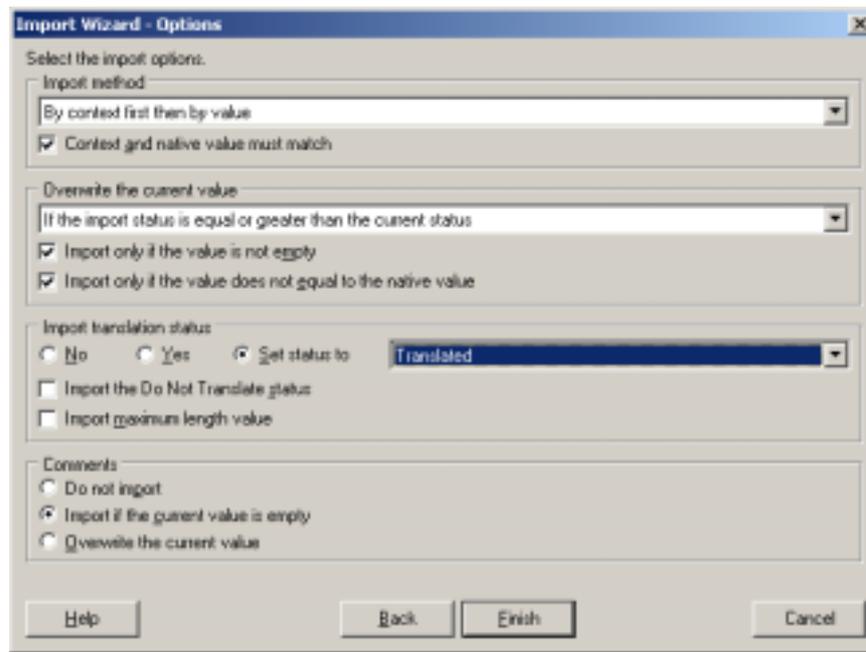


Figure 33: Specifying Import options.

Import method tells how Multilizer performs the look-up of the native string. The user can specify to match the Native column string only, or the user can force Multilizer to do context-specific matching. This means that translations are imported in correct context. This is the recommended way of importing Multilizer projects that were sent off with Exchange Wizard.

Overwrite options specify the rules for importing a translation if Native look-up is successful. In order to ensure the quality, the user should never allow Import Wizard to import translations with lower (quality) status than those already in the project.

Translation status should be imported as such from (Setting: Yes) the project.



Import options are highly dependent on the file format of an imported file. Import options for typical import file formats are discussed in the next chapter.

Options for typical file imports

Typically, the following file formats are imported with Import Wizard:

- Multilizer Project (MPR)
- Text file (TXT)
- Translation Memory Exchange file (TMX)
- Comma Separated Values file (CSV)
- Multilizer dictionary file

All of the abovementioned formats serve a specific purpose, and therefore special attention should be paid on the import options.

Multilizer Project (MPR)

Import settings for MPR are discussed in Import Wizard steps, p. 38.

Translation Memory Exchange file (TMX)

Translation Memory Exchange File format is used for exchanging data between Translation Memory products.

When opening a TMX file for import, Multilizer first detects the languages included in the file.

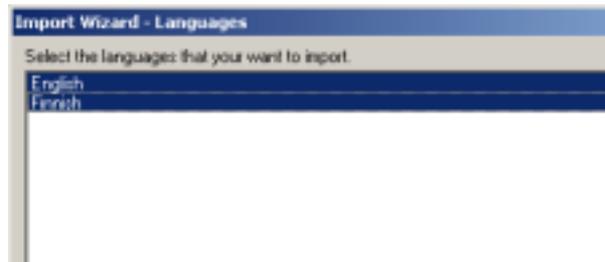


Figure 34: Specifying languages to be imported from TMX file.

After specifying languages, the user can decide the options for importing translation data.

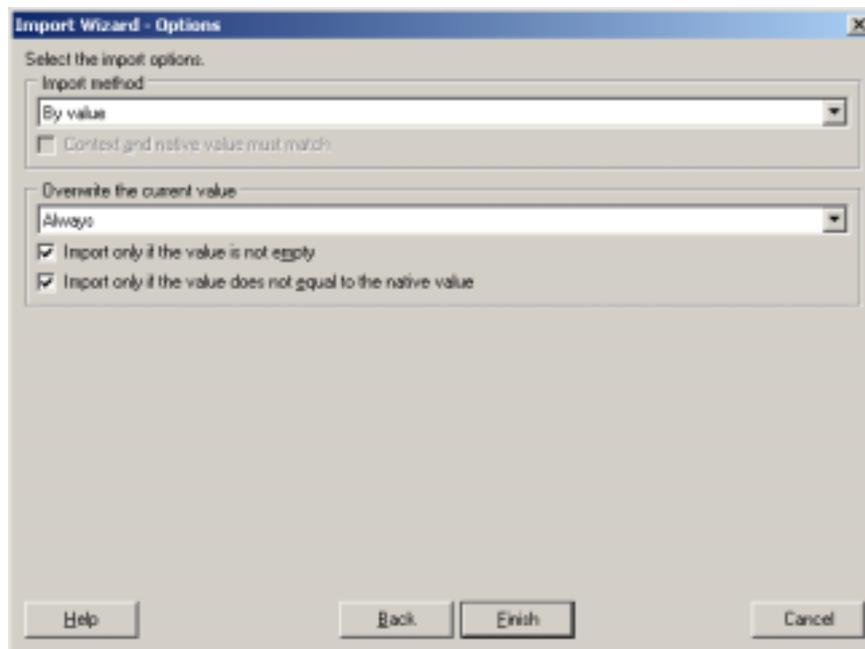


Figure 35: Import options for TMX.

Import method specifies the look-up method for matching Multilizer project's Native column with source language in the TMX.

For example, 'By value' means that the string found in Multilizer project's Native column is matched with the source language segment contents of the TMX file.

Overwrite settings specify when a translation should be imported.



NOTE!

TMX format is based on XML, thus implying many vendor-specific variations. In addition, there are many versions of this format. Therefore, finding of correct import/export settings for this format can be time-consuming. If you use this format a lot, we recommend subscribing to Multilizer SUMA (→ Multilizer Support and Maintenance, p. 9), which includes certain amount of free consulting. Multilizer consults have years of experience on working with localization and human languages-specific file formats.

Text file (TXT)

Text file import work with files where each translation record is on one line in the text file. Each line should contain at least a source term and target term separated by a delimiter specified by the user. This filter imports CSV-files as well.

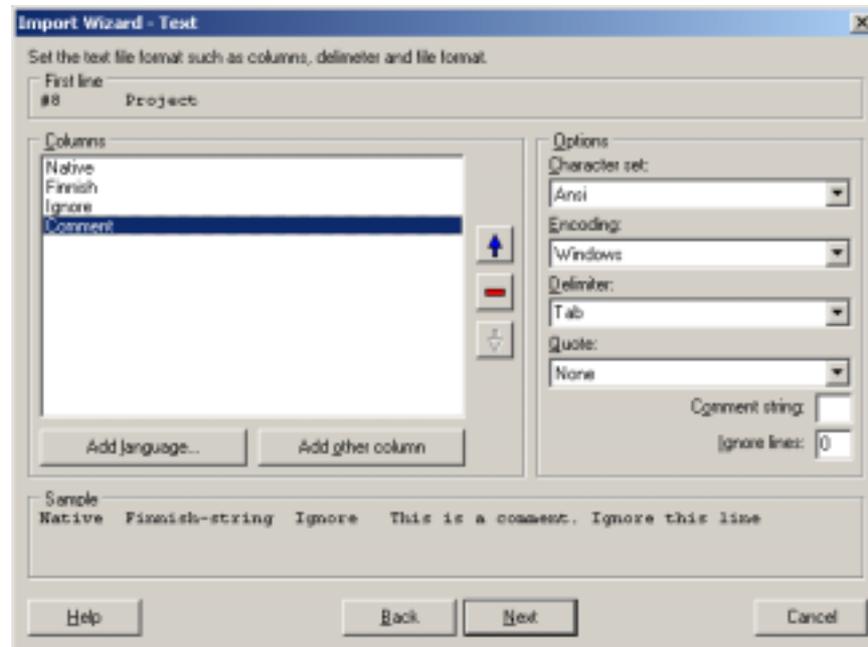


Figure 36: Specifying file format for importing text file.

For example, in the picture above, import filter is configured to import translations from a text-file where each line contains four columns. Columns are separated with TAB character. The Native column corresponds to native column in the open Multilizer project. The Finnish column should include Finnish translations in the imported text file. The 3rd column is ignored, and the 4th column includes a comment.

Upon completing the Wizard, Multilizer will match Native columns of the file and open the Multilizer project. If there are matches, Finnish translations will be imported from the text file. If there's a comment in the text file, it will be imported as well.

Comma Separated Values file (CSV)

Multilizer CSV support is intended for importing Microsoft glossaries, either as translations to current project or to Multilizer Translation Memory.

In order to import other CSV files, use TXT-import instead; it includes all configurability required to CSV-import. (→ Text file (TXT), p. 42)

When using CSV-import to import Microsoft glossaries, Multilizer attempts to detect the language of the selected glossary.

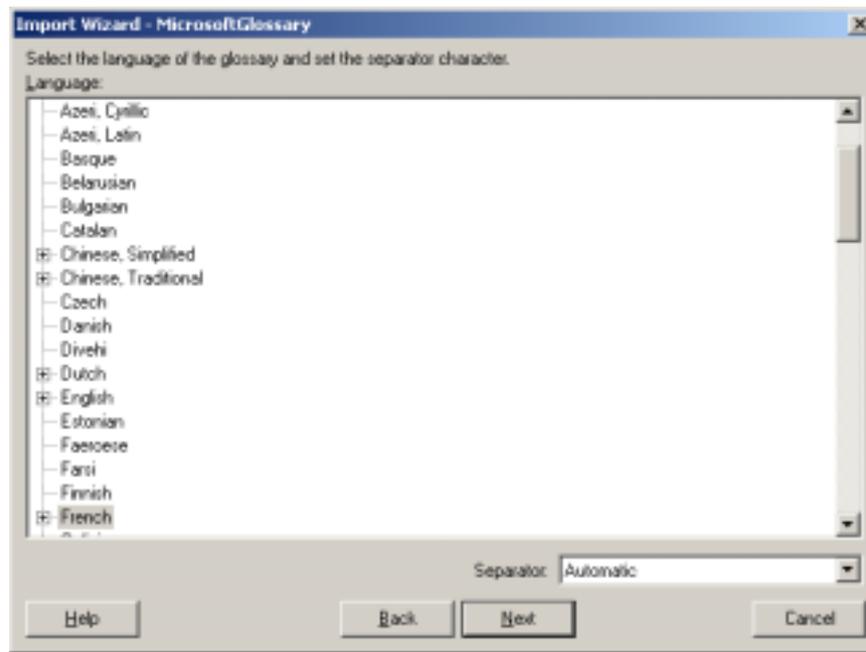


Figure 37: Selecting language for importing Microsoft glossary.

If language detection fails, the user can change it manually.

Multilizer Dictionary File

Multilizer Dictionary Files (MLD) were widely in use in previous Multilizer versions. They were used both as glossaries as well as a resource in multilingual Delphi, C++Builder, and Visual Basic applications.

5

Translate

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java Multilizer Translator Edition Pro Multilizer Translator Edition
User's role in process:	Translator
Wizards:	None

The Multilizer project can be translated with any Multilizer edition.

Restrictions

The items that the translator can access depend on how the project was prepared for translation (→ Prepare project for translation, p. 26).

Possible restrictions are:

- Translation length is limited to a certain pixel or character amount. If translation doesn't fit, it should be reported in the comment.
- String is locked; translation can't be added.
- Wysiwyg editor is read-only; translations can be edited, but forms position and size can't be edited.

The translator should always check the availability of possible comments; there may be translation-specific information.



Restrictions can't be disabled in sub-projects. If translation is done directly on a main project (i.e., Exchange Wizard has not been used), then restrictions can be disabled as described in the chapter 'Prepare project for translation', p. 26.

Translation work-place

Translation work-place is the part of Multilizer user interface where translations are done.

Project Tree enables easy navigation between project parts, and corresponding strings (and other localizable data) is shown in translation grid. Multilizer Translation Grid always shows Native language and target language.

New (native) strings are marked with a yellow dot in the grid margin.

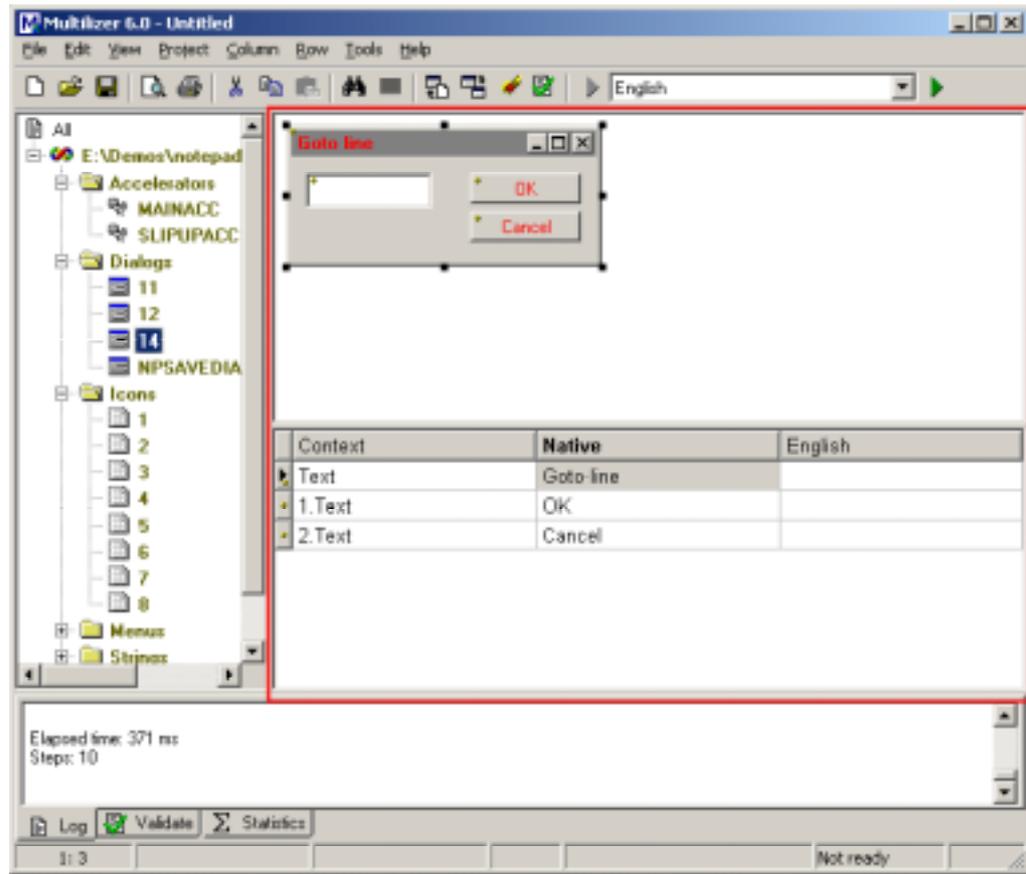


Figure 38: Translation view in Multilizer.

If selected node is a dialog or menu, Multilizer displays it as Wysiwyg, as in image above. Untranslated strings are shown in red.

Selecting visible columns

Choosing **Columns...** from the Translation Grid's context menu (or selecting **View→Columns...** from main menu) brings the following dialog that lets the user specify the visible columns:

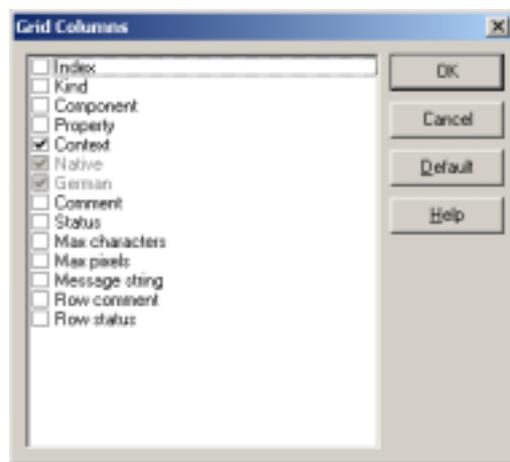


Figure 39: Choosing visible columns in translation grid.



In order to view more than one language, select any of the languages listed in View menu.

Row filtering

Filtering is used to view only a part of the localizable data in translation grid. Users can specify how to filter rows by translations status, row status, and data type. Choosing **View→Filter...** shows the Grid Filter dialog.

Data type filter

Translating strings is one part of localization. In addition to these, there are other kinds of data that need to be changed in the localized software. Multilizer can show in the translation grid various data types.

On Data Types tab user can define what data types are shown in the translation grid.

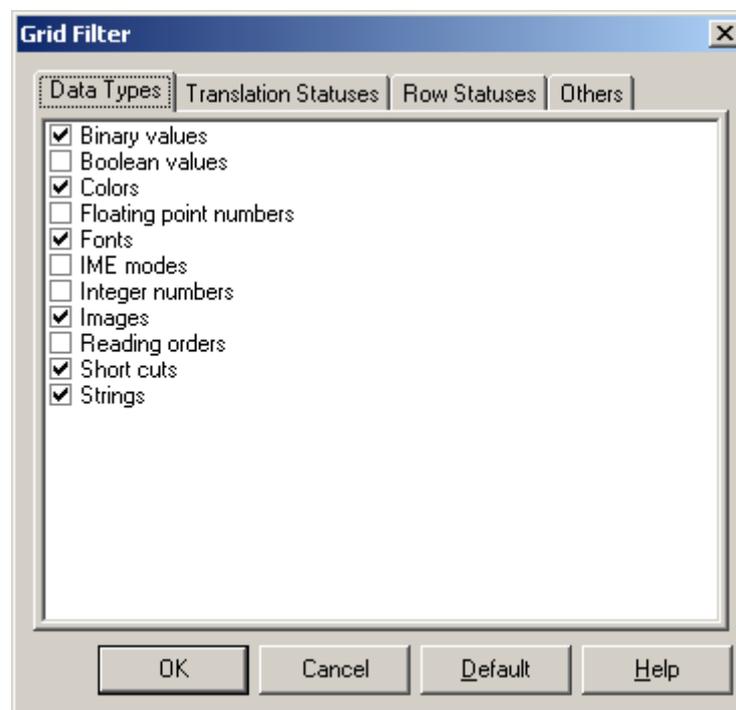


Figure 40: Options for filtering by data type.

Generally, the translator should not translate non-textual data, except if asked separately. Translating non-textual data can damage the localized software, preventing it from working correctly.

Translation status filter

On Translation Statuses tab user can specify, how to filter rows by the target language's status. If all statuses are checked, rows are shown regardless translation's status.

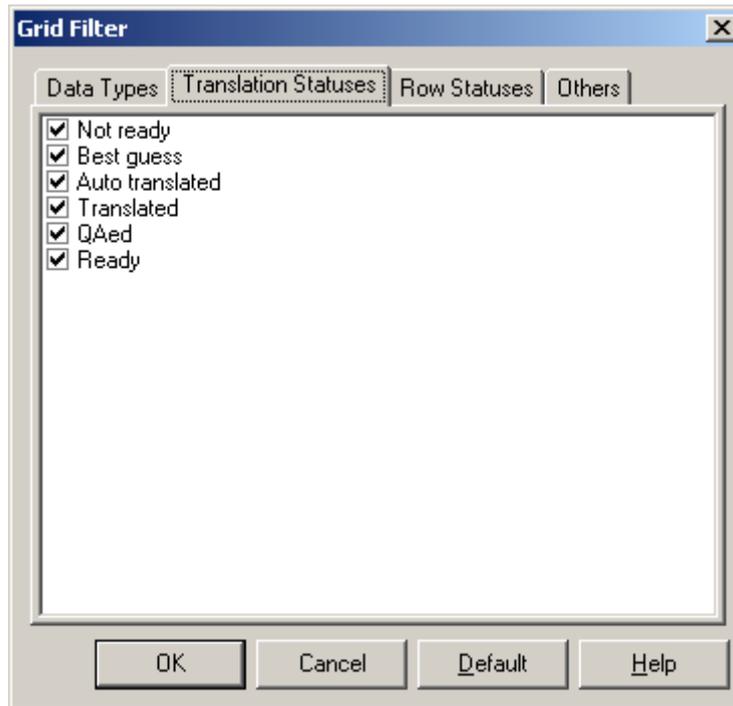


Figure 41: Options for filtering by translation status.

Row status filter

On Row Statuses tab user can define, how to filter by row status.

New rows contain localizable data that was not present in last re-scan (→ Re-scan project, 25) of targets. Unused rows are those that are not found in the targets. In use strings are those that are found

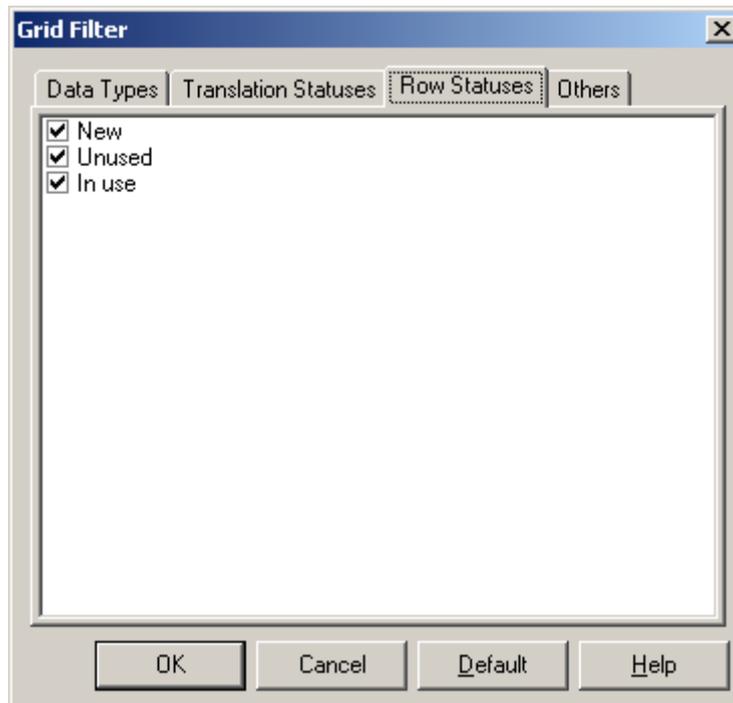


Figure 42: Options for filtering by row status.

Other filters

Others tab contains other attributes that can be used for filtering.

Strings can be filtered by Do not translate -status as shown in figure below.

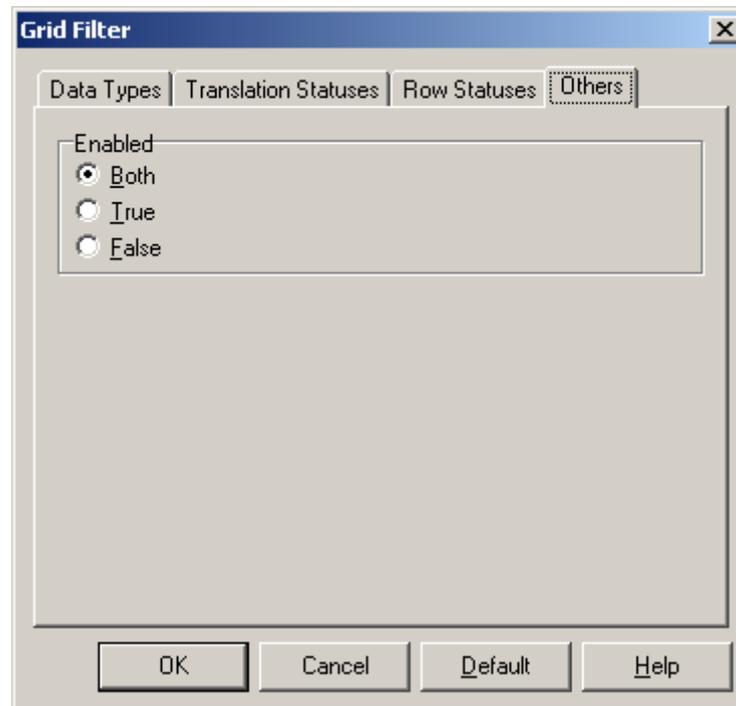


Figure 43: Options for filtering strings by Do not translate -status.

Translation grid options

Right-clicking the translation grid area and selecting **Options...** from the context menu (Main menu: **Tools**→**Options**→**Grid...**) shows the grid options dialog. Grid options affect how translations are shown in the grid, and what additional information is shown.

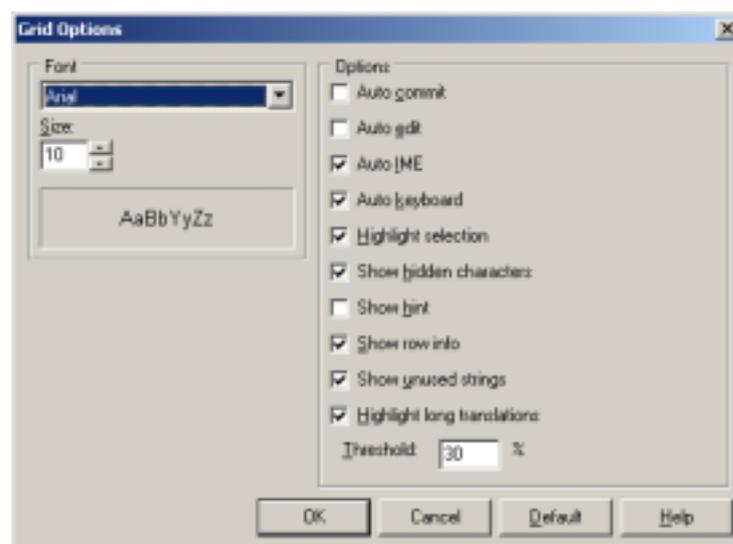


Figure 44: Translation grid options.

Visual Editors, Wysiwyg

Visual Editors enable editing of both non-textual data as well as translation.

Translations of dialogs and menu-items can be edited so that results of localization are shown visually during translation. This feature -- also known as Wysiwyg (What you see is what you get) -- reduces the time spent on localization QA.

Multilizer visual editors allow two-way navigation. When user navigates in translation grid corresponding control in wysiwyg gets focused. When user works in wysiwyg, corresponding cell in translation grid gets focus.

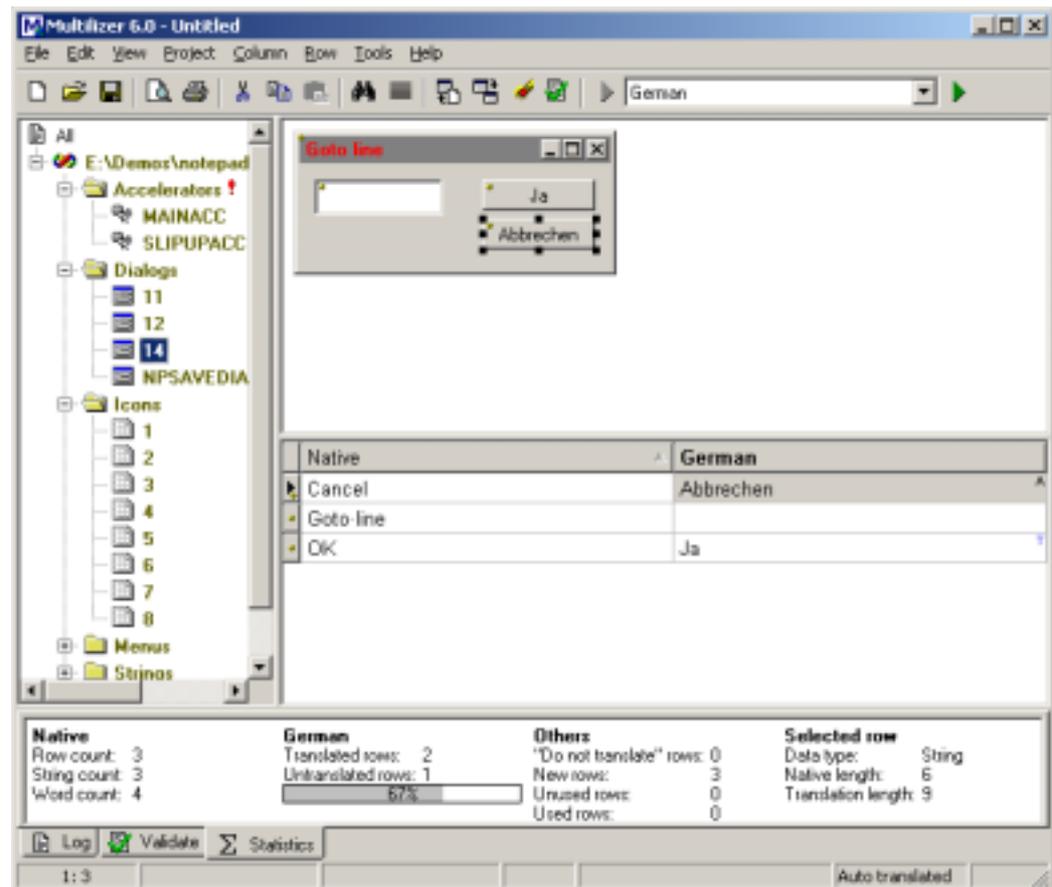


Figure 45: Visual editor for forms and dialogs.

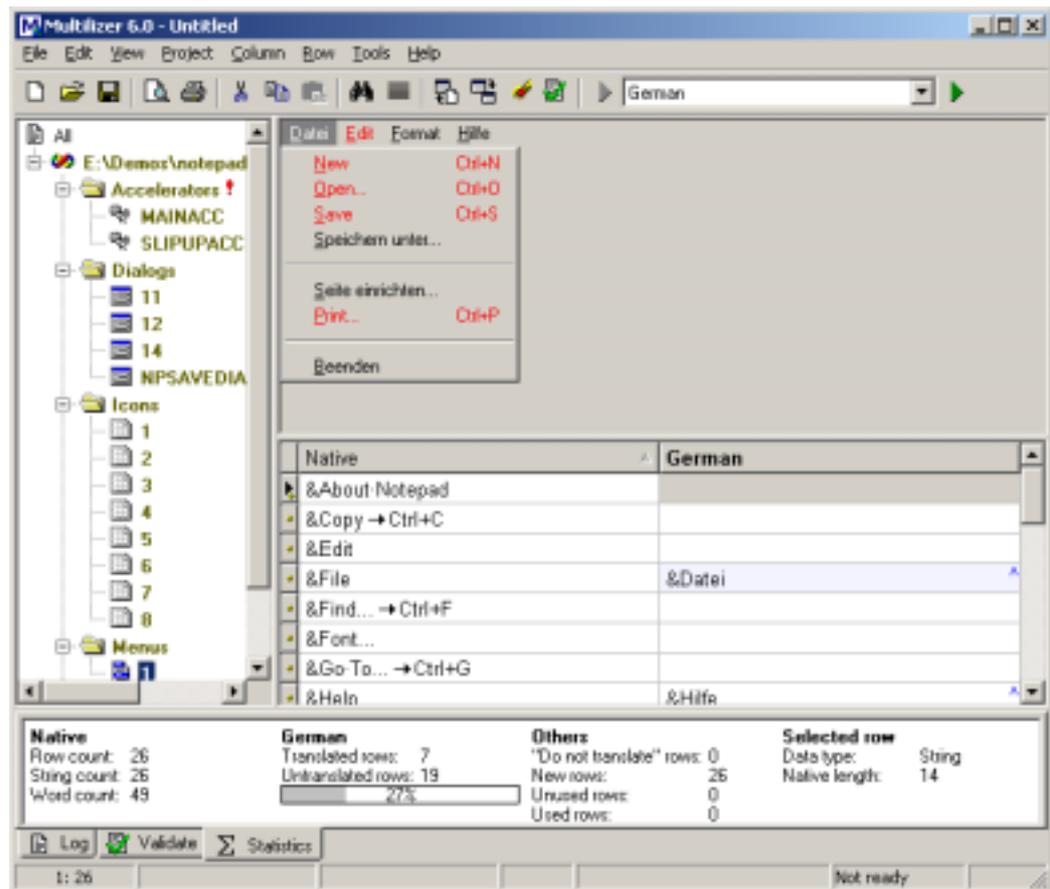


Figure 46: Visual editor for menus.

Visual Editor (Wysiwyg) settings

Right-clicking the wysiwyg area and selecting **Options...** from the context menu (Main menu: **Tools**→**Options**→**Graph...**) shows the wysiwyg options dialog.

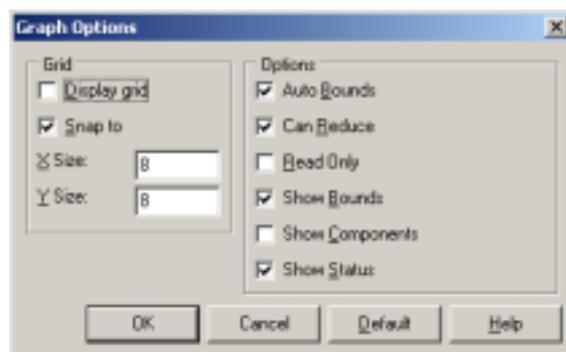


Figure 47: Options for visual dialog editors.

Options

- **Auto bounds** makes placeholder resize to the same size as translated string.
- **Can reduce** allows manually decreasing the size of placeholder.
- **Read only** locks the editor in the way that only translation is allowed; any modifications on layout are disabled.
- **Show bounds** enables color coding control boundaries according to changes in size and position of localized placeholder's (→ Control boundary colors, p. 69)

- **Show components** will show non-visual components as well. This setting applies to component-based environments, such as VB, Delphi, C++Builder, and VS.NET.
- **Show status** will show status of wysiwyg components.

Grid

Grid options allow users define grid size and whether to show it or not.

Localization of strings

Editing translations is easy; translations are simply entered in translation grid, just as in Excel spreadsheet for example.

You can start editing the contents of a cell by double-clicking the cell with the mouse, pressing the **F2** key, or simply starting to type.

You can stop editing the contents of a cell by clicking outside the cell with the mouse or by pressing the **F2**, **UP**, **DOWN**, or **TAB** key. If the native cell contains line feeds, you have to press the **Ctrl+UP**, **Ctrl+DOWN**, or **Ctrl+TAB** keys.

Pressing **ENTER** stops editing and moves the cursor to the next cell if the native cell doesn't contain line feeds. If it does, then pressing **ENTER** adds a line-feed to the cell. Press **Ctrl+ENTER** to stop editing and moving to the next cell.

During editing, the cell has a light yellow background, and it shows non-visible characters.

	Native	English	T
	Oracle Forms Opti		
untii	Runtime		
untii	Forms <u>5</u> Runtime	·Forms·&5·	
untii	...		
untii	Forms <u>6</u> Runtime		
untii	...		

Figure 48: Translation grid with cells showing non-visible characters.

Localization of accelerators

If selected node is Accelerators, Multilizer displays them in Accelerator table.

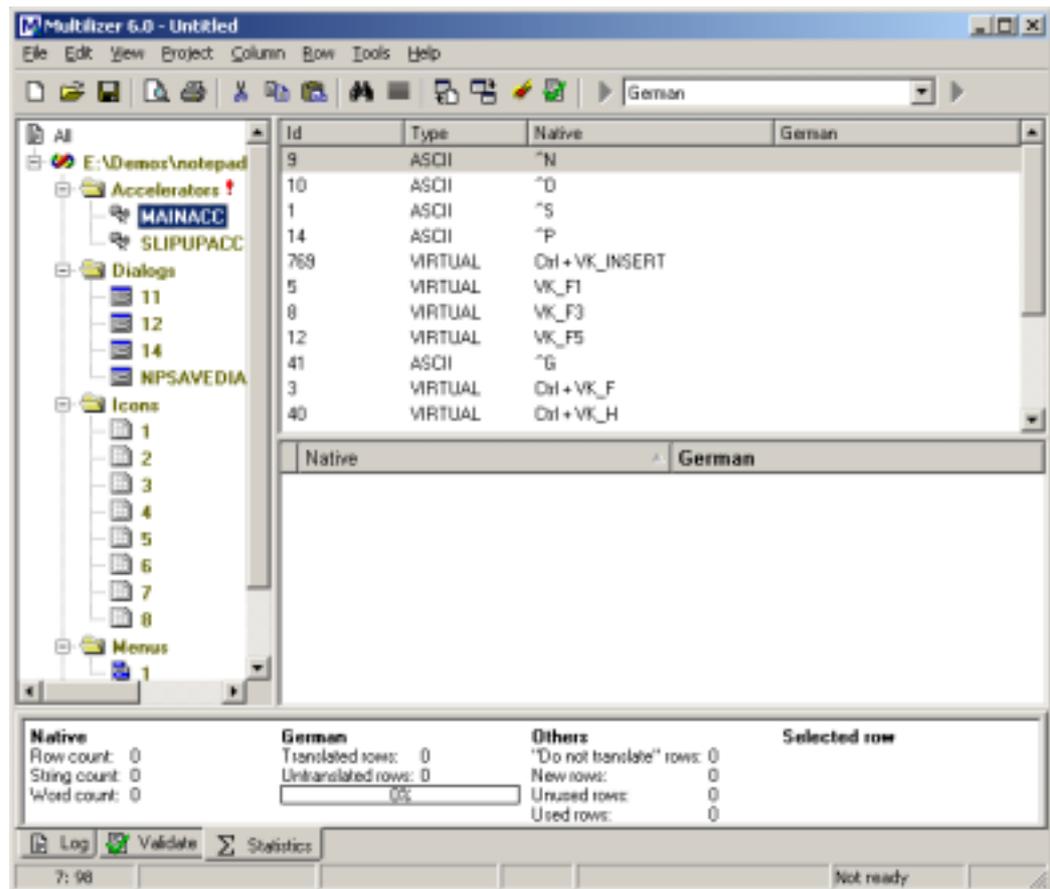


Figure 49: Accelerators view.

Double-clicking an accelerator enables translation of it.



Figure 50: Localizing an accelerator.

Localization of images

In addition to text translation, images can be “translated,” which means that images of the original software can be replaced with new ones in the localized software. Bitmap resources, cursor resources, and icon resources are images.

In order to see images, ensure that *Images* is selected in grid options (→ **Error! Reference source not found.**, p. **Error! Bookmark not defined.**).

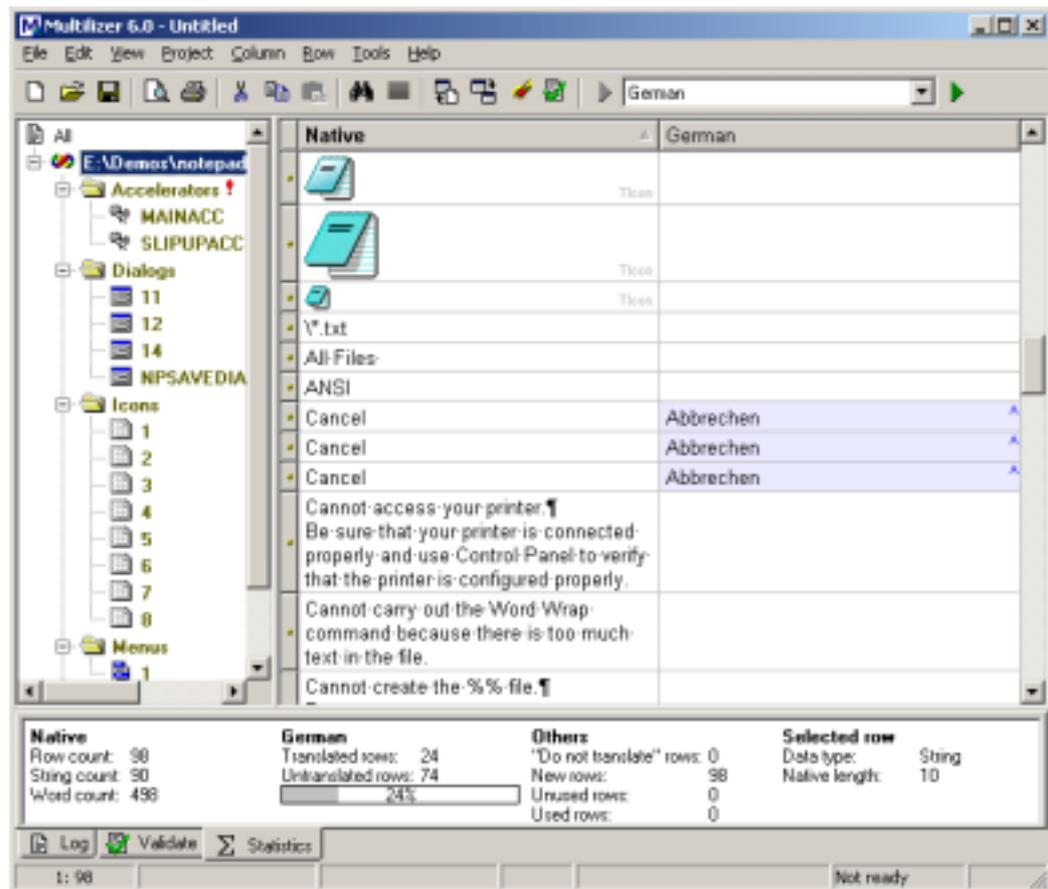


Figure 51: Localizing images.

Native language image can be copied to target language choosing **Edit→Paste Native....** If image will be the same in Native language as in target languages, it shouldn't be copied.

In order to localize the image, right-click target language cell and choose **Load....** User will be prompted for the location of localized image.

Localization of AVI and other custom resources

Multilizer localized Windows custom resources. Windows custom resources can contain any data, such as AVI animations, Wave sound files, etc.

Custom resources are binary data. In order to see custom resources, ensure that *Binary values* is selected in grid options (→ **Error! Reference source not found.**, p. **Error! Bookmark not defined.**).

Multilizer shows a place-holder for binary data in translation grid.

In order to localize custom resource, right-click target language cell and choose **Load....** User will be prompted for the location of localized custom resource.

Software translation specifics

There are major differences between the translation of words and phrases in software and conventional translation work. The clearest differences are the following:

- There are a lot of one-word translations to do. The strings to be translated are mostly very short.
- The GUI (Graphical User Interface) elements have standard and mostly explicit translations.
- Strings may include characters and codes that have a special purpose in the context of the software functionality.
- The GUI may require a certain maximum length of translations.

In addition, there are many other features that can be derived from those mentioned above. Multilizer includes many features that help you do the translation work.

Characters with a special purpose

Depending on the programming language and the programming technique that the developer has used, the strings in the dictionary may include special characters.

Sample string	Explanation
Cannot create file %s	%s is used to denote a string inside another string. E.g., if 'temp.txt' is assigned to %s, the software would show the following: 'Cannot create file temp.txt' So, %s must exist also in the translation.
%s (%s, line %d)	This is like the example above. There can be multiple special characters in one string to be translated. If you have to change the order of the special characters in your translation, inform the developer of this.
%0:s (%1:s, line %2:d)	If the code is property internationalized, there should not be a string like in the above row but like in the left. As you can see, every variable has been indexed so you can freely change the order of variables.
&File	The & sign is used in menu items and button captions to show the hotkey, i.e., the character that is underlined and is used to trigger the menu. In the example at the left, the text would be shown as File in the menu. It's up to you to decide which hotkey you want to use in the translation.
CODEBASE_ENUM	This kind of dictionary items can normally be left untranslated. The developer might be using it in a way related to software functionality. Normally the developer should mark strings that need no translations. These strings appear on a gray background in the dictionary.

Maximum length of translations

One common issue is the length of the strings to be translated. The string layout in the original software may accept only slight changes in string length. Therefore, the translator has to pay special attention to this.

Multilizer helps in showing possible troublesome translations: it makes the cell background darker the more the translation's length exceeds the native string's length. In addition, Wysiwyg gives the translator immediate feedback of whether the translation fits in a dialog or not.

