sgcLocalization

Getting Started

Translate a Delphi application and switch its language while it runs, from one localizer on a form to a project that translators and AI services work on, one small demo at a time

About this guide

This guide is for Delphi developers who know how to build a VCL form and want their application to speak more than one language. sgcLocalization finds the texts of your forms, keeps their translations in one project file and switches the language of the running application with one line of code. You do not need to know anything about translation formats or plural rules in advance.

The guide follows the three QuickStart demos and the translation editor. Each demo adds one idea to the previous one, from a form that switches between four languages to an application with a frame, a data module, a form created later and a language combo. Chapter 4 walks through them one at a time: what you will build, the components on the form, the code, how it works and a few changes to try.

Read it with Delphi open next to it. Open a demo, run it, switch the language, then read its section and try the suggested changes. Chapters 1 to 3 explain what sgcLocalization is, how to install it and the ideas behind it, chapter 5 lists the main components, chapter 6 collects common problems and their fixes, and chapter 7 shows where to go next.

The demos are in the `Demos\Localization\Delphi\00.QuickStart` folder of your sgcUI installation. Each demo has its own subfolder, from `01.Hello` to `03.LanguageCombo`, with a Delphi project (`.dpr`) that you can open and run directly, and with its translations already in place. All of them are VCL applications for Windows, and all of them run offline: no network, no server and no API key. The same demos exist for FireMonkey, Lazarus and C++Builder, see chapter 7.

The screenshots of the translation editor show a sample project named Orders, with German, Spanish and French as target languages. The editor follows the language of the sgcUI texts, so on your machine its captions may be in another language. This guide uses the English names.

Contents

1. What sgcLocalization is

The idea in one minute
Your first translated application in five steps
What you need
Part of sgcUI
Coming from resource DLLs and hand made tables

2. Install and the palette

Installing
The packages
The palette page
Editions
Where the demos are
Running your first demo

3. Core concepts

One project file holds every language
Scanning finds the texts
Ignore rules and the built in ignores
Switching the language at run time
Texts in code: snippets
Plurals
The translation editor
States, marks and incremental work
Validation
XLIFF: working with translators
AI translation gives proposals
Storage and deployment

4. The QuickStart demos

4.1 Demo 01: Hello
4.2 Demo 02: Snippets and Plurals
4.3 Demo 03: Language Combo
4.4 The standalone translation editor

5. Main components

Reference table
The localizers
The language combos
The global functions
FireMonkey, Lazarus and C++Builder

6. Common problems

Nothing is translated
Single texts and forms
Scanning
Translators and AI

7. Where next

The other demos
Connecting an AI service
Help topics
On the website
Getting help

1. What sgcLocalization is

sgcLocalization translates Delphi, C++Builder and Lazarus applications at run time, for the VCL, for FireMonkey and for the LCL. You drop a localizer on the main form, scan the forms of the project, translate the texts in the translation editor and switch the language of the running application with one line of code. No resource DLLs are built and the application is not restarted.

The localizer finds the texts of the forms, frames and data modules by itself: captions, hints, the lines of a list, the columns of a collection. Nothing has to be marked as translatable. Texts that live in code are translated with three small functions, `sgcT`, `sgcTC` and `sgcTN`, the last one with the plural rules of the target language.

This guide is for Delphi developers who are new to sgcLocalization. By the end you will know how the pieces fit together, which properties to reach for first, and how to avoid the mistakes that most people make in their first hour.

The idea in one minute

Every sgcLocalization application is built from three things:

- **The project.** One file with the extension `.sgcloc` holds every translatable text of the application, called an item, with its translation in every language. There is one file for all languages, not one for each.
- **The localizer.** A `TsgcVCLLocalizer` is a non visual component that loads the project, scans forms into it and applies a language to every form on screen. An application normally has one, on its main form or on a data module that is created first.
- **The translation editor.** A window where the texts are translated, reviewed and validated. It opens from the localizer at design time, and it also runs as a program of its own for translators who do not have Delphi.

Your forms stay as they are, written in one language, the source language. At run time the localizer replaces the texts of the forms with the ones of the language you ask for, and puts the source texts back when you ask for the source language. That is the entire idea; the rest of sgcLocalization is about where the translations come from (you, a translator or an AI service) and about keeping them correct while the application changes.

Your first translated application in five steps

Open any VCL project of yours, or create a new one with a few labels, buttons and a menu.

1. **Drop the localizer.** Drop a `TsgcVCLLocalizer` from the **SGC Localization** palette page on the main form. `SourceLanguage` is `en` by default; change it when your forms are written in another language. Set `FileName` to the name of the project file, for example `MyApp.sgcloc`.
2. **Set the languages.** Add the target languages to `Languages`, one code per line: `de`, `es`, `fr`.
3. **Scan.** Right click the localizer and choose **Scan project**. The forms, frames and data modules of the project are read and their texts become items of the project. **Scan this form** reads only the form in the designer.
4. **Translate.** Right click the localizer and choose **Translation editor...**, or double click the component. Select a row, type the translation in the edit panel and set its state to translated or reviewed. Save. The project file is written to the folder of the project.
5. **Switch.** Add a button or a menu item and call `sgcSetLanguage`:

```
uses
  sgcLocalization_Engine;

procedure TfrmMain.mnuGermanClick(Sender: TObject);
begin
  // every open form changes at once, and the forms created later follow
  sgcSetLanguage('de');
end;

procedure TfrmMain.mnuEnglishClick(Sender: TObject);
begin
  // the source language puts the original texts back
  sgcSetLanguage('en');
end;
```

Run the application and click the button: every form on screen changes language. This is how the localizer of the first QuickStart demo is stored in the form file:

```
  object Localizer: TsgcVCLLocalizer
    FileName = 'Hello.sgcloc'
    Languages.Strings = (
      'de'
      'es'
      'fr')
    Rules = <
      item
        ComponentMask = 'btnLang*'
      end>
  end
```

Section 4.1 walks through this demo in detail. To start in the language of the user, set the `Language` property of the localizer to `auto` at design time: the language of the system is used when the project has it, and the source language otherwise.