Translation Memory maintenance

Multilizer Translation Memory allows storing of translations made in the project, and importing translations from glossaries. In order to keep its translations consistent, it needs to be maintained. (→Translation Memory, p. 56)

Sending back translations

If translations were made on a sub-project (i.e., based on a Localization Kit made with Exchange Wizard), translations must be sent back to be integrated in the main project. Choosing **File→Exchange...** from the main menu will instruct in this.

6

Translation Memory

Required product(s):	Any
User's role in process:	Terminologist, QA personnel, Translator.
Wizards:	–

Introduction

All Multilizer products include Translation Memory.

The idea of Translation Memory is to store translations for easy re-use. Reuse of translations saves time, and translations become more accurate as the same translation is applied in every occurrence of the same native string.

Multilizer Translation Memory stores translations in a database. The database can be either local database (such as DBISAM) or server database (E.g., SQL Server, Oracle). If there is no commercial database system installed, Multilizer uses its own database (DBISAM) for storing translations.

Ensuring the Translation Memory quality

Because Translation Memory automates translation, users should pay extra attention to the quality of translations that are stored in it. If non-sense data gets stored in it, automated translations may also return bad results. This would compromise the benefits of using Translation Memory at all.



Before adding any data in Translation Memory, check out following chapter 'Store translations' (p. 60). It explains how to ensure that translations of decent quality get stored in Translation Memory, and how to use block words to improve fuzzy match performance in both speed and quality.

Using Translation Memory

See 'Translate using Translation Memory' chapter (p. 25) for information on how to translate open Multilizer project.

Multilizer Translation Memory configurations are found in **Tools→Translation Memories...** menu. It will display the Translation Memory configuration dialog.

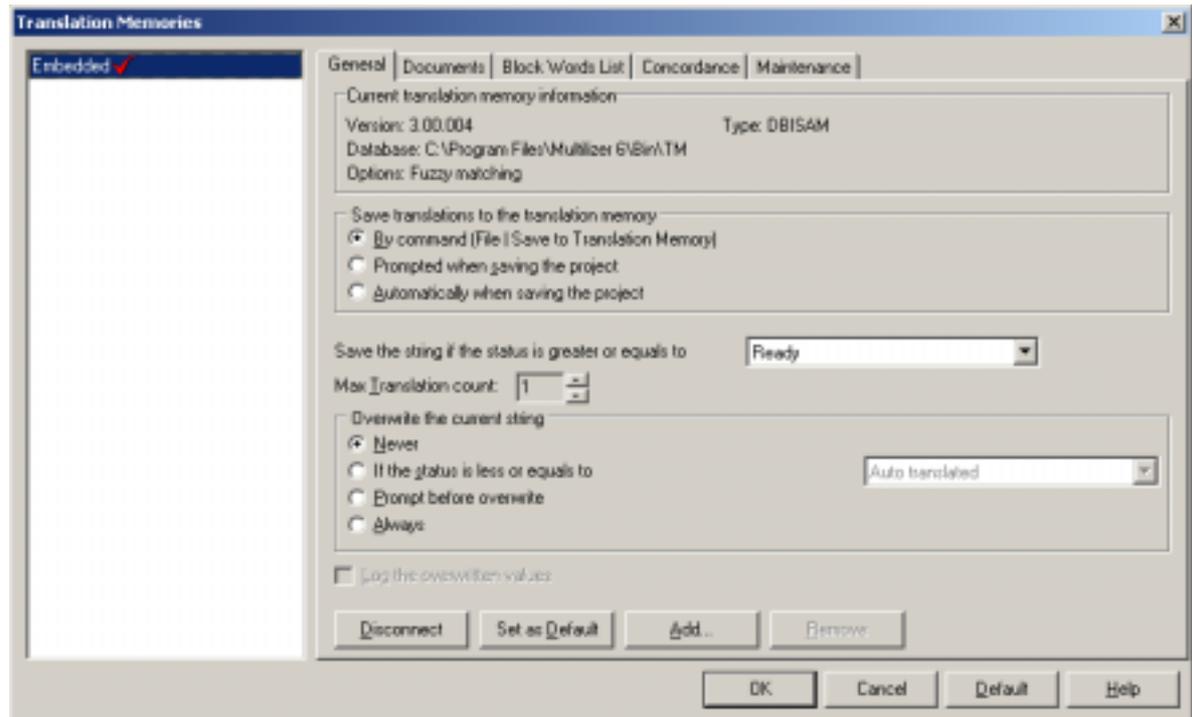


Figure 52: Translation Memory administration.

Multilizer allows using of several translation memories. Each configured translation memory is shown in left-hand side list box. Default translation memory is checked in the list. Translation Memories that have no active database connection are shown in gray.

In order to be able to configure any of the translation memories, you must be connected to it. Once connected, there are five tabs:

- **General**
These settings affect the way that active Translation Memory is used from Multilizer; it specifies on which condition project translation is stored in Translation Memory, and on which condition Translation Memory returns translations for translating project's native string (→ Translate using Translation Memory, p. 25).
- **Documents**
All translations stored in Translation Memory are grouped by documents. Documents can be either Multilizer projects whose translations are saved to translation memory, or glossaries that have been imported by using Import Wizard. (→ Store translations, p. 60)
- **Block Words List**
Block words are used in fuzzy match searches. Besides excluding common words (and hence preventing unwanted matches) using them improves the performance. (→ Block words, p. 62)
- **Concordance**
On concordance tab matching method is specified. If configured Translation Memory supports fuzzy match, user can set threshold for returned translations. (→ Finding Translations, 58)
- **Maintenance**
Maintenance tab contains Translation Memory maintenance specific functions, such as backup, restore, and clear. (→ Maintenance, p. 62)

Finding Translations

Multilizer supports three different ways for finding translations for project's native string.

- Fuzzy matching looks for strings that have something in common with project's native string. User can specify a threshold percentage, in order to control how similar strings have to be.
- *Minimal difference match* looks up for strings that have the same words. Capitalization, punctuation and special characters are ignored.
- *Perfect match* looks for strings that are exactly same as the native string.

Close-to matches and perfect matches are bidirectional; if English to German translations are stored in Translation Memory, Multilizer can find German to English translations too.

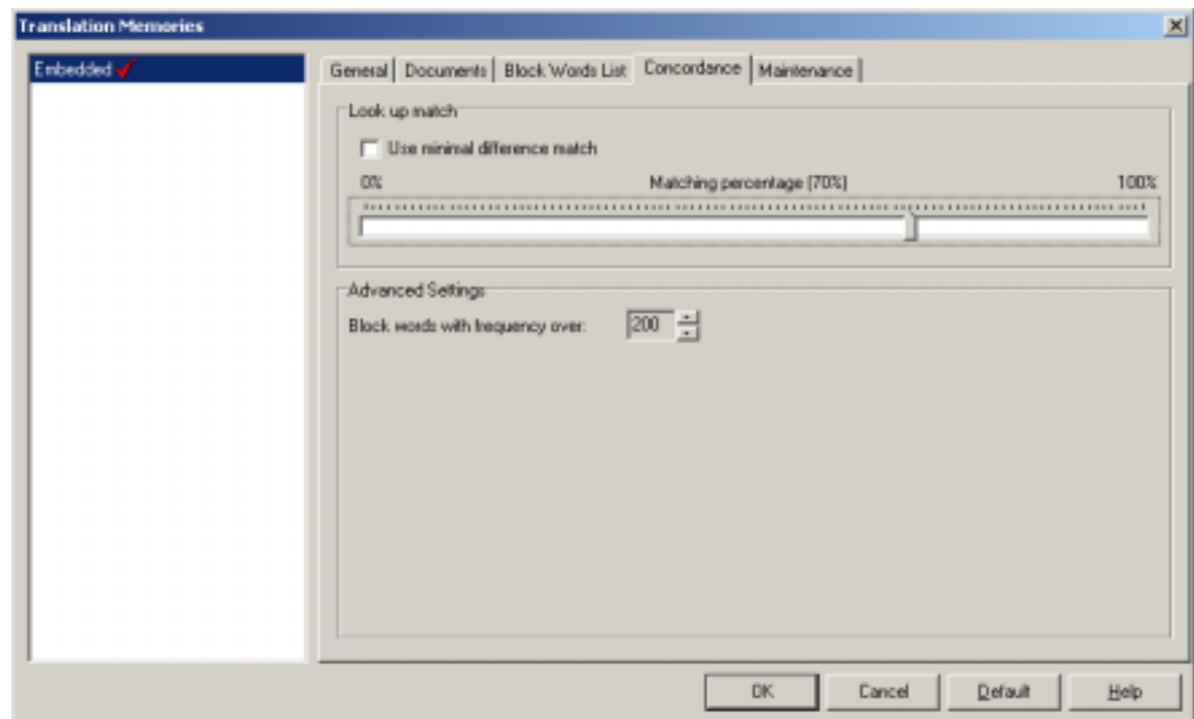


Figure 53: Translation Memory; setting matching options.

Installation of Multilizer Translation Memory

When running Multilizer for the first time it creates a local DBISAM Translation Memory. Depending on the specific needs, user can afterwards add any number and any type of Translation Memories.

New Translation Memories can be added either on local database or on database server. Read chapters 'Create Local Translation Memory' (p. 59) and 'Create Server Translation Memory' (p. 59) before adding a Translation Memory.

Click Add... button in Translation Memories dialog to add new Translation Memory.

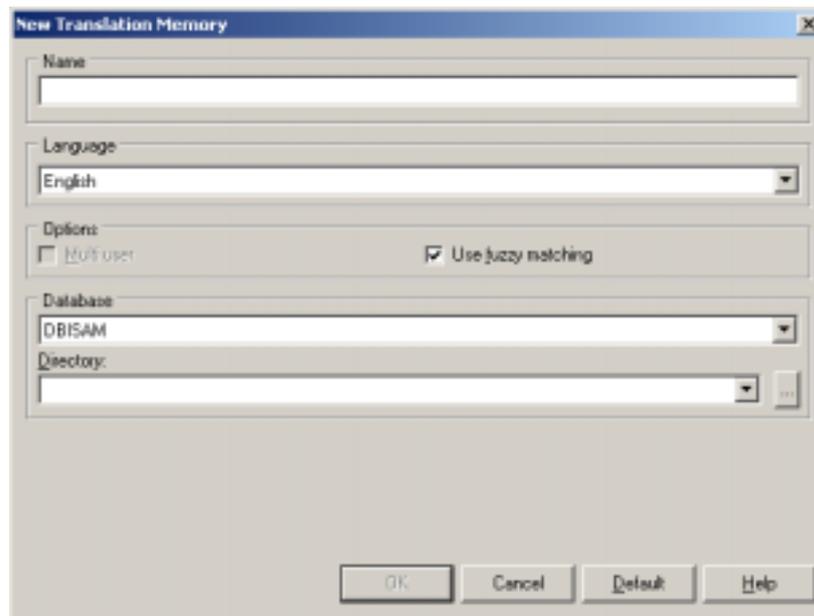


Figure 54: Adding properties of a new Multilizer Translation Memory.

Name is a descriptive name for the Translation Memory.

Language is the source language by which the Translation Memory is indexed.

Options allow configuration of basic functionality; multi-user support, and fuzzy-matching. Multi-user option is available only for database servers.



Selected options affect the format of Multilizer Translation Memory. They can't be changed after creating it.

Depending on the selected database, connection parameters vary. To get correct parameters, ask your system administrator. If DBISAM database is used, directory should point to an empty directory if you want to create a new Translation Memory. If there's an existing Multilizer 6 Translation Memory in the directory, that will be used.

Create Local Translation Memory

Before creating a new Translation Memory from Multilizer, user has to create an empty database for it. The only exception is DBISAM, where Multilizer creates the database.

In New Translation Memory dialog (Figure 54: ***Adding properties of a new Multilizer Translation Memory.***) user then specifies the connection parameters to the empty database.

Create Server Translation Memory¹

Multilizer Translation Memory can be installed on Database Servers, such as Oracle, SQL Server, and Interbase for example. This enables better performance with big amount of translations.

Furthermore this enables multiple users to use the same Multilizer Translation Memory.

¹ Feature enabled in Multilizer Enterprise.

To create Multilizer Translation Memory on database server following steps need to be done:

1. Create a user on database with sufficient rights to create and drop database. Create an empty database that will contain the tables required by the Translation Memory. (This is done with database administration tools).
2. In New Translation Memory dialog (Figure 54: **Adding properties of a new Multilizer Translation Memory.**) user then specifies the connection parameters to the empty database. Multilizer will create the tables required by Translation Memory. In addition Multilizer adds default Translation Memory user with full Translation Memory administrator rights to it.
3. Add/modify Translation Memory users. This is done with Multilizer Translation Memory user management (→ Next chapter).



Because Multilizer Translation Memory has its own user management, there is no need – from Multilizer Translation Memory user management point of view – to specify the users with database administration tools; Multilizer's own user management allows specifying of appropriate access rights for different users connecting to Multilizer Translation Memory.



In case of assigning users using database administration tools, required database access rights are discussed on SQL Statement level here: , p. 60.

Store translations

Translations are stored to Translation Memory either by saving project translations to it (File→Save to Translation Memory...) or by importing translations as documents.

Save project translations

When choosing **File→Save to Translation Memory...** in Multilizer, the translations of open project will be stored in default Translation Memory. Project translations can be configured to be saved in Translation Memory automatically.

Regardless the way of storing translations, user can define that only strings with certain status are stored in translation memory. For example, only QA-ed or Ready strings are stored. This ensures that only verified strings are stored.

² Feature enabled in Multilizer Enterprise.

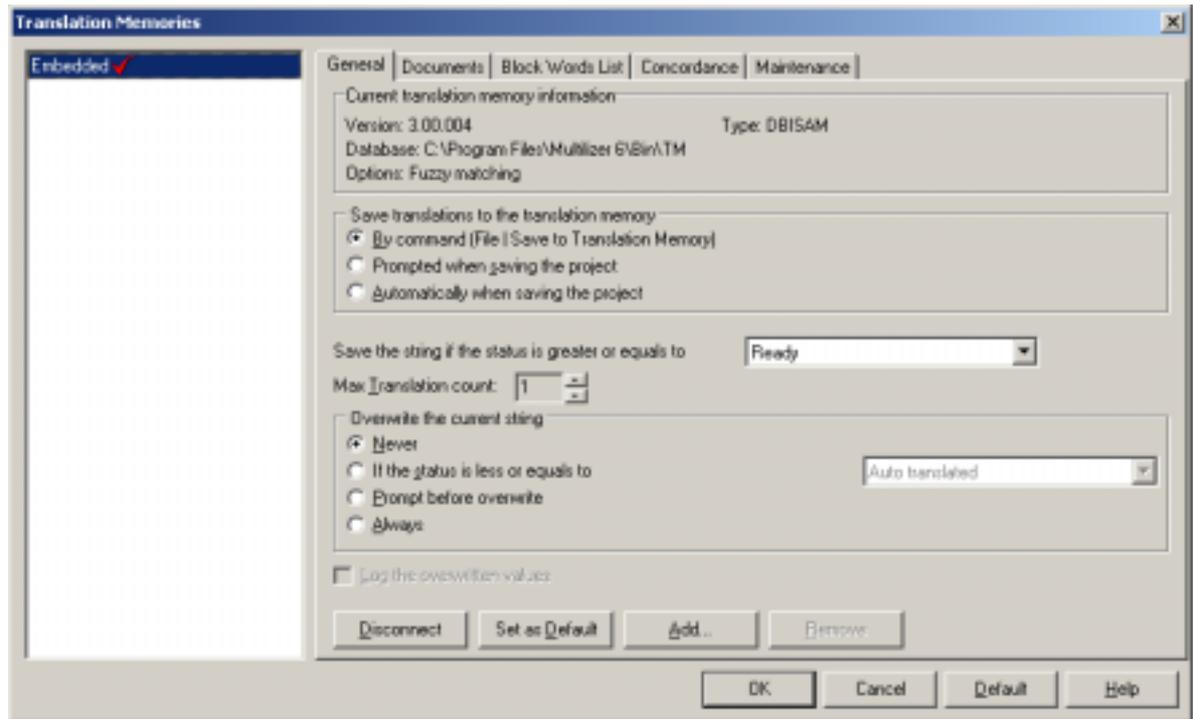


Figure 58: Translation Memory; general settings.

Import documents

Click **Add...** on documents tab to import translations.

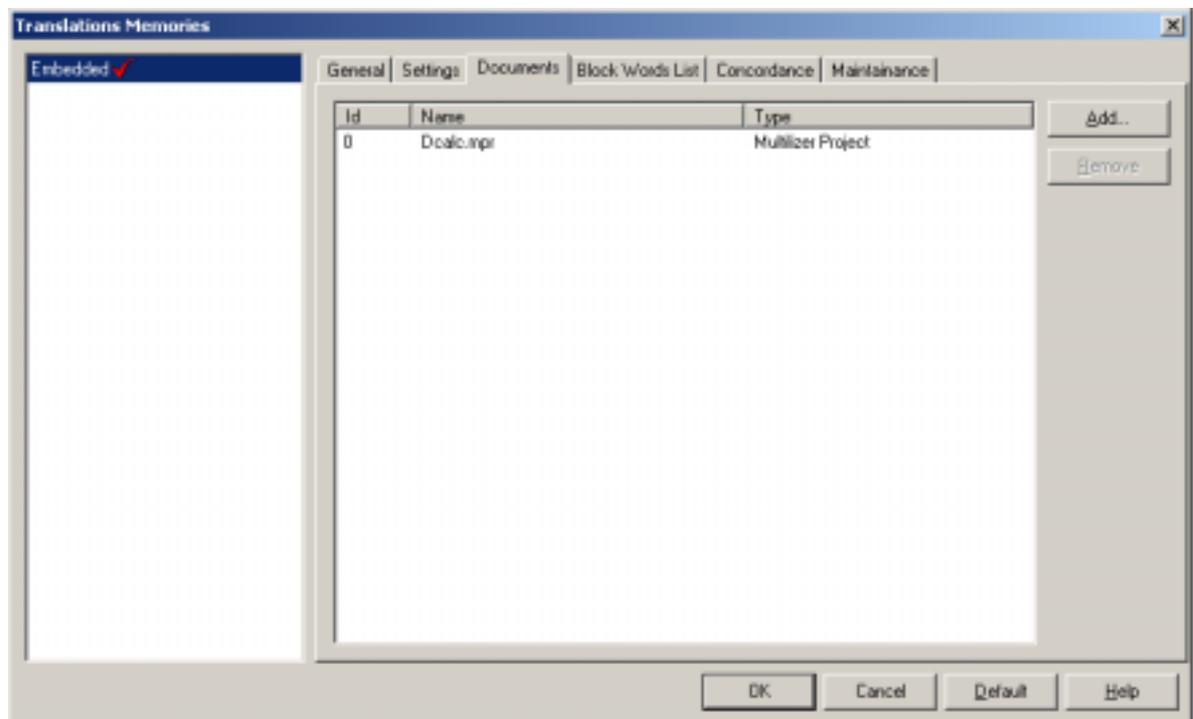


Figure 59: Translation Memory; importing of documents.

When importing text, it is passed through Multilizer Translation Memory's segmentation; text is being split in shorter strings, which improves fuzzy match search results.

Segmentation

The idea of segmentation is to split longer source language texts into segments by using segmentation rules. As a result one or more source language segments point to corresponding translation. This helps to easier find existing translations.

A segment is normally equivalent to a sentence. Based on this, pre-defined set of segmentation rules aim to split text in sentences. E.g., a full stop, exclamation mark, question mark, or colon indicate the end of a sentence when they are followed by a space.

Block words

Segments are stored in a way that each word of it – except a block word – is indexed. Block words are typically the most common words of a spoken language. Common words exist in most sentences, hence not differentiating two sentences in any way.

Specifying block words will improve the results of fuzzy-matching and speed up searches.

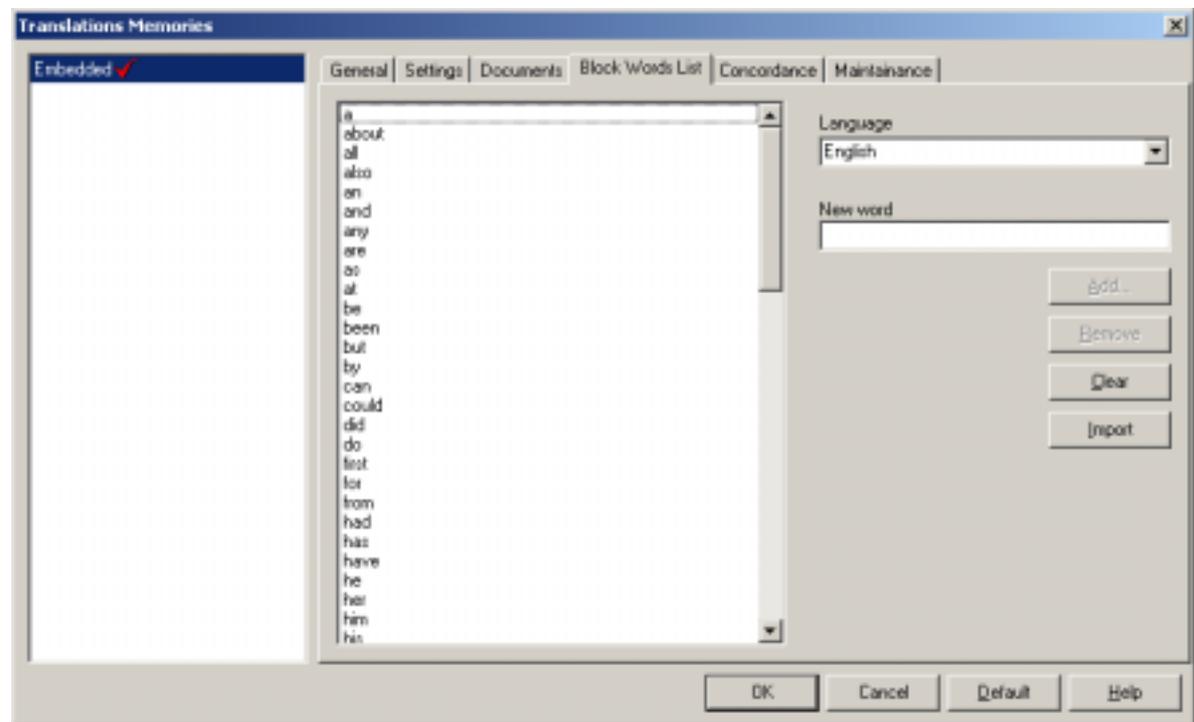


Figure 60: Translation Memory; specifying block words.

Maintenance

Maintenance tab includes tools that affect the entire Translation Memory.

³ Feature enabled in Multilizer Enterprise.

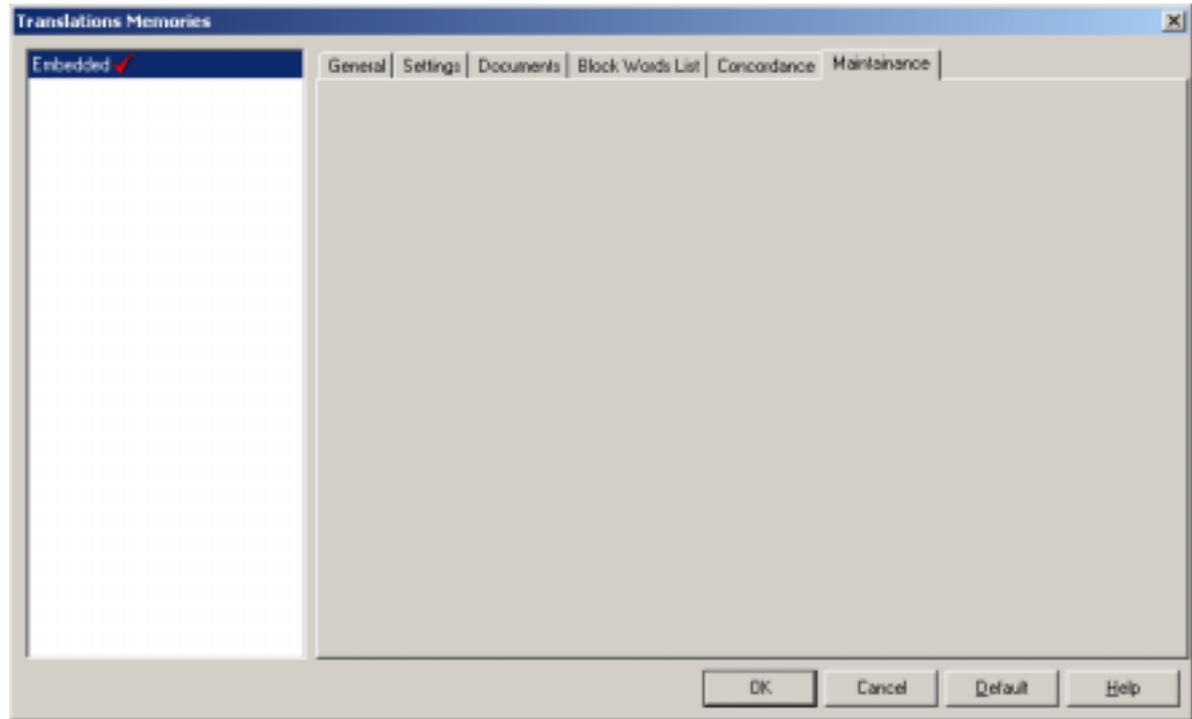


Figure 62: *Translation Memory maintenance tab.*

- **Clear**
Clear will erase all translations from Translation Memory; it clears all tables of Translation Memory database.
- **Backup**
Backup will export Translation Memory database contents to an XML-file. It makes perfect copy of table structures and data.
- **Restore**
This restores the translations from existing backup file. Restore erases existing translations.
- **Repair**
Repair checks out for unlinked segments and other garbage data in Translation Memory, and attempts to make corrections.

7

Quality assurance

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java Multilizer Translator Edition Pro Multilizer Translator Edition
User's role in process:	QA personnel, Translator
Wizards:	Validation Wizard

Quality assurance features are available in all Multilizer versions. However, complete testing of localization requires the possibility to build (→Build localized versions, p. 73) localized versions of software/content.

There are both automated and informative quality assurance features in Multilizer.

- Validation Wizard is an automated QA feature.
- Informative QA features give the user visual feedback, such as cell colors, on-line statistics, etc.

Validation Wizard

Validation Wizard automates validation of common QA issues in localization. It includes a comprehensive set of tests performed against localized versions of software/content.

Selecting **Project→Validate Options...** lets user choose the validation routines to run. Results of validation can either be directed to log window, or it can be saved as a report for later review. If validation settings differ from default options, they are stored in the Multilizer project; this ensures that the same validation is applied consistently in the project.

Validation types

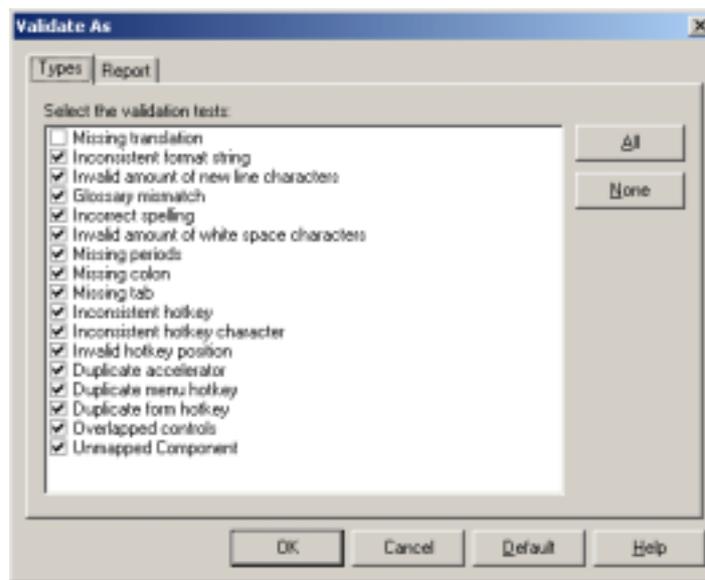


Figure 63: Selecting validations to perform.

Missing translation. This validation tells if native string is translated or not. This validation is useful after re-scanning the project; the user will immediately get feedback of the location of new strings.

Inconsistent format string. This validates that same formatting strings (arguments) are present both in native string and translation. (Press %S to..., place holders)

Invalid amount of new line characters. This validates that the number of New Line (NL) characters match in native string and in translation.

Glossary mismatch. This validation checks if translations match with translation memory contents.

Incorrect spelling. This validation checks spelling using MS Office spell-check. For further spell-check support, please refer to MS Office documentation.

Invalid amount of white space characters. This checks that both native string and translation include the same number of White Spaces at string start and end.

Missing periods. This checks that both native string and translation include the same number of periods at the end of the string. (Useful in menus; File...or Open...)

Missing colon. This validation checks that if native string includes a colon at the end, translation should also. This is important, because label captions generally should include a colon, and corresponding buttons should not. If native software follows these guidelines, this validation ensures the same quality in localized versions.

Missing Tab. This validation checks if native string and translation have the same amount of tab-characters.

Inconsistent hotkey. This validation checks if native string and translation have the same amount of tab-characters.

Inconsistent hotkey character. This validation checks, if hotkey is valid. For example hotkey can't precede a line break.

Invalid hotkey position. Hotkey can't be the last character of a string.

Duplicate accelerator. Checks that there are no duplicated accelerators.

Duplicate menu hotkey. Checks that there are no duplicated menu hotkeys; in each drop-down menu hotkeys must be unique.

Duplicate form hotkey. Checks that there are no duplicated hotkeys on the form or dialog. The hotkeys must be unique on each form/dialog.

Overlapped controls. Checks that placeholders of visual controls are not overlapping.

Unmapped component. Checks that visual components are mapped to a visual representation. This mapping ensures that 3rd-party and custom controls are shown correctly in Wysiwyg. This validation is useful in highly component-based development environments, such as Delphi (→ p. 115) or Visual Studio .NET (→ p. 138) for example.

In order to generate report of validation results, check *Generate report* on Report tab (→ Validation reports, p. 71).

Working with validation results

Validation results are displayed in validation log of info page.

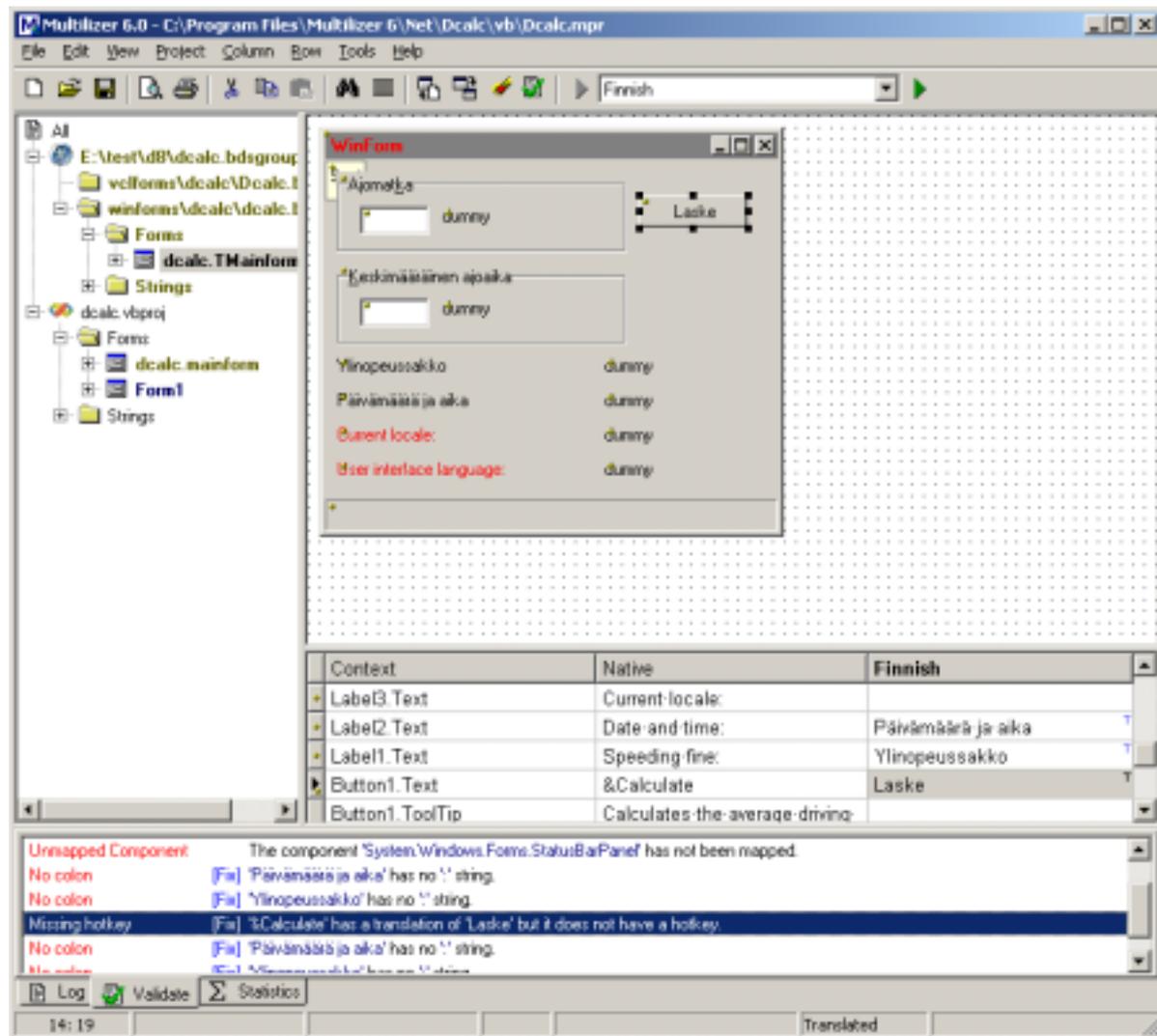


Figure 64: Displaying validation results.

Quick fix

Multilizer offers quick fix for several validation issues. Just click Fix to have Multilizer correct the issue automatically. If fix doesn't correct the issue, you can still manually fix the problem.

Navigation

When a row of validation log is clicked, corresponding translation is shown both in translation grid and visual editor. In addition Project tree shows current node. This auto-navigation feature makes it extremely fast to process validation results.

Change translation status

Whenever an issue is corrected, translation status should be changed to QAed (or any other status used by the company). This makes it possible to later filter out validated parts of project. (→ Translation Status, p. 69)



If you correct issues immediately, it's possible to make Multilizer change translation status automatically; Select **Tools**→**Options...**→**Status**, enable Default status on select, and set desired status. Now clicking a row in validation log will change status automatically.

Pseudo Languages

Multilizer includes comprehensive support for test and pseudo languages. Each language in the project can be populated with pseudo translations. This enables complete localization of software/content before translation of the project has even begun.

To use Pseudo translation, right-click language column and choose **Fill Pseudo Translation...** (Main menu: **Column**→**Fill Pseudo Translation...**). This shows the configuration dialog for Pseudo Translation.

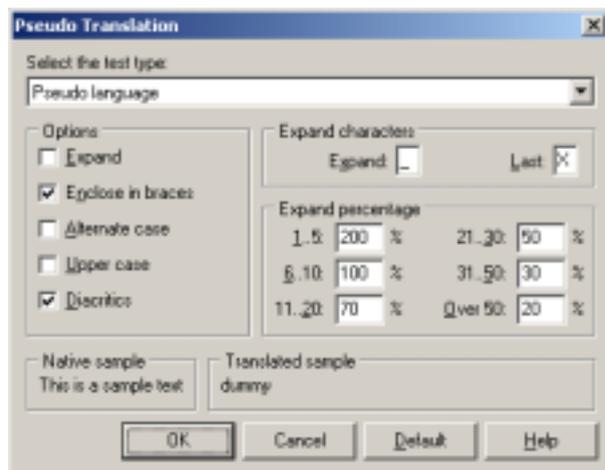


Figure 65: Defining Pseudo Translation properties.

There are several tests with different purposes, and for detecting different issues. After defining a test for a language, it can easily be turned on/off.

Cover

This test replaces all characters of a string with the same number of user-defined characters. This test helps to detect potential UI issues; it lets users immediately see non-localized parts of the UI, hence locating possible internationalization issues.

Minimum

This test replaces all strings with a single user-defined character. This test helps to detect potential UI issues.

Pseudo language

Pseudo language test is the most sophisticated test language. Besides allowing more customization, it also helps in testing localization to different character sets.

Pseudo language supports single-byte LTR (left-to-right) character sets, single-byte RTL (right-to-left) character sets, DBCS (Double-byte character sets), and Unicode®.



Informative QA features

Informative QA features give the user feedback of translations. The user's responsibility is to interpret the information, and understand possible issues involved.

Cell coloring

Cell colors give the user immediate feedback of relative change in the translation length; the longer translation is compared with native string, the darker the shade of blue of translation background.

Display of non-printing characters

In cell-editing mode, Multilizer shows non-printing characters exactly as in Microsoft Word; this helps translators to see duplicated spaces, tabulators, etc.

Statistics panel

Statistics panel shows statistical info of native string and its translation.

Control boundary colors

In Visual Form editors, Multilizer displays control boundaries in red, if control size or location is modified in localization. This helps to get a quick overview of changes in UI.

Colors are applied as follows:

Left: Control has been moved horizontally.

Top: Control has been moved vertically.

Right: Control width has been modified.

Bottom: Control height has been modified.

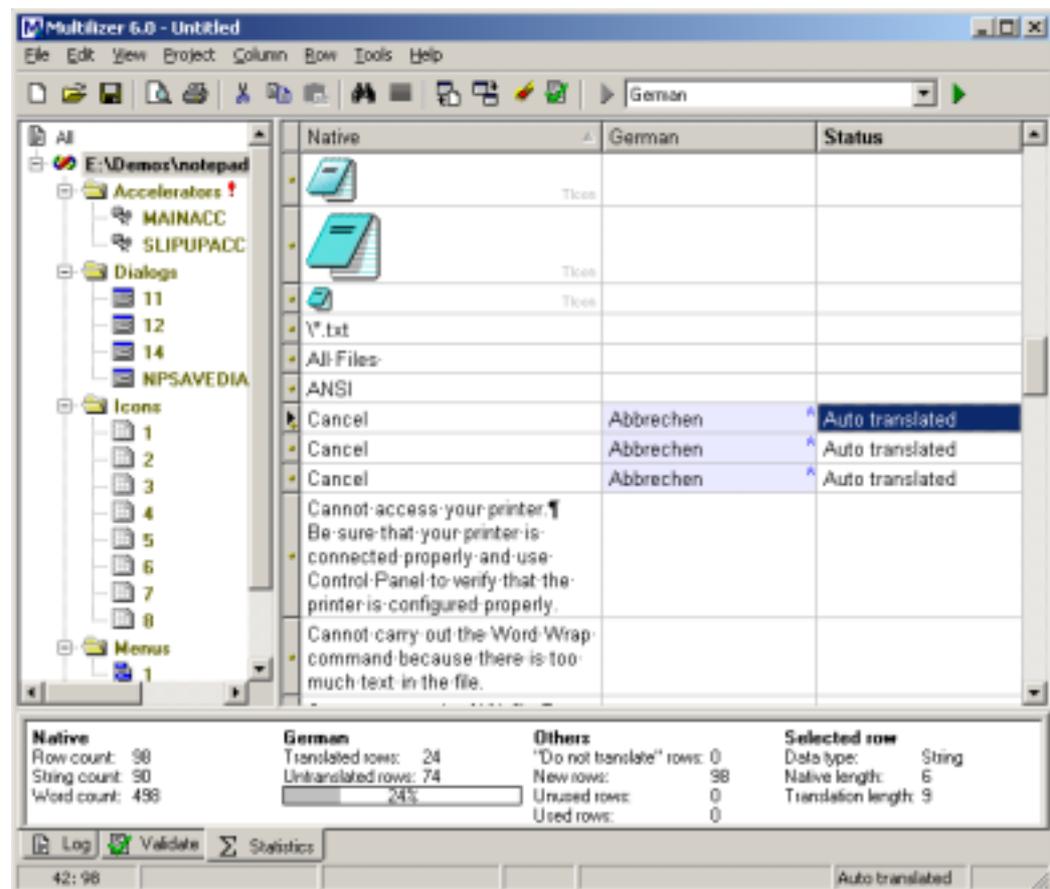
For example, if localized control has been resized horizontally and moved horizontally, control is shown with red boundaries at the left and right sides of it.

Translation Status

Each translation in Multilizer has a status. Status is shown in its own column, and as an abbreviation in translation column.

Maintaining status information in localization project simplifies working with big projects. As discussed earlier, translation status can be used for filtering project translations. In addition, with appropriate use of status information, project reports (→ Reports, p. 71) give a more realistic picture of entire project's status.

Set status automatically



Translation Status changes automatically when translating:

- When using glossaries, Translation Memory, or using duplicates, translation status become 'Auto translated'.
- When translating manually, status becomes 'Translated'.

Aforementioned statuses can be modified (Main menu: **Tools**→**Options...**→**Status**).

Set status manually

When validating a Multilizer localization project, the QA person needs to set status either to *QAed* or to *Ready* (just before release).

There are several ways of setting status:

- Right-click translation, and set status.
- Toggle status by clicking space bar on cell in status column.
- Toggle translations status with Ctrl+T.
- Set status by selecting cell. You need to enable 'Default status on select' for this option (Main menu: **Tools**→**Options...**→**Status**).

Reports

Multilizer produces two kinds of reports:

- Project reports with statistics of project.
- Validation reports with statistics of last validation.

All reports that Multilizer produce are in XML format. This ensures that the data can easily be imported in other systems, such as corporate resource-management systems.

Project reports

Multilizer Project reports (Main menu: **File**→**Project Report...**) give a good insight in localization project status; it shows translation counts by status, and groups localizable items by targets and by languages.

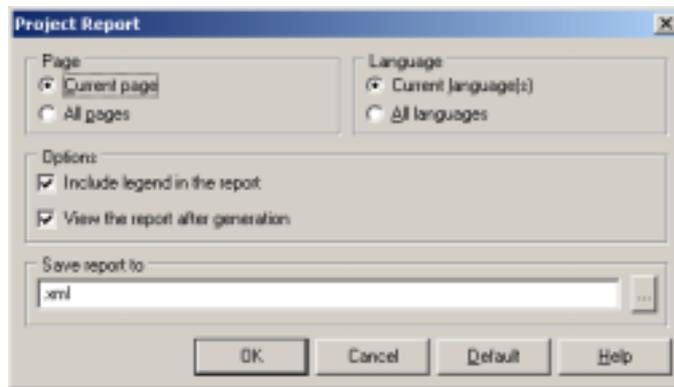


Figure 66: *Project report.*

Validation reports

In addition to showing validation results on-line, Multilizer can write out detailed validation reports. To generate a report select **Project**→**Validate As...**, and select **Report** tab. Check **Generate report**.

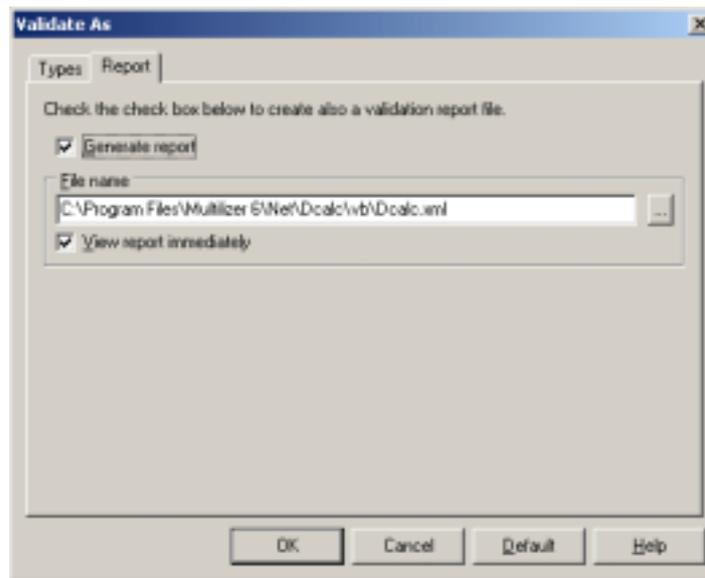


Figure 67: Validation report options.