What you need

- **Delphi.** The packages in the `Packages` folder cover Delphi 7, 2007, 2009, 2010, XE to XE8, 10 Seattle to 10.4 Sydney, 11, 12 and 13. C++Builder uses the same components.
- **VCL.** `TsgcVCLLocalizer` and `TsgcVCLLanguageCombo` work in every one of these versions.
- **FireMonkey.** `TsgcFMXLocalizer` and `TsgcFMXLanguageCombo` have their own packages from Delphi XE6 on, for Windows, macOS, iOS and Android, and for Linux with FMXLinux.
- **Lazarus.** The LCL uses the components with the VCL names, from the package `sgcLocalizationLazarus`, on Windows and Linux.
- **sgcUICore.** The shared runtime package of the sgcUI products. The installer installs it together with sgcLocalization, you do not install it separately.

sgcLocalization works with any VCL, FireMonkey or LCL control. Your application does not have to use other sgcUI controls, and the QuickStart demos use only standard controls.

> **Delphi 7 to 2007.** These compilers have ANSI strings. The project file is always UTF-8, and the texts are converted to the code page of the system when they are applied. A target language written with another script, such as Russian, Greek or Japanese, needs a system whose code page holds that script. Languages of the Latin script of the system code page work everywhere.

Part of sgcUI

sgcLocalization is one product of sgcUI, the eSeGeCe suite of visual components (grids, charts, dashboards, planners, rich text and more). It ships with every sgcUI edition and with every single product of the suite. In practice this means three things for you:

- A switch also sets the language of the built in texts of the sgcUI controls, such as the buttons of a dialog or the filter menu of a grid. The property `SyncUILanguage`, True by default, controls it.
- A form that mixes standard controls and sgcUI controls scans cleanly. The properties of the sgcUI controls that hold data and not user text, such as field names, formats and expressions, are on a built in ignore list.
- For Arabic, Hebrew, Persian and Urdu the application changes to right to left, and back to left to right for every other language. The property `AutoBiDi`, True by default, controls it.

Coming from resource DLLs and hand made tables

The classic way to translate a Delphi application builds one resource DLL for every language. The language is chosen when the application starts, a change needs a restart, and every new version means building and testing the DLLs again. With sgcLocalization the translations are data: one readable file that the application loads, which can be updated or extended with a language without building the application again.

Many applications also grew their own solution: a table of texts and a loop over the components of a form in `OnCreate`. That works until someone forgets a form, a column title or a hint, or until a translator breaks a `%d`. sgcLocalization scans every published text for you, keeps a state for every translation, validates format specifiers, accelerators and plural forms, and never raises in `Format` because of a faulty translation.

2. Install and the palette

This chapter gets sgcLocalization onto your Tool Palette and the first demo running. If the components are already installed, skim the palette section and jump to Running your first demo.

Installing

sgcLocalization is installed by the sgcUI installer, the same installer for every sgcUI product:

1. Run the sgcUI installer and sign in with the user name and password of your eSeGeCe account.
2. Choose the Delphi versions to install into.
3. On the components page, **sgcLocalization: translate your applications** is selected by default, whatever your licence is. You can leave it out there. sgcUICore has no checkbox: it is installed automatically.
4. The installer copies the files, compiles the packages on your machine for each selected Delphi version and installs the design time packages.

Everything goes into one folder, by default `Documents\esegece\sgcUI`. The installer adds its `Source` folder and the folder of the compiled units to the library path of each selected Delphi version, so a new project finds the sgcLocalization units without any setting on your side.

The packages

You only need this section if you build the packages yourself. sgcLocalization is split into a framework neutral package and one package per framework, so a FireMonkey application never links the VCL and the other way round. Every Delphi package name ends with the suffix of the Delphi version, for example `D13` for Delphi 13 or `D7` for Delphi 7.

Package	Kind	Contains
`sgcLocalization<V>`	Runtime	The project, the engine, the rules, the validation, the plural rules, XLIFF and the AI translator. No VCL, no FireMonkey.
`sgcLocalizationVCL<V>`	Runtime	`TsgcVCLLocalizer`, `TsgcVCLLanguageCombo` and the translation editor.
`sgcLocalizationFMX<V>`	Runtime, XE6 and later	`TsgcFMXLocalizer` and `TsgcFMXLanguageCombo`.
`dclsgcLocalization<V>`	Design time	The palette registration, the component editor with the scan commands and the property editors.
`dclsgcLocalizationFMX<V>`	Design time, XE6 and later	The palette registration of the FireMonkey components.
`sgcLocalizationLazarus`	Lazarus	The Lazarus package, runtime and design time.

Build them in dependency order: sgcUICore first, then `sgcLocalization`, then `sgcLocalizationVCL`, then `dclsgcLocalization`. Then install the design time package and add `sgcUI\Source` to the library path of your Delphi version (**Tools** > **Options** > **Language** > **Delphi Options** > **Library**).