Validation reports show results by validation types, by localization targets, and by languages.

Validation report can be either shown immediately, or any time later by selecting **Project**→**Validation report....**

Modifying reports

Report data is displayed using XSL (XML Style sheet). Users can change the layout by applying another style sheet to the report. In order to change style sheet reference of generated XML, select **Project**→**Options....**

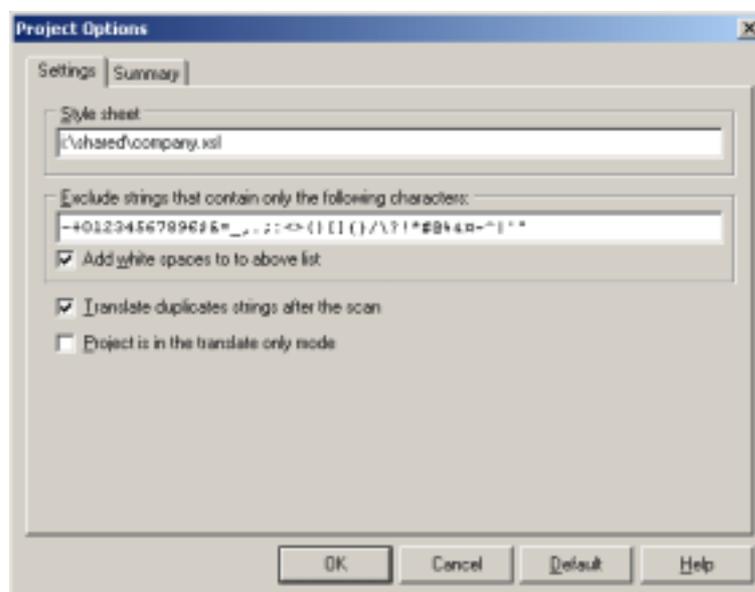


Figure 68: Specifying style sheet for reports.

By specifying style sheet location, Multilizer will generate all reports with referral to the specified one. If *Style sheet* field is left empty, Multilizer uses default style sheet.

7

Build localized versions

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java Multilizer Translator Edition Pro
User's role in process:	QA personnel
Wizards:	None

Build is the last step in the Multilizer localization process; it creates localized items of targets.

How does build work

Localized software items are created by reading original software, and writing localized items using localization information found in the Multilizer project. Multilizer never overwrites the original software. The result of localization depends on the software platform and type; refer to the tutorials for detailed information.

Localized data files are created by reading the original data file, and writing localized data files using localization information found in the Multilizer project. The structure of a data file remains exactly the same. Multilizer never overwrites original data files.

In database localization, localized data is written either in new records, localized tables, or localized fields. The way Multilizer works depends on the database localization type. Refer to the database localization tutorial for more info.

Part II: Multilizer Tutorials

This part includes tutorials for localizing software/content on specific platforms, and each tutorial describes the respective target in depth.

Tutorials:

- **Windows tutorial**
Localization of Windows binaries (EXE, DLL, OCX) that include standard Windows resources. (Software is most commonly developed with Visual C++)
- **VCL tutorial**
Localization of Windows binaries (EXE, DLL, OCX) developed with Delphi or C++Builder.
- **.NET tutorial**
Localization of Visual Studio .NET projects, C#Builder projects, or individual ResX resource files.
- **Database tutorial**
Localization of database contents.
- **XML tutorial**
Localization of XML-files.
- **Source localization**
Localization of single source files.
- **Data file localization**
Localization of INI, SHL, and Key (TXT) files.

8

Windows Tutorial

This tutorial describes localization of Windows software.

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for Visual C++
Sample(s):	<mldir>/VCPP/dcalc
Tutorial(s):	-

Because the resource format is the same for Windows software and Windows CE software, both can be localized using binary localization. For testing Windows CE software, Multilizer supports running localized software in emulator.

- To learn the basics of localization of Windows (C++) software and localization prerequisites, go through the entire tutorial. It requires that you have Visual C++ (4-6) or eMbedded Visual C++ installed on your computer.
→ Introduction, p. 75.
- To learn how to use Multilizer for Windows software localization, you can localize any of the sample applications.
→ Create Multilizer Project, p. 84.

Introduction

For localization of Windows software, Multilizer supports both binary- and RC-localization. Binary localization applies directly to software executable, and RC-localization is done on source code (RC and RC2 files). Binary localization projects are simpler to maintain, because the amount of files to localize is much smaller than in RC-localization.

Localization requires that all localizable data is put in resources during development. This is normally the case in developing Windows software with Visual C++, for example.

Binary applications, libraries, or components contain the resource data in the application files (e.g., .exe), library files (e.g., .dll), or component files (e.g., .ocx). Multilizer creates the localized application files from the original file. The following picture describes the binary localization process:

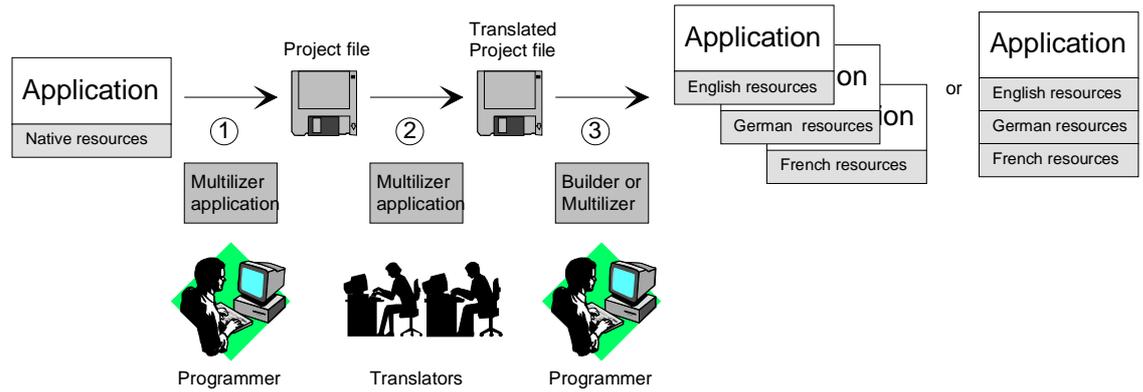


Figure 69: Binary localization process.

The programmer uses Multilizer to extract localizable resources from the original application file (1). Multilizer saves these to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (1). The programmer uses Multilizer or Builder to create the localized application files (2). As a result, there will be one application file for each localized language and/or a single binary file containing all languages.

Open Tutorial Application

We could start from scratch but in most cases it is a completed application or at least some specific application under construction that you want to globalize. This is what we are going to do. The `<mldir>\VCP\Samples\Tutorial\dcalc.dsw` contains the project file of Dcalc sample application for Visual C++. Compile and run the application.

The application should look like this:

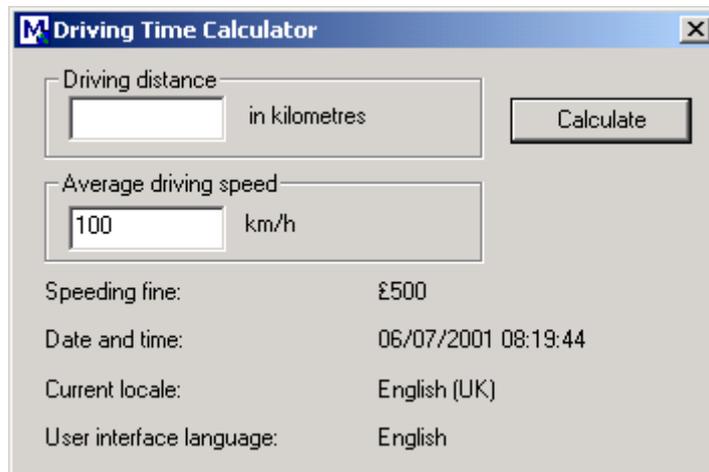


Figure 70: Driving time calculator with English user interface.

The `<mldir>\EVCPP\Samples\Tutorial\dcalc.vcw` contains the project file of Dcalc sample application for Embedded Visual C++. Compile and run the application.

The application should look like this:



Figure 71: English Visual C++ Windows CE application.

The user interface language is UK English and the applications use UK format with currency, date and time. In the following chapters, we will turn Dcalc into a truly multilingual application, step-by-step.

Internationalization

This chapter describes the binary internationalization process. Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software. Internationalization (I18N) takes place at the level of program design and document development.

Open the Tutorial application, `<mldir>\VCP\Samples\Tutorial\dcalc.dsw`, or `<mldir>\EVCPP\Samples\Tutorial\dcalc.vcw`.

Study the source code of the application to familiarize yourself with it. It is not a complex application, so you should get the idea fairly quickly.

The most important part of the internationalization (I18N) is resourcing. This means removing all hard coded strings from the application's source code. Traditionally, hard coded strings are turned into resources by moving the strings from the actual code into the resource strings.

Select the ResourceView sheet. Select the dcalc resource leaf from the tree and click the right mouse button. Choose Insert. The Insert Resource dialog appears.

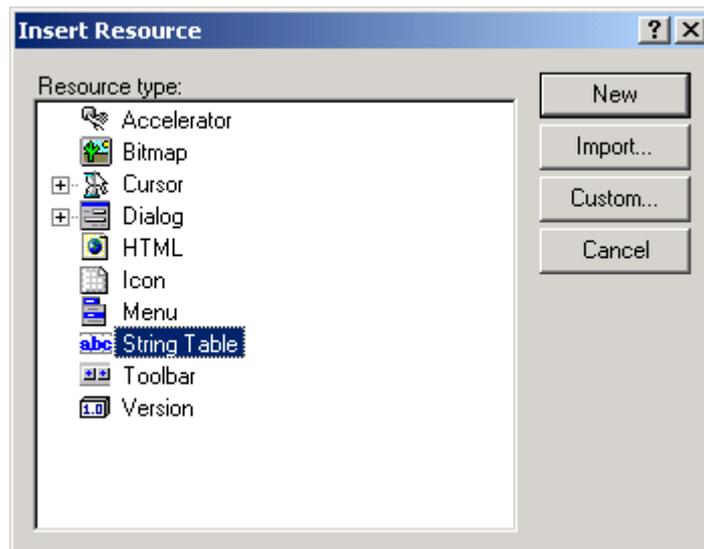


Figure 72: Insert Resource dialog box.

Select String Table resource and press New. The String Table editor appears. Add the following resource strings to the table:

ID	Value	Caption
IDS_ABOUTBOX	101	%s About OCalc...
IDS_INVALID_DISTANCE	102	'%s' is not a valid distance!
IDS_INVALID_SPEED	103	'%s' is not a valid speed!
IDS_RESULT	104	The average driving time is %d hours and %d minutes.
IDS_METRIC_DISTANCE	105	in kilometres
IDS_US_DISTANCE	106	in miles
IDS_METRIC_SPEED	107	km/h
IDS_US_SPEED	108	mph
IDS_LANGUAGE	109	English
IDS_INFORMATION	110	Information

Figure 73: String Table editor.

The next step is to set the right value to the user interface labels. The original application shows the speeding fine in Pounds, the date and time in UK format, the locale and language labels have been hard coded to English (UK) and English. In addition, the application requires the input in kilometers and in kilometers per hour.

An essential part of internationalization is to make the code locale independent. This means that the code is not hard coded to a single locale (e.g., English (UK)) but works with any locale.

Windows contains NLS API. It is a collection of locale functions that have access to the locale database. GetLocaleInfo function is used to get locale specific data such as measurement system, data format, etc.

To prepare your code to locale enabling, we have to write some helper functions.

```

Visual C++    void CDcalcDlg::SetLabel(int control, int resourceId)
                {
                    CString str;

                    str.LoadString(resourceId);
                    SetDlgItemText(control, str);
                }

                void CDcalcDlg::SetLocaleLabel(int control, int localeItemId)
                {
                    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
                    LPTSTR str = (LPTSTR)malloc(len + 2);

                    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
                    SetDlgItemText(control, str);
                    free(str);
                }

                int CDcalcDlg::GetLocaleInfoInt(int localeItemId)
                {
                    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
                    LPTSTR str = (LPTSTR)malloc(len + 2);
                    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
                    int value = atoi(str);
                    free(str);

                    return value;
                }

EVC          void CDcalcDlg::SetLabel(int control, int resourceId)
                {
                    CString str;

                    str.LoadString(resourceId);
                    SetDlgItemText(control, str);
                }

                void CDcalcDlg::SetLocaleLabel(int control, int localeItemId)
                {
                    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
                    LPTSTR str = (LPTSTR)malloc(len + 2);

                    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
                    SetDlgItemText(control, str);
                    free(str);
                }

                int CDcalcDlg::GetLocaleInfoInt(int localeItemId)
                {
                    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, NULL, 0);
                    LPTSTR str = (LPTSTR)malloc(len + 2);
                    GetLocaleInfo(LOCALE_USER_DEFAULT, localeItemId, str, len);
                    int value = _wtoi(str);
                    free(str);

                    return value;
                }

```

SetLabel function sets the label of a user interface element to a value found from the resource string. *SetLocaleLabel* sets the label of a user interface element to a value found from the locale database. *GetLocaleInfoInt* function returns an integer value from the locale database.

Now we can update the user interface items to match the current locale. Keep in mind that the system has a default locale. This locale is given to all applications currently running. You can change the default locale from the Control Panel.

OnInitDialog method is used to initialize the dialog box. Add the following code to the end of the *OnInitDialog* method.

Setlocale function sets the formatting functions of the C run-time library to use the default locale. The original *Dcalc* uses kilometers and km/h. In United States, miles and miles per hour are used. *LOCALE_IMEASURE* value of the locale database contains the measurement system of the locale. *GetLocaleInfoInt* gets the measurement system. If the system is metric, kilometers are used otherwise miles are used.

There are four different ways to show the currency value. They are 500 \$, 500\$, \$500, and \$ 500. You can put the currency label before or after the value and use a space between or not. *LOCALE_ICURRENCY* value if the locale database contains this information. The switch-case block formats the speeding fine according to the current locale.

CTime class has the *Format* method that returns the date and time as a string that has been formatted according to the current locale.

The final step is to update the locale and language labels. *LOCALE_SLANGUAGE* returns the current locale as a string. *IDS_LANGUAGE* resource string contains the name of the language in its own language (e.g., English, Deutch, suomi).

Visual C++

```

BOOL CDcalcDlg::OnInitDialog()
{
    ...
    // Sets the locale depend format function to use the default locale

    setlocale(LC_ALL, "");

    // Gets the measurement system
    // Sets the Driving distance label: km or miles
    // and the Average driving speed label: km/h or mph

    if (GetLocaleInfoInt(LOCALE_IMEASURE) == 0)
    {
        // Metric

        SetLabel(IDC_DISTANCE, IDS_METRIC_DISTANCE);

        SetDlgItemText(IDC_SPEED_EDIT, "100");
        SetLabel(IDC_SPEED, IDS_METRIC_SPEED);
    }
    else
    {
        // US

        SetLabel(IDC_DISTANCE, IDS_US_DISTANCE);

        SetDlgItemText(IDC_SPEED_EDIT, "65");
        SetLabel(IDC_SPEED, IDS_US_SPEED);
    }

    // Set the fine value: $500, 500 mk, etc

    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, NULL,
0);
    LPTSTR currStr = (LPTSTR)malloc(len + 2);

    GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, currStr, len);

    LPTSTR buffer = (LPTSTR)malloc((strlen(currStr) + 5)*sizeof(TCHAR));

    switch (GetLocaleInfoInt(LOCALE_ICURRENCY))
    {
        case 0:

```

```

        sprintf(buffer, "%s500", currStr);
        break;

    case 1:
        sprintf(buffer, "500%s", currStr);
        break;

    case 2:
        sprintf(buffer, "%s 500", currStr);
        break;

    case 3:
        sprintf(buffer, "500 %s", currStr);
        break;
}

SetDlgItemText(IDC_FINE, buffer);
free(currStr);
free(buffer);

// Set the date and time

SetDlgItemText(IDC_DATETIME, CTime::GetCurrentTime().Format("%c"));

// Sets the current locale and user interface language

SetLocaleLabel(IDC_LOCALE, LOCALE_SLANGUAGE);
SetLabel(IDC_LANGUAGE, IDS_LANGUAGE);

return TRUE;
}

```

OnInitDialog function in the Embedded Visual C++ is almost identical. We use COleDateTime instead of CTime.

EVC

```

BOOL CDcalcDlg::OnInitDialog()
{
    CDialog::OnInitDialog();

    SetIcon(m_hIcon, TRUE);
    SetIcon(m_hIcon, TRUE);

    // Gets the measurement system
    // Sets the Driving distance label: km or miles
    // and the Average driving speed label: km/h or mph

    if (GetLocaleInfoInt(LOCALE_IMEASURE) == 0)
    {
        // Metric

        SetLabel(IDC_DISTANCE, IDS_METRIC_DISTANCE);

        SetDlgItemText(IDC_SPEED_EDIT, L"100");
        SetLabel(IDC_SPEED, IDS_METRIC_SPEED);
    }
    else
    {
        // US

        SetLabel(IDC_DISTANCE, IDS_US_DISTANCE);

        SetDlgItemText(IDC_SPEED_EDIT, L"65");
        SetLabel(IDC_SPEED, IDS_US_SPEED);
    }

    // Set the fine value: $500, 500 mk, etc

```

```

    int len = GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, NULL,
0);
    LPTSTR currStr = (LPTSTR)malloc(len + 2);

    GetLocaleInfo(LOCALE_USER_DEFAULT, LOCALE_SCURRENCY, currStr, len);

    wchar_t buffer[20];

    switch (GetLocaleInfoInt(LOCALE_ICURRENCY))
    {
        case 0:
            swprintf(buffer, L"%s500", currStr);
            break;

        case 1:
            swprintf(buffer, L"500%s", currStr);
            break;

        case 2:
            swprintf(buffer, L"%s 500", currStr);
            break;

        case 3:
            swprintf(buffer, L"500 %s", currStr);
            break;
    }

    SetDlgItemText(IDC_FINE, buffer);
    free(currStr);

    // Set the date and time

    SetDlgItemText(IDC_DATETIME,
COleDateTime::GetCurrentTime().Format());

    // Sets the current locale and user interface language

    SetLocaleLabel(IDC_LOCALE, LOCALE_SLANGUAGE);
    SetLabel(IDC_LANGUAGE, IDS_LANGUAGE);

    CenterWindow(GetDesktopWindow());

    return TRUE;
}

```

The CalculateButtonClick event needs a little bit more rewriting. Let's study the code that generates the driving distance message:

```

text = CString("The average driving time is ") +
    itoa(hours, buffer1, 10) +
    " hours and " +
    itoa(minutes, buffer2, 10) +
    " minutes.";

```

This seems to be just OK, but it will actually make the localization hard or even impossible. The reason is that the above logic assumes that the message always starts with the "The average driving time is " string, and then contains the hours, hour label, minutes, and minute label. However, not all languages use the same order of words in a sentence. For example, the order might be: minute label, minutes, hour label, hours, and text part. Reordering of the parts of the message is impossible if we use the code shown above.

Fortunately, we can use CString's Format function. It uses message pattern that contains placeholders for the dynamic parameters. At run-time, the function combines the pattern with the parameters to compose the message. Because the pattern is a single string, it

can be added to the resource strings, and it can then be translated as a single item. The following code contains the internationalized CalculateButtonClick event:

```
void CDcalcDlg::OnCalculate()
{
    // Calculates the driving time and shows it in a message box

    CString text;
    CString distanceS;
    CString speedS;

    GetDlgItemText(IDC_DISTANCE_EDIT, distanceS);
    GetDlgItemText(IDC_SPEED_EDIT, speedS);

    int distance = atoi(distanceS);
    int speed = atoi(speedS);

    if (distanceS == "" || distance < 0)
        text.LoadString(IDS_INVALID_DISTANCE);
    else if (speedS == "" || speed <= 0)
        text.LoadString(IDS_INVALID_SPEED);
    else
    {
        int hours = distance/speed;
        int minutes = (int)((double)distance/speed - hours)*60;

        text.Format(IDS_RESULT, hours, minutes);
    }

    CString str;

    str.LoadString(IDS_INFORMATION);

    MessageBox(text, str);
}
```

The dialog resource contains several strings that are obsolete because they all get set at run-time. A good practice is to replace these strings with “dummy” strings and then exclude these strings from the localization project.

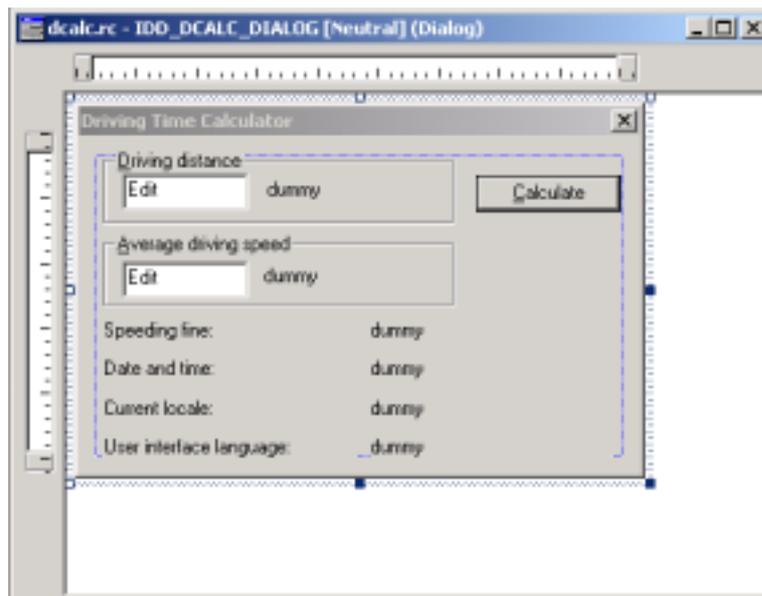


Figure 74: Replace dynamic items with dummy strings.

Most translations get longer when translated from English to other European languages. The final internationalization step is to change the user interface in such a way that it can

accommodate long translations. The easiest way is to set every user interface item as wide as possible. The following figure contains the reworked user interface.

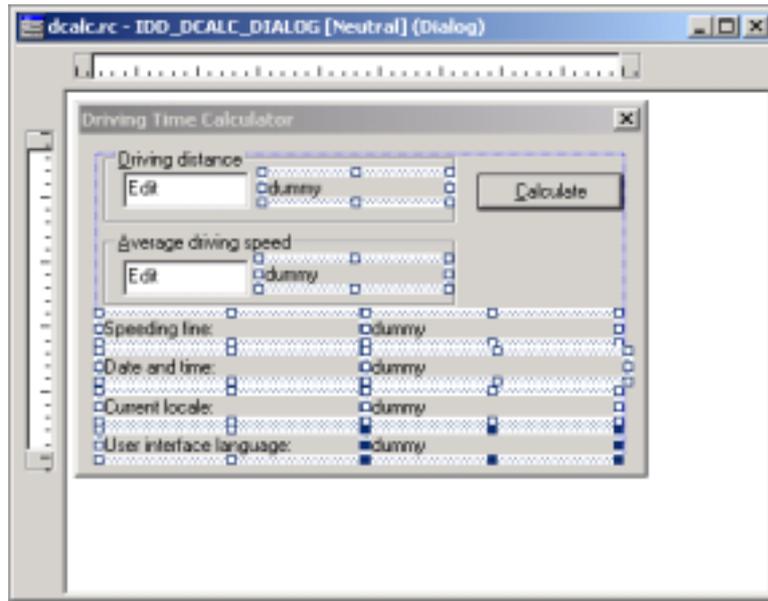


Figure 75: Resize dynamic items for long translations.

We now have internationalized application's code, and it is ready to be localized. Now it is time to launch the Multilizer.

Create Multilizer Project

In order to localize Windows or Windows CE software, you have to create a Multilizer project. This is described in the first part of the manual, in the chapter 'Create Project,' p. 11.

What you need to do is to simply let Project Wizard guide you in this.

Specify Localization options

After finishing the Wizard, you have to specify the localization options for the software. Normally default options are the most useful – and follow Windows suggested way of localization – but in this tutorial, we will review the options.

Right-click the localization target in Project Tree, and click properties to see Windows Binary File Target options.



All Windows software-specific options are gathered under the Windows Binary File Target dialog. If there are many targets in one project, you can set different localization options to all, if needed.

Encodings

Encodings tab lets you specify codepages for target languages. In addition, you can force Multilizer to read the localization target with certain language and codepage settings.

Normally default values should be used; they are based on the information that Multilizer detects from the Windows software.

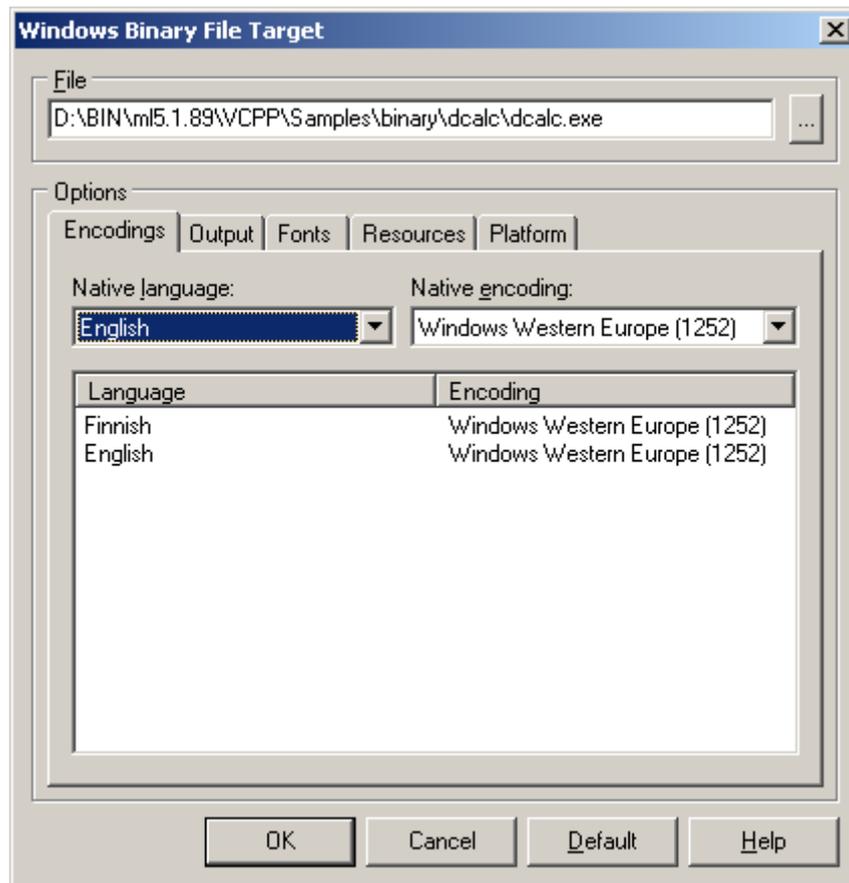


Figure 76: Encoding options for target languages.

Output

The far most important option in localization is to specify location and type of localized files.

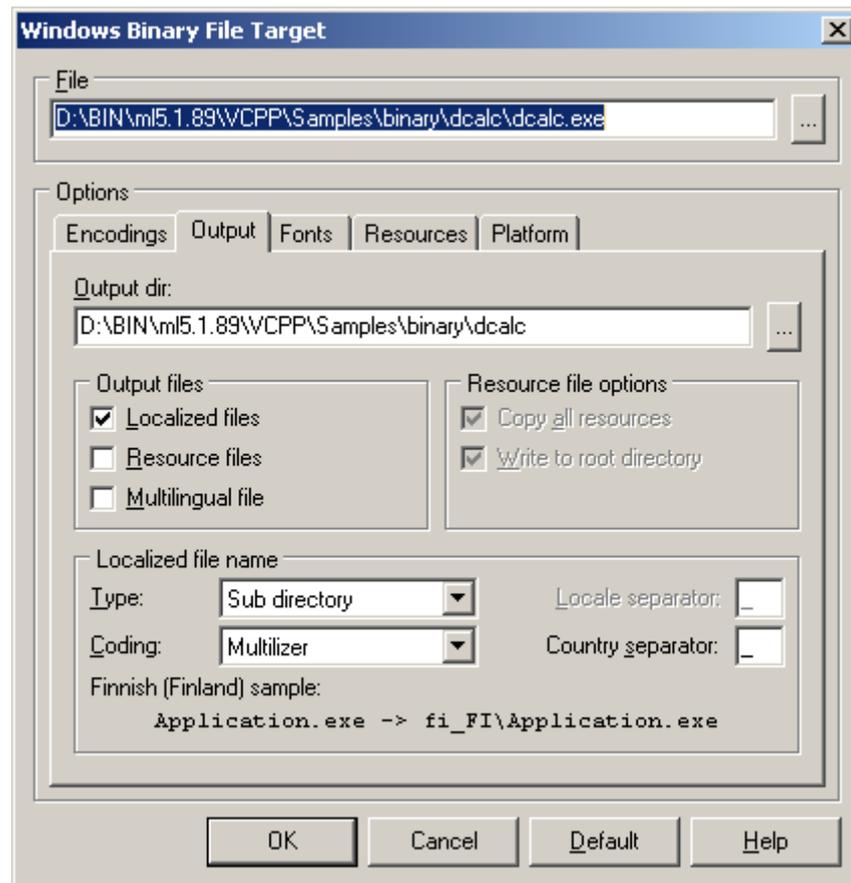


Figure 77: Output options for localized files.

The following example figure shows the files that Multilizer uses on the C++ binary localization process in Windows:

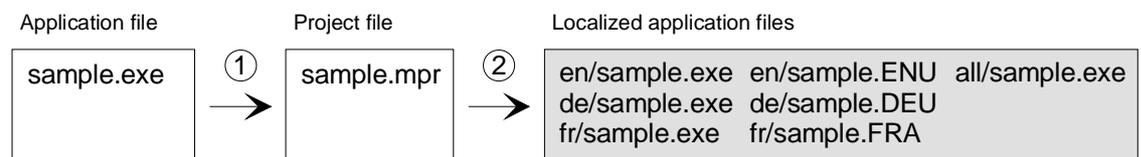


Figure 78: The files of the binary C++ localization process in Windows.

When deploying the application, you can either deploy the localized binary file (e.g., de\sample.exe), the multilingual binary file (all\sample.exe), or the original binary file (e.g., sample.exe) and the localized resource DLL(s) (e.g., de\sample.DEU).

By default, Multilizer creates localized files. It creates subdirectories under the original file folder containing the localized file(s). I.e., there might be subfolders called ..\en\



Besides applications built with C++, Multilizer binary localization type can be applied to applications compiled with other compilers. Multilizer automatically detects projects compiled with Delphi, C++Builder, and Visual Basic. Refer to the corresponding tutorials, if you localize applications with any of the aforementioned compilers.

Fonts

On Fonts tab, the user can specify the font of the localized software. Furthermore, rules can be set to apply fonts on certain conditions.

Default settings are recommended, because they are strictly based on Windows standards.

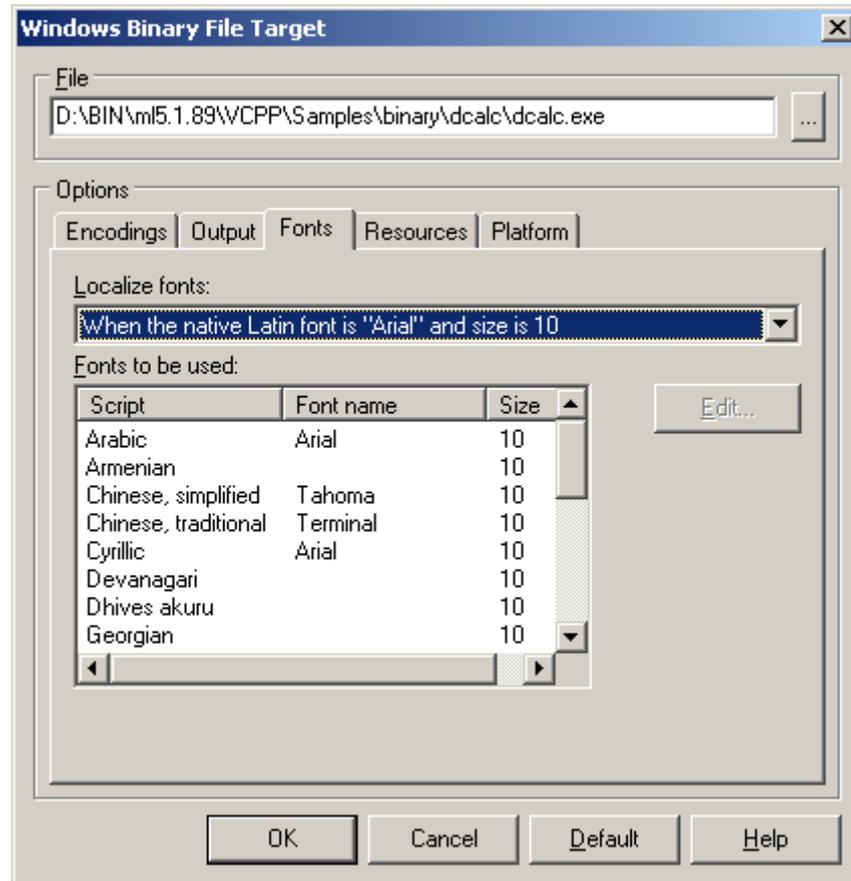


Figure 79: Font options for localized software.

Resources

On Resources tab, you can specify what kind of resources you want to localize.

Multilizer detects the resources of the Windows executable, and lets the user choose what to localize. Typically dialogs and string resources are localized, because both contain texts that need translation.

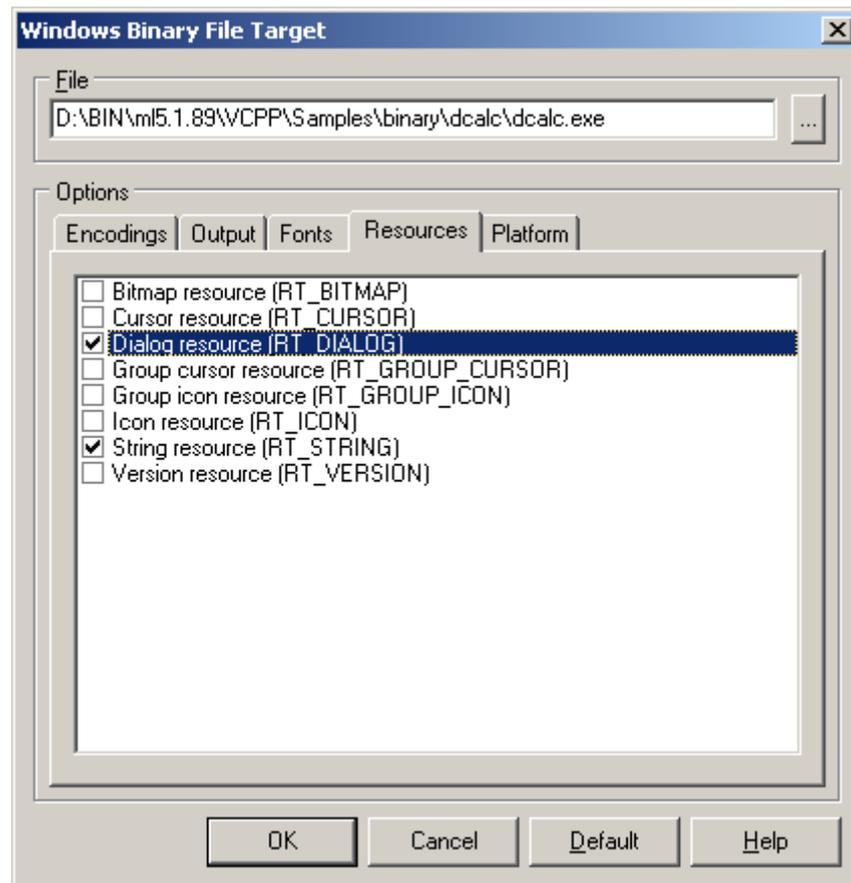


Figure 80: Specifying the resources types to localize.

Platform

Platform tab shows the target platform of the localized software. Multilizer sets the values automatically when creating a project, so these settings shouldn't be changed.

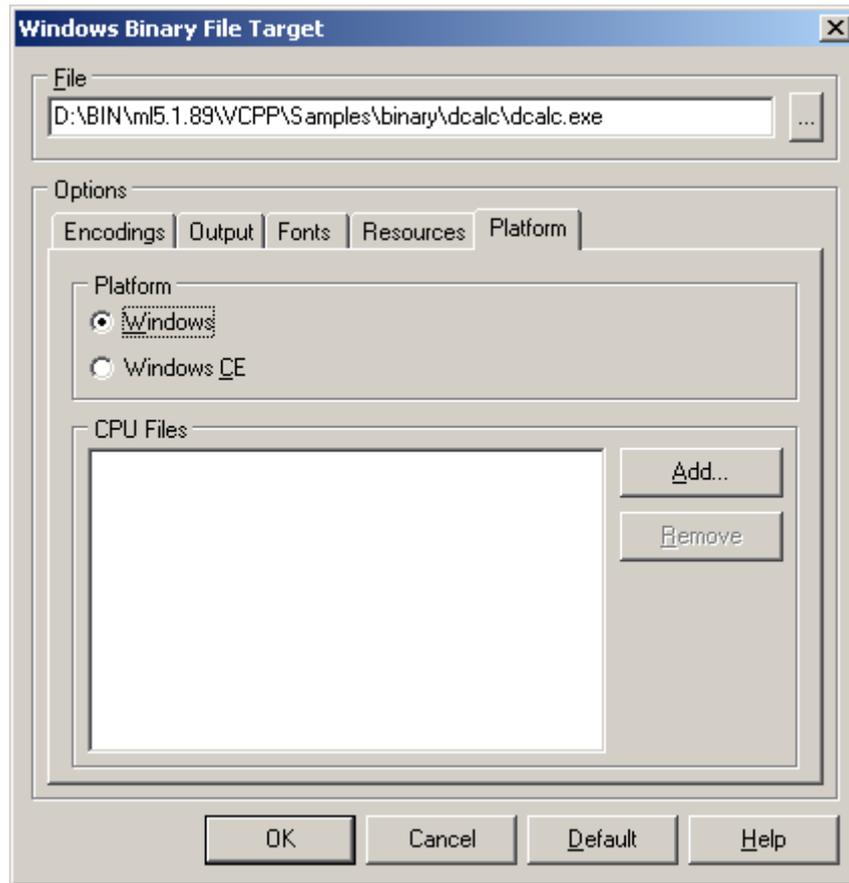


Figure 81: Specifying the platform of localized software.

Translate Project

For testing purposes, you can translate the software by using pseudo languages (C.f. Pseudo language, p. 68); right-click language column, choose properties, and select the pseudo language options. This will fill the translation grid with pseudo language translations.

Wysiwyg

Besides just translating, Multilizer allows editing of UI (user interface) elements. This is useful in cases, where the original software was not designed for localization, and translated strings don't fit in the placeholders.

Translation with Multilizer showing visually the changes in UI is referred to as Wysiwyg in this manual.

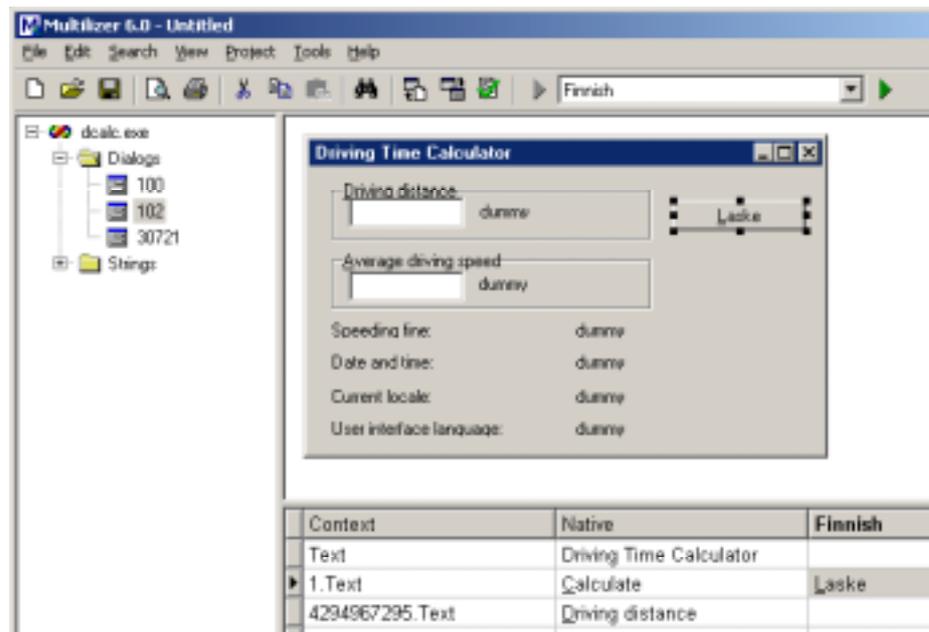


Figure 82: Localizing forms visually.

More info



Refer to the following parts of the manual for more information on translating software, and sharing translation work between team members:

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized application files by choosing **Project | Build Localized Files**. This creates the localized files based on the target options (→Output, p. 85).

Finally, you can run the localized application by right-clicking the column header (e.g., Finnish) and by choosing Run.

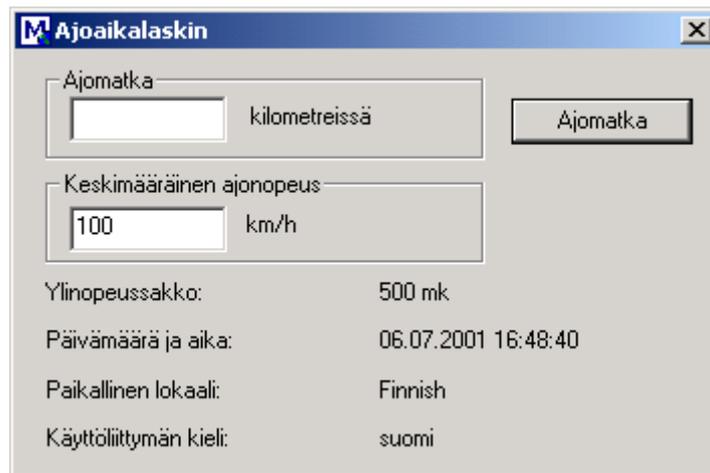


Figure 83: Localized Dcalc application (Windows).

Localized Windows CE version should look like this:



Figure 84: Localized Dcalc application (Windows CE).

9

VCL Tutorial

This tutorial describes localization of Delphi and C++Builder software.

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for VCL
Sample(s):	<mldir>/vcl/delphi/dcalc <mldir>/vcl/CBuilder/dcalc
Tutorial(s):	–

- To learn the basics of localization of Delphi/C++Builder software and localization prerequisites, go through the entire tutorial. It requires that you have Delphi (2-7) or C++Builder installed on your computer.
→ Introduction, p. 92.
- To learn how to use Multilizer for Delphi/C++Builder software localization, you can localize any of the sample applications.
→ Create Multilizer Project, p. 104.

Introduction

This tutorial is written for Delphi 7, and C++Builder 6. Using an older version is almost identical. Some menu commands may vary.

Compiled VCL applications (.exe or .dll) contain resource data. When doing binary localization, Multilizer scans the original binary files and creates localized binary files as copies of the original files. The following picture describes the binary localization process:

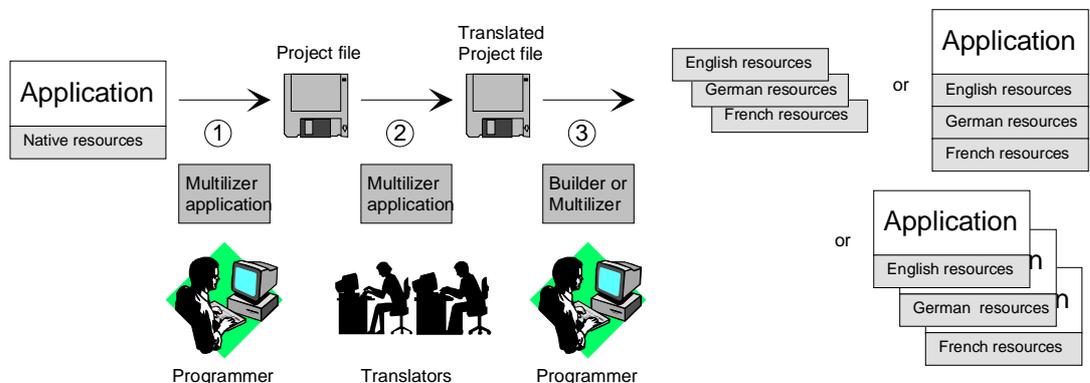


Figure 85: Binary localization process of a VCL application.

The programmer uses Multilizer to extract strings from the original binary file(s) (1). Multilizer saves these strings to a project file. The programmer uses Multilizer to send the

project file to the translator(s) who uses Multilizer to translate the project file, and then sends the translated project file back to the programmer (2). The programmer then uses Multilizer to create localized binary files (3). As a result, there will be one resource file for each localized language. Multilizer can also produce a single multilingual binary file containing all the languages of the project, or one binary file for each language.

Open Tutorial Application

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize. The `<mldir>\vcl\<compiler>\dcalc\dcalc.dpr` contains the project file of the Dcalc sample application. `<compiler>` is *Delphi*, or *CBuilder* depending on your compiler. Compile and run the application. By default, Dcalc uses the English language.

The application should look like this:

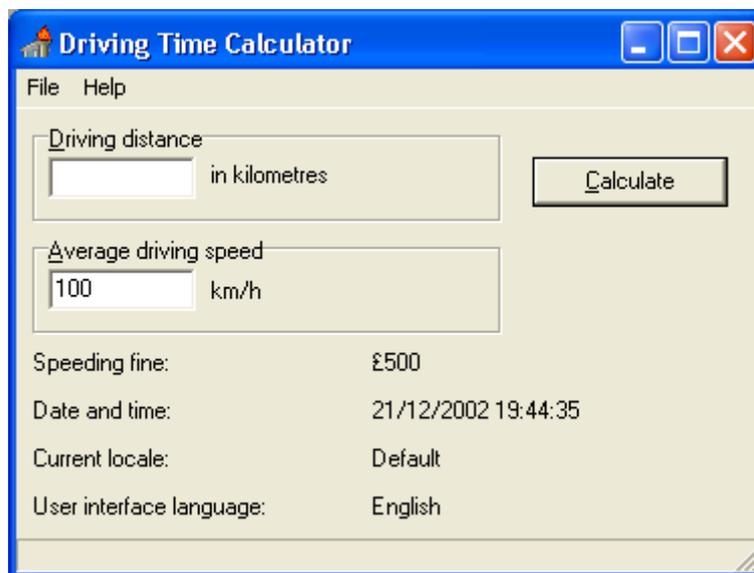


Figure 86: Dcalc application using an English user interface.

The user interface language is English (UK) and the application formats currency, date and time according to English (UK) standards. In the following chapters, we will turn Dcalc into a truly multilingual application, step-by-step.

Internationalization

This chapter describes the internationalization process. Internationalization is the process of generalizing a product so that it can handle multiple languages and cultural conventions without the need for re-design; re-engineering source code so that products and applications are compatible with country-specific operating systems and software. Internationalization (I18N) takes place at the level of program design and document development.

Open the Tutorial application,
`<mldir>\<compiler>\Samples\Tutorial\dcalc.dpr`.

Study the source code of the application to familiarize yourself of its behavior. It is not a complex application, so you should get the idea fairly quickly.

The main form contains some labels that are locale dependent. The label on the right side of the edit box contains the distance unit. Not every country uses kilometers. That's

why we must update the label at run-time using a resource string, to make sure that a correct unit is used. Similarly, we assign the screentip text of the edit control and the label containing the current language at run-time. We could add the initialization code into the OnCreate event of the main form but let's prepare the run-time language switch and write a separator function for the initialization.

Delphi

```

procedure TMainForm.InitFrom;
resourcestring
  SLanguage = 'English';
  SDefault = 'Default';

  SMetricDistanceHint = 'Give the driving distance in kilometres';
  SMetricSpeedHint = 'Give the average driving speed in kilometres per
hour';
  SMetricDistanceLabel = 'in kilometres';
  SMetricSpeedLabel = 'km/h';

  SUsDistanceHint = 'Give the driving distance in miles';
  SUsSpeedHint = 'Give the average driving speed in miles per hour';
  SUsDistanceLabel = 'in miles';
  SUsSpeedLabel = 'mph';
begin
  Application.OnHint := DisplayHint;

  SpeedingFine.Caption := Format('%m', [500.0]);
  CurrentTime.Caption := DateTimeToStr(Now);

  CurrentLocale.Caption := GetLocaleStr(
    LOCALE_USER_DEFAULT,
    LOCALE_SNATIVELANGNAME,
    SDefault);

  CurrentLanguage.Caption := SLanguage;

  if GetMeasurementSystem = ivmsMetric then
  begin
    DistanceEdit.Hint := SMetricDistanceHint;
    DistanceLabel.Caption := SMetricDistanceLabel;

    SpeedEdit.Text := '100';
    SpeedEdit.Hint := SMetricSpeedHint;
    SpeedLabel.Caption := SMetricSpeedLabel;
  end
  else
  begin
    DistanceEdit.Hint := SUsDistanceHint;
    DistanceLabel.Caption := SUsDistanceLabel;

    SpeedEdit.Text := '65';
    SpeedEdit.Hint := SUsSpeedHint;
    SpeedLabel.Caption := SUsSpeedLabel;
  end;
end;

```

C++Builder

```

void __fastcall TMainForm::InitForm()
{
  Application->OnHint = DisplayHint;

  SpeedingFine->Caption = Format("%m", OPENARRAY(TVarRec, (500.0)));
  CurrentTime->Caption = DateTimeToStr(Now());

  CurrentLocale->Caption = GetLocaleStr(
    LOCALE_USER_DEFAULT,
    LOCALE_SNATIVELANGNAME,
    LoadStr(SDefault));

  CurrentLanguage->Caption = LoadStr(SLanguage);

  if (GetMeasurementSystem() == ivmsMetric)
  {
    DistanceEdit->Hint = LoadStr(SMetricDistanceHint);
    DistanceLabel->Caption = LoadStr(SMetricDistanceLabel);

    SpeedEdit->Text = "100";

```

```

    SpeedEdit->Hint = LoadStr(SMetricSpeedHint);
    SpeedLabel->Caption = LoadStr(SMetricSpeedLabel);
}
else
{
    DistanceEdit->Hint = LoadStr(SUsDistanceHint);
    DistanceLabel->Caption = LoadStr(SUsDistanceLabel);

    SpeedEdit->Text = "65";
    SpeedEdit->Hint = LoadStr(SUsSpeedHint);
    SpeedLabel->Caption = LoadStr(SUsSpeedLabel);
}
}

```

The most important part of internationalization (I18N) is resourcing. This means removing all hard coded strings from the application's source code. Traditionally, hard coded strings are turned into resources by moving the strings from the actual code into resource strings.

Delphi makes this extremely easy because of its built-in support for resource strings, with the `resourcestring` clause. It defines one or more resource strings. The `resourcestring` block contains the resource strings used in the function. If you are not familiar with resource strings in Delphi, refer to the VCL documentation. To put it briefly, you use them almost exactly as you would use string constants.

With C++Builder, things are a little bit more complicated because you have to use the old-fashioned resource scripts. The following paragraph contains the resource script header file `dcalcres.h`. It specifies the ID of each resource string.

C++Builder

```

#define SAboutMsg          0
#define SLanguage         1
#define SDefault          2

#define SMetricDistanceHint  3
#define SMetricSpeedHint    4
#define SMetricDistanceLabel 5
#define SMetricSpeedLabel   6

#define SUsDistanceHint    7
#define SUsSpeedHint       8
#define SUsDistanceLabel   9
#define SUsSpeedLabel      10

#define SInvalidDistance   11
#define SInvalidSpeed      12
#define SCalculateMsg0     13
#define SCalculateMsg1     14
#define SCalculateMsgN     15

```

The resource script file is shown below.

```

C++Builder      #include "dcalcres.h"

STRINGTABLE
BEGIN
    SAboutMsg "Dcalc is a multilingual application that calculates the
average driving time";
    SLanguage "English";
    SDefault "Default";

    SMetricDistanceHint "Give the driving distance in kilometres";
    SMetricSpeedHint "Give the average driving speed in kilometres per
hour";
    SMetricDistanceLabel "in kilometres";
    SMetricSpeedLabel "km/h";

    SUsDistanceHint "Give the driving distance in miles";
    SUsSpeedHint "Give the average driving speed in miles per hour";
    SUsDistanceLabel "in miles";
    SUsSpeedLabel "mph";

    SInvalidDistance "\"%s\" is not a valid distance!";
    SInvalidSpeed "\"%s\" is not a valid speed!";
    SCalculateMsg0 "The average driving time is %d minutes.";
    SCalculateMsg1 "The average driving time is one hour and %d minutes.";
    SCalculateMsgN "The average driving time is %0:d hours and %1:d
minutes.";
END

```

The second code section in the beginning section of the `IniForm` function formats the speed and time in locale independent ways. The `Format` and `DateTimeToStr` functions convert the value to a string value using the formatting rules of the current locale.

The next code section sets the caption of the current locale label to match the current locale. The name is given in the native language of the locale.

The next code section sets the caption of the current language label to match the current language. The `SLanguage` resource string contains the name of the language in its native language (e.g., English, Deutsch, suomi, svenska, etc).

The last code section sets the initial values, labels, and screentips for distance and speed. The metric system uses kilometers and km/h. The US system uses miles and mph. Unit `lv18N` contains the `GetMeasurementSystem` function that returns the measurement system of the current locale.

To do the first initialization, we call the initialization function from the `OnCreate` event.

```

Delphi          procedure TMainForm.FormCreate(Sender: TObject);
begin
    InitForm;
end;

```

```

C++Builder     void __fastcall TMainForm::FormCreate(TObject *Sender)
{
    InitForm();
}

```

The `CalculateButtonClick` event needs a little bit more rewriting. Let's study the code that generates the driving distance message:

Delphi `'The average driving time is ' + IntToStr(hours) + ' hours and ' + IntToStr(minutes) + ' minutes.'`,

C++Builder `"The average driving time is " + IntToStr(hours) + " hours and " + IntToStr(minutes) + " minutes."`,

This seems to be just ok, but it will actually make the localization hard or even impossible. The reason is that the above logic assumes that the message always starts with the “The average driving time is “ string, and then contains the hours, hour label, minutes, and the minute label. However, not all languages use the same order of words in a sentence. For example, the order might be: minute label, minutes, hour label, hours, and the text string. Reordering of the parts of the message is impossible if we use the code shown above.

Fortunately, we can use VCL’s Format function. It uses message pattern that contains placeholders for the dynamic parameters. At run-time, the function combines the pattern with the parameters to compose the message. Because the pattern is a single string, it can be added to the resource strings, and it can then be translated as a single item. The above code after the internationalization is:

Delphi `Format(SCalculateMsg, [hours, minutes]),`

C++Builder `Format(LoadStr(SCalculateMsg), OPENARRAY(TVarRec, (hours, minutes))),`

SCalculateMsg is the pattern, and the hours and minutes are parameters.

The next step is to internationalize the calculate event. The following line of code contains the Calculate event:

Delphi

```

procedure TMainForm.CalculateButtonClick(Sender: TObject);
resourcestring
  SInvalidDistance = '"%s" is not a valid distance!';
  SInvalidSpeed = '"%s" is not a valid speed!';
  SCalculateMsg0 = 'The average driving time is %d minutes.';
  SCalculateMsg1 = 'The average driving time is one hour and %d
minutes.';
  SCalculateMsgN = 'The average driving time is %0:d hours and %1:d
minutes.';
var
  str: String;
  distance, speed, hours, minutes: Integer;
begin
  distance := StrToIntDef(DistanceEdit.Text, -1);
  if distance < 0 then
    begin
      MessageDlg(
        Format(SInvalidDistance, [DistanceEdit.Text]),
        mtError,
        [mbOK],
        0);
      DistanceEdit.SetFocus;
      Exit;
    end;

    speed := StrToIntDef(SpeedEdit.Text, -1);
    if speed <= 0 then
      begin
        MessageDlg(
          Format(SInvalidSpeed, [SpeedEdit.Text]),
          mtError,
          [mbOK],
          0);
        SpeedEdit.SetFocus;
        Exit;
      end;

      if GetMeasurementSystem = ivmsUS then
        begin
          distance := Trunc(MILE_IN_METERS*distance/1000);
          speed := Trunc(MILE_IN_METERS*speed/1000);
        end;

        hours := distance div speed;
        minutes := Round(60*(distance mod speed)/speed);

        case hours of
          0: str := Format(SCalculateMsg0, [minutes]);
          1: str := Format(SCalculateMsg1, [minutes]);
        else
          str := Format(SCalculateMsgN, [hours, minutes]);
        end;

        MessageDlg(str, mtInformation, [mbOK], 0);
      end;

```

C++Builder

```

void __fastcall TMainForm::CalculateButtonClick(TObject *Sender)
{
  int distance = StrToInt(DistanceEdit->Text);
  if (distance < 0)
  {
    MessageDlg(
      Format(
        LoadStr(SInvalidDistance),
        OPENARRAY(TVarRec, (DistanceEdit->Text))),
      mtError,
      TMsgDlgButtons() << mbOK,
      0);
  }

```

```

        DistanceEdit->SetFocus();
        return;
    }

    int speed = StrToInt(SpeedEdit->Text);
    if (speed <= 0)
    {
        MessageDlg(
            Format(
                LoadStr(SInvalidSpeed),
                OPENARRAY(TVarRec, (SpeedEdit->Text))),
            mtError,
            TMsgDlgButtons() << mbOK,
            0);
        SpeedEdit->SetFocus();
        return;
    }

    if (GetMeasurementSystem() == ivmsUS)
    {
        distance = MILE_IN_METERS*distance/1000;
        speed = MILE_IN_METERS*speed/1000;
    }

    int hours = distance/speed;
    int minutes = float(60)*(distance%speed)/speed;

    AnsiString str;

    switch (hours)
    {
        case 0:
            str = Format(LoadStr(SCalculateMsg0), OPENARRAY(TVarRec,
                (minutes)));
            break;

        case 1:
            str = Format(LoadStr(SCalculateMsg1), OPENARRAY(TVarRec,
                (minutes)));
            break;

        default:
            str = Format(LoadStr(SCalculateMsgN), OPENARRAY(TVarRec, (hours,
                minutes)));
    }

    MessageDlg(str, mtInformation, TMsgDlgButtons() << mbOK, 0);
}

```

The hard coded error message has been replaced with the Format message.

If the selected locale uses miles instead of kilometers, we have to treat the value in the edit box as miles. Then we also have to convert distance from miles to kilometers before calculating the driving time. It is always a good idea to internally use the metric system and convert the input and output to the US system when application is run on a US locale, because that makes the calculations easier.

After we have calculated the average driving speed, we have to show it to the user. As described previously, we are going to use the Format function and message patterns. However, we want to make the message grammatically correct. That's why we need three message patterns. The first one is for the case when the time is less than an hour, another for the case when the time is between one and two hours, and last for the case when the time is two hours or more. This is because in most languages the single and plural forms are handled in different ways. For example, "one hour" vs. "two hours."

The following figure contains the message when the time is less than one hour. Note that there is no hour string present.



Figure 87: Message displayed when time is less than 1 hour.

The following figure contains the message when the time is more than one hour but less than two hours. Note that the value for hour is not given as a number but written in letters.



Figure 88: Message displayed when time is 1 hour.

The following figure contains the message when the time is more than two hours. Both hours and minutes are shown as numbers and in plural form.



Figure 89: Message displayed when time is more than 1 hour.

Even this solution is not perfect because:

- There might be a language that has a specific word for two hours. The above logic assumes that only 0, 1, and 2 or more are handled each in different ways.
- We should use the same logic for minutes as well, but this would require 3 by 3 equals 9 message patterns.

The final task left is to resource the message used in the about box:

Delphi

```

procedure TMainForm.AboutMenuClick(Sender: TObject);
resourcestring
  SAboutMsg = 'Dcalc is a multilingual application that calculates the
  average driving time';
begin
  MessageDlg(SAboutMsg, mtCustom, [mbOK], 0);
end;

```

C++Builder

```

void __fastcall TMainForm::AboutMenuClick(TObject *Sender)
{
  MessageDlg(
    LoadStr(SAboutMsg),
    mtCustom,
    TMsgDlgButtons() << mbOK,
    0);
}

```

Because we set many property values dynamically at run-time, the original design time values in the form files become obsolete. They cause no harm, but it makes the translator's job easier if we remove them. We could set those values to empty, but this would make it harder to edit the form files because the labels would no longer be visible. A good solution is to set all dynamic visible property values to "dummy."

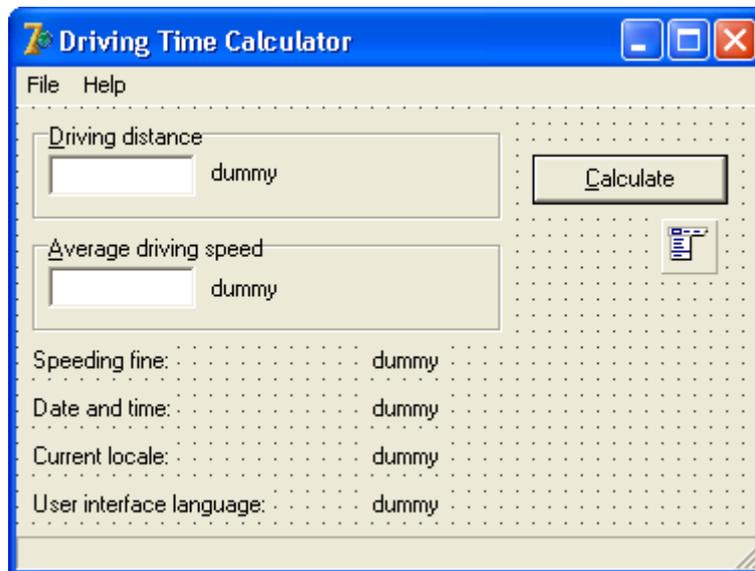


Figure 90: The internationalized Dcalc form on Delphi IDE.

Now the Dcalc application has been internationalized. Compile and run it to see that it works before moving on.

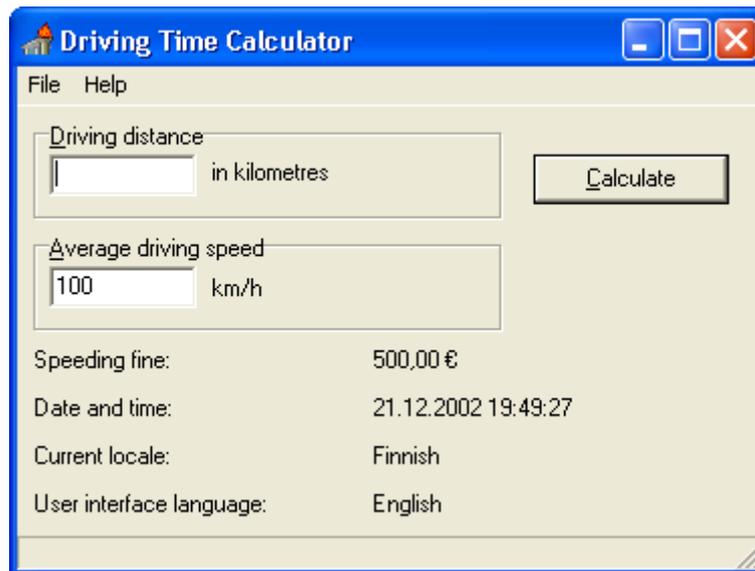


Figure 91: The internationalized Dcalc application running with Finnish locale.

This simple internationalization demonstrates three of the most important issues to take into consideration in internationalization: resourcing, dynamic messages, and unit conversions. There are quite many other things that you need to know about internationalization as well. Refer to Delphi's online help and/or an I18N book to get more information about internationalization.

Run-time language switch

The final task is to implement run-time language switch. This can be done if the resource DLLs are used. Add Language... menu to the File menu and write the following code:

Delphi

```
procedure TMainForm.LanguageMenuClick(Sender: TObject);
begin
  if SelectResourceLocale then
  begin
    InitForm;
    SetCurrentDefaultLocaleReg;
  end;
end;
```

C++Builder

```
void __fastcall TMainForm::LanguageMenuClick(TObject *Sender)
{
  if (SelectResourceLocale())
  {
    InitForm();
    SetCurrentDefaultLocaleReg();
  }
}
```

The `SelectResourceLocale` function shows a dialog box that shows the available resource language and loads the selected resource DLL. This will remove all run-time modifications of the forms. That's why we have to call the `InitForm` function again. Finally, we save the selected language to the system registry.

When starting the application, VCL is going to select the resource DLL matching to the current locale settings of the user. We want better control over the initial language. The first choice would be a command line parameter (e.g., `dcalc.exe en_US`). If that is not present, then we would like to use the previous language stored in the system registry under the `HKEY_CURRENT_USER\Software\Borland\Locales` key. This registry key is the built-in feature of VCL. Only if that does not exist, we would like to use the default language. Add the following code in the initialization part of the main form:

Delphi

```

var
  locale: Integer;
initialization
  if GetCommandLineLocale(locale) then
    SetNewResourceDll(locale);
end;

```

C++Builder

```

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
  try
  {
    int locale;
    if (GetCommandLineLocale(locale))
      SetNewResourceDll(locale);

    Application->Initialize();
    Application->CreateForm(__classid(TMainForm), &MainForm);
    Application->Run();
  }
  catch (Exception &exception)
  {
    Application->ShowException(&exception);
  }
  return 0;
}

```

Enable DRC generation in Delphi

We will perform one more task to make the localization easier. When using the resourcestring clause, Delphi puts the strings to the string resources automatically. However, it does not let you choose what string ids will be used. In addition, Delphi will most likely change those ids next time you compile your application. What remains constant are the resource string names (e.g., `SInvalidDistance`). Unfortunately, the compiled binary file (.exe or .dll) does not contain the resource string name but only the string ids. Fortunately, it is possible to make Delphi create a resource string file that contains all the resource string's name and ids used by the application. To create such a file, open a Delphi project, choose **Project | Options**, select the Linker tab, and check Detailed in the Map file radio group. Rebuild the application by choosing **Project | Build DCalc**. Delphi generates the resource string file called `dcalc.drc`.

Create Multilizer Project

In order to localize Windows software developed with Delphi/C++Builder, you have to create a Multilizer project. This is described in the first part of the manual, in the chapter 'Create Project,' p. 11.

Delphi-specific settings

For Delphi localization projects, Project Wizard will display one extra page with target-specific information. At a minimum, the user should specify here the location of the Delphi-project and DRC-file, if available. This and other Delphi-specific settings are discussed in the next chapter.

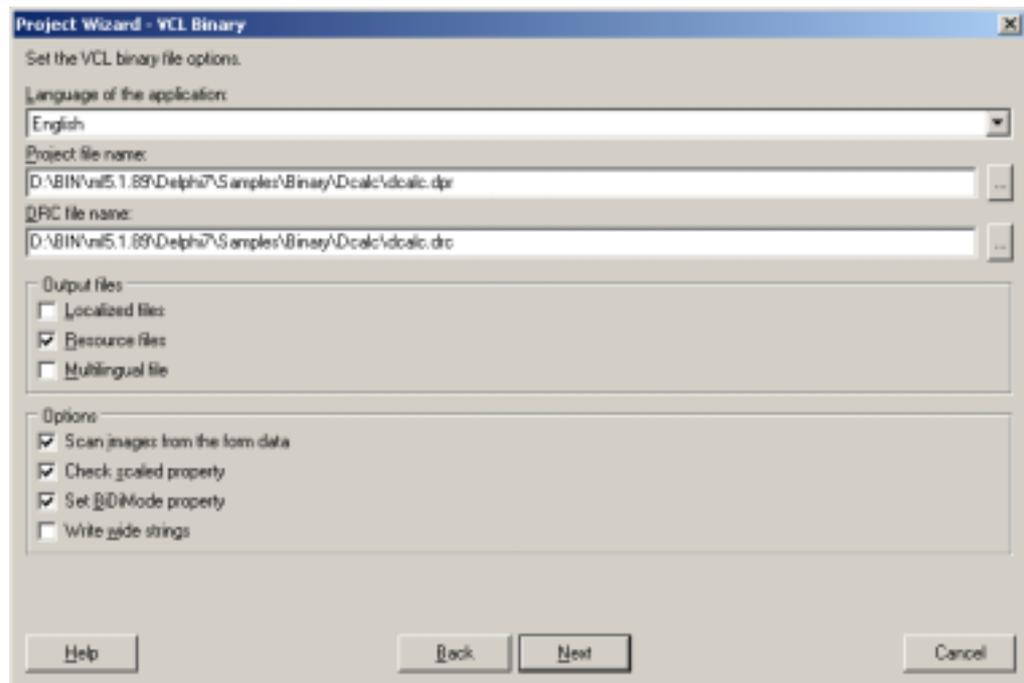


Figure 92: Delphi-specific settings for VCL binary target.

Integrated Translation Environment

If you have previously used Borland Integrated Translation Environment (ITE) to localize your application, the following message box will appear in Project Wizard:

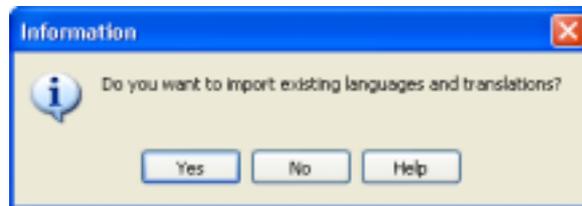


Figure 93: Message box telling that there are existing ITE translations.

Press **Yes** to import the initial languages and translations from ITE-generated resource DLLs. From now on you do not have to use ITE anymore. You can delete ITE directories, projects files, and project groups.

Specify Localization options

After finishing the Wizard, you have to specify the localization options for the software. Normally default options are the most useful – and follow Delphi/C++Builder suggested way of localization – but in this tutorial, we will review all options.

Right-click the localization target in Project Tree, and click properties to see Delphi Target options.

All Delphi-specific options are gathered under the Delphi Target dialog. If there are many targets in one project, you can set different localization options to all, if needed.



Project

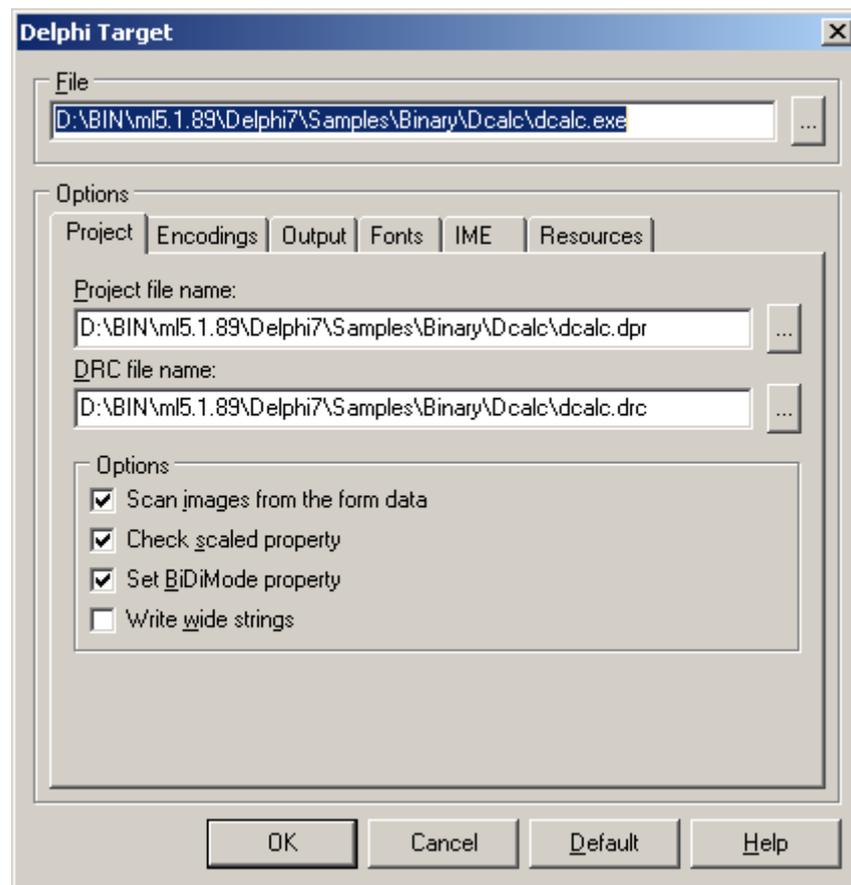


Figure 94: Delphi target options.

Project file name

Project file refers to the Delphi project from which the executable was compiled. If the project file is specified, Multilizer is able to automatically resolve visual inheritance of the software. This enables visual inheritance in the Multilizer project, which minimizes translation work and maximizes translations consistency.

DRC file name

DRC-file is needed to resolve Delphi resource string names. If DRC-file is specified, Multilizer will assign resourcestring constant names to the localization context. In addition, the context will include the name of the unit, where resourcestring is defined.



If DRC-file is not specified, Multilizer uses resource id (integer) as the context. Because Delphi-compiler changes these ids on re-compilation of the software, localization context changes, which can result in loss of translations.

DRC-file is created by Delphi compiler (→ Enable DRC generation in Delphi, p. 104).

Options

Scan images from the form data enables Multilizer to scan images (glyphs) from *form resource* data. Most VCL components with images store the bitmaps in form data, and checking this option enables localization of them.

Check scaled property enables Multilizer to ensure that scaled property of forms is set to false. This prevents VCL from scaling the form during run-time, which occurs if a different script is applied than on design-time.

Set BiDi Mode Property enables Multilizer to set the BiDi Mode Property for RTL (Right-to-Left) languages.

Write Wide strings enables Multilizer to write strings using Unicode encoding. If this option is unchecked, strings are written using the encoding as in the original form.

Encodings

Encodings tab lets you specify codepages for target languages. In addition, you can force Multilizer to read the localization target with certain language and codepage settings.

Normally default values should be used; they are based on the information that Multilizer detects from the Windows software.

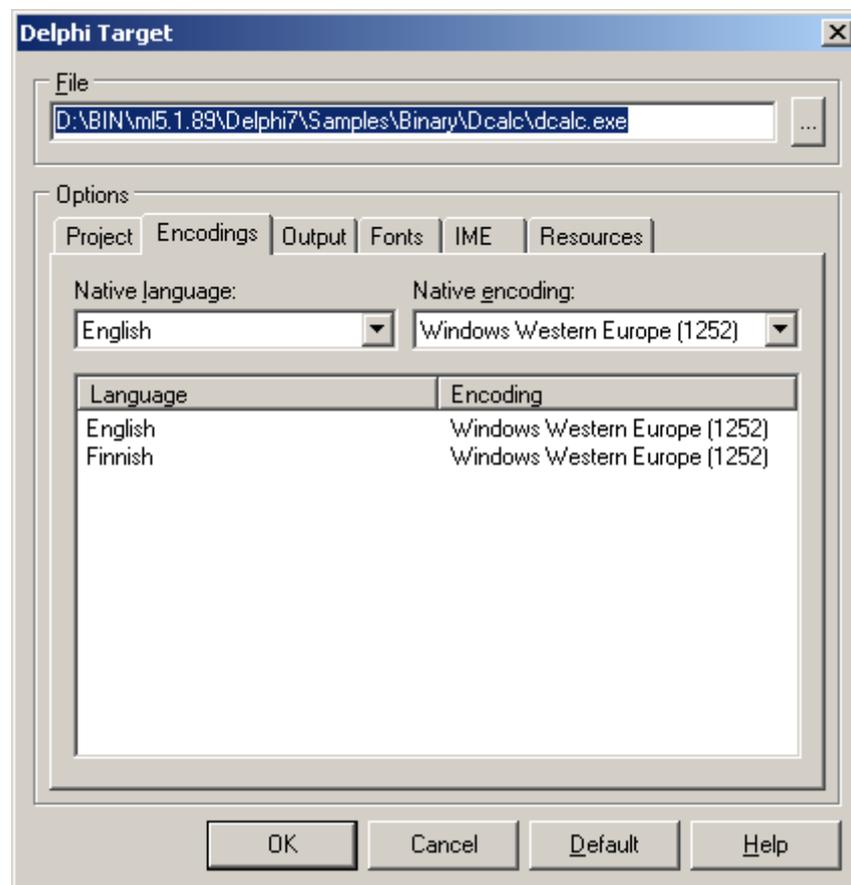


Figure 95: Encodings for target languages.

Output

The far most important option in localization is to specify location and type of localized files.

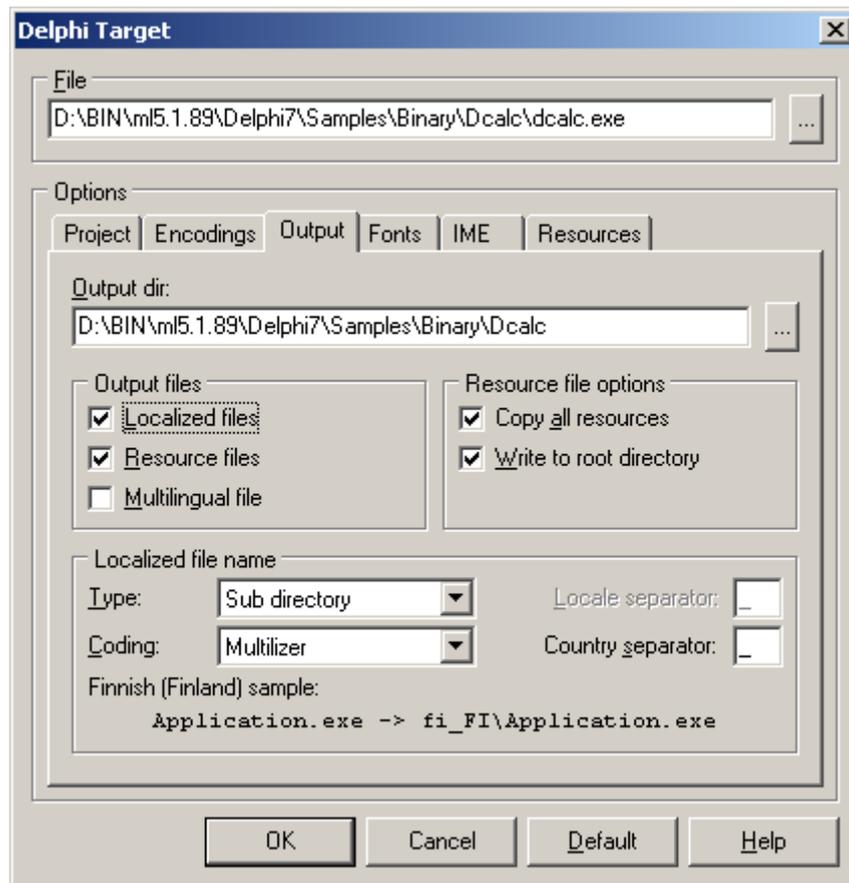


Figure 96: Output options for localized Delphi software.

The following figure shows the files that Multilizer uses in the binary localization process of a VCL application:

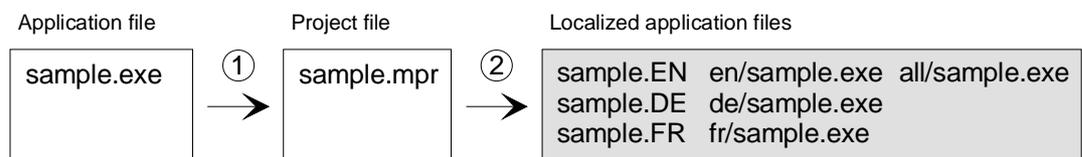


Figure 97: The files of the binary localization process of a VCL application

When deploying the application, you can either deploy the original application file with the selected resource file(s), the localized application file(s), or the multilingual application file.

By default, Multilizer creates resource files. This is the way of localizing Delphi/C++Builder recommended by Borland.



If you choose Resource files, you can enable run-time language switching in the software (→ Internationalization, p. 93).

Fonts

On Fonts tab, the user can specify the font of the localized software. Furthermore, rules can be set to apply fonts on certain conditions.

Default settings are recommended, because they are strictly based on Windows standards.

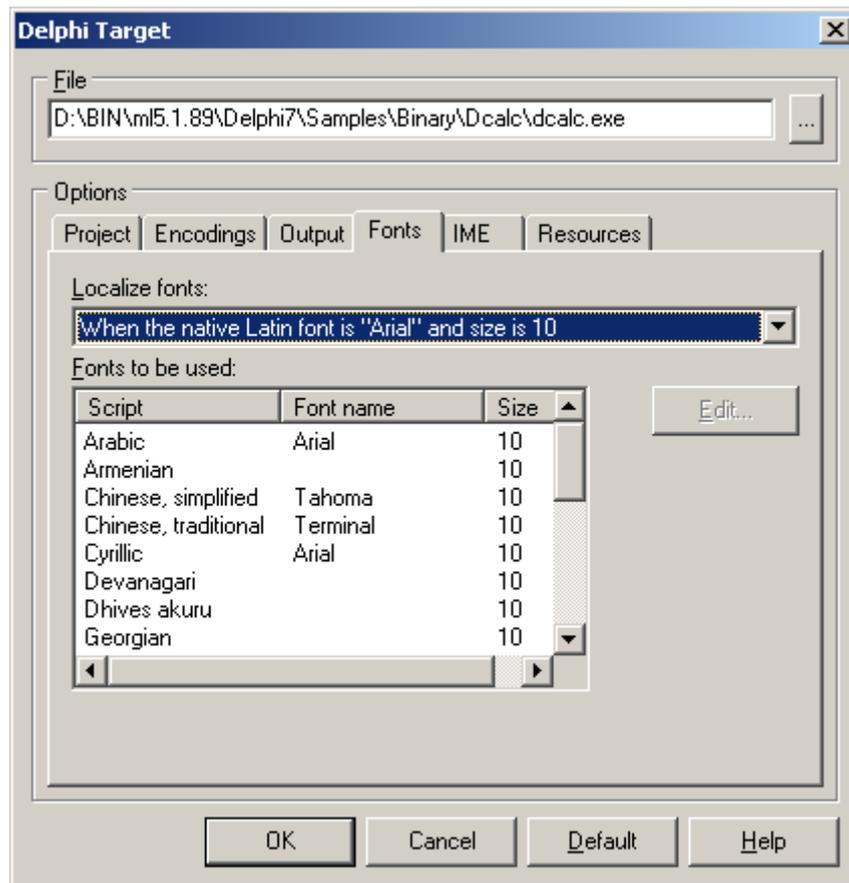


Figure 98: Font options for localized software.

IME

IME (Input Method Editor) settings specify the input method editor (IME) to use for converting keyboard input to Asian language characters.

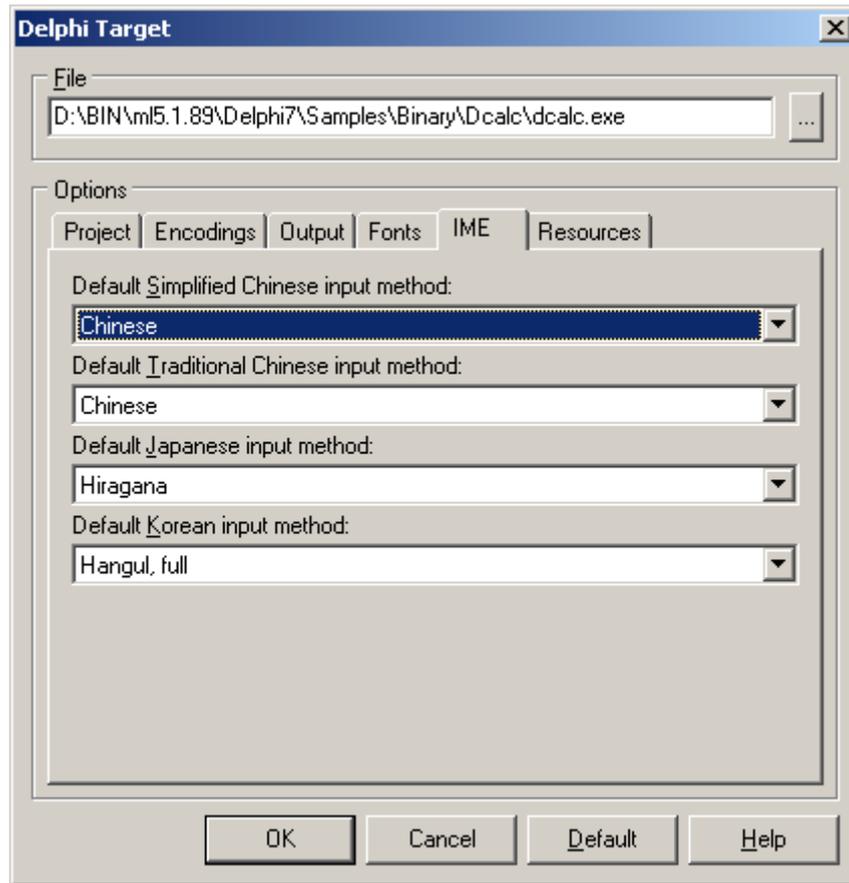


Figure 99: IME options for software localized to Far Eastern languages.

The IME settings here will apply corresponding value to IMEMode property in the software localized to Chinese, Japanese, and Korean.

Resources

On Resources tab, you can specify what kind of resources you want to localize. Multilizer detects the resources of the executable, and lets the user choose what to localize.

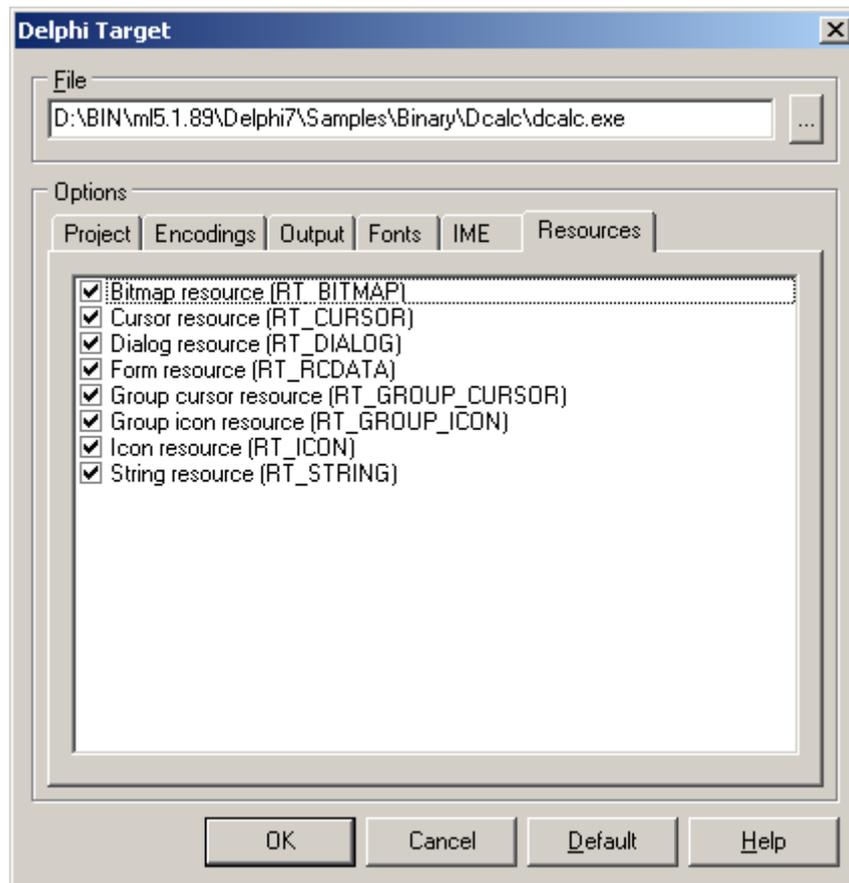


Figure 100: Specifying the resources to be localized.