An application that links the units directly, without packages, needs `sgcUI\Source` on its search path and two units in a `uses` clause:

```
uses
  sgcLocalization_Engine, // sgcSetLanguage, sgcT, sgcTC, sgcTN
  sgcLocalization_VCL;    // TsgcVCLLocalizer, TsgcVCLLanguageCombo
  // FireMonkey: sgcLocalization_FMX
```

The palette page

After the installation the Tool Palette has a page named **SGC Localization**. It is a short page:

Component	Visual	What it does
`TsgcVCLLocalizer`	no	Holds the translation project, scans forms, frames and data modules, and applies the active language to them. VCL and LCL.
`TsgcVCLLanguageCombo`	yes	A drop down list of the languages of its localizer. A selection switches the application.
`TsgcFMXLocalizer`	no	The localizer of FireMonkey applications, with the same properties, methods and events.
`TsgcFMXLanguageCombo`	yes	The FireMonkey language combo.

A VCL project shows the VCL components and a FireMonkey project the FireMonkey ones. The localizer has a context menu in the designer with three commands, and this guide uses their names as they appear in the IDE:

Command	What it does
Translation editor...	Opens the translation editor on the project of the localizer. A double click on the component does the same.
Scan this form	Reads the texts of the form, frame or data module in the designer into the project.
Scan project	Reads every form, frame and data module of the project in Delphi and C++Builder. In Lazarus it reads the forms that are open in the designer.

The Object Inspector helps as well: `FileName` opens a file dialog, `Language` and `SourceLanguage` show a drop down list of the built in languages, and `Rules` opens the rules dialog.

Editions

sgcLocalization is included with every sgcUI edition and with every single product of the suite, so there is nothing extra to buy or download. It is not sold separately. Everything in this guide, the editor, XLIFF and the AI translation included, is part of it.

One optional feature needs another product: AI translation through `TsgcUIAIChatProvider` uses the AI components of sgcWebSockets, the eSeGeCe networking library. Without sgcWebSockets, `TsgcUIAIEventProvider` connects the AI translation to any service through your own code. See AI translation gives proposals.

Where the demos are

The demos live in `Demos\Localization\Delphi` under your sgcUI folder. The QuickStart is the first subfolder:

Folder	Project	What it shows
00.QuickStart\01.Hello	`Hello.dpr`	The smallest translated application: a localizer, a project file and four language buttons
00.QuickStart\02.SnippetsAndPlurals	`SnippetsAndPlurals.dpr`	Texts set in code with `sgcT`, `sgcTC` and `sgcTN`, with the plural forms of German and Polish
00.QuickStart\03.LanguageCombo	`LanguageCombo.dpr`	The language combo, a frame used twice, a data module with actions, a form created later and the embedded storage
01.TranslationEditor	`TranslationEditor.dpr`	The translation editor as a program of its own, for translators who do not have the IDE

Every demo folder holds the project, its unit and form, the `.sgcloc` project file with the finished translations and a `readme.html` with a short description. The `readme.html` in `00.QuickStart` itself is the index of the demos. The folders `Demos\Localization\FMX`, `Lazarus` and `CBuilder` hold the same demos for the other frameworks and compilers, listed in chapter 7.

Running your first demo

1. In Delphi, choose **File** > **Open Project** and open `Demos\Localization\Delphi\00.QuickStart\01.Hello\Hello.dpr`.
2. Press **F9**. The window opens in English. Click the German, Spanish and French buttons: the menu, the labels, the hints and the items of the combo box change at once.
3. Back in the IDE, open the form, right click `Localizer` and choose **Translation editor...**. The grid shows every text of the form with its three translations.

> **Tip.** To see every property of a form at once, right click the form in the designer and choose **View as Text**. The DFM excerpts in this guide come from that view. Choose **View as Form** to go back to the designer.

3. Core concepts

Twelve ideas cover almost everything you do with sgcLocalization in your first weeks. The QuickStart demos meet most of them; this chapter puts them in one place so that the demos make sense at a glance.

One project file holds every language

All translations of an application live in one project file with the extension `.sgcloc`: the items, every language, the states, the notes, the rules and the version marks. Every translatable text is an **item**. A property item is identified by the class of its form, the path of the component and the path of the property:

```
TfrmMain:btnOK.Caption
TfrmMain:grdOrders.Columns[2].Title.Caption
snippet:|File not found
```

The last line is a snippet, a text of the code, identified by its context and its text. A frame is keyed by its own class, so a frame is translated once, whatever the number of forms that host it.

The file is JSON, UTF-8, with one item on every line, sorted by id. Saving a project that was not changed gives the same bytes, so the file works well with version control: a diff shows the items that changed, and two developers rarely touch the same line. It is readable on purpose, but edit it with the translation editor, which keeps the states consistent.

Scanning finds the texts

A scan reads the texts of forms, frames and data modules and merges them into the project. It looks at the published string properties of the form and of every component the form owns, at the lines of string lists, at the items of collections and at nested objects. The caption of a grid column or the items of a combo box are found without any registration.

Where	Command	Scope
Design time	**Scan this form**	The form, frame or data module in the designer.
Design time	**Scan project**	Every form, frame and data module of the project. In Lazarus, the forms that are open in the designer.
Translation editor	**Scan**	The same scan at design time. In a running application, every live form and data module.
Code	`ScanRoot(aRoot)`, `ScanApplication`	One root, or every live form and data module. Both return the number of items added or changed.