In Delphi and C++Builder projects, forms are stored in form resource. Therefore, at least *Form resource* and *String resource* should be checked, in order to localize the texts of the software.

VCL components store a lot of non-localizable data in strings (string properties). Multilizer excludes a lot of this kind of strings automatically by applying rules based on component names and properties. These settings are configurable (→ Excluding Properties from localization, p. 113).

Translate Project

For testing purposes, you can translate the software by using pseudo languages (C.f. Pseudo language, p. 68); right-click language column, choose properties, and select the pseudo language options. This will fill the translation grid with pseudo language translations.

Wysiwyg

Besides just translating, Multilizer allows editing of UI (user interface) elements. This is useful in cases, where original software was not designed for localization, and translated strings don't fit in the placeholders.

Translation with Multilizer showing visually the changes in UI is referred to as Wysiwyg in this manual. Wysiwyg is enabled both in forms editing as well as menu editing, as shown in the following images:

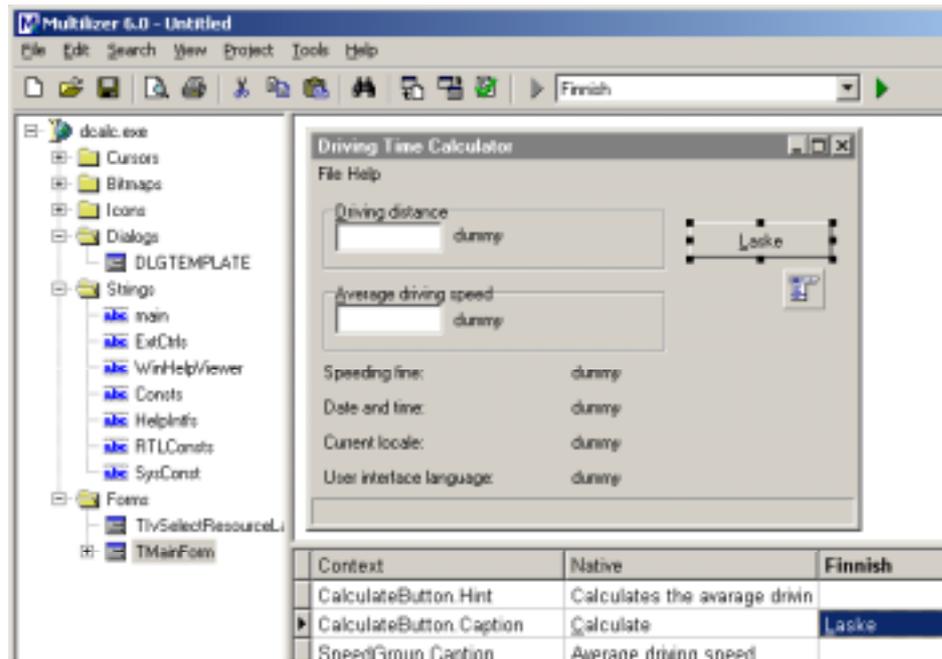


Figure 101: Localizing Delphi forms visually in Multilizer.

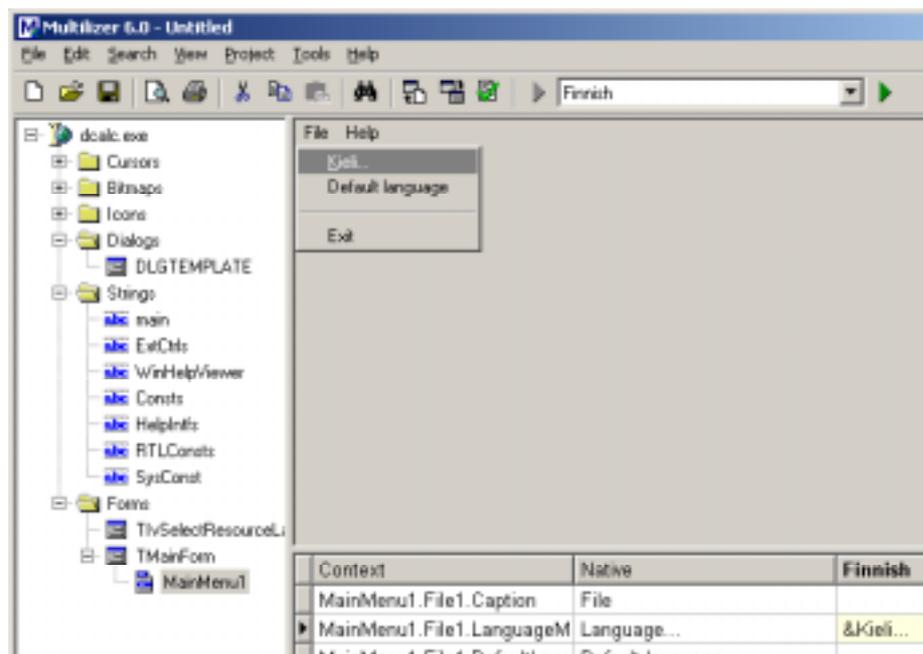


Figure 102: Localizing menus visually.

More info



Refer to the following parts of the manual for more information on translating software, and sharing translation work between team members:

- Pre-translate project, p. 25
- Prepare project for translation, p. 26

- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized application files by choosing **Project | Build Localized Files**. This creates the localized files based on the target options (→ Output, p. 107).

Finally, you can run the localized application by right-clicking the column header (e.g., Finnish) and by choosing Run.

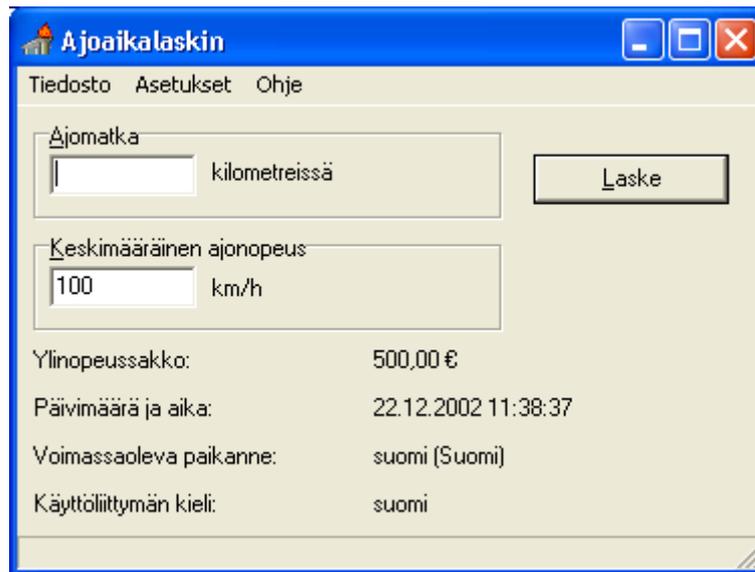


Figure 103: Running localized software.

Excluding Properties from localization

Not all string properties are intended for localization. Localization of them can even result in a crash of the software. In order to prevent this from happening, Multilizer excludes by default some of the string properties.

In order to exclude strings from localization, you can:

- Specify what string-type properties are excluded for a VCL component
- Specify what properties are always excluded

These settings will prevent Multilizer from adding them in the localization project.

The settings discussed here affect the way that Multilizer scans *form resource* data.

Exclude properties by components

To exclude properties by components, choose **Tools→Options...**, select **Delphi and C++Builder, Components**.

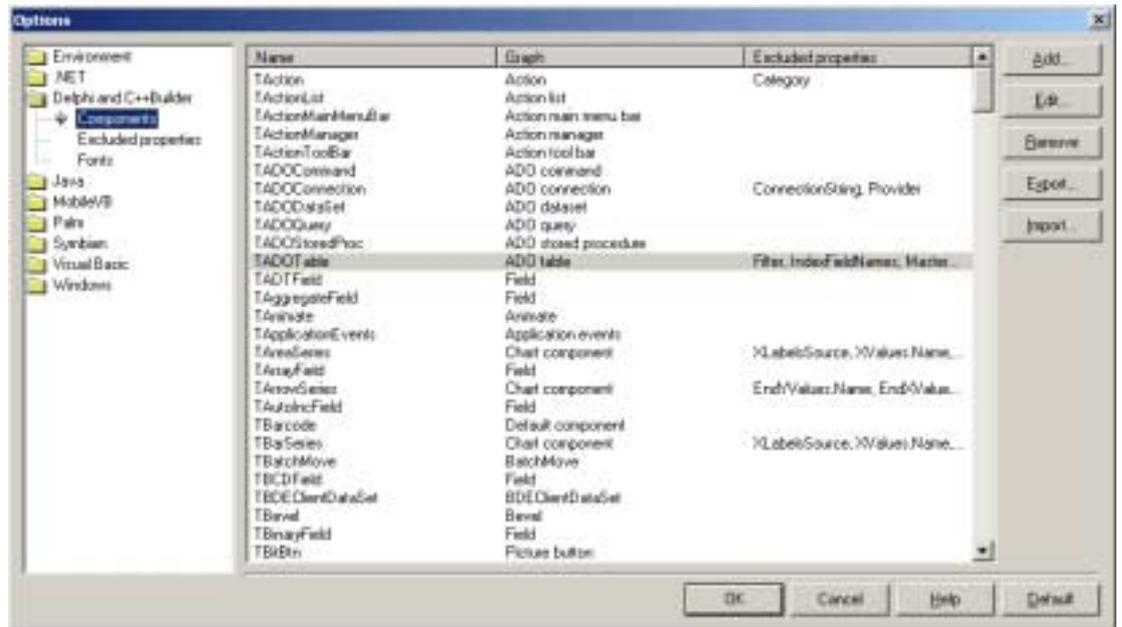


Figure 104: Excluding properties by components in VCL binary localization.

This dialog lets you specify the excluded properties of specific components. You can specify one or more properties that are excluded from scanning for each component.

This dialog is also used to configure the visual representation of each component (→Visual Representation, p. 115).

Exclude properties by name

To exclude properties by name, choose **Tools→Options...**, select **Delphi and C++Builder, Excluded properties**.

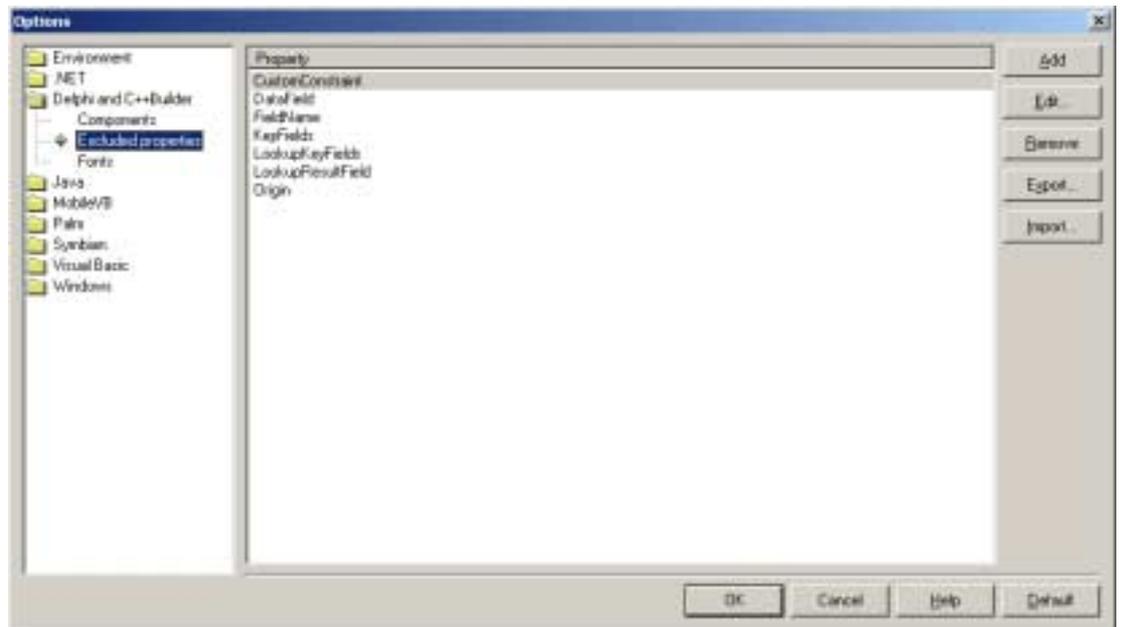


Figure 105: Excluding properties by name in VCL binary localization.

This dialog lets you specify the properties that are not localized.

Visual Representation

Delphi and C++Builder are component-based RAD (Rapid Application Development) programming environments, where user interfaces are designed with components.

There are tens of standard components, and more than a thousand of third-party components. In order to display any component correctly, Multilizer allows users to configure the visual representation of any VCL component.

All standard VCL components and most common 3rd-party components are defined already.

Components whose visual representation is not defined are shown with yellow placeholders in Wysiwyg.

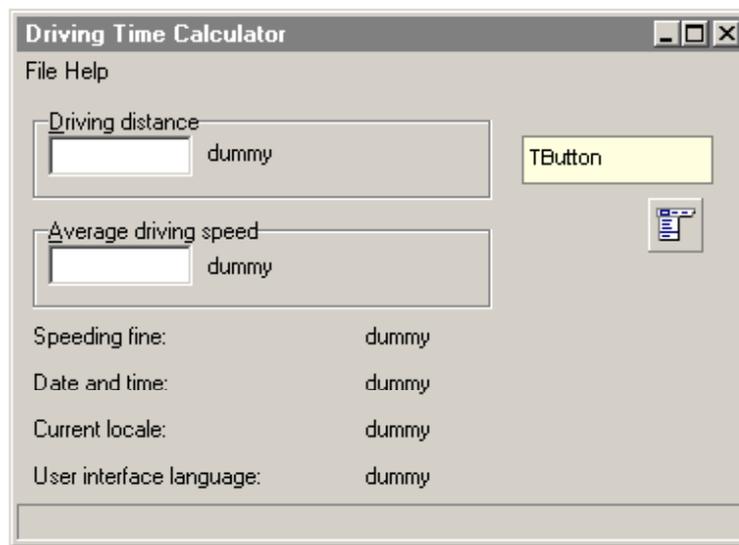


Figure 106: Visual form editor with an unknown VCL component.

To exclude properties by components, choose **Tools**→**Options...**, select **Delphi and C++Builder, Components**. Choose **Add..** to add new component.

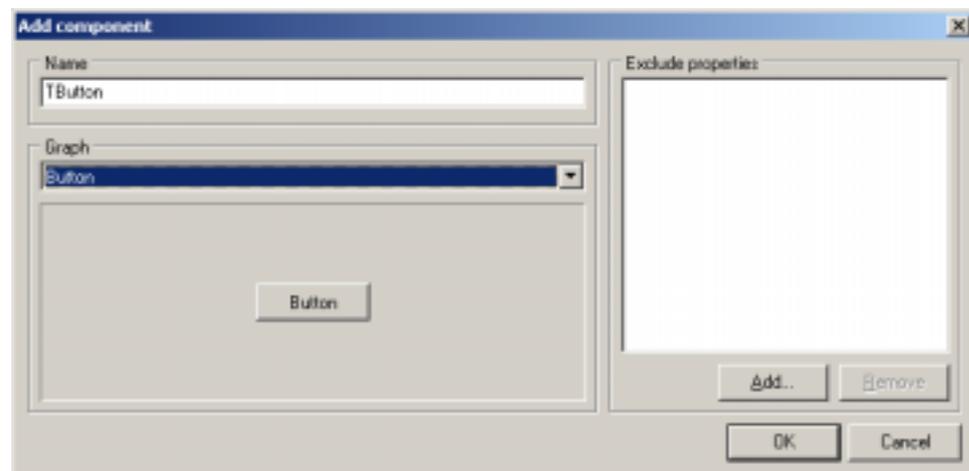


Figure 107: Specifying visual representation for VCL control.

Write the component name as specified in Delphi/C++Builder. Choose from Graph the appropriate representation; a preview is shown automatically.

After the next re-scan of project selected, visual representation is applied in Wysiwyg, as shown in the next picture.

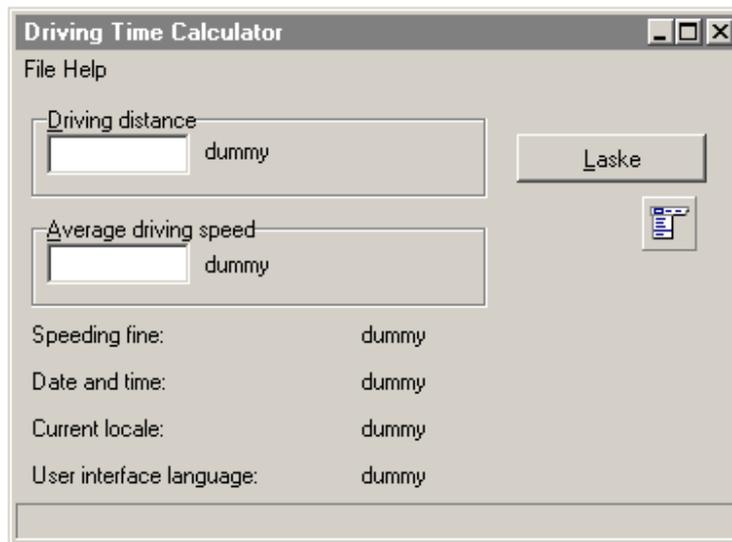


Figure 108: Visual Editor recognizing all VCL controls.



Visual Representation configurations can be shared between team members; you can both import and export the settings as *Multilizer item files*.

10

.NET Tutorial

This tutorial describes localization of .NET software.

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET
Sample(s):	<mldir>/NET/Dcalc/cs/ <mldir>/NET/Dcalc/vb/
Tutorial(s):	<mldir>/NET/Tutorial/cs/ <mldir>/NET/Tutorial/vb/

- To learn the basics of localization of .NET software and localization prerequisites, go through the entire tutorial. It requires that you have Visual Studio .NET installed on your computer.
→ Introduction, p. 117.
- To learn how to use Multilizer for .NET software localization, you can localize any of the sample application.
→ Create Multilizer Project, p. 128.

Introduction

.NET software includes localizable data in resource files (ResX, TXT). Multilizer localizes these resources.

In order to simplify localization, Multilizer supports localization of

- Visual Studio .NET Solutions,
- Visual Studio .NET projects,
- C#Builder and Delphi 8 Project Groups
- C#Builder and Delphi 8 Projects

Localizing any of abovementioned Solutions, Projects, or Project Groups makes Multilizer pick all localizable resources automatically; without support for this, user would need to do this manually.

This tutorial is written for Visual Studio .NET; both C# and Visual Basic code samples are included.

Localization of Visual Studio .NET software

Multilizer localizes Visual Studio Solutions, Visual Studio Projects, and .NET resources.

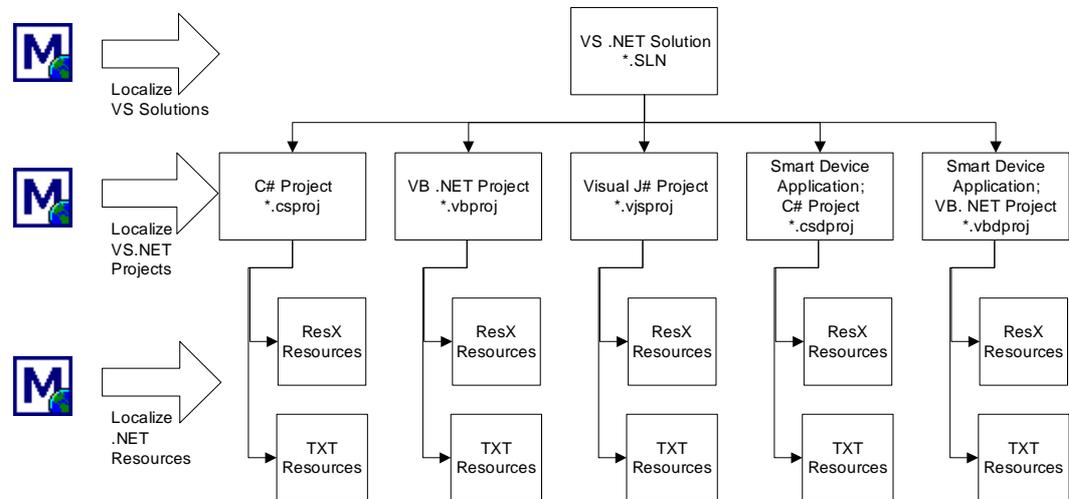


Figure 109: Visual Studio .NET file hierarchy.

Localization of Visual Studio .NET solutions

Visual Studio .NET introduced the concept of Software Solution: each solution contains any number of Visual Studio .NET projects. While .NET projects are programming language dependent, solutions are not.

Multilizer supports the localization of a .NET solution. It detects all projects in the solution, and from each project it detects the localizable resources.

By localizing a solution with Multilizer, user doesn't need to pick localizable resources manually, but Multilizer finds them automatically and adds them in Multilizer project. All translation and other localization work is done conveniently in Multilizer.

When creating localized versions, Multilizer again takes care of creating all localized files required in the .NET Solution.

Localization of Visual Studio .NET projects

Visual Studio .NET projects contain the resource data in the resource files (e.g. `.resx` or `.txt`). Multilizer creates the localized resource files from the original files. The following picture describes the localization process.

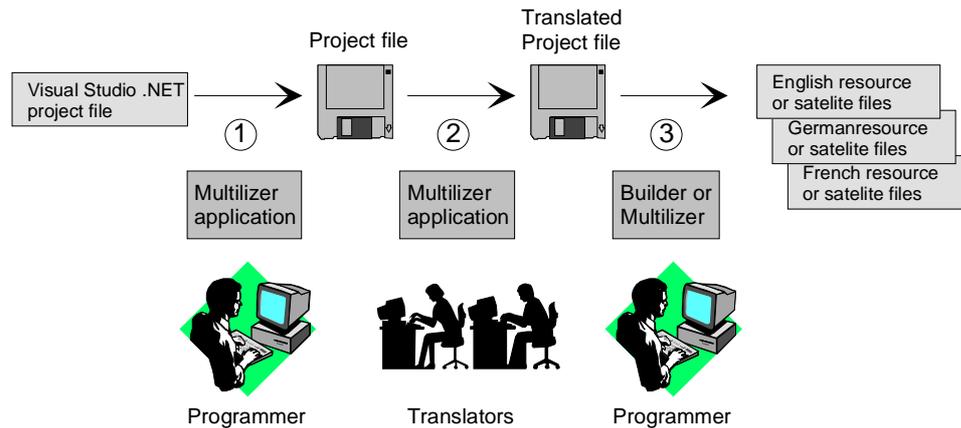


Figure 110: Visual Studio .NET project file localization process.

The programmer uses Multilizer to extract strings from the original resource file(s) belonging to the project (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized resource files and/or the localized satellite assembly files (3). As a result there will be one resource file or one satellite assembly file for each localized language.

Multilizer creates subdirectories under original project file folder containing the localized satellite assembly files. I.e. there might be subfolders called `..\en\MySample.resources.dll` (English) and `..\fi\MySample.resources.dll` (Finnish).

When deploying the application you can either deploys the original application file with the localized satellite assembly (e.g. `de\MySample.resources.dll`), or you can link the localized resource file (e.g. `MySample.de.resources`) to the original application file.

The following example figure shows the files that Multilizer uses on the Visual Studio .NET project file localization process.

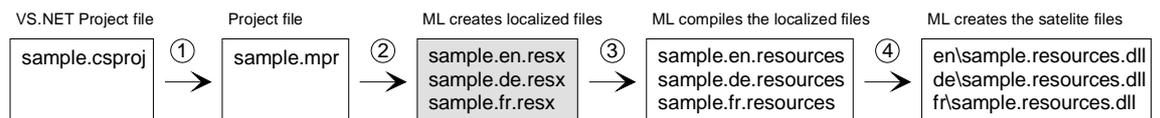


Figure 111: The files of the Visual Studio .NET project file localization process.

Using this localization process Multilizer globalizes the resources of any .NET application.

Localization of Delphi 8 / C#Builder software

Multilizer localizes .NET projects and project groups of Borland® Delphi 8 and C#Builder. Thus instead of working with individual ResX resources, you work with the same project and project group concept as in Borland .NET development tools.

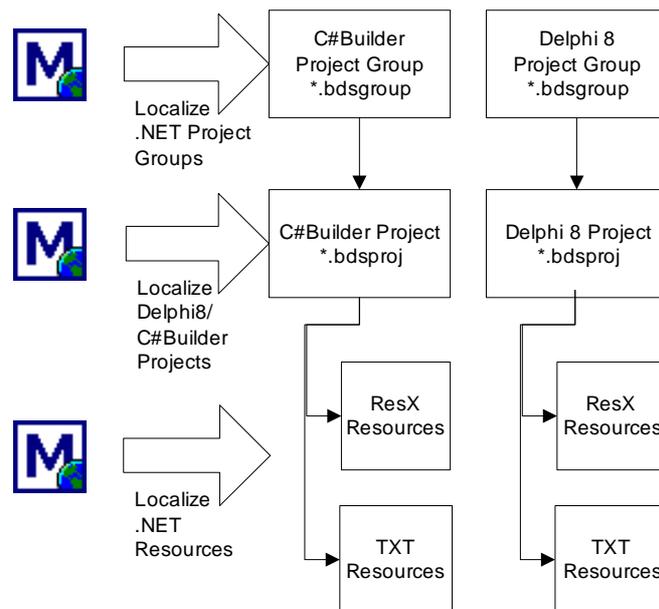


Figure 112: Borland Delphi 8/C#Builder file hierarchy.

Localization of Delphi 8 / C#Builder project groups

Multilizer supports the localization of Delphi 8 / C#Builder project groups.

Multilizer detects all projects in the project group, and from each project it detects the localizable resources.

By localizing a project group with Multilizer, user doesn't need to pick localizable resources manually, but Multilizer finds them automatically and adds them in Multilizer project. All translation and other localization work is done conveniently in Multilizer.

When creating localized versions, Multilizer again takes care of creating all localized files required in the project group.

Localization of Delphi 8 and C#Builder projects

Delphi 8 and C#Builder projects contain the resource data in the resource files (e.g. `.resx` or `.txt`). Multilizer creates the localized resource files from the original files.

Using this localization process Multilizer globalizes the resources of any Delphi 8 or C#Builder application.

Localization of .NET resources

If you do not use Visual Studio .NET, C#Builder or Delphi 8 but the command line tools of the .NET SDK or some other .NET development tool, you have to use the resource file localization process.

The following picture describes the localization process of .NET resources.

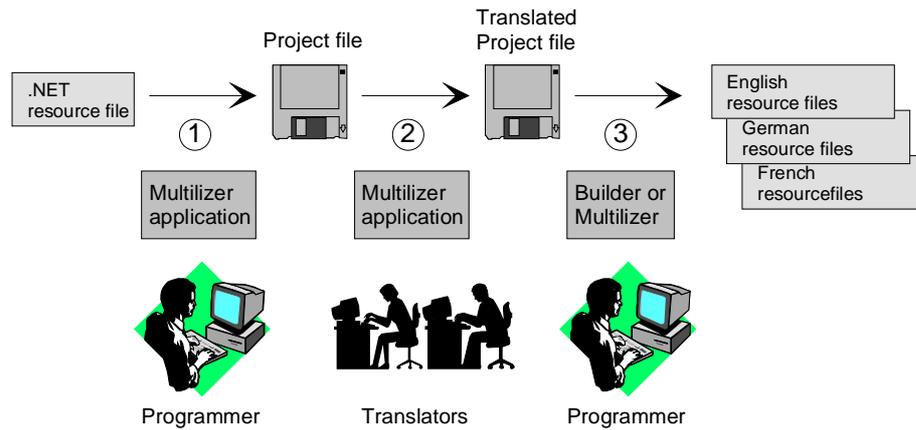


Figure 113: .NET resource file localization process.

Programmer uses Multilizer to extract localizable data from the original resource file(s) (1). Multilizer saves them to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized resource files (3). As a result there will be one resource file for each localized language.

When deploying the application you can either link the localized resource file (e.g. `MySample.de.resources`) to the original application file or create a satellite assembly file containing the localized resource file(s).

The following example figure shows the files that Multilizer uses on the .NET resource file localization process.

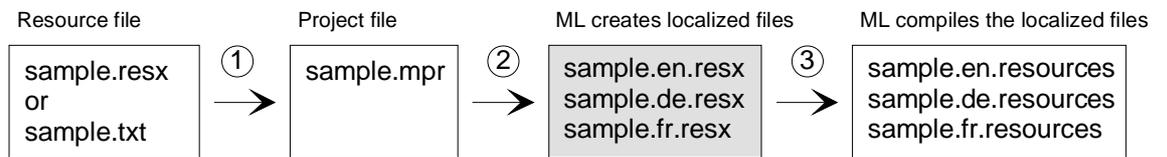


Figure 114: The files of the .NET resource file localization process.

Using this localization process Multilizer globalizes all localizable data from the .NET resource files (`.resx`) and strings found from the text resource files (`.txt`).

Open Tutorial Application

We could start from scratch but in most cases it is a completed application or at least an application under construction that you want to globalize.

```
<mldir>\NET\Tutorial\cs\Dcalc.csproj (C#) and
<mldir>\NET\Tutorial\vb\Dcalc.vbproj (Visual Basic)
```

contain the project file of the Dcalc sample application. Compile and run the application. By default, Dcalc uses English language.

The application should look like this:

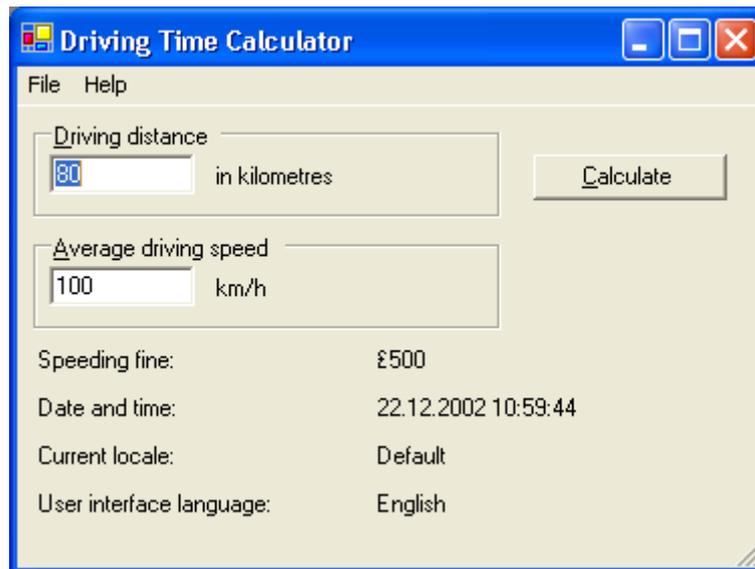


Figure 115: Dcalc application using an English user interface.

This version of Dcalc is not internationalized and many of the locale-dependent features are hard-coded. For example the currency symbol is always pounds, distances are given in kilometers and speed in kilometers per hours.

Internationalization

Microsoft .NET localization model is based on localizing resources. This means that before localizing the software, all culture (locale) dependent data must be separated from the code and put in resource files. This is called internationalization. Internationalization tasks are discussed more in detail in chapter Overview.

Internationalization of forms

Internationalization of forms is easy, because .NET generates automatically the source code for them. To internationalize a form, you just need to set the *Localizable* property of the form to *true*. .NET will automatically create a resource file (.resx) for the forms elements and create the code that references to it.

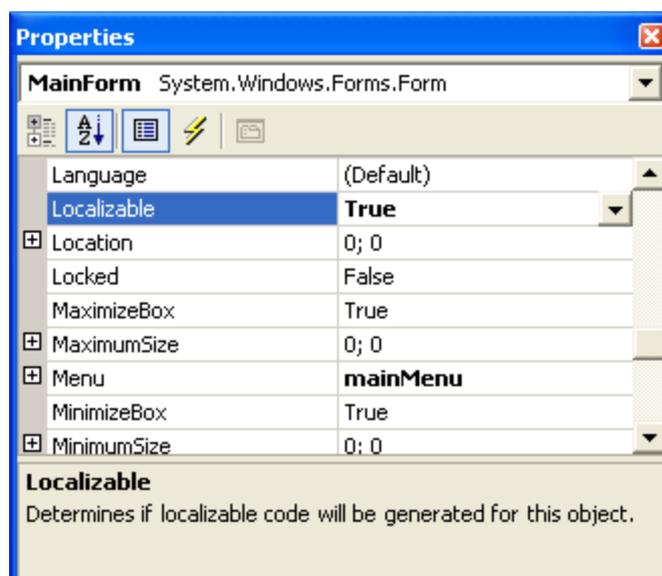


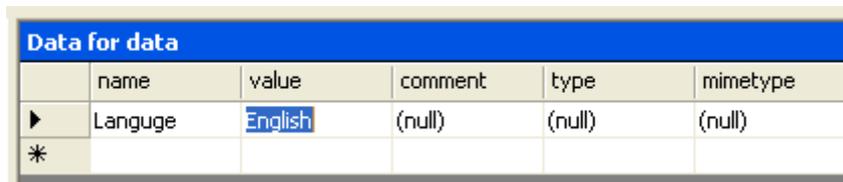
Figure 116: To localize the form set the *Localizable* property to true.

Internationalization of code

If there are hard-coded strings in the user-written part of code, you have to do the internationalization manually. You will need to create a resource file for strings, and call the internationalized strings from your code.

The first step is to create a resource file for the string. Choose Project | Add New Item from the Visual Studio .NET. From the Template list select Assembly Resource File. Rename the file to `Resource.resx`. Finally press the Open button to create the file. Visual Studio .NET creates an empty resource file.

Let's add a string to the resource file. Double click `Resource.resx` from the Solution Explorer tree. This opens the resource table editor. Type "Language" to the first row of the name column and "English" to the value column. This adds one row the file where "Language" is the key and "English" is the translation.



Data for data					
	name	value	comment	type	mimetype
▶	Language	English	(null)	(null)	(null)
*					

Figure 117: Resource file after adding the first item.

During this internationalization process we are going to add several new items to the resource file.

To access the resource file during run-time we have to add a Resource Manager object to the application. Add the following lines to the MainForm class.

C#

```
public class MainForm : System.Windows.Forms.Form
{
    private ResourceManager rm;
    ...
}

...

private void MainForm_Load(object sender, System.EventArgs e)
{
    rm = new ResourceManager("Dcalc.Resource", this.GetType().Assembly);
    ...
}
```

Visual Basic

```
Public Class mainform
    Inherits System.Windows.Forms.Form

    Dim rm As New Resources.ResourceManager("Dcalc.Resource",
    GetType(mainform).Assembly)
    ...
End Class
```

The `rm` object is used to get translation from the resource file.

Hard-coded strings

Open the Click event of the `aboutMenu`. It contains two hard coded strings. To remove the hard coding we have to add the string to the resource file and replace the direct access to the string by the access to the resource file.

Add both strings to the resource file and wrap them by the `rm.GetString` method.

```
C#
private void aboutMenu_Click(object sender, System.EventArgs e)
{
    MessageBox.Show(
        rm.GetString("Dcalc is a multilingual application that calculates the
average driving time"),
        rm.GetString("About Dcalc"),
        MessageBoxButtons.OK,
        MessageBoxIcon.Information);
}
```

```
Visual Basic
Private Sub AboutMenu_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles AboutMenu.Click
    MsgBox(rm.GetString("Dcalc is a multilingual application that
calculates the average driving time"), vbOKOnly, rm.GetString("About
Dcalc"))
End Sub
```

Format strings

If you look at the calculate event you will see that the following lines are used to format the message string that show the driving time to the user.

```
C#
MessageBox.Show(
    "The average driving time is " + Convert.ToString(hours) + " hours and
" +
    Convert.ToString(minutes) + " minutes.",
    "Driving time",
    MessageBoxButtons.OK,
    MessageBoxIcon.Information);
```

```
Visual Basic
MsgBox("The average driving time is " + Convert.ToString(hours) + " hours
and " + Convert.ToString(minutes) + " minutes", vbOKOnly, "Driving time")
```

The above code contains both hard coded string and bad design. First of all the message is split into several string that do not contain a real sentence. That's why it is hard to translate them. Also the code assumes that the word order is text plus hours plus some other text plus minutes plus some other text. This works with English but may not work with some other language. That's why we have to use the Format method of the String class. It uses a message pattern and variables.

```
C#
MessageBox.Show(
    String.Format(
        rm.GetString("The average driving time is {0} hours and {1}
minutes."),
        Convert.ToString(hours),
        Convert.ToString(minutes)),
    rm.GetString("Driving time"),
    MessageBoxButtons.OK,
    MessageBoxIcon.Information);
```

```
Visual Basic
MsgBox(String.Format(rm.GetString("The average driving time is {0} hours
and {1} minutes"), hours, minutes), vbOKOnly, rm.GetString("Driving
time"))
```

The calculate event shows a message if the distance value is invalid. It also contains hard coded string and a dynamic message.

C#

```
MessageBox.Show(
    distanceText.Text + " is not a valid distance!",
    "Error",
    MessageBoxButtons.OK,
    MessageBoxIcon.Error);
```

Visual Basic

```
MsgBox(distanceText.Text + " is not a valid distance!", vbOKOnly,
"Error")
```

Use the same approach as with the result message.

C#

```
MessageBox.Show(
    String.Format(
        rm.GetString("{0} is not a valid distance!"),
        distanceText.Text),
    rm.GetString("Error"),
    MessageBoxButtons.OK,
    MessageBoxIcon.Error);
```

Visual Basic

```
MsgBox(String.Format(rm.GetString("{0} is not a valid distance!"),
DistanceText.Text), vbOKOnly, rm.GetString("Error"))
```

Resource the message box for the invalid speed in the same way.

Culture-specific issues

The Load event initializes some user interface items such as the speeding fine label and the current time label.

C#

```
private void MainForm_Load(object sender, System.EventArgs e)
{
    speedingFine.Text = "£500";
    currentTime.Text = DateTime.Now.ToString();
}
```

Visual Basic

```
Private Sub Form1_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    speedingFine.Text = "£500"
    currentTime.Text = DateTime.Now.ToString()
End Sub
```

The current time is properly formatted. The ToString method converts the time to a string using the default formatting rules of the current culture. However the fine is hard code to £500. It can be fixed using the int.ToString method.

In US miles are used instead of kilometers. The code below checks if the current culture is English (US). If it is the labels and initial values are set to US system. Otherwise the metric system is used.

C#

```

private void MainForm_Load(object sender, System.EventArgs e)
{
    rm = new ResourceManager("Dcalc.Resource", this.GetType().Assembly);

    int fine = 500;
    speedingFine.Text = fine.ToString("C");
    currentTime.Text = DateTime.Now.ToString();

    currentLocale.Text = CultureInfo.CurrentCulture.NativeName;
    currentLanguage.Text = rm.GetString("Language");

    if (Application.CurrentCulture.LCID == 0x0409)
    {
        distanceLabel.Text = rm.GetString("in miles");
        speedLabel.Text = rm.GetString("mph");
        distanceText.Text = "100";
        speedText.Text = "55";
    }
    else
    {
        distanceLabel.Text = rm.GetString("in kilometres");
        speedLabel.Text = rm.GetString("km/h");
        distanceText.Text = "120";
        speedText.Text = "100";
    }
}

```

Visual Basic

```

Private Sub MainForm_Load(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles MyBase.Load
    SpeedingFine.Text = FormatCurrency(500)
    CurrentTime.Text = FormatDateTime(Now)

    CurrentLocale.Text = Application.CurrentCulture.NativeName()
    CurrentLanguage.Text = rm.GetString("Language")

    If Application.CurrentCulture.LCID = &H409 Then
        DistanceLabel.Text = rm.GetString("in miles")
        SpeedLabel.Text = rm.GetString("mph")

        DistanceText.Text = "100"
        SpeedText.Text = "55"
    Else
        DistanceLabel.Text = rm.GetString("in kilometres")
        SpeedLabel.Text = rm.GetString("km/h")

        DistanceText.Text = "120"
        SpeedText.Text = "100"
    End If
End Sub

```

Now we have fully internationalized the source code. Make sure that you added every single string inside the `rm.GetString` method to the resource file.

Determine startup language at command-line

In order to test different locale we add one command line parameter to the application. The parameter `-lang:id`, where `id` is the culture identifier that specifies the culture that the application uses.

This parameter is passed from Multilizer when running localized version of the software from within Multilizer. The parameter to pass from Multilizer can be altered from target settings.

C#

```
static void Main(string[] args)
{
    // The application can have a command line parameter that specifies the
    // culture
    bool found = false;
    if (args.Length > 0)
    {
        for (int i = 0; i < args.Length; i++)
        {
            string str = args[i];
            int index = str.IndexOf("-lang:");

            if (index == 0)
            {
                str = str.Remove(0, 6);

                if (str != "")
                {
                    CultureInfo ci = new CultureInfo(str);
                    Thread.CurrentThread.CurrentUICulture = ci;
                    if (!ci.IsNeutralCulture)
                        Thread.CurrentThread.CurrentCulture = ci;
                }

                found = true;
                break;
            }
        }
    }

    if (!found)
        Thread.CurrentThread.CurrentUICulture = CultureInfo.CurrentCulture;

    Application.Run(new MainForm());
}
```

Visual Basic

```
Public Sub New(ByVal culture As String)
    If culture <> "" Then
        Try
            Thread.CurrentThread.CurrentUICulture = New CultureInfo(culture)
        Catch e As ArgumentException
            MessageBox.Show(Me, e.Message, "Bad command-line argument")
        End Try
    End If

    InitializeComponent()
End Sub

<System.STAThreadAttribute()> _
Public Shared Sub Main()
    '
    ' Main takes an optional argument identifying the culture you'd like
    ' displayed.
    '
    Dim args() As String = System.Environment.GetCommandLineArgs()
    Dim i
    Dim str As String
    Dim strCulture As String = ""
    If args.Length > 0 Then
        For i = 0 To args.Length - 1
            str = args(i)
            If str.IndexOf("-lang:") = 0 Then
                str = str.Remove(0, 6)
                If str <> "" Then
                    strCulture = str
                End If
            End If
        Next
    End If
    Application.Run(New MainForm(strCulture))
End Sub
```

Miscellaneous**Add icon**

In order to be able to localize icons, software must not use .NET default icon; specifying an icon other than the default one will make it possible to localize the icon later with Multilizer application.

Unicode

Although not related to internationalization, .NET support for Unicode® improves localization quality. There are no more code-page related issues, such as non-readable characters appearing in the software.

Create Multilizer Project

In order to localize .NET software, you have to create a Multilizer project. This is described in the first part of the manual, chapter Create Project, p. 11.

Specify Localization options

After finishing the Wizard, you have to specify the localization options for the software. Normally default options are the most useful, but in this tutorial we will review all options.

Right-click the localization target in Project Tree, and click properties to see Delphi Target options.



All .NET-specific options are gathered under the corresponding target dialog. If there are many targets in one project, you can set different localization options to all, if needed.

Encodings

Encodings tab shows the languages of project. Because .NET is Unicode-based, the only encoding for all languages is UTF-8. All .NET ResX resources are UTF-8 encoded.