Scan again whenever the forms change. A scan never deletes a translation:

- A text that the project does not know becomes a new item.
- A text whose source changed keeps its translations, and they get the state changed.
- An item that is no longer found on a scanned form becomes obsolete. It is hidden and kept, and it comes back with its translations when a later scan finds it again.

Two limits are worth knowing from the start. A scan covers the components that the form owns, which is everything placed in the designer; a component created in code with another owner is not scanned, so set its texts with `sgcT`. And a control that takes its caption from an action has no item of its own: the item lives on the action, and the translated action passes its text to the control.

Ignore rules and the built in ignores

Not every string property is a text for a translator. Without any rule a scan already ignores an empty text, a text without any letter, such as `0` or `...`, and a built in list of property names that never hold user text: `Name`, `FieldName`, `DataField`, `SQL`, `ConnectionString`, `FileName`, `EditMask`, `DisplayFormat`, `Password` and many more. The list also knows the properties of the sgcUI controls that hold data, so a form with sgcUI controls scans cleanly without any rule of yours.

The `Rules` collection of the localizer refines the defaults. Every rule has four masks (class, component, property and value) and an action, ignore or include. The masks know `*` for any run of characters and `?` for one character, and an empty mask matches everything. The built in defaults are evaluated first, then the rules in their order, and the last matching rule wins, so an include rule can bring back a property of the built in list.

```
uses
  sgcLocalization_Types, sgcLocalization_Rules;

// class, component, property and value mask, then the action
// nothing of a dataset is translated
Localizer.Rules.AddRule('TDataSet', '', '', '', lraIgnore);
// the Text of the edits whose name starts with edtCode
Localizer.Rules.AddRule('', 'edtCode*', 'Text', '', lraIgnore);
// texts that look like a version number
Localizer.Rules.AddRule('', '', '', 'v?.*', lraIgnore);
// but the display format of this one column is wanted
Localizer.Rules.AddRule('', 'grdOrders', 'DisplayFormat', '', lraInclude);
```

At design time the `Rules` property opens the rules dialog, the same one as **Rules** in the translation editor. It lists the rules in their order, shows the built in ignores read only, and has a test line that tells whether a class, component, property and value would be translatable. The rules are saved in the project file.

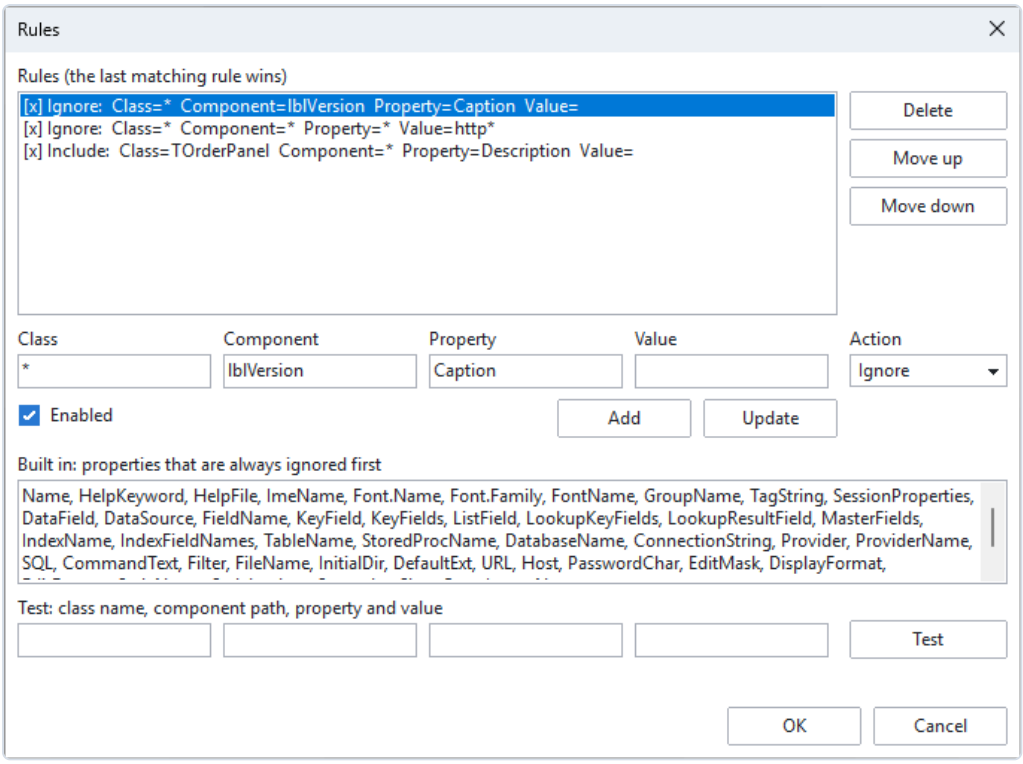


Figure 3.1. The rules dialog. Two ignore rules, one include rule, and the built in list below them.

The QuickStart demos use one rule: the language buttons show the name of each language in that language, so the component mask `btnLang*` keeps them out of the project.

Switching the language at run time

The language of a running application changes with one line. Every open form, frame and data module is translated at once, without a restart and without creating the forms again.

```
uses
  sgcLocalization_Engine;

sgcSetLanguage('de');        // the default localizer
Localizer.Language := 'de';  // the same through the component
sgcSetLanguage('en');        // the source language: the original texts
sgcSetLanguage('auto');      // the language of the system
```

`sgcSetLanguage` works on the default localizer, the first one the application creates. A regional tag falls back to its base, so `de-AT` uses the texts of `de` where the project has no Austrian ones. `Language` keeps what was asked, `auto` for example, and `ActiveLanguage`, or the function `sgcLanguage`, returns the language that is really in use.