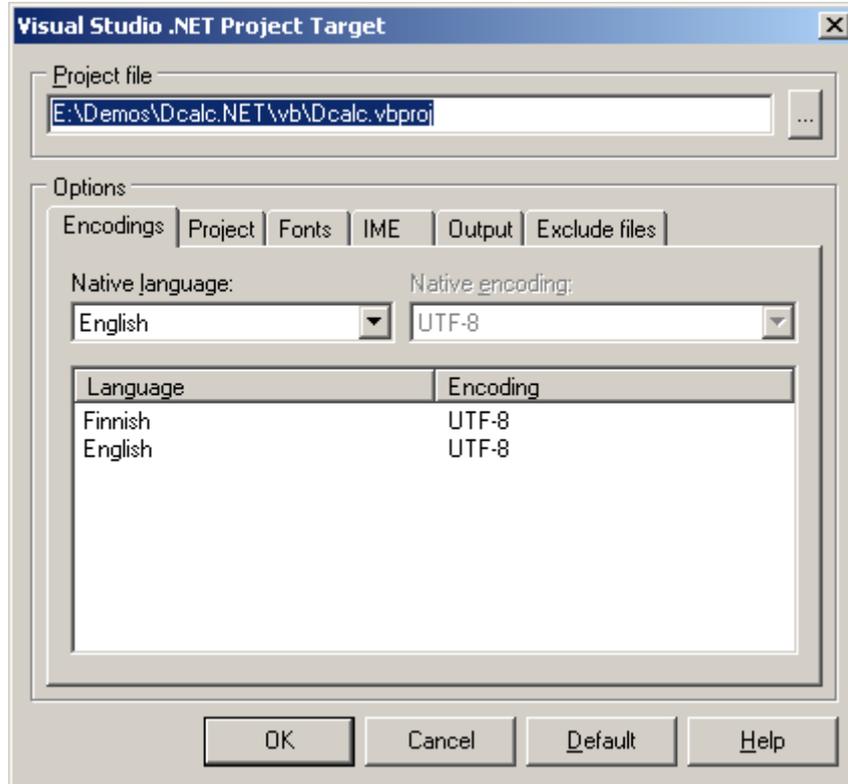


Figure 118: Encodings for localized software.

Project

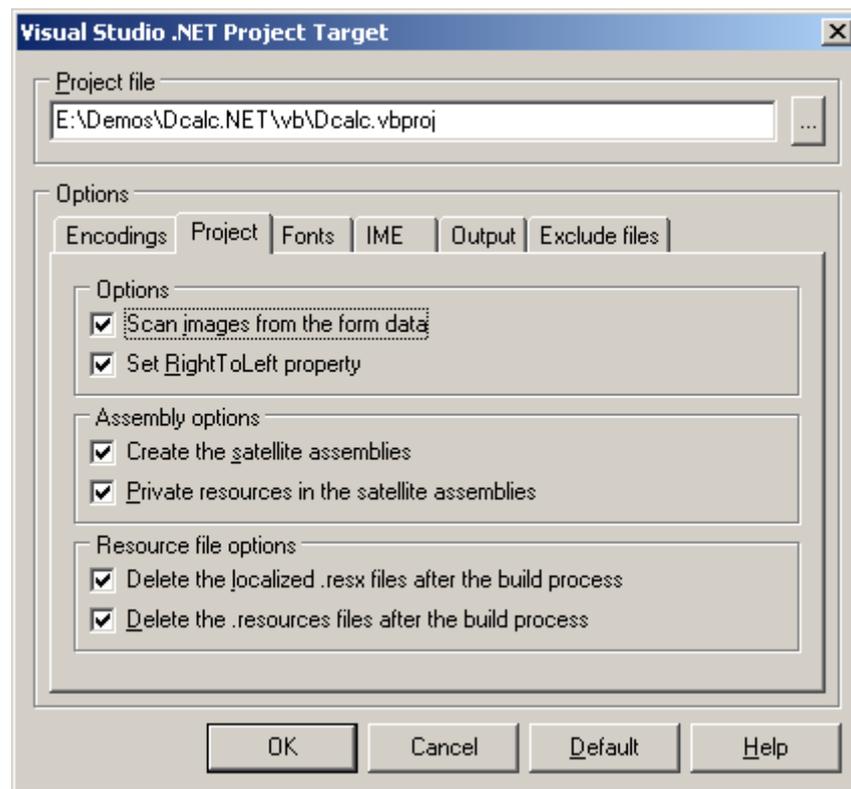


Figure 119: .NET Localization options

Options

Scan images from the form data enables all pictures embedded in the form resources to be scanned.

Set RightToLeft property affects localization of RTL (right-to-left) languages; this property affects how texts are aligned in its placeholder. Having this feature checked ensures that text alignment is mirrored when localization to RTL-languages, such as Hebrew and Arabic.

Assembly options

Check 'Create the satellite assemblies', if you want that Multilizer creates the satellite dll's. In order to do this, you have to specify the paths to Assembly Linker and Resource Generator (→ Specify .NET tools location, p. 136). If this option is not checked, Multilizer creates localized resource files.

Check 'Private resources in the satellite assemblies', if you want that generated resources are not visible to other assemblies.

Resource file options

When Multilizer creates localized satellite assemblies, there are a lot of intermediary files created. By checking both options here, all intermediary files created by Multilizer will be deleted after build.



Fonts

On Fonts tab user can specify the font of localized software. Furthermore, rules can be set to apply fonts on certain conditions.

Default settings are recommended, because they are strictly based on Windows standards.

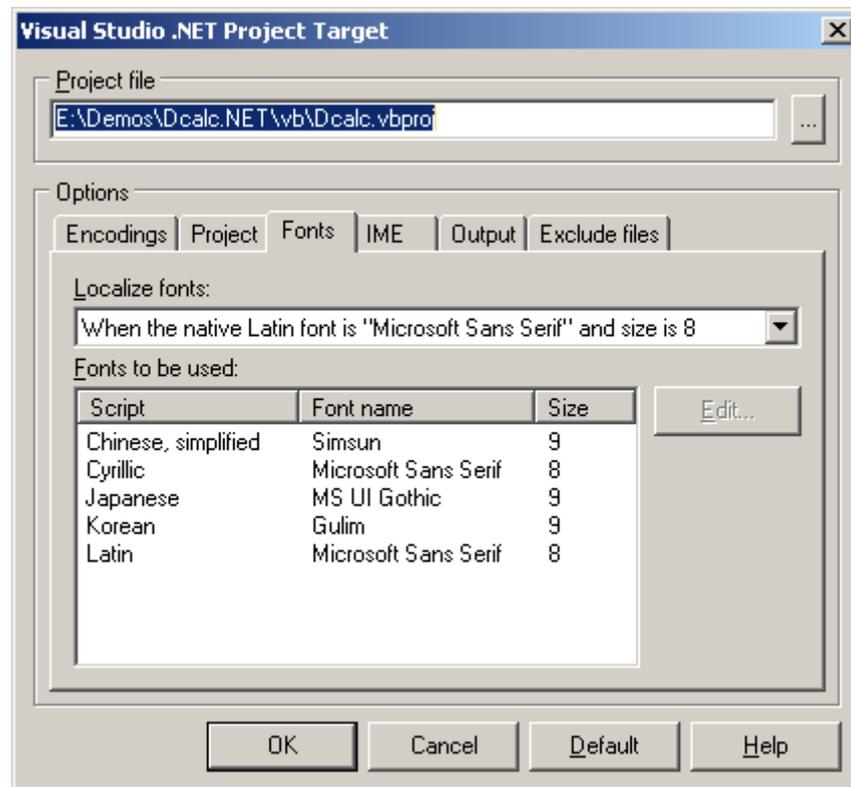


Figure 120: Font options for localized .NET software.

IME

IME (Input Method Editor) settings specify the input method editor (IME) to use for converting keyboard input to Asian language characters.

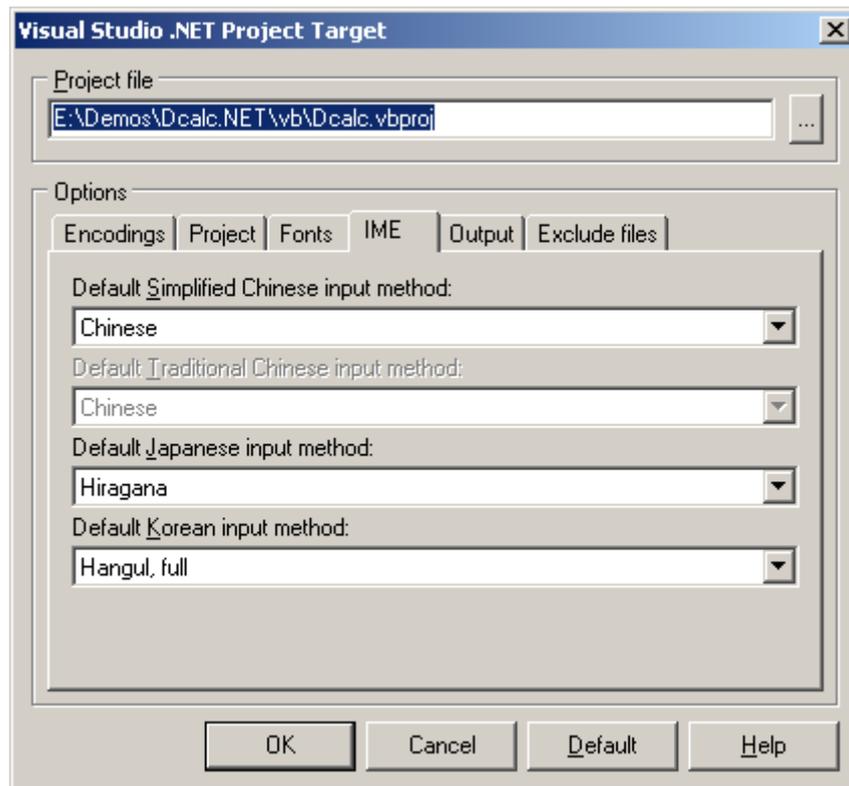


Figure 121: IME options for software localized to Far Eastern languages.

IME settings are enabled only if Multilizer project includes languages that use IME.

The IME settings here will apply corresponding value to `IMEMode` property in software localized to Chinese, Japanese, and Korean.

Output

Output directory lets use specify where to store localized items.

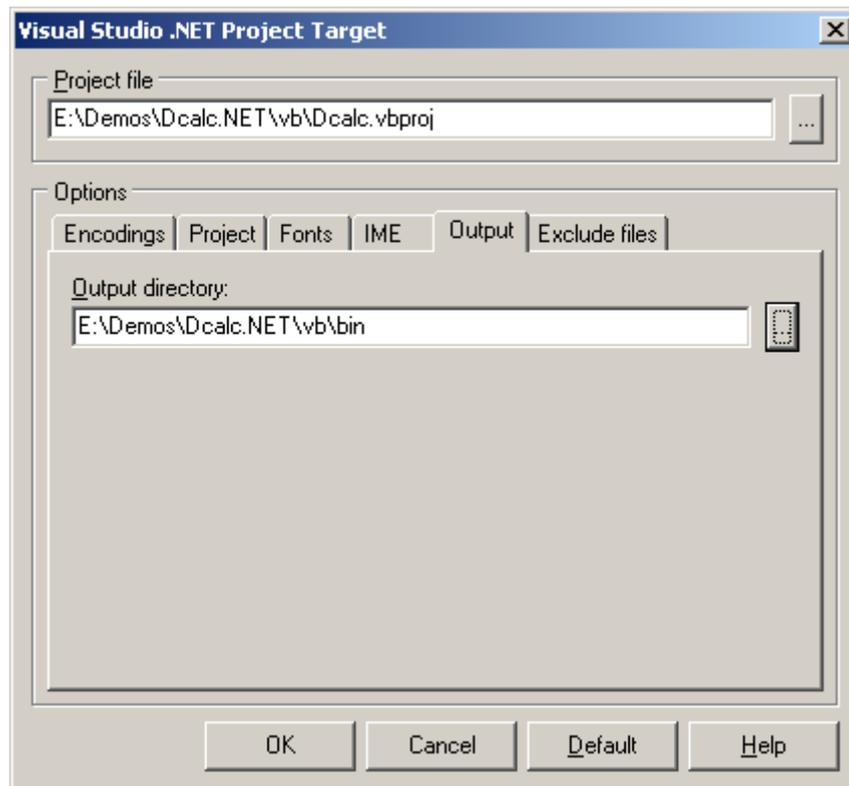


Figure 122: Output options for .NET software.

Translate Project

For testing purpose, you can translate the software by using pseudo-languages (C.f. Pseudo language, p. 68); right-click language column, choose properties, and select the pseudo-language options. This will fill translation grid with pseudo-language translations.

Wysiwyg

Besides just translating, Multilizer allows editing of UI (user interface) elements. This is useful in cases, where original software was not designed for localization, and translated strings don't fit in the placeholders.

Translation with Multilizer showing visually the changes in UI is referred as Wysiwyg in this manual. Wysiwyg is enabled both in forms editing as well as menu editing, as shown in following images.

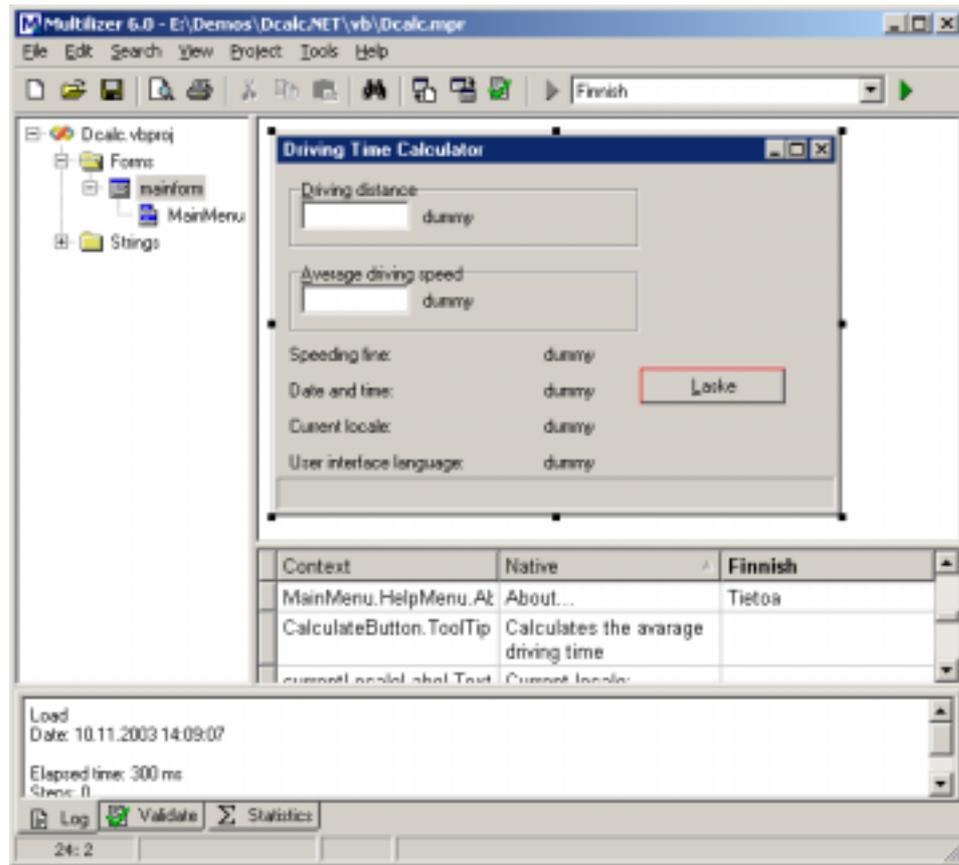


Figure 123: Localizing forms of .NET software visually.

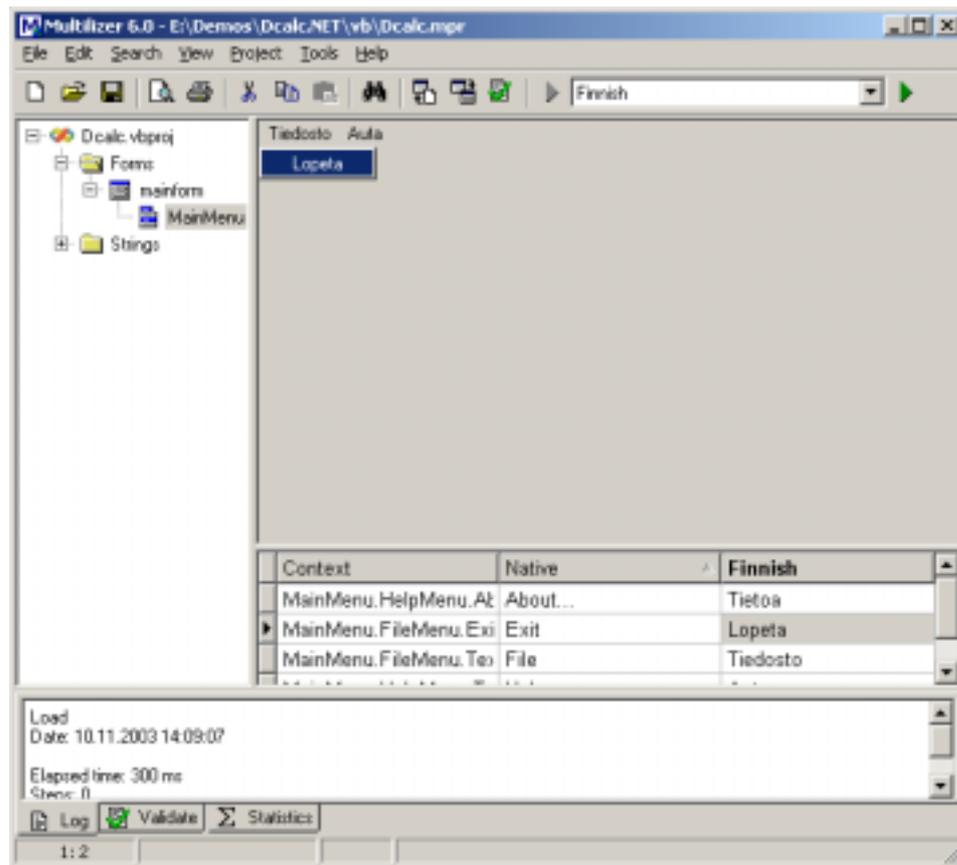


Figure 124: Localizing menus of .NET software visually.

More info



Refer to following parts of the manual for more information on translating software, and sharing translation work between team members.

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized application files by choosing **Project | Build Localized Files**. This creates the localized files.

Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run.

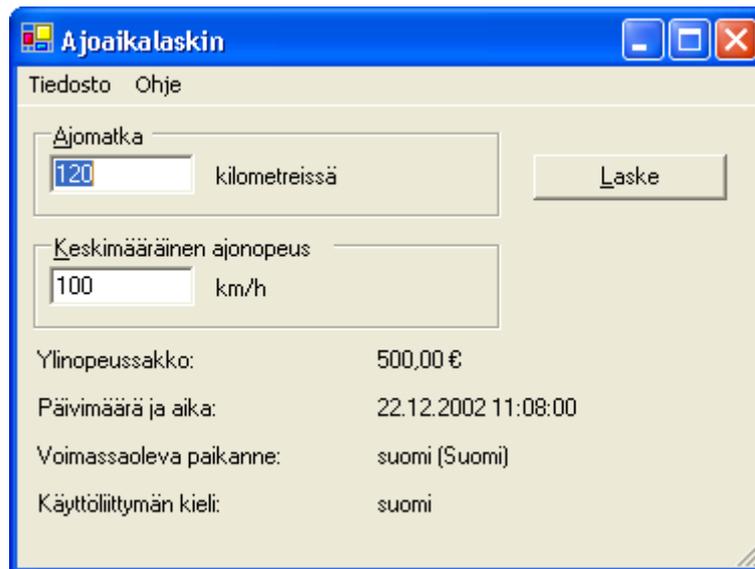


Figure 125: Localized .NET software.



In order to use the satellite assembly files they must locate on the sub directories of the main assembly file (`Dcalc.exe`). By default Visual Studio .NET place the EXE file to the `bin` or `bin\Debug` subdirectory. When running the application from Multilizer, Multilizer copies the application file to the project's main directory (e.g. `bin\Debug\Dcalc.exe -> Dcalc.exe`). Run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run.

Specify .NET tools location

In order to create assembly DLLs, Multilizer needs to know the location of .NET tools.

To modify the settings, choose **Tools**→**Options...**, select **.NET, Tools**.

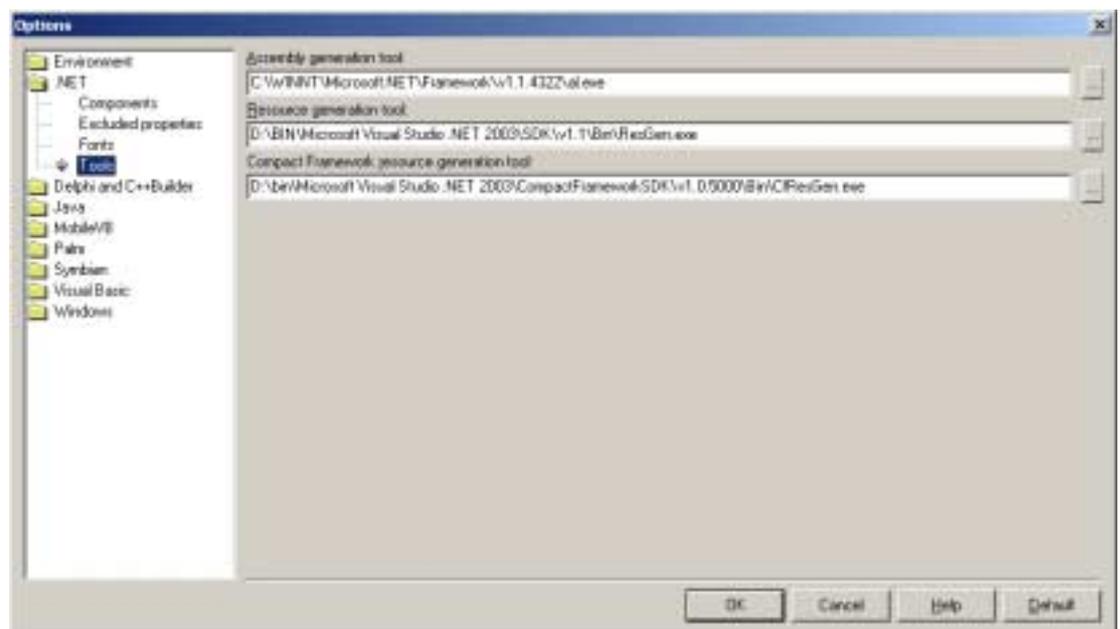


Figure 126: Specifying of .NET tools.

Excluding Properties from localization

Not all string properties are intended for localization. Localization of them can even result in crash of the software. In order to prevent this from happening, Multilizer excludes by default some of the string properties.

In order to exclude strings from localization, you can

- Specify what string-type properties are excluded for a .NET component
- Specify what properties are always excluded

These settings will prevent Multilizer from adding them in localization project.

The settings discussed here affect the way that Multilizer scans *ResX* data.

Exclude properties by components

To exclude properties by components, choose **Tools→Options...**, select **.NET, Components**.

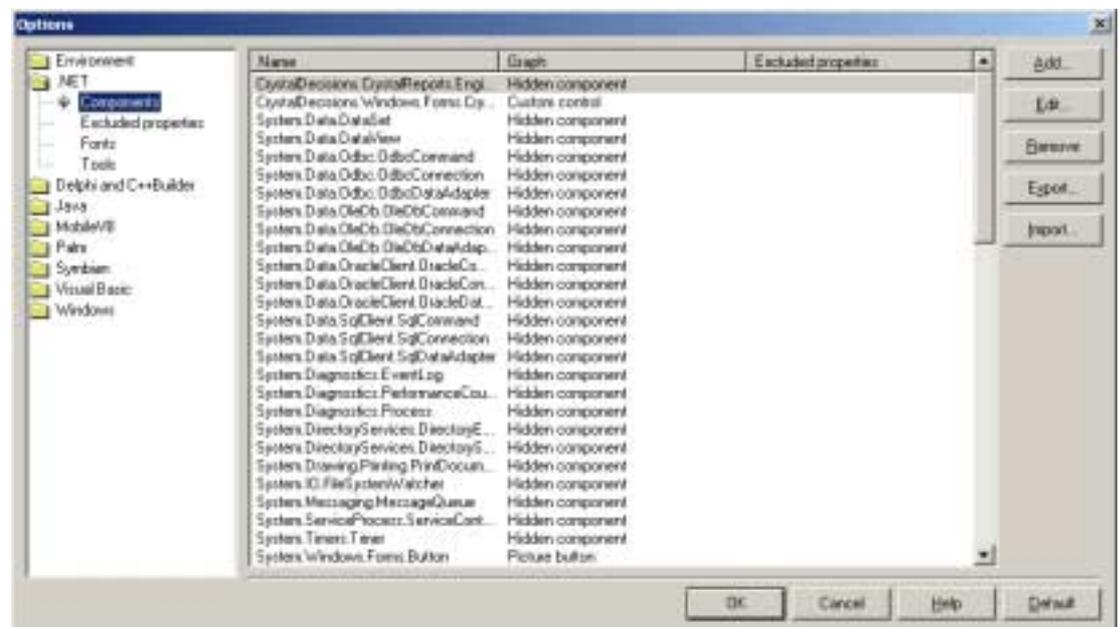


Figure 127: Excluding properties by components in .NET localization.

This dialog lets you specify the excluded properties of specific components. You can specify one or more properties that are excluded from scanning for each component.

This dialog is also used to configure the visual representation of each component (→Visual Representation, p. 115).

Exclude properties by name

To exclude properties by name, choose **Tools→Options...**, select **.NET, Excluded properties**.

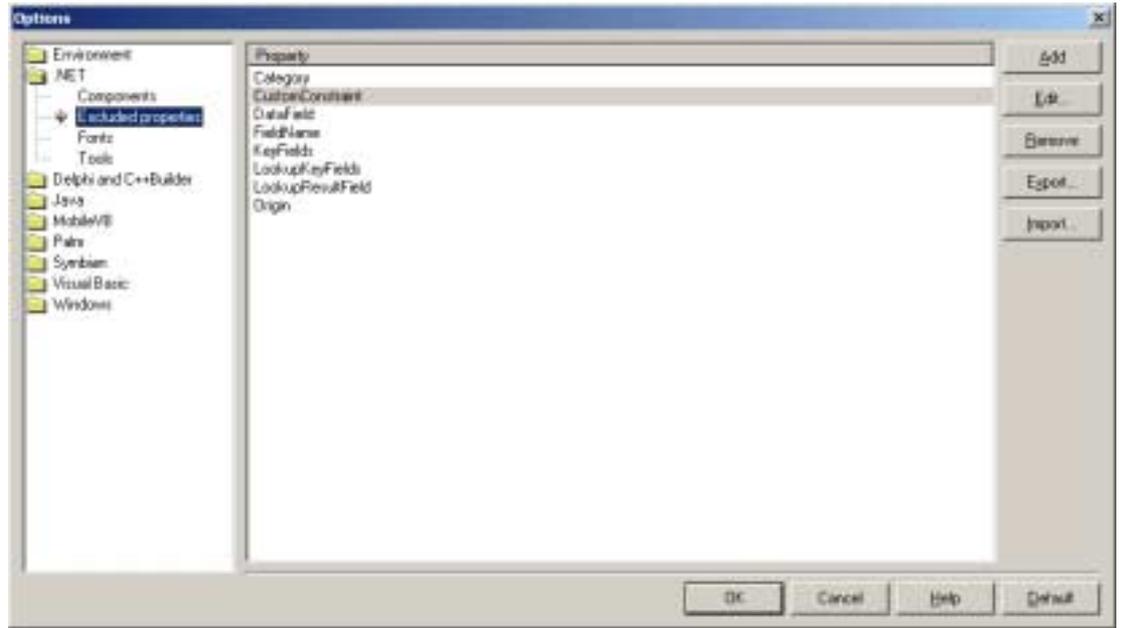


Figure 128: Excluding properties by name in VCL binary localization.

This dialog lets you specify the properties that are not localized, and never included in Multilizer project.

Visual Representation

User interface of .NET software is developed using visual components.

There are tens of standard components, and an increasing number of third-party components. In order to display any component correctly, Multilizer allows users to configure the visual representation of any component.

All standard components and a few 3rd-party components are defined already.

Components whose visual representation is not defined are shown with yellow placeholder in Wysiwyg.

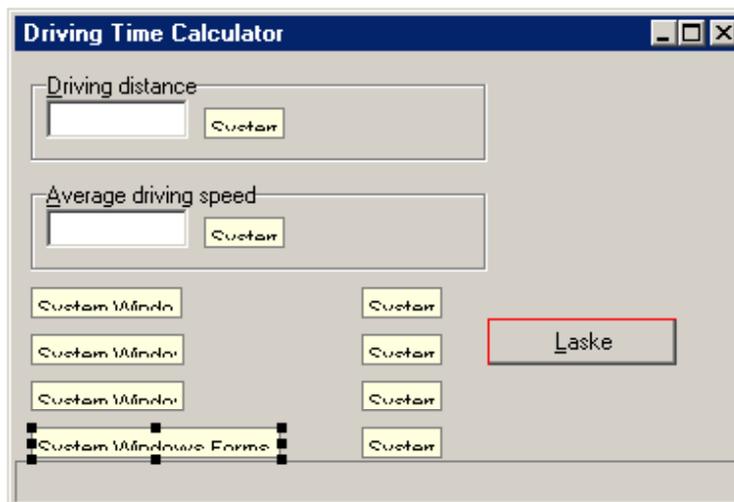


Figure 129: Visual form editor with unknown .NET controls.

To exclude properties by components, choose **Tools**→**Options...**, select **.NET, Components**. Choose **Add..** to add new component.

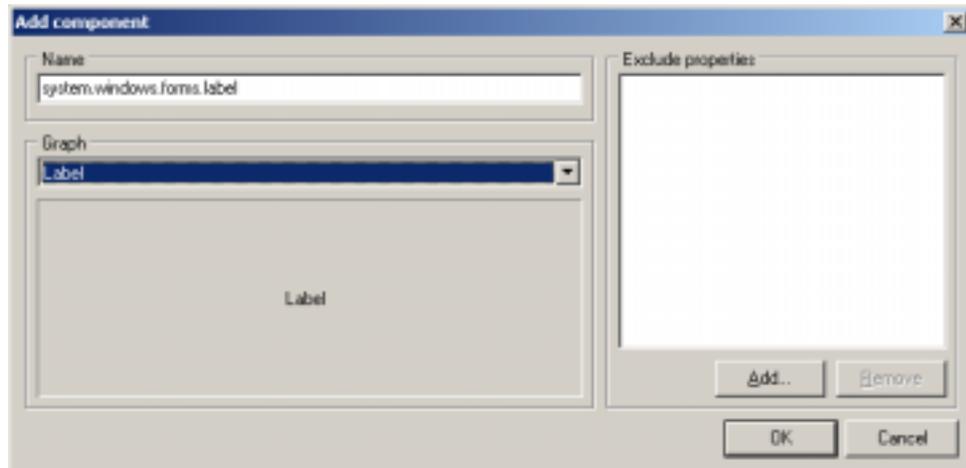


Figure 130: Specifying visual representation for .NET control.

Write component name as specified in .NET namespace. Choose from Graph the appropriate representation; a preview is shown automatically.

After next re-scan of project selected visual representation is applied in Wysiwyg, as shown in next picture.

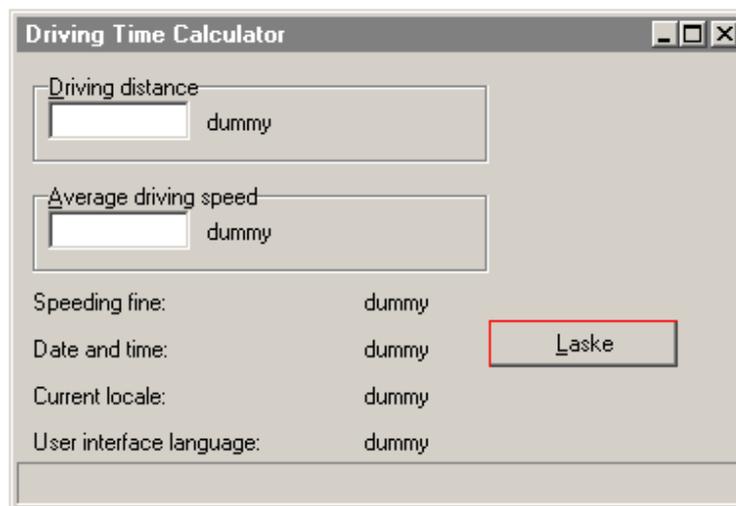


Figure 131: Visual Editor recognizing all .NET controls.



Visual Representation configurations can be shared between team members; you can both import and export the settings as *Multilizer item files*.

11

Java Tutorial

This tutorial describes localization of J2SE/J2EE software.

Required product(s):	Multilizer Enterprise Multilizer for Java
Sample(s):	<mldir>/java/dcalc/
Tutorial(s):	–

- To learn the basics of localization of Java software and localization prerequisites, go through the entire tutorial. It requires knowledge of Java programming, and that you have an existing Java development environment installed on your computer.
→ Introduction, p. 140.
- To learn how to use Multilizer for Java software localization, create a Multilizer project based on the sample application.
→ Create Multilizer Project, p. 143.

Introduction

Localization of Java software is based on translating resource bundles. Resource bundles contain all strings intended for localization. Resource Bundles can be either *List resource bundles* or *Property Resource Bundles*. Both are supported by Multilizer.

Multilizer allows localization of the following Java application files containing resource bundles:

- JBuilder project files (.jpr, .jpx)
- Java archive files (.jar)
- Resource Bundles (.properties)
- Java list resource bundles (.java)

In addition Multilizer supports localization of .java source code files (→ Source Localization Tutorial, p. 167).

Because Java code is also embedded in XML, Multilizer supports localization of Java code embedded in XML (→ XML Tutorial, p. 163).

Localization of Resource Bundles

Java standard edition provides support for internationalization in `java.util` and `java.text` packages. Localization is mainly done through resource bundles. This chapter covers only the basic internationalization (I18N). Prefer I18N books and web sites to get more information about. An excellent start is the Java tutorial at <http://www.javasoft.com/>.

The following picture describes the Java standard edition localization process with resource bundles and Multilizer.

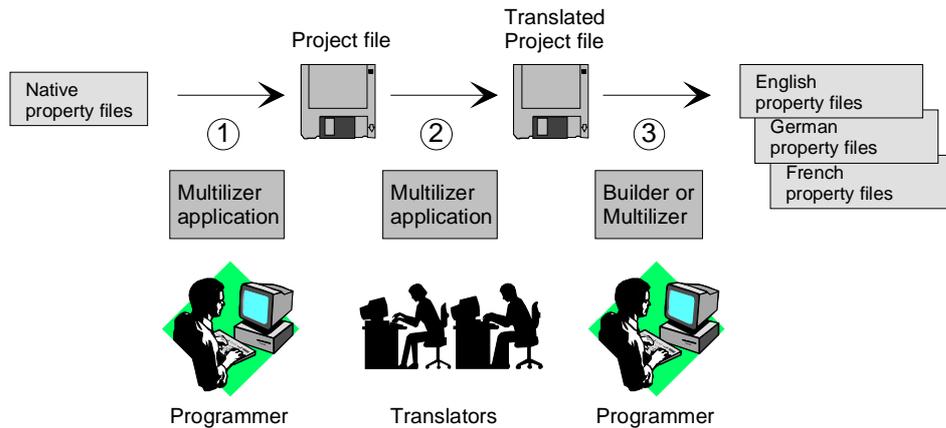


Figure 132 Java localization process with resource bundles

The programmer uses Multilizer to extract strings from the original resource bundle (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized resource bundles (3). As the result there will be one resource bundle for each localized language.

The following figure shows the files that Multilizer uses in the Java standard edition localization process.

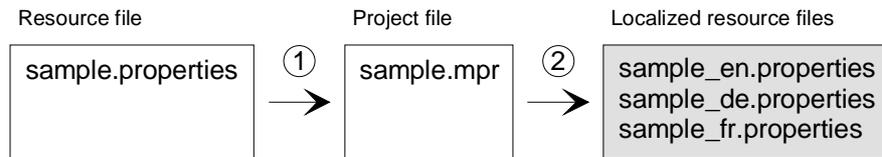


Figure 133 The files of the Java localization process with property resource bundles

Internationalization

Before localizing a Java application, developer needs to put all localizable strings in resource bundles. This is called resourcing, and implies removing hard-coded strings. In addition he must use ResourceBundle class to get strings from the resource bundles.

This all is called internationalization.

The result of internationalization is modified .java files, and one or many .properties files that contain the strings.

Internationalization traditionally involves a lot of manual work, but modern Java development environments include Internationalization Wizards that do the work for the developer.

JBuilder Resource Wizard

In JBuilder you can create the resource bundle easily by using a wizard: **Wizards | Resource Strings**. The wizard scans your source code, creates the bundle and makes the necessary modifications in your source code.

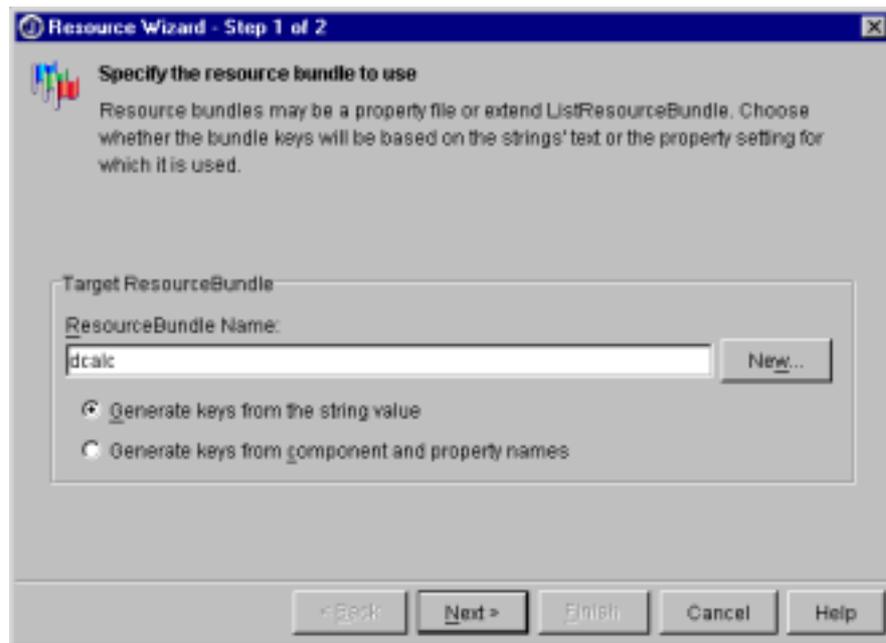


Figure 134 JBuilder Resource wizard

Press New and change the Name to dcalc and Type to PropertyResourceBundle. Press OK. Press Next twice. The wizard extracts strings from the source code. Press Finish to complete the wizard.

NetBeans IDE 3.5.1

NetBeans IDE includes Internationalization Wizard that takes care of internationalization of hard-coded strings. It is invoked from main menu: Tools→Internationalization→Internationalization Wizard...

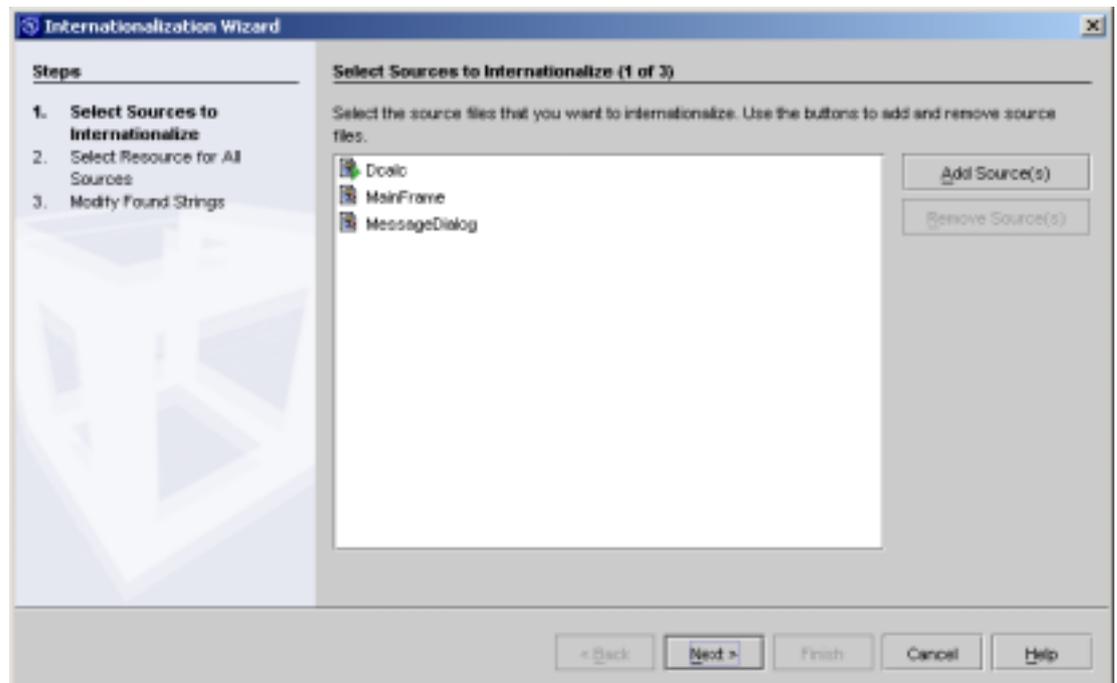


Figure 135: Internationalization Wizard of netBeans IDE.

After running the Wizard

```
helpMenu.setLabel("Help");
```

becomes

```
helpMenu.setLabel(java.util.ResourceBundle.getBundle("dcalc").getString("Help"));
```

Manual internationalization

In plain JDK create the PropertyResourceBundle by hand and take it in use.

```
public class MainFrame extends Frame
{
    static ResourceBundle res = ResourceBundle.getBundle("dcalc");
```

Wrap all hard coded strings inside the `res.getString()` method. Add all keys and their native translations to the bundle. Remember that key values in a bundle aren't allowed to contain e.g. space characters and you have to use Unicode escapes for non-ASCII characters.

```
fineLabel.setText(res.getString("dummy"));
label4.setText(res.getString("Speeding_fine_"));
label15.setText(res.getString("Date_and_time_"));
dateLabel.setText(res.getString("dummy"));
label7.setText(res.getString("Current_locale_"));
localeLabel.setText(res.getString("dummy"));
label9.setText(res.getString("User_interface"));
languageLabel.setText(res.getString("English"));
```

The last String ("English") should not be localized by translating it in a resource bundle. Remove the line and add the following code to the constructor of MainFrame.

```
// Update the user interface language
languageLabel.setText(res.getLocale().getDisplayLanguage());
```

Use also elsewhere in the MainFrame `Locale.getDefault()` instead of `res.getLocale()`. For example:

```
// Update the locale label
localeLabel.setText(res.getLocale().getDisplayName());
```



`ResourceBundle.getLocale()` is supported only by JDK 1.2 or later. With JDK 1.1.8 you have to use `Locale.getDefault()` instead of `res.getLocale()`.

Create Multilizer Project

In order to localize Java software, you have to create a Multilizer project. This is described in the first part of the manual, chapter Create Project, p. 11.

Multilizer allows localization of the following Java application files:

- JBuilder project files (.jpr, .jpx)
- Java archive files (.jar)

- Resource Bundles (.properties)
- Java list resource bundles (.java)

Specify Localization options

After finishing the Wizard, you have to specify the localization options for the software. Normally default options are the most useful, but in this tutorial we will review all options.

Right-click the localization target in Project Tree, and click properties to see Delphi Target options.



All .NET-specific options are gathered under the corresponding target dialog. If there are many targets in one project, you can set different localization options to all, if needed.

Encodings (all Java targets)

Encodings tab shows the languages of project. Because Java uses Unicode escapes in resource bundles, the only encoding for all languages is Unicode escapes.

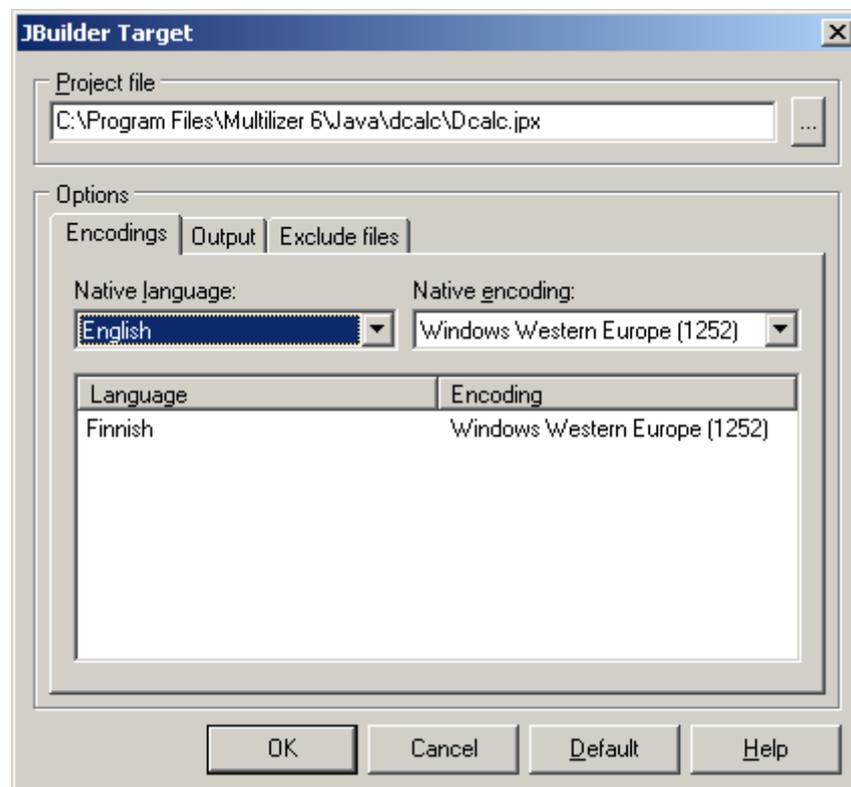


Figure 136: Encodings for localized software.

Output (all Java targets)

Output directory lets use specify where to store localized items.

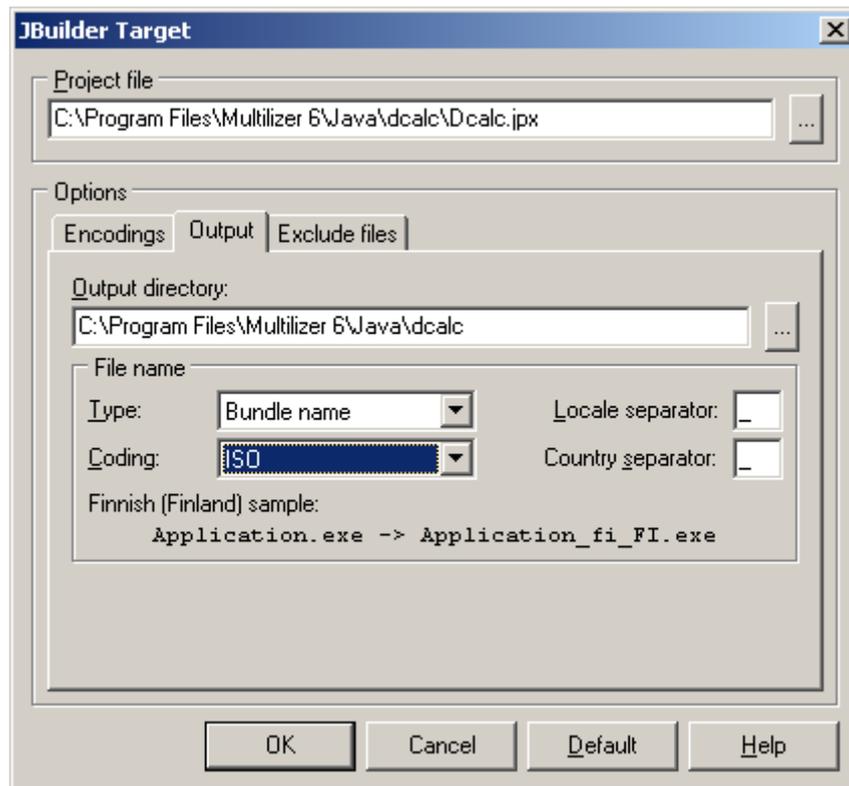


Figure 137: Output options for Java software.

Exclude files (JBuilder targets)

When localizing JBuilder projects there may be files that must not be localized. On Exclude files tab you can specify which files should be excluded from localization.

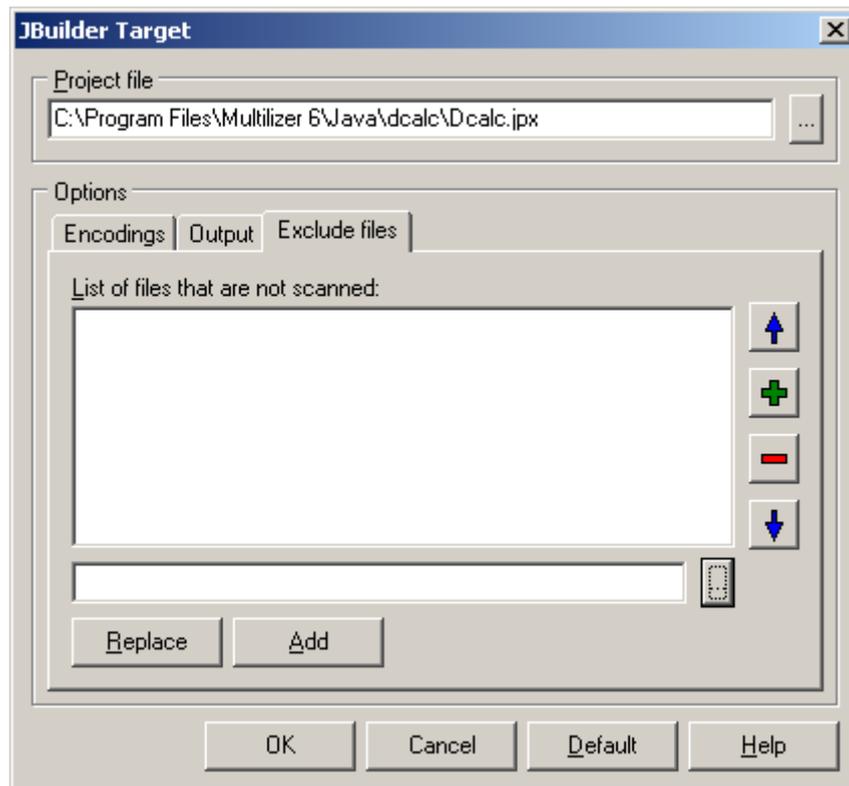


Figure 138: Excluded files for Java software.

Translate Project

For testing purpose, you can translate the software by using pseudo-languages (C.f. Pseudo language, p. 68); right-click language column, choose properties, and select the pseudo-language options. This will fill translation grid with pseudo-language translations.

More info



Refer to following parts of the manual for more information on translating software, and sharing translation work between team members.

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized application files by choosing **Project | Build Localized Files**. This creates the localized files. Finally you can run the localized application by right-clicking the column header (e.g. Finnish) and by choosing Run.

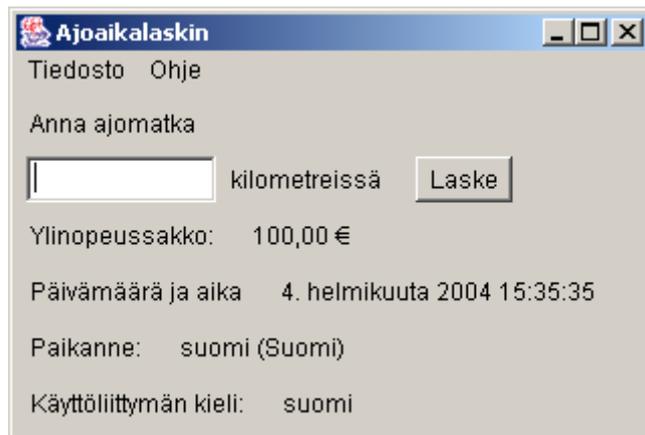


Figure 139: Localized Java software.



In order to run localized software, you need to specify location of JDK. See next chapter for more info.

Specify Java tools location

Running localized software from within Multilizer requires that location of Java Virtual Machine is specified. To specify it, select Tools→Options...→Java, and click J2SE/J2EE options. Select desired Virtual Machine from drop-down list.

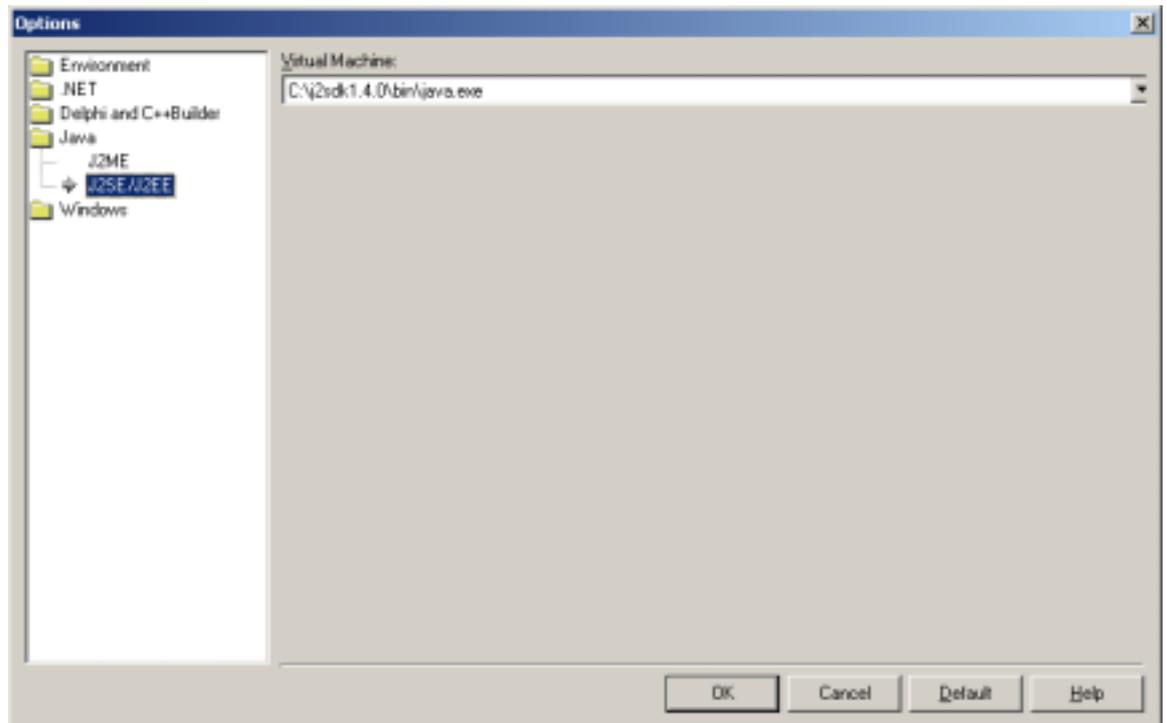


Figure 140: Specifying Java Virtual Machine location.

12

J2ME Tutorial

This tutorial describes localization of J2ME software. For localization of J2SE/J2EE software, see Java Tutorial, p. 140.

Required product(s):	Multilizer Enterprise Multilizer for Java
Sample(s):	<mldir>/j2me/dcalc/
Tutorial(s):	–

- To learn the basics of localizing J2ME applications using Multilizer resource bundle classes, go through the entire tutorial. It requires knowledge of Java programming, and that you have an existing Java development environment installed on your computer.
→ Introduction, p. 148.
- To learn how to use Multilizer for J2ME software localization, create a Multilizer project based on the sample application.
→ Create Multilizer Project, p. 152.



There is more information in Multilizer J2ME help (located at <mldir>/j2me/Multilizer_me.chm).

J2ME applications can be also localized by using source localization (→ Source Localization Tutorial, p. 167). While this tutorial explains how to use the flexible resource bundle classes for localization, source localization may be the most useful for localization of tiny applications with only a couple of localized strings.

Introduction

Java Standard Edition (J2SE) contains very rich support for localization. J2SE contains locale class, resource bundles and formatting classes. Unfortunately Java Micro Edition (J2ME) does not contain these classes. It only contains very low-level resource class that the application can use to access resource file. The current CLDC-configuration has the following I18N-support:

- `java.lang.Class.getResourceAsStream(String name)` that returns the input stream for the resource file.
- `java.lang.System.getProperty(String name)`. When passed "microedition.locale" as a parameter this returns the system locale of the configuration.

In theory this could be just enough support for I18N for simple applications. However using the `getResourceAsStream` to get the localized user interface strings is rather

complicated. This is because `getResourceAsStream` is a very low level function. It just gives you the access to raw resource file data. The J2ME programmer needs to write a large amount of code to get the localized user interface strings from the resource file.

J2SE contains two kinds of resource files: property files and list files. The property file contains the translations of the strings in one language, one translation in a row. List files use similar approach but they represent data inside a Java class. Without having the resource bundle class a J2ME programmer has two choices to localize his or her application.

- Write own code on the top of `java.lang.Class.getResourceAsStream`. This is a complicated task and the memory consumption might be too high.
- Change the strings in the java source code. This might seem as an easy solution but it leads the programmer into troubles if the application source code changes. Either the programmer needs to do the same changes to every single localized java code or retranslate the localized java source codes again. Both approaches are difficult to implement, slow and error prone.

Because J2ME doesn't support property files (through `PropertyResourceBundle`) nor list files (through `ListPropertyFiles`), Multilizer contains a small footprint resource bundle class and format that is suitable for J2ME.

Localization of Resource Bundles

With Multilizer you are able to globalize your J2ME applications. The `multilizer.microedition.Properties` class has the main role. Using it is pretty similar to using J2SE `PropertyResourceBundles`. It's even possible to use your old J2SE property files directly through it.

`multilizer.microedition` works on a top of the CLDC configuration. `multilizer.microedition` is profile independent so it works with any CLDC profile such as MIDP and PDA profile. It is also small (3 Kbytes) and memory efficient.

Property files may be UTF-8 or ISO8859-1 encoded. This makes the `Properties` class backward compatible to ISO8859-1 encoded property files. Using UTF-8 encoding makes the files more understandable, because you don't have to use Unicode escapes for non-ASCII characters. UTF-8 also helps conserving the memory because none English UTF-8 files are considerably smaller than Unicode escaped ASCII files.

The following picture describes the J2ME localization process.

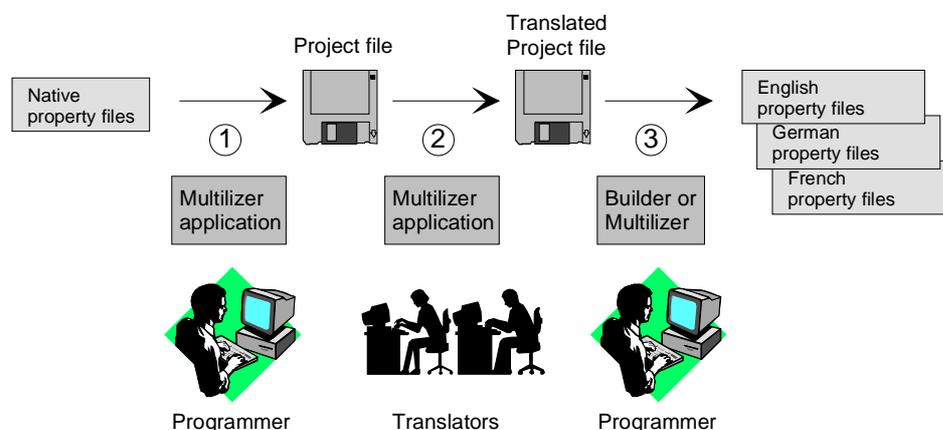


Figure 141 J2ME localization process

The programmer uses Multilizer to extract strings from the original property file (1). Multilizer saves these strings to the project file. The programmer sends the project file to the translator(s) that use Multilizer to translate the project file (2). The programmer uses Multilizer or Builder to create the localized property files (3). As the result there will be one property file for each localized language.

The following figure shows the files that Multilizer uses on the J2ME localization process.

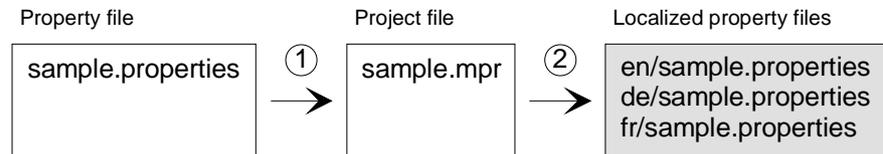


Figure 142 The files of the J2ME localization process

Add the localized property file (e.g. `de/sample.properties`) instead of the original property file (`sample.properties`) to the setup package when building a localized application.

Application with an English User Interface

There is one J2ME sample in `<mdir>j2ME/dcalc` directory. It is an English version of Dcalc. Open it, compile it, and finally run it.

Depending on the selected emulator, the application looks like this for example:

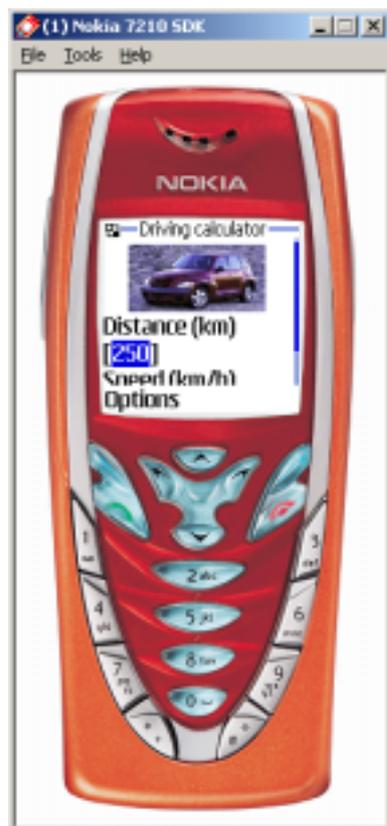


Figure 143 J2ME application with an English UI

The user interface language is English. In the following chapters we will review how Dcalc was localized.

Internationalization

To globalize a J2ME application you have to resource it. Resourcing means getting rid of hard coded strings. Most applications contain strings that have been inserted inside the source code. These strings are hard coded. It is impossible to localize such code without changing and recompiling the code. When you resource the code you take the strings from the source code and place them to a resource file that can easily be translated.

The main source code file of the Dcalc sample application is `dcalc/Dcalc.java`.

Use property files

First task is to create the property file and take it in use in the midlet. The property file used by Dcalc has the same name as the midlet: `dcalc.properties`. Using the property file in the midlet is easy. Just create a `multilizer.microedition.Properties` instance and pass the property file name as the parameter.

```
public class Dcalc extends MIDlet implements CommandListener
{
    private Properties prop = new Properties("/dcalc/Dcalc.properties");
    ...
}
```

Use `Properties(String)` if your property files are in UTF-8 format. This is the case in the code above.

If you want to use the same format (ISO8859-1) as J2SE property files use `Properties(String, int)` instead.

Add strings to property files

The property file format is `key<separator>value`. Where the key is the string inside the `getString` method and the value is the translation. The separator can either be a tab or the '=' character.

All hard-coded strings of Dcalc were added in property file, which looks like this:

Distance (km)	Distance (km)
Speed (km/h)	Speed (km/h)
Driving calculator	Driving calculator
Calc	Calc
About	About
Exit	Exit

Use format function

In `Message.java` file there was originally a concatenated string like this:

```
alert.setString(
    new Integer(hours) + " hours " +
    new Integer(minutes) + " minutes");
```

We could resource "hours" and "minutes" by wrapping them with the `getString` method. However that would not be very good internationalization because the above code always assumes that the message has the following form <hour value> <hour label> <minute value> <minute label>. There are languages that use the label first and the value next. Also some countries prefer to show the minutes first and hours next.

To solve this problem `multilizer.microedition.MessageFormat` class is used; there the whole message is put in the message pattern and the pattern is combined with data at run-time to produce the actual message.

```
Object[] args = {new Integer(hours), new Integer(minutes)};

alert.setString(MessageFormat.format(
    prop.getString("{0} hours {1} minutes"),
    args));
```

Now the property file contains the message pattern, `{0} hours {1} minutes`. The translator can easily relocate the items in the pattern.

Load strings from property files

Resourcing the code is done with the `getString` method. In general you should wrap every single string that need to be translated inside the `getString` method. Thus instead of writing code like this

```
private StringItem distanceLabel = new StringItem(
    null,
    "Distance (km)");
```

... Dcalc uses this

```
private StringItem distanceLabel = new StringItem(
    null,
    prop.getString("Distance (km)"));
```

Conclusions

As shown in the code samples above, Dcalc source code is totally language-independent. To localize the sample, the only task will be to create localized property files. This is discussed in next chapter.



This chapter has covered only basic internationalization (I18N). Refer to I18N books and web sites to get more information. An excellent start is the Java tutorial at www.javasoft.com.

Create Multilizer Project

In order to localize J2ME software, you have to create a Multilizer project. This is described in the first part of the manual, chapter Create Project, p. 11.

Localizable data of J2ME application is located in Resource Bundles (`.properties`).

Specify Localization options

After finishing the Wizard, you have to specify the localization options for the software. Normally default options are the most useful, but in this tutorial we will review all options.

Right-click the localization target in Project Tree, and click properties to see Delphi Target options.



All J2ME-specific options are gathered under the corresponding target (Java Resource Target) dialog. If there are many targets in one project, you can set different localization options to all, if needed.

Encodings

Encodings tab shows the languages of project. UTF-8 is the native encoding of Multilizer's Java property files.

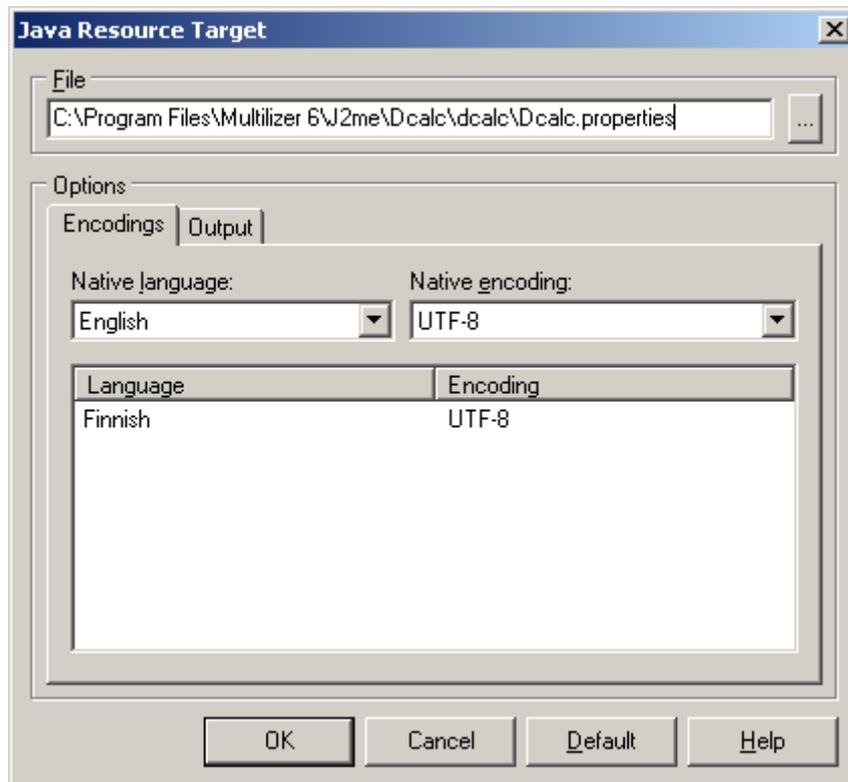


Figure 144: Encodings for localized software.

Output

Output directory lets use specify where to store localized items.

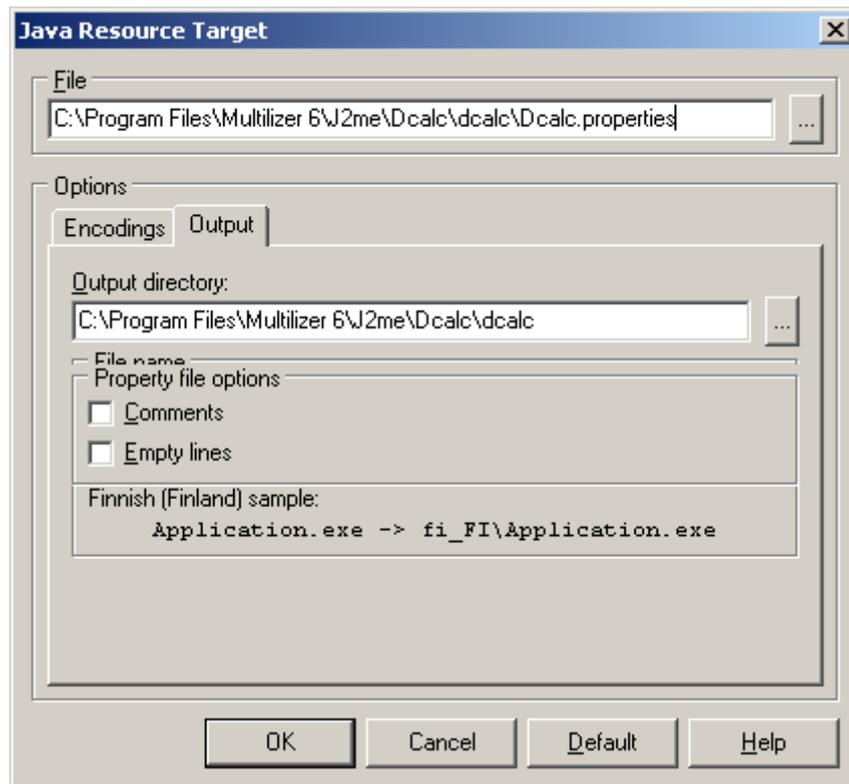


Figure 145: Output options for Java software.

Translate Project

For testing purpose, you can translate the software by using pseudo-languages (→ Pseudo language, p. 68); right-click language column, choose properties, and select the pseudo-language options. This will fill translation grid with pseudo-language translations.

More info



Refer to following parts of the manual for more information on translating software, and sharing translation work between team members.

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized application files by choosing **Project | Build Localized Files**. This creates the localized files.

After this, the localized versions of the application can be created. Edit for example the build script to include the right property files for each language's setup packages.

Finally run localized software in emulator to see how it works. Depending on the selected emulator, the application looks like this for example:

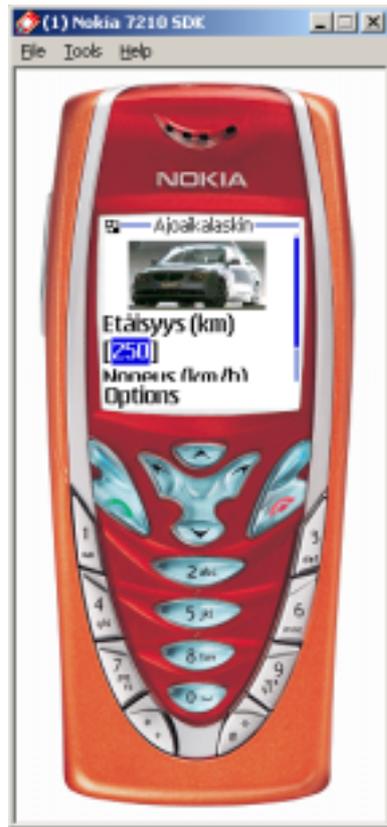


Figure 146: Localized Java software.



In order to run localized software, you need to specify location of JDK. See next chapter for more info.

Configure J2ME options

J2ME emulator and other specific options are configured in Multilizer options dialog; select **Tools**→**Options...**→**Java**, and click **J2ME** options.

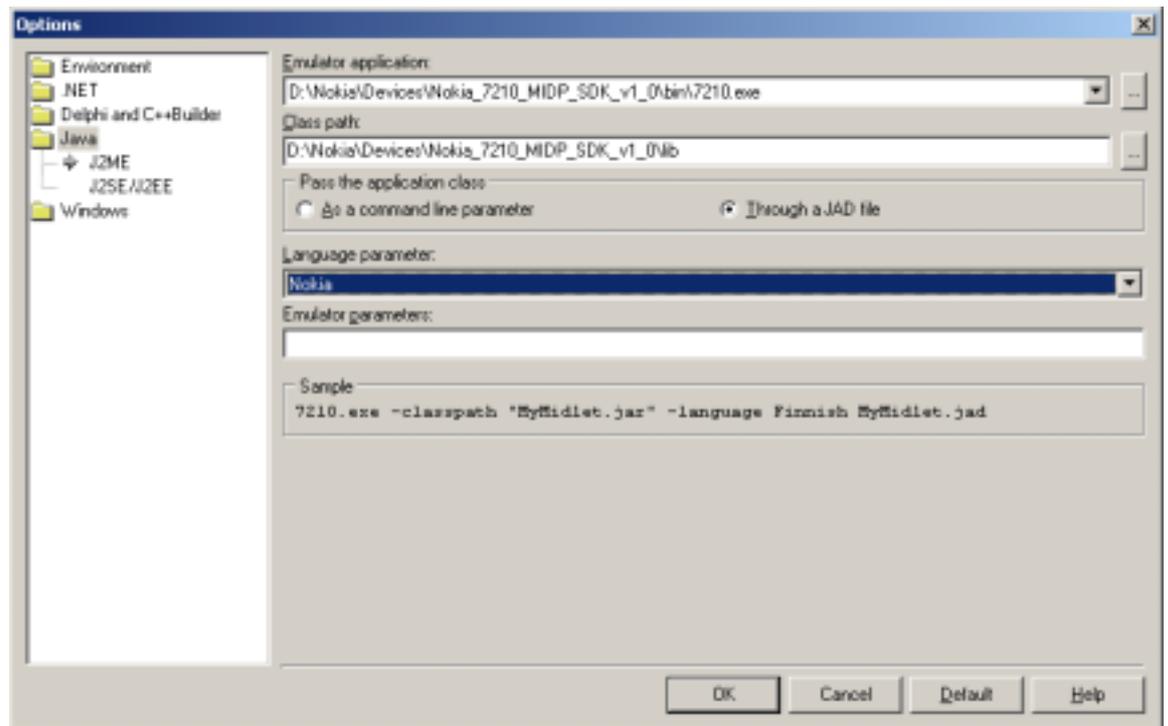


Figure 147: Specifying J2ME options.

Refer to Multilizer on-line help for more info.

13

Database Tutorial

This tutorial describes localization of databases.

Required product(s):	Multilizer Enterprise Multilizer for Windows
Sample(s):	<none>
Tutorial(s):	<none>

Introduction

Multilizer supports localization of Interbase databases.

Localization of databases

Database cloning

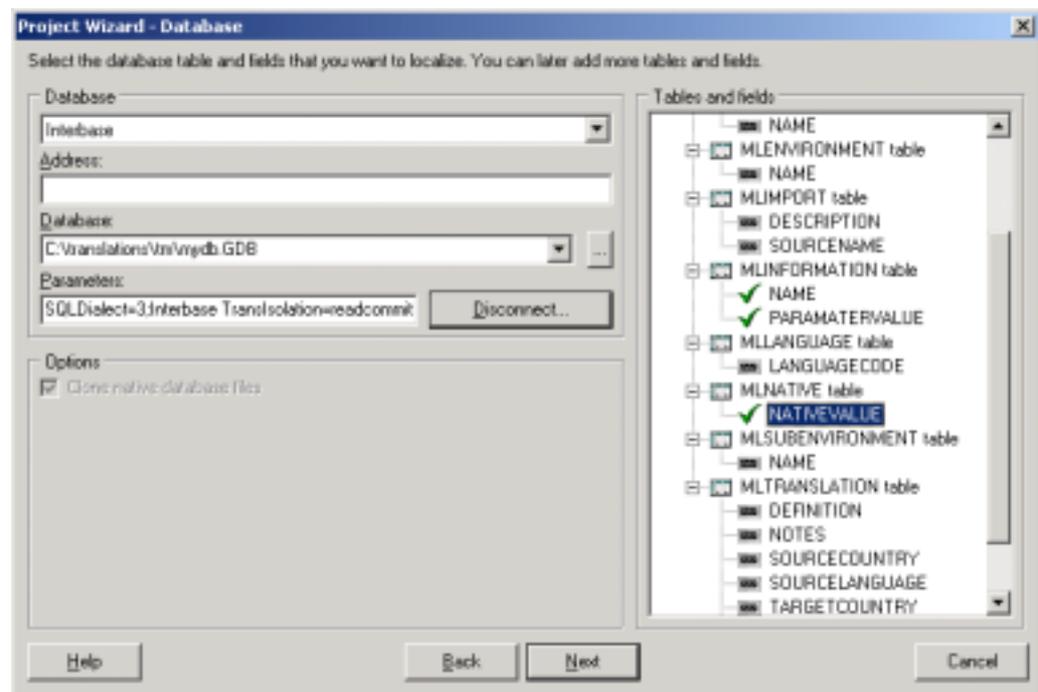
Multilizer localizes databases by copying the original database and populating the database with translations of Multilizer project. This approach is called database cloning; the structure of original database and localized database are identical.

Create Multilizer Project

In order to localize a database, you have to create a Multilizer project. This is done with Project Wizard. Start Project Wizard by choosing **File→New....** Choose 'Localize a database' on first page of Wizard.

On next page user defines what database, and what fields, to localize.

Database cloning settings



Database must be either Interbase or MS Access. Database cloning is not support for other databases.

Address

In database cloning Address is left empty.

Database

Ensure that you can browse to the database file. If you can't browse to the database file, database cloning will not work.

Parameters

If database doesn't include date values, SQLDialect can be 1. Otherwise SQLDialect 3 needs to be used.

Tables and Fields.

Doubleclick fields that you want to localize. Right-clicking a field lets you specify, if certain fields are added as localizable fields or comments in Multilizer project.

Fields that are not marked are cloned as such when building localized versions.

More info

More info on Project Wizard is found in the first part of the manual, chapter Create Project, p. 11.



Specify Localization options

After finishing the Wizard, you can review the localization options for database localization. Normally default options are the most useful, but in this tutorial we will review all options.

Right-click the localization target in Project Tree, and click properties to see Database Target options.

Encodings

Encodings tab shows the languages of project. You can also review the original language and character set here.

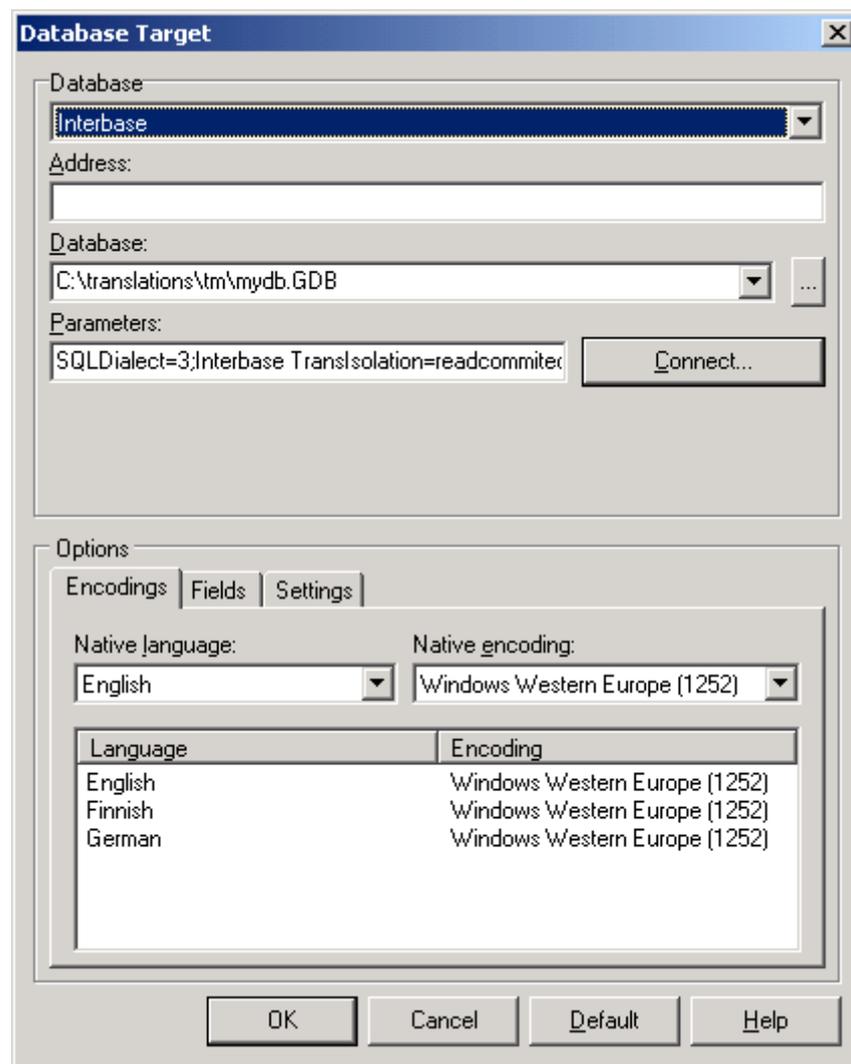


Figure 148: Encodings of database.

Fields

In order to review fields of the database, you have to press connect button. After connecting you will have the same view to fields as in Project Wizard.

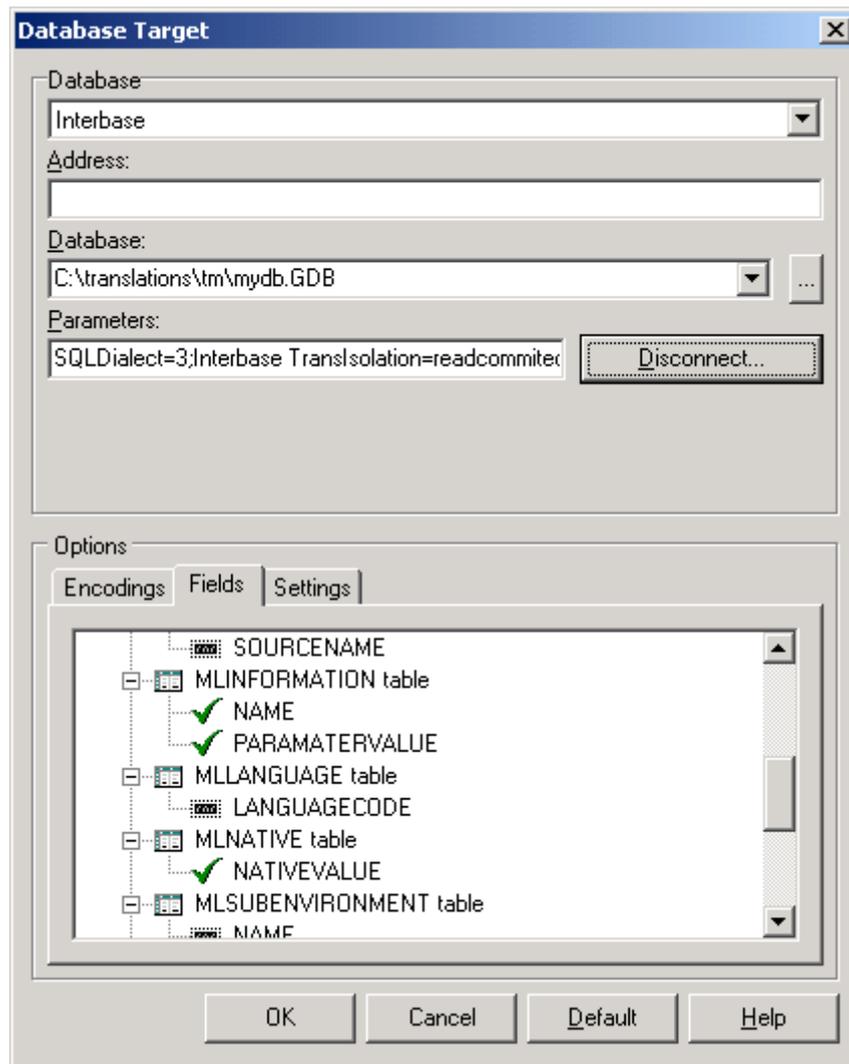


Figure 149: Localizable fields.

Settings

For database cloning there is only one setting to choose from: enable/disable size checks on build.

Enabling this setting will check that translations fit in the fields.

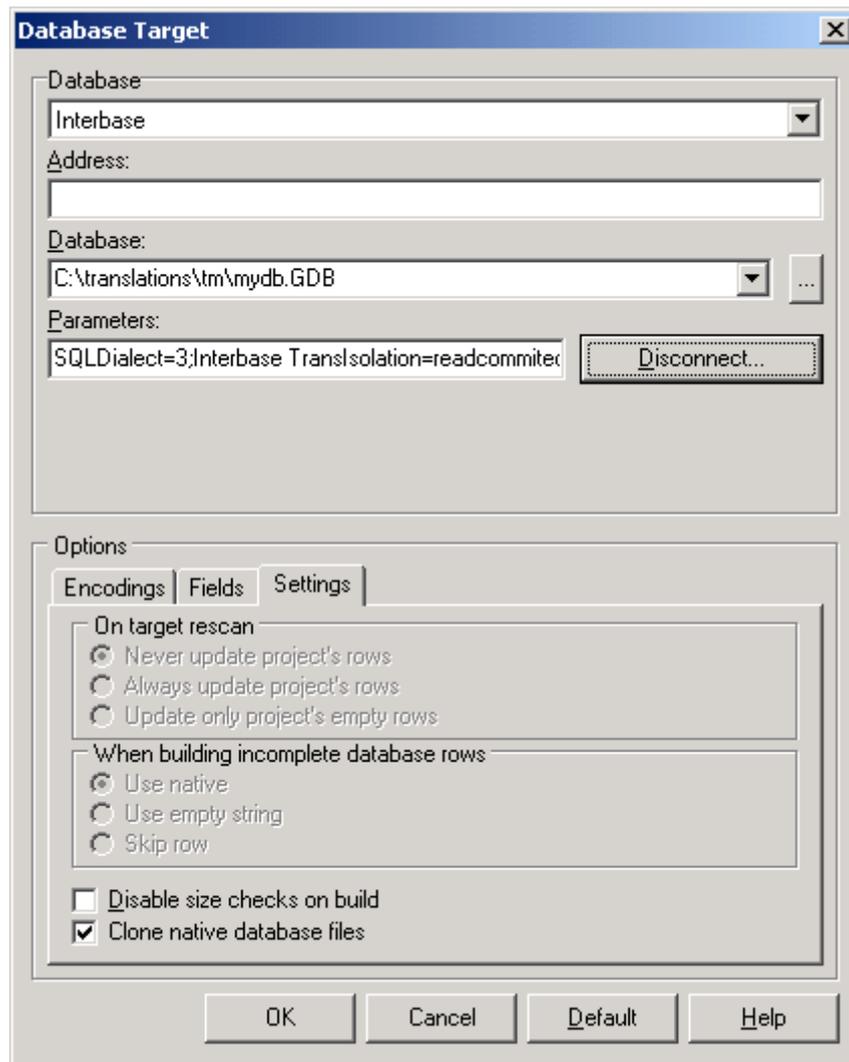


Figure 150: Database cloning settings.

Translate Project

For testing purpose, you can translate the database by using pseudo-languages (C.f. Pseudo language, p. 68); right-click language column, choose properties, and select the pseudo-language options.

This will fill translation grid with pseudo-language translations.

Translation view

Translation view is optimized for simple navigation in database localization projects; Project tree shows the localizable tables and fields, and lets the user navigate between them.

Translation grid shows the original (native) language and user can choose the visible target language.