Four things follow a switch without code of yours:

- **Forms created later.** With `AutoTranslate`, True by default, a form, frame or data module that is created after the switch is translated before it is painted for the first time.
- **Selections are kept.** Translating the items of a combo box or a list box keeps its selection: `ItemIndex`, a multiple selection, and the text of an editable combo box when it showed the selected line.

- **Values changed by code are kept.** A switch writes a property only when it still holds the source text or the text of the language that was active before. A caption that your code replaced in the meantime, a file name in a title bar for example, is left alone.
- **The sgcUI controls and the text direction.** The built in texts of the sgcUI controls change to the same language, and Arabic, Hebrew, Persian and Urdu set the application to right to left.

`OnLanguageChanged` fires last. It is the place to refresh the texts that your code builds, and to remember the choice of the user:

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
  // 'auto' on the first start
  Localizer.Language := FSettings.ReadString('UI', 'Language', 'auto');
end;

procedure TfrmMain.LocalizerLanguageChanged(Sender: TObject);
begin
  Caption := sgcT('Orders of %s', [FCustomerName]);
  FSettings.WriteString('UI', 'Language', Localizer.Language);
end;
```

Texts in code: snippets

A snippet is a text that lives in code: a message, a status line, a caption built at run time. The snippet functions of the unit `sgcLocalization_Engine` return its translation in the active language. The source text is the key, so the code stays readable and works without a project file.

Function	What it returns
`sgcT(aText)`	The translation of a text.
`sgcT(aText, aArgs)`	The translation, formatted with the arguments like `Format`.
`sgcTC(aContext, aText)`	The translation of a text in a context. The same word can have two translations, `sgcTC('menu', 'Open')` and `sgcTC('state', 'Open')`.
`sgcTN(aSingular, aPlural, aCount)`	The plural form that the active language uses for the count.
`sgcTN(aSingular, aPlural, aCount, aArgs)`	The plural form, formatted with the arguments.

```
uses
  sgcLocalization_Engine;

ShowMessage(sgcT('The file was saved.'));
lblUser.Caption := sgcT('Welcome, %s', [vUserName]);
btnState.Caption := sgcTC('state', 'Open');
lblCount.Caption := sgcTN('%d file copied', '%d files copied', vCount, [vCount]);
```

A scan reads forms, not source code, so a snippet is added to the project by hand: **Add snippet** in the translation editor asks for the text, an optional context, the plural text and a note. A snippet used with `sgcTN` has to be added with its plural text. While an application is tested, the event `OnMissingSnippet` can collect them for you. It reports every snippet function that was called with a text the project does not hold, once for every text:

```
procedure TfrmMain.LocalizerMissingSnippet(Sender: TObject;
  const aContext, aText: string);
begin
  // a developer build collects the texts, the project is saved on exit
  Localizer.Project.AddSnippet(aContext, aText);
end;
```

Safe by design. Without a localizer, or for a snippet the project does not hold, the functions return the source text, formatted when arguments are given. A translation whose format specifiers do not match the source is not used either, the source text is shown in its place, so a faulty translation never raises in `Format`.

Plurals

Languages do not agree on how many plural forms a noun has. English has two, Polish has four, Japanese has one, Arabic has six. sgcLocalization follows the CLDR plural categories for integer counts: zero, one, two, few, many and other. The source always has two texts, the singular and the plural. A translation has one text for every category that its language uses, and `sgcTN` picks the form of the count.

Languages	Categories
en, de, es, it, nl, sv, da, nb, fi, el, hu, ca, bg, tr	one, other
fr, br, pt	one (for 0 and 1), other
ru, uk, pl	one, few, many, other
cs, sk, hr, sr, lt, ro	one, few, other
sl	one, two, few, other
lv	zero, one, other
he	one, two, other
ar	zero, one, two, few, many, other
ja, ko, zh and every other language	other

38 languages are built in with their names and rules. A regional tag uses the rule of its base language, `pt-BR` the one of `pt`. In the translation editor a plural item shows one tab for every category of the current language, so the translator sees exactly which forms are needed.

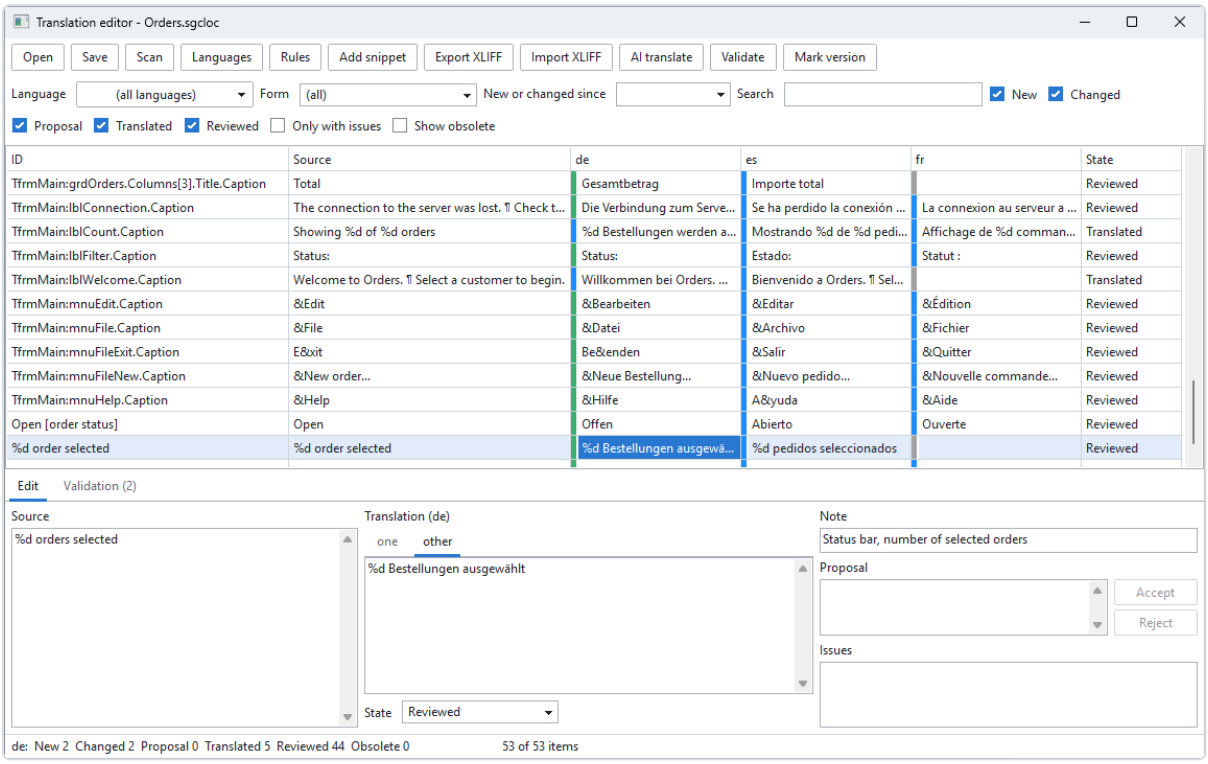


Figure 3.2. A plural snippet in the editor. German needs two forms, so the edit panel shows the tabs one and other.

The translation editor

The translation editor is the window where the texts of a project are translated, reviewed and validated. It is the same window at design time, inside your own application and as a standalone tool:

- **Design time.** Right click the localizer and choose **Translation editor...**, or double click the component.
- **Your application.** Call `sgcLocalizationShowEditor(Localizer)` of the unit `sgcLocalization_Editor_VCL`. The window is modal and the function returns True when the project was changed and the changes were kept.
- **Standalone.** The demo `01.TranslationEditor` is the editor as a program of its own, see section 4.4.

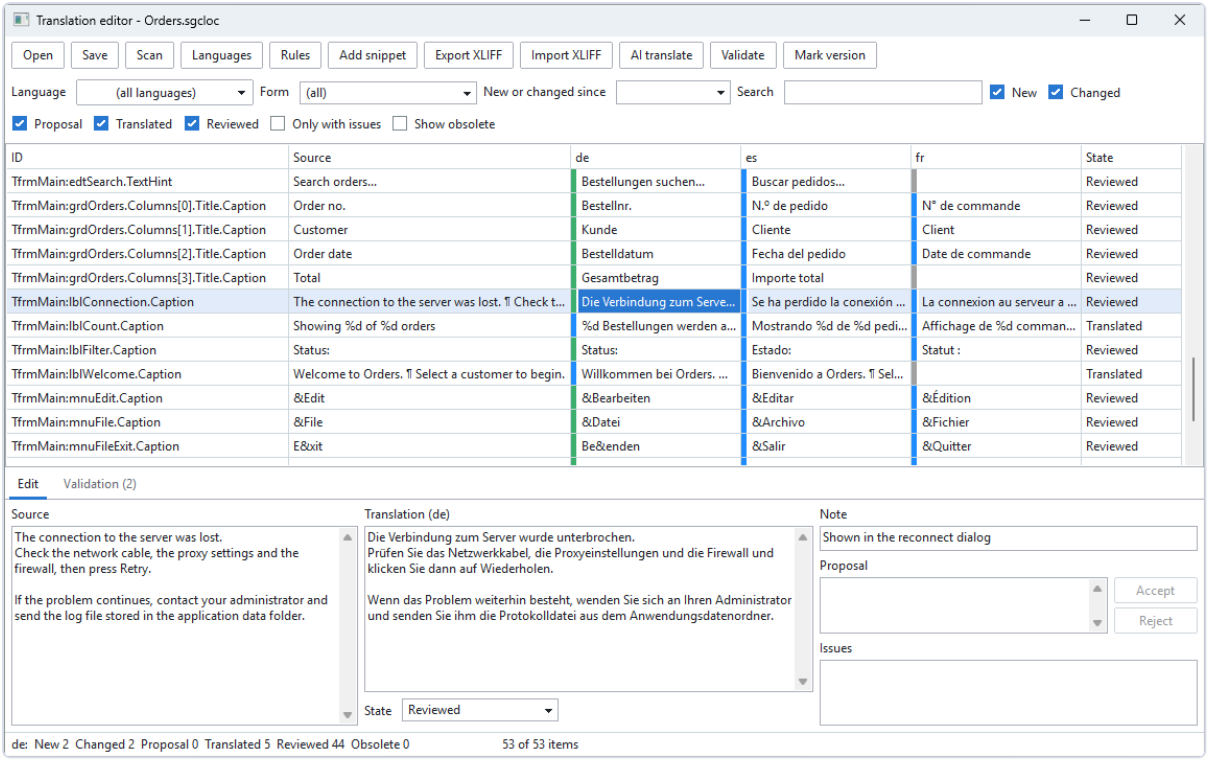


Figure 3.3. The translation editor. The toolbar, the filters, the grid with one column per language, and the edit panel of the selected row.

The **grid** has the columns ID, Source, one column for every visible language, and State. A coloured bar in a language cell shows the state of that translation. A line break inside a cell is shown as a return mark, and the full text with its lines is in the edit panel. A snippet is listed by its text with the context in square brackets, for example `Open [order status]`.

The **edit panel** below the grid holds the source text, the translation in the language of the selected cell, the state, a note for the translator, and a proposal with Accept and Reject. The issues of the text are listed while you type. Changes go to the project while you type, and the caption of the window shows a `*` while there are unsaved changes.

The **filters** keep a large project readable:

- **Language**: a tick list of the language columns to show.
- One check box for every state to show: New, Changed, Proposal, Translated and Reviewed.
- **Form**: the items of one form, frame or data module. The last entry, Snippet, shows the texts of the code.
- **New or changed since**: a version mark, to see only what was added or changed after a release.
- **Search**, **Only with issues** and **Show obsolete**.

Several rows can be selected together, and a right click opens a menu for the selection: Mark as translated, Mark as reviewed, Accept proposals, AI translate and Select all.

Toolbar command	What it does
Open, Save	Loads and writes the project file.
Scan	Scans and merges. The status line tells how many items were added or changed.
Languages	The source language and the target languages of the project.
Rules	The ignore and include rules, with a test line.
Add snippet	Adds a text of the code: text, optional context, plural text and note.
Export XLIFF, Import XLIFF	The exchange with translators.
AI translate	Asks the AI provider for proposals.
Validate	Checks the current language and fills the Validation tab.
Mark version	Names the current state of the project.

The editor itself is localized in the 12 languages of the suite, and it is drawn in light or in dark colours after the theme of the sgcUI application.

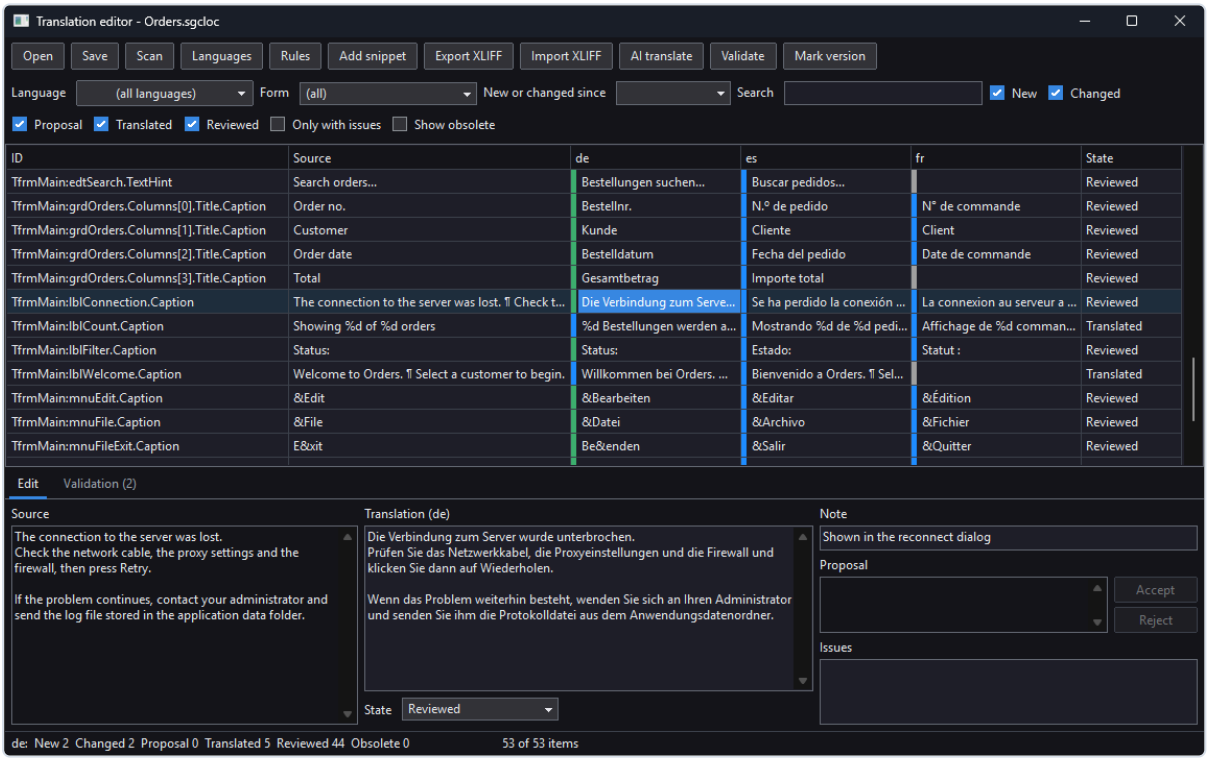


Figure 3.4. The same project in the dark theme.

States, marks and incremental work

An application changes with every version, and only the difference should go back to the translators. sgcLocalization tracks this with a state for every translation and with marks on the versions you release.

State	Meaning
New	No translation yet. The application shows the source text.
Changed	The source text changed after it was translated. The old translation is kept and shown until it is checked.
Proposal	Only a proposal exists, from the AI translation, an import or the memory of the project. The application shows the source text until the proposal is accepted.
Translated	A translation was entered.
Reviewed	The translation was checked. A text with a validation error cannot reach this state.
Obsolete	The item was not found by the last scan of its form. It is hidden unless the filter shows it, and it is not exported.