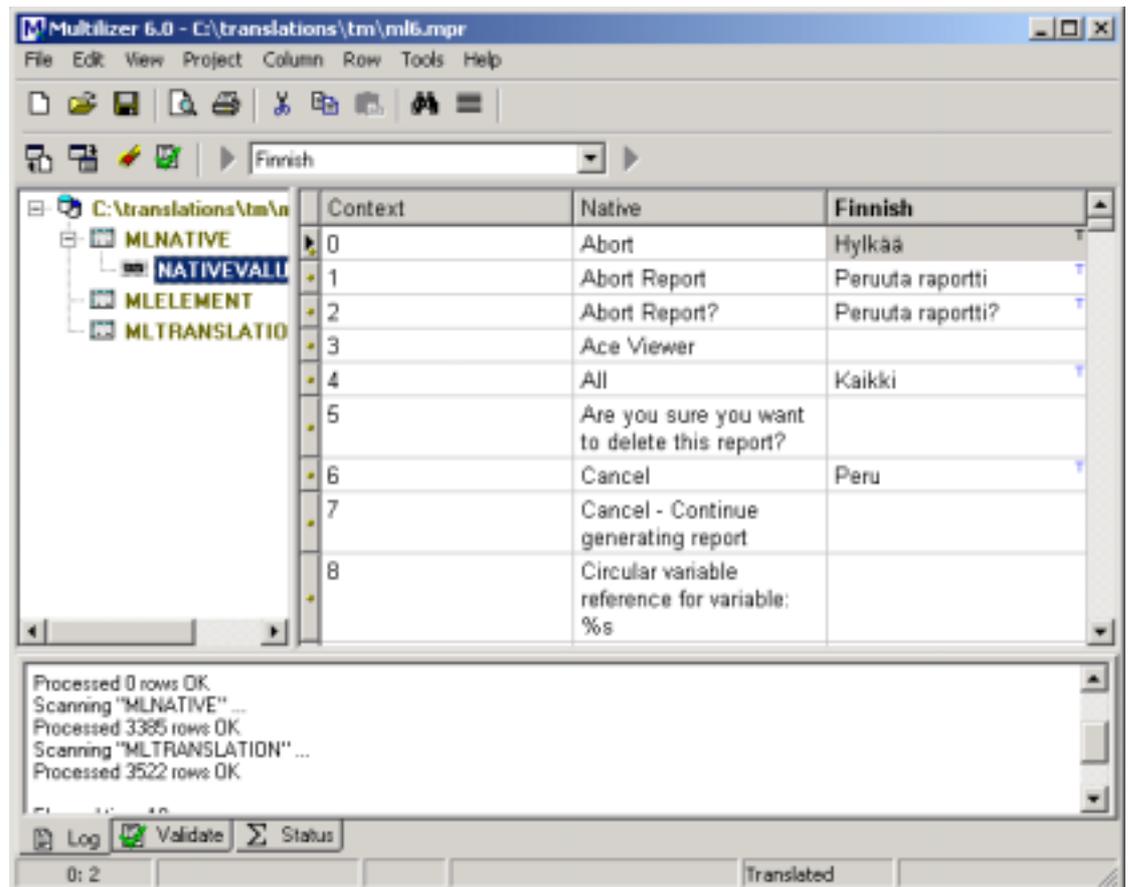


Figure 151: Localization view for translating database contents.

More info



Refer to following parts of the manual for more information on translating software, and sharing translation work between team members.

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized files by choosing **Project | Build Localized Files**. This creates the localized database files.

14

XML Tutorial

This tutorial describes localization of any XML file.

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java
Sample(s):	<none>
Tutorial(s):	<none>

Introduction

XML (Extensible Markup Language) files consist of markup tags that form elements. There can be data inside the elements, or nested elements. Inside tags there can be attributes.

XML-files are normally Unicode-encoded, but other encodings are widely in use. XML-files can contain just any data.

Multilizer is able to localize XML-file. It also shows XML-file visually, and bitmap data is shown as an image, for example. Multilizer's wysiwyg for XML makes it extremely easy to translated XML contents in context.

Localization of XML

Multilizer localizes XML by reading in original file, and writing out either localized files or one multilingual file. For multilingual files, Multilizer uses xml:lang attribute as defined in XML-standards.

Multilizer never overwrites original source code file.

Create Multilizer Project

In order to localize a source file, you have to create a Multilizer project. This is done with Project Wizard. Start Project Wizard by choosing **File→New....** Choose 'Localize a file' on first page of Wizard.

On next page, select 'XML file' filter, and specify the source code file to localize.

Click next after selecting the XML file to localize.

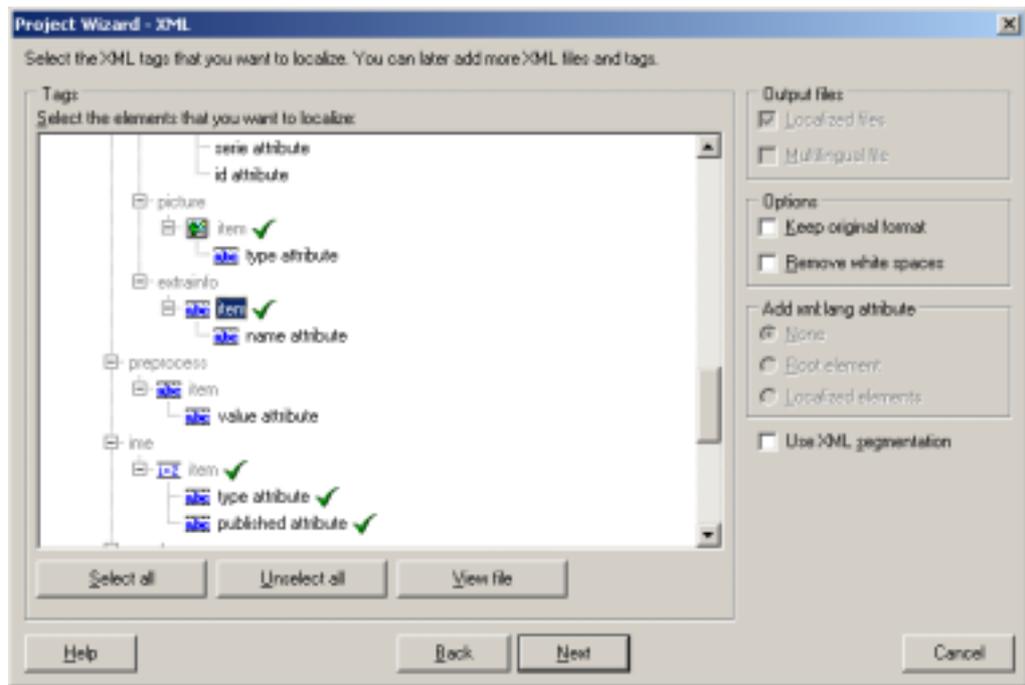


Figure 152: XML-options in Project Wizard.

Multilizer lets use define all tags and attributes that need localization.

By default Multilizer interprets data as plain text. Because XML can contain just any data, Multilizer lets users define the way to interpret data; right-click any node and select properties to define it.

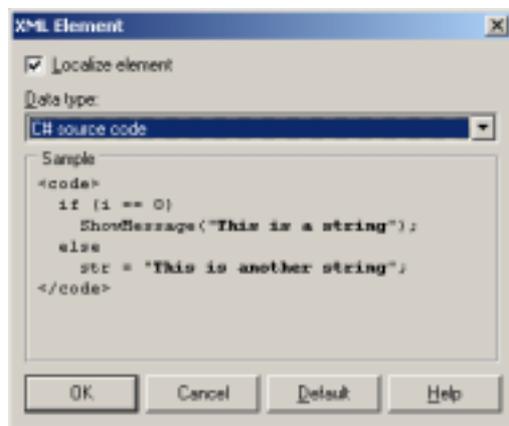


Figure 153: Defining an element containing C# source code.

Source code embedded in XML

If XML element contains source code, Multilizer can parse it. This enables translation of strings in source code without touching the code part. (→ Source Localization Tutorial, p. 167).

In order to interpret a XML element as source code, right-click a node and select any of the source code formats.

Translate Project

It's extremely easy to translate XML; Multilizer shows the file as Wysiwyg, allowing translators to see the context. Translations are simply written in the grid, and translator will see it visually too.

User can navigate in the XML; strings are high-lighted and clicking a string will scroll the translation grid to corresponding location.

User can also navigate in translation grid, and corresponding string is shown in XML view.

Translation of embedded source code

Strings in embedded source code are translated just as any string in the project. Just as explained in Source Localization Tutorial (p. 167), Multilizer shows the strings in source code and allows the translation of them without touching the code itself.

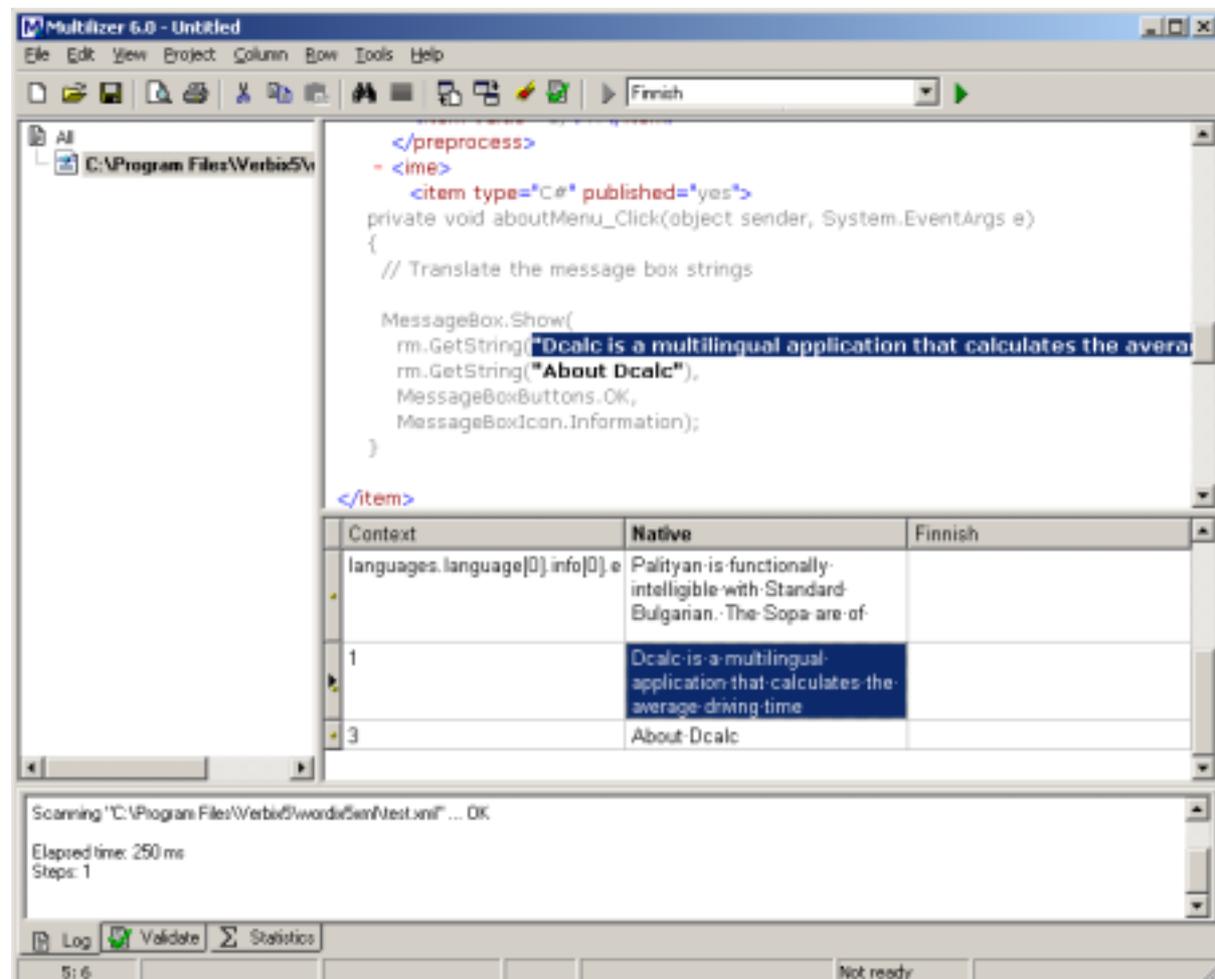
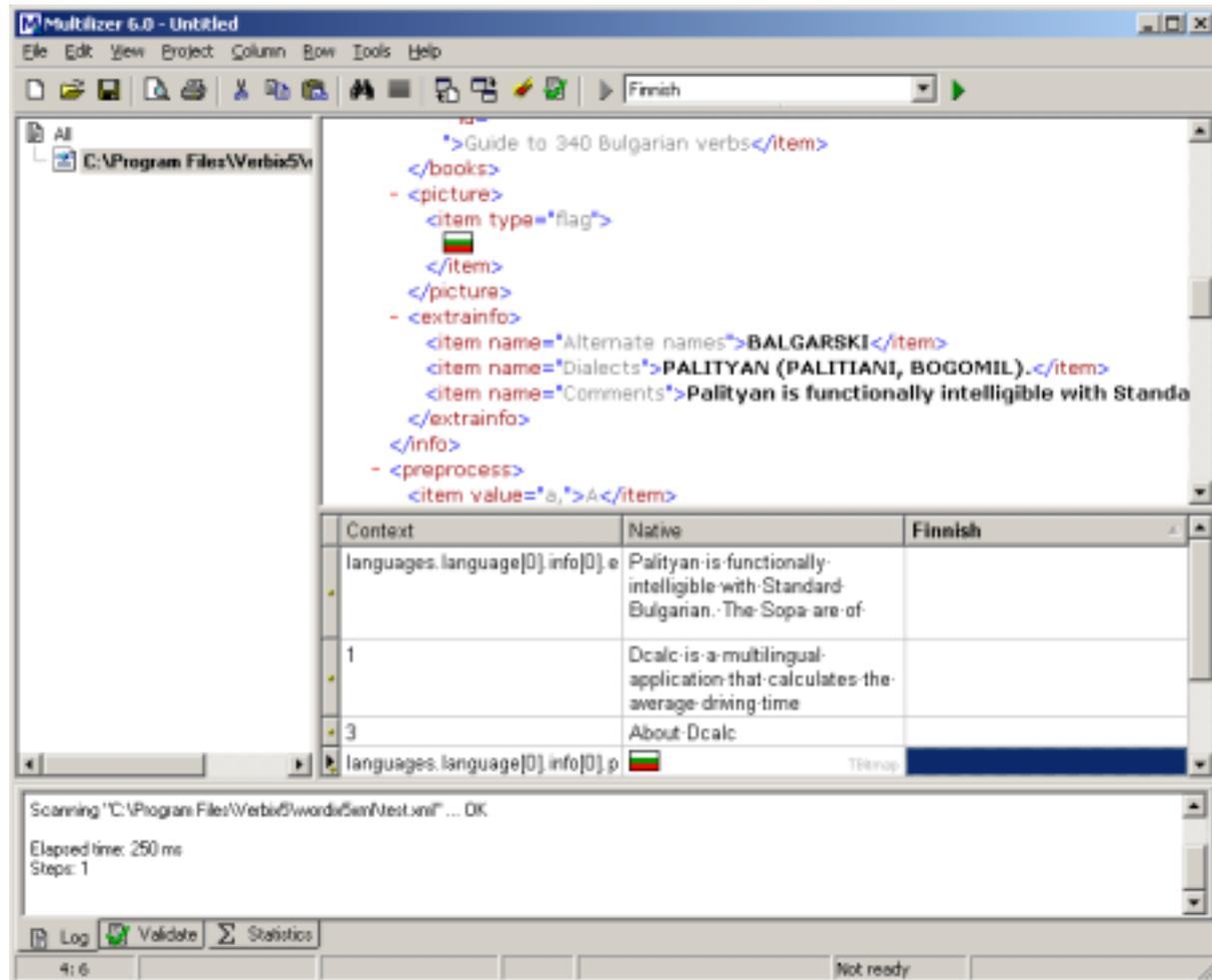


Figure 154: Localization of strings in C# source code embedded in XML.

Localization of bitmaps

Multilizer is able visualize the localization of embedded bitmaps.



More info



Refer to following parts of the manual for more information on translating software, and sharing translation work between team members.

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized files by choosing **Project | Build Localized Files**. This creates the localized files.

15

Source Localization Tutorial

This tutorial describes localization of any source file formats that can be localized with Multilizer.

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java
Sample(s):	<none>
Tutorial(s):	<none>

Introduction

Localization of single source code files, as described in this tutorial, should be used in special cases only. If possible, use binary localization or any other platform specific localization as described in other tutorials.

Localization of source codes

Multilizer detects strings from various source code formats, and using Multilizer for translation makes the work safe; Multilizer doesn't allow modification of code, but just translation of strings.

Multilizer localizes source code files by reading in original source file(s), and writing out localized files, one for each language. Multilizer never overwrites original source code file.

Create Multilizer Project

In order to localize a source file, you have to create a Multilizer project. This is done with Project Wizard. Start Project Wizard by choosing **File→New....** Choose 'Localize a file' on first page of Wizard.

On next page, select 'Source code file' filter, and specify the source code file to localize.

Then continue Project Wizard as usually. More info on Project Wizard is found in the first part of the manual, chapter Create Project, p. 11.

Translate Project

It's extremely easy to translate source code file; Multilizer shows source code as Wysiwyg, allowing translators to see the context. Yet, the source code can't be altered. Translations are simply written in the grid.

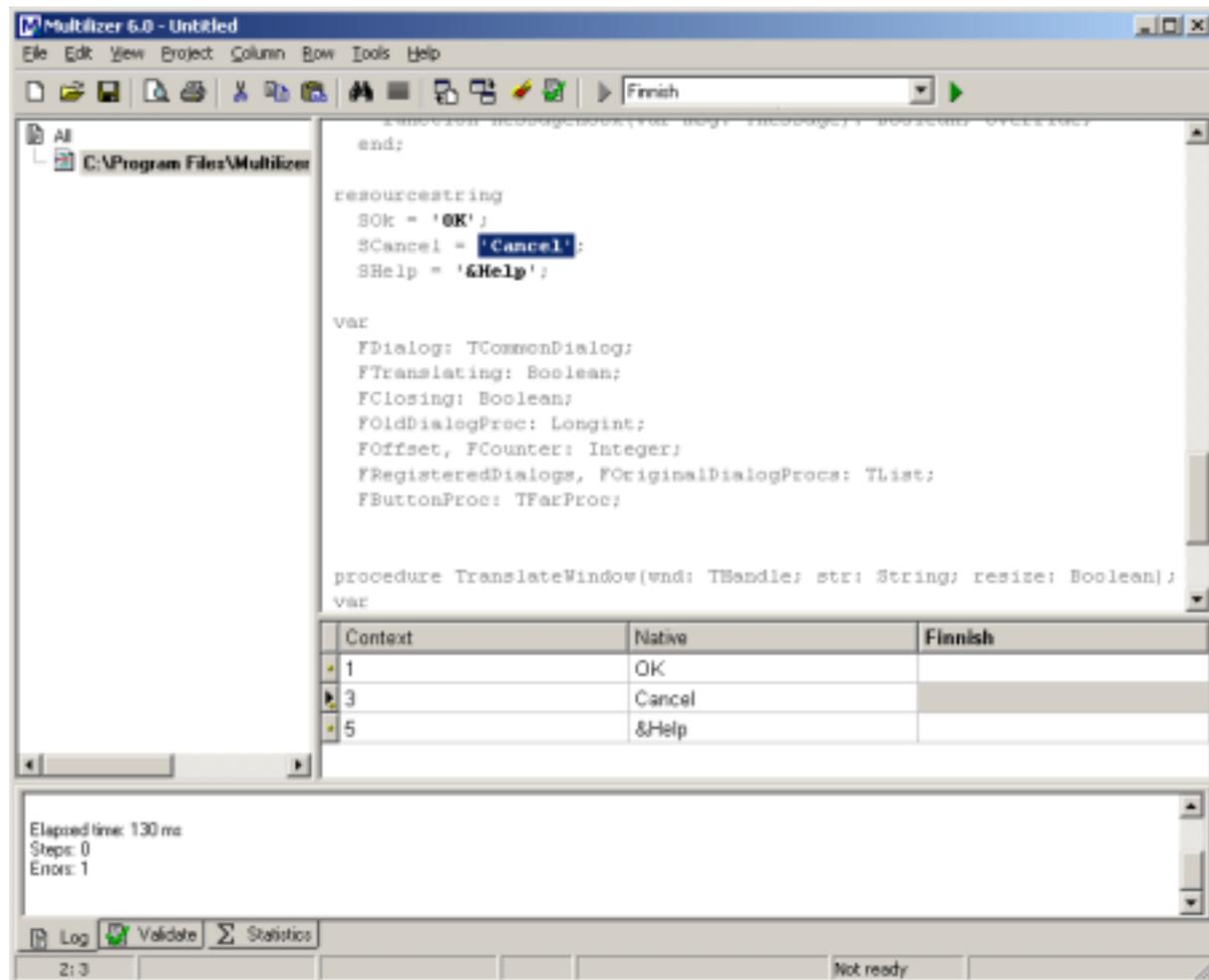


Figure 155: Localization view for translating source code.

User can navigate in the source code; strings are high-lighted and clicking a string will scroll the translation grid to corresponding location.

User can also navigate in translation grid, and corresponding string is shown in code view.

More info



Refer to following parts of the manual for more information on translating software, and sharing translation work between team members.

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized files by choosing **Project | Build Localized Files**. This creates the localized files.

16

Data File Localization Tutorial

This tutorial describes localization of any data files that can be localized with Multilizer.

Required product(s):	Multilizer Enterprise Multilizer for Windows Multilizer for .NET Multilizer for Visual C++ Multilizer for VCL Multilizer for Java
Sample(s):	<mldir>/data/key/product/sample.txt <mldir>/data/ini/product/sample.txt
Tutorial(s):	<none>

Introduction

Besides localizing resource files and source code files, Multilizer also localizes data files, such as INI-files, SHL-files, KEY-files, and XML.

INI-files are typical configuration files for Windows software. SHL-files are used by InstallShield®, and they contain strings shown in installation software.

This tutorial covers localization of INI, SHL, and KEY files. XML localization is described in another tutorial.

Localization of data files

Multilizer detects strings from various data file formats, and using Multilizer for translation makes the work safe; Multilizer doesn't allow modification anything but the strings.

Multilizer localizes data files by reading in original file(s) and writing out localized files, one for each language. Multilizer never overwrites original file.

Create Multilizer Project

In order to localize a data file, you have to create a Multilizer project. This is done with Project Wizard. Start Project Wizard by choosing **File→New....** Choose 'Localize a file' on first page of Wizard.

On next page, select 'Ini file' or 'Key file' filter, and specify the file to localize.

Ini/shl file specifics

Project Wizard allow users to specify which sections and keys in INI/SHL files require localization. Only selected keys will appear in Multilizer translation grid.

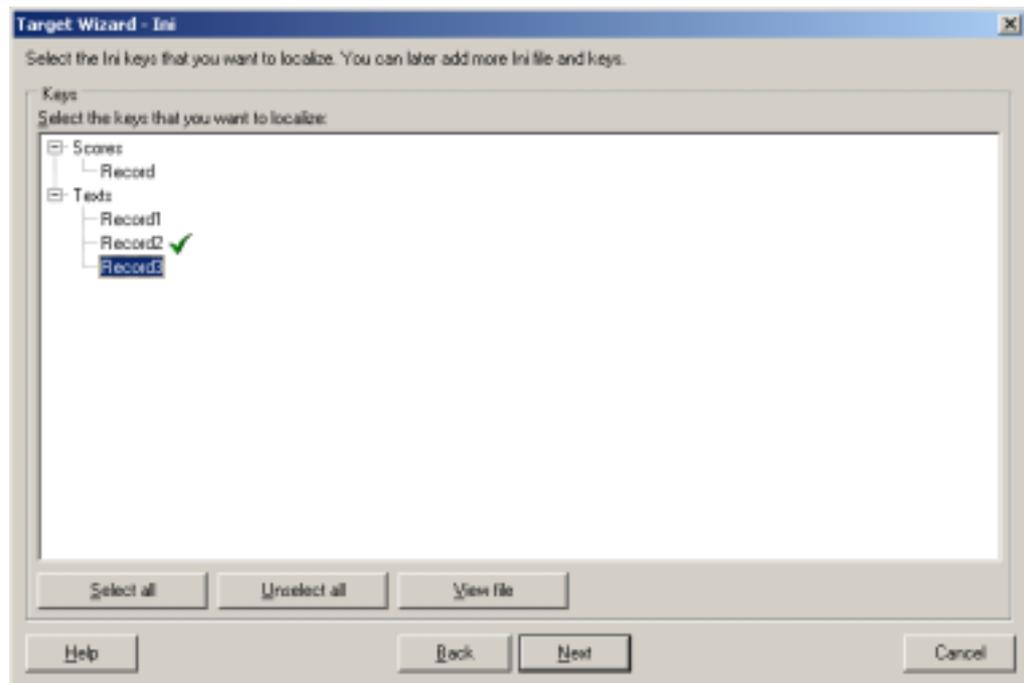


Figure 156: Selecting INI-file keys for translation.

Then continue Project Wizard as usually. More info on Project Wizard is found in the first part of the manual, chapter Create Project, p. 11.

Translate Project

Translation is done in translation grid. Only those strings defined for translation are shown in the grid.

INI/SHL file properties

In order to redefine keys for translation, right-click corresponding target in project view, and choose properties. In target dialog 'Keys' tab will show the keys that are marked for localization.

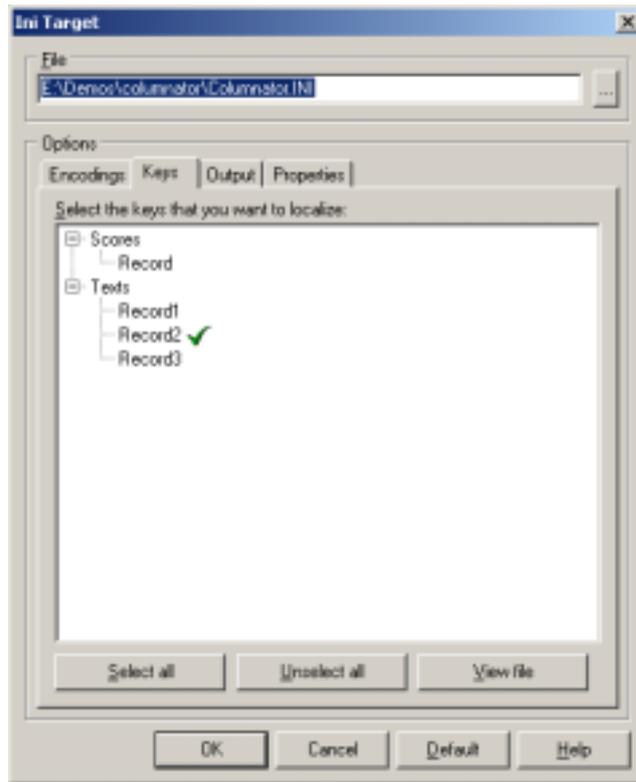


Figure 157: INI-file target; keys marked for localization.

Key file properties

Key-files are extremely simple text files. Each row contains a key-value pair. In order to redefine exact format, right-click corresponding target in project view and choose properties. In target dialog 'Format' tab will show the format specification.



Figure 158: Key-file target; defining key file format.

Key-files are extremely simple text files. Each row contains a key-value pair. In order to redefine exact format, right-click corresponding target in project view and choose properties. In target dialog 'Format' tab will show the format specification.

Another important option is the encoding shown on 'Encodings' tab; it allows user to specify encoding for each target language. User can also set the encoding for original file, if Multilizer failed in detecting it correctly.

More info



Refer to following parts of the manual for more information on translating software, and sharing translation work between team members.

- Pre-translate project, p. 25
- Prepare project for translation, p. 26
- Share translation work, p. 27
- Translate, p. 44

Build Localized Versions

Create the localized files by choosing **Project | Build Localized Files**. This creates the localized data files.

Part III: Appendices

Index

- .
- .NET, 7
- A**
- Auto-navigation, 67
- B**
- Binary, 92
- C**
- CLDC, 148
- Close-to matches, 58
- D**
- DateTimeToStr, 97
- DLL, 103
- E**
- Enterprise, 7
- Exchange package
 - Secure the contents, 30
- F**
- filter strings, 30, 37
- Format, 97, 100
- Fuzzy matching, 58
- I**
- Info page, 18
- Installation, 8
- InstallShield, 169
- Internationalization, 77, 93, 122
- ITE, 105
- J**
- J2SE, 148
- K**
- Kilometer, 97
- km/h, 97
- L**
- language
 - native, 11
- Localization
 - binary, 92
- Localization Kit, 29
- localization targets, 31
 - target, 11
- M**
- Metric system, 100
- Mile, 97
- mph, 97
- MPR, 11
- Multilizer editions, 7
- Multilizer package
 - MLP, 32
- Multilizer Project, 11
- Multilizer project file
 - creating, 28
- N**
- native*, 11
- P**
- Perfect matches, 58
- Project
 - Statistics, 71

project tree, 18

Project view, 18

Q

quick fix, 67

R

Resource string, 94

resource type, 19

resources

 bitmap, 20

 cursors, 20

 Dialog, 20

 Form, 20

 icons, 20, See

S

SDK, 120

SDLX, 36

Statistics, 71

Stylesheet

 XML, 72

System

 registry, 103

T

target, 11, 20

add, 21

modify, 21

remove, 21

text-file, 35

TMX, 34

TRADOS, 35

translation grid, 20

Translation Kit

 Secure the contents, 30

translation work-place, 18

Translator's Workbench, 35

U

USA, 97, 100

UTF-8, 149

V,W

VCL, 7

Visual C++, 75

wysiwyg, 20

Wysiwyg, 89, 111, 133

X

XML, 71

XSL, 72

Table of figures

Figure 1: Organizing the files to localize.....	12
Figure 2: Selection between localizing file and localizing databases.....	13
Figure 3: Specifying files to be included in project.....	13
Figure 4: Specifying file type.....	14
Figure 5: Specifying file type.....	15
Figure 6: Specifying project information for new project.....	16
Figure 7: Selecting languages for project.....	16
Figure 8: Multilizer project view, with project tree, translation work-place (translation grid and visual editor), and info page.....	19
Figure 9: Project tree with mainform resource shown in bold.....	20
Figure 10: <i>Dialog displaying project targets (Project→Targets...)</i>	21
Figure 11: <i>Adding new target by file type</i>	22
Figure 12: <i>Translation Workplace; translation grid and visual editor</i>	23
Figure 13: <i>Info Page</i>	24
Figure 14: <i>Log view shows information of scan, make, and build processes</i>	24
Figure 15: <i>Statistics panel with quick info of translations</i>	25
Figure 16: Default work-flow with Multilizer.....	28
Figure 17: Running Exchange Wizard.....	29
Figure 18: Specifying the targets and resources to be exchanged.....	30
Figure 19: Filtering rows that are exchanged.....	30
Figure 20: Specifying project info, password protection.....	31
Figure 21: Adding targets in exchange package.....	31
Figure 22: Specifying additional files to be included in exchange package.....	32
Figure 23: Specifying the name for exchange package.....	32
Figure 24: Building the exchange package.....	33
Figure 25: Running Export Wizard.....	34
Figure 26: Specifying export file and TMX file format.....	34
Figure 27: Export settings for TRADOS-compatible TMX.....	35
Figure 28: Specifying export file and text file format.....	36
Figure 29: Filtering of rows that are exported.....	37
Figure 30: Running Import Wizard.....	38
Figure 31: Specifying the file to be imported.....	39
Figure 32: Selecting file type for the file to be imported.....	39
Figure 33: Specifying Import options.....	40
Figure 34: Specifying languages to be imported from TMX file.....	41
Figure 35: Import options for TMX.....	41
Figure 36: Specifying file format for importing text file.....	42
Figure 37: Selecting language for importing Microsoft glossary.....	43
Figure 38: Translation view in Multilizer.....	45
Figure 39: Choosing visible columns in translation grid.....	45
Figure 40: Options for filtering by data type.....	46
Figure 41: Options for filtering by translation status.....	47
Figure 42: Options for filtering by row status.....	47
Figure 43: Options for filtering strings by Do not translate -status.....	48
Figure 44: Translation grid options.....	48
Figure 45: Visual editor for forms and dialogs.....	49
Figure 46: Visual editor for menus.....	50
Figure 47: Options for visual dialog editors.....	50
Figure 48: Translation grid with cells showing non-visible characters.....	51
Figure 49: Accelerators view.....	52
Figure 50: Localizing an accelerator.....	52
Figure 51: Localizing images.....	53
Figure 52: <i>Translation Memory administration</i>	57
Figure 53: <i>Translation Memory; setting matching options</i>	58
Figure 54: <i>Adding properties of a new Multilizer Translation Memory</i>	59
Figure 58: <i>Translation Memory; general settings</i>	61
Figure 59: <i>Translation Memory; importing of documents</i>	61

Figure 60: <i>Translation Memory; specifying block words.</i>	62
Figure 62: <i>Translation Memory maintenance tab.</i>	63
Figure 63: Selecting validations to perform.	65
Figure 64: <i>Displaying validation results.</i>	67
Figure 65: Defining Pseudo Translation properties.	68
Figure 66: <i>Project report.</i>	71
Figure 67: <i>Validation report options.</i>	72
Figure 68: <i>Specifying style sheet for reports.</i>	72
Figure 69: Binary localization process.	76
Figure 70: Driving time calculator with English user interface.	76
Figure 71: English Visual C++ Windows CE application.	77
Figure 72: Insert Resource dialog box.....	78
Figure 73: String Table editor.	78
Figure 74: Replace dynamic items with dummy strings.	83
Figure 75: Resize dynamic items for long translations.	84
Figure 76: Encoding options for target languages.	85
Figure 77: Output options for localized files.	86
Figure 78: The files of the binary C++ localization process in Windows.	86
Figure 79: Font options for localized software.	87
Figure 80: Specifying the resources types to localize.	88
Figure 81: Specifying the platform of localized software.	89
Figure 82: Localizing forms visually.	90
Figure 83: Localized Dcalc application (Windows).	91
Figure 84: Localized Dcalc application (Windows CE).	91
Figure 85: Binary localization process of a VCL application.	92
Figure 86: Dcalc application using an English user interface.	93
Figure 87: Message displayed when time is less than 1 hour.	101
Figure 88: Message displayed when time is 1 hour.	101
Figure 89: Message displayed when time is more than 1 hour.	101
Figure 90: The internationalized Dcalc form on Delphi IDE.	102
Figure 91: The internationalized Dcalc application running with Finnish locale.	103
Figure 92: Delphi-specific settings for VCL binary target.	105
Figure 93: Message box telling that there are existing ITE translations.	105
Figure 94: Delphi target options.	106
Figure 95: Encodings for target languages.	107
Figure 96: Output options for localized Delphi software.	108
Figure 97: The files of the binary localization process of a VCL application.	108
Figure 98: Font options for localized software.	109
Figure 99: IME options for software localized to Far Eastern languages.	110
Figure 100: Specifying the resources to be localized.	111
Figure 101: Localizing Delphi forms visually in Multilizer.	112
Figure 102: Localizing menus visually.	112
Figure 103: Running localized software.	113
Figure 104: Excluding properties by components in VCL binary localization.	114
Figure 105: Excluding properties by name in VCL binary localization.	114
Figure 106: Visual form editor with an unknown VCL component.	115
Figure 107: Specifying visual representation for VCL control.	115
Figure 108: Visual Editor recognizing all VCL controls.	116
Figure 109: Visual Studio .NET file hierarchy.	118
Figure 110: Visual Studio .NET project file localization process.	119
Figure 111: The files of the Visual Studio .NET project file localization process.	119
Figure 112: Borland Delphi 8/C#Builder file hierarchy.	120
Figure 113: .NET resource file localization process.	121
Figure 114: The files of the .NET resource file localization process.	121
Figure 115: Dcalc application using an English user interface.	122
Figure 116: To localize the form set the Localizable property to true.	122
Figure 117: Resource file after adding the first item.	123
Figure 118: Encodings for localized software.	129
Figure 119: .NET Localization options.	130
Figure 120: Font options for localized .NET software.	131

Figure 121: IME options for software localized to Far Eastern languages.....	132
Figure 122: Output options for .NET software.....	133
Figure 123: Localizing forms of .NET software visually.....	134
Figure 124: Localizing menus of .NET software visually.....	135
Figure 125: Localized .NET software.....	136
Figure 126: Specifying of .NET tools.....	136
Figure 127: Excluding properties by components in .NET localization.....	137
Figure 128: Excluding properties by name in VCL binary localization.....	138
Figure 129: Visual form editor with unknown .NET controls.....	138
Figure 130: Specifying visual representation for .NET control.....	139
Figure 131: Visual Editor recognizing all .NET controls.....	139
Figure 132: Java localization process with resource bundles.....	141
Figure 133: The files of the Java localization process with property resource bundles.....	141
Figure 134: JBuilder Resource wizard.....	142
Figure 135: <i>Internationalization Wizard of netBeans IDE</i>	142
Figure 136: Encodings for localized software.....	144
Figure 137: Output options for Java software.....	145
Figure 138: Excluded files for Java software.....	146
Figure 139: Localized Java software.....	147
Figure 140: <i>Specifying Java Virtual Machine location</i>	147
Figure 141: J2ME localization process.....	149
Figure 142: The files of the J2ME localization process.....	150
Figure 143: J2ME application with an English UI.....	150
Figure 144: Encodings for localized software.....	153
Figure 145: Output options for Java software.....	154
Figure 146: Localized Java software.....	155
Figure 147: <i>Specifying J2ME options</i>	156
Figure 148: Encodings of database.....	159
Figure 149: Localizable fields.....	160
Figure 150: Database cloning settings.....	161
Figure 151: Localization view for translating database contents.....	162
Figure 152: XML-options in Project Wizard.....	164
Figure 153: Defining an element containing C# source code.....	164
Figure 154: Localization of strings in C# source code embedded in XML.....	165
Figure 155: Localization view for translating source code.....	168
Figure 156: Selecting INI-file keys for translation.....	170
Figure 157: INI-file target; keys marked for localization.....	171
Figure 158: Key-file target; defining key file format.....	171

Glossary

Accelerator resource

Accelerator resources include key combinations used as shortcuts in application. MULTILIZER extracts accelerator resources from executables, and allows localization of them.

Bitmap resource

Bitmap resource includes an image. MULTILIZER extracts bitmap resources from executables, and allows localization of them.

Cursor resource

Cursor resource includes application-specific cursors. MULTILIZER extracts cursors from executables, and allows localization of them. Cursors are bitmaps, so the localization is done with visual editor.

Icon resource

Icon resource includes application-specific icons. MULTILIZER extracts icons from executables, and allows localization of them. Icons are bitmaps, so the localization is done with visual editor.

Localization Kit

Localization Kit is a compressed file created with Exchange Wizard. It contains in a minimum a sub-project. It [can](#) also include MULTILIZER Translator Edition and any user-specified files. If Localization Kit includes MULTILIZER Translator Edition, it is a self-installing executable; when executing it, it installs and runs MULTILIZER Translator Edition and opens the sub-project. If MULTILIZER Translator Edition is not included, the Localization Kit is a compressed MULTILIZER package file (MLP) that can be opened in any [MULTILIZER Edition](#), in order to extract the contents of it.

MULTILIZER Edition(s)

MULTILIZER Editions are different MULTILIZER products that can be purchased separately (Note: MULTILIZER Translator Edition is free, and included in Localization Kit). MULTILIZER Editions differ in the role in a localization process, and in support for different platforms.

Project

A MULTILIZER project must be created in order to localized any software/content. MULTILIZER project keeps all the information required for localizing software/content. It contains 1...N targets along with associated data. In addition MULTILIZER project keeps information of target languages and general project [information](#). MULTILIZER project can be created with following [MULTILIZER editions](#): MULTILIZER Enterprise, MULTILIZER for Oracle, MULTILIZER for Windows, MULTILIZER for .NET, MULTILIZER for VCL, and MULTILIZER for Java.

Project Tree

Project tree is the tree-structure shown at left-hand side [of](#) MULTILIZER application's Project View.

Project View

The main view of MULTILIZER is called [Project](#) View. It is shown upon opening a MULTILIZER project, or creating a new one.

Resources

Resources is data that is kept separate from program code. Resources can contain localizable data, such as string tables to translate. MULTILIZER is able to detect localizable resource types from within Windows executables and add them in MULTILIZER project. Some resource types support Wysiwyg.

Sub-project

Sub-project is a MULTILIZER project that is created with Exchange Wizard. Sub-project contains a sub-set of [MULTILIZER](#) project's languages, and sub-set of the targets. In addition translations may be filtered out with Exchange Wizard. Sub-projects are maintained normally only for translation. After translation, sub-project is imported in MULTILIZER project using Import Wizard.

Target

Target (MULTILIZER Localization Target) is a file/database that is supported and localized in MULTILIZER . A target can be a software project file, software executable (EXE, DLL, etc.), single source file, database, etc. MULTILIZER project targets are shown as root-nodes in [Project Tree](#). For each target you can specify native language (source language).

Target language

Target language is the language(s) into which the software/content is localized. Target language should not be mixed with [target](#).

Translation [Work-place](#)

Translation work-place is the part of MULTILIZER application where translations are edited. It shows strings to translate and accompanying info, such [_as](#) resources shown in [Wysiwyg](#).

Wysiwyg

Wysiwyg (What you see is what you get) stands for the UI editors that allow users modify size [and](#) position of dialog resources, form resources, edit bitmap resources, and menu resources visually.

Supported file types

This appendix shows the file types with native support in Multilizer.



Different file types can have the same file extension. In order to localize the file properly, user has to know the file type. In order to simplify this, Multilizer attempts to auto-detect (→ File type, p. 14) the format.

Following list shows the file types with referral to starting page of the tutorial(s) that discuss the localization of it.

File extension	File type	Tutorial page
bas	[Visual] Basic source code file	167
bdsgroup	C#Builder and Delphi 8 project group file	117
bdsproj	C#Builder and Delphi 8 project file	117
bpl	C++Builder binary file	92
bpl	Delphi binary file	92
c	Source code file	167
cpp	Source code file	167
cs	Source code file	167
csdproj	Visual Studio .NET project file	117
csproj	Visual Studio .NET project file	117
dll	.NET assembly file	117
dll	C++Builder binary file	92
dll	Delphi binary file	92
dll	Visual Basic binary file	–
dll	Windows binary file	75
ebp	Embedded Visual Basic project file	–
exe	.NET assembly file	117
exe	C++Builder binary file	92
exe	Delphi binary file	92
exe	Visual Basic binary file	–
exe	Windows binary file	75
h	Source code file	167
hpp	Source code file	167
ini	Ini-file	169
jar	Java archive file	–
java	Java resource file	–
java	Java source code file	167
jpr	JBuilder project file	–
jpx	JBuilder project file	–
ocx	C++Builder binary file	92
ocx	Delphi binary file	92
ocx	Windows binary file	75
pas	[Object] Pascal source code file	167
properties	Java resource file	–
rc	Windows resource file	75
rc2	Windows resource file	75
resx	.NET resource file	117
shl	InstallShield string table file	169
sln	Visual Studio .NET solution file	117
txt	.NET resource file	117
txt	Key file	169
vb	Visual Basic [.NET] Source code file	167
vbdproj	Visual Studio .NET project file	117
vbp	Visual Basic project file	–
vbproj	Visual Studio .NET project file	117
vjsproj	Visual Studio .NET project file	117

xml

XML file

163

Localization Walkthrough Quick Reference

The Multilizer localization process is extremely straight-forward; as its simplest entire localization process is carried out in four short steps.

1	Create project	To do: <ul style="list-style-type: none">○ Run Project Wizard.○ Specify the software you wish to localize.○ Specify target languages. → Create Project, p. 11
2	Prepare project for translation	To do: <ul style="list-style-type: none">○ Remove strings that shouldn't be localized at all.○ Lock strings that shouldn't be translated.○ Use Translation Memory for pre-translation of project. → Translate using Translation Memory, p. 25
3	Share translation work & translate!	To do: <ul style="list-style-type: none">○ Create a Localization Kit, and send it to translator.○ Translate the strings.○ Modify dialog layouts in WYSIWYG.○ After translation, use Import Wizard to import translators. → Share translation work, p. 27
4	Create localized files	To do: <ul style="list-style-type: none">○ Use Validation Wizard to check that everything is all right.○ Click "build" to create localized files. → Build localized versions, p. 73