A **mark** is a name for the state of the project at one moment. **Mark version** in the translation editor asks for the name, the version number of the release for example. Later the filter **New or changed since** with that mark shows exactly what was added or changed after the release. A release cycle then looks like this:

1. Develop, then scan the project.
2. Filter by the states new and changed, or by the mark of the last release.
3. Translate the difference: in the editor, with AI proposals for the new and changed texts, or with an XLIFF export that holds only those texts.
4. Review, then run **Validate** for every language.
5. Release and set a mark with the version number.

Validation

A translation can break an application: a missing `%s` raises an exception in `Format`, a lost accelerator makes a menu unreachable from the keyboard. The validation compares every translation with its source and reports such defects before the application ships.

Rule	Severity	Checks
V1 Format specifiers	Error	The translation reads the same `Format` arguments with the same types as the source. Indexes are known, so `%1:d of %0:s` matches `%s has %d`.
V2 Placeholders	Error	The tokens in braces, `{0}` or `{name}`, are the same set in both texts.
V3 Accelerator	Warning	A source with an accelerator, `&Open`, needs exactly one in the translation.
V4 Duplicate accelerator	Warning	Two texts of the same form use the same accelerator letter. Off by default.
V5 Line breaks	Warning	The same number of line breaks.
V6 Whitespace	Warning	The same leading and trailing whitespace, the space after a colon for example.
V7 Empty target	Error	An empty translation of a source that is not empty.
V8 Plural forms	Error	Every plural category of the language has a form.

The validation runs while you type in the edit panel, on every row of a review list, and on demand with **Validate**, which fills the Validation tab with the issues of the current language. A double click on an issue selects its item. An error blocks the state reviewed; a warning is a hint that a person decides on.

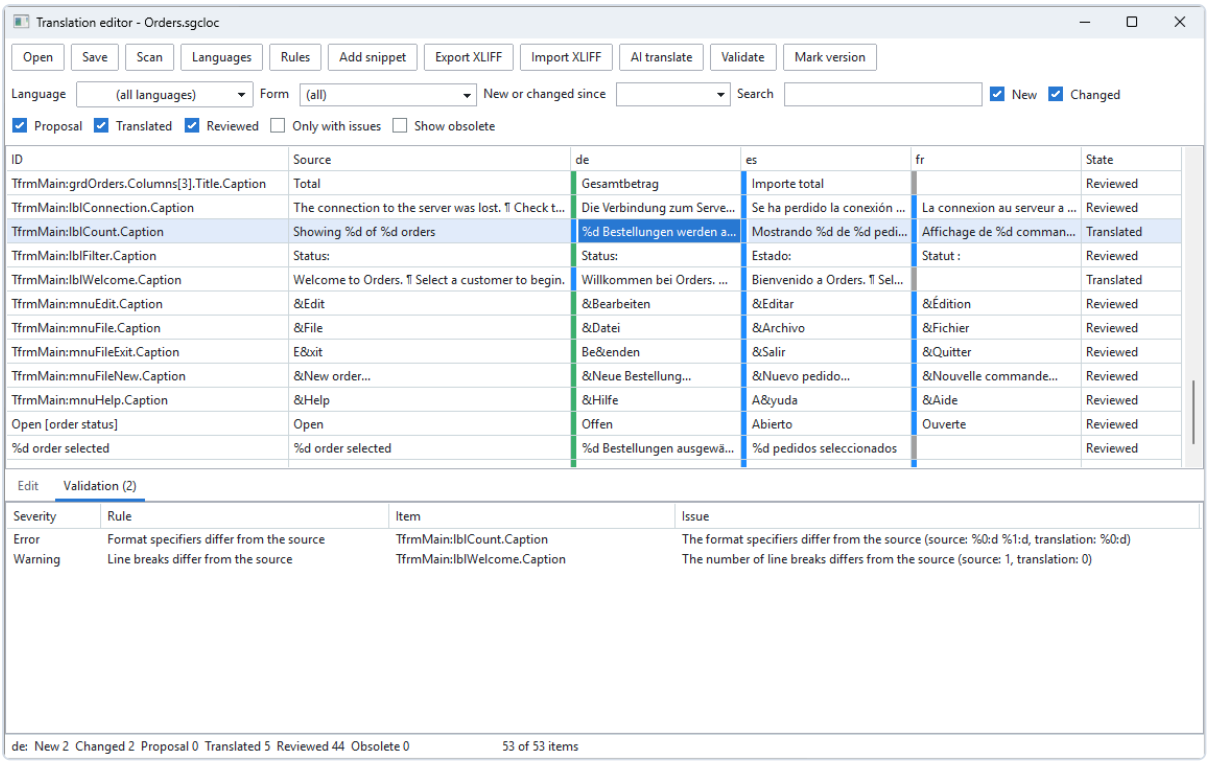


Figure 3.5. The Validation tab: one error and one warning for German, with the rule, the item and the reason.

XLIFF: working with translators

XLIFF 1.2 is the exchange format that translation tools and agencies read. sgcLocalization exports one file for every target language, and imports the translated file through a review list, so nothing reaches the project unseen.

Export XLIFF in the translation editor asks for the language, the file name and the option **Only new and changed texts**, which sends the translator just the work that is open. The file is UTF-8 and carries the source, the target, the state and the note of every item. Obsolete items are not exported.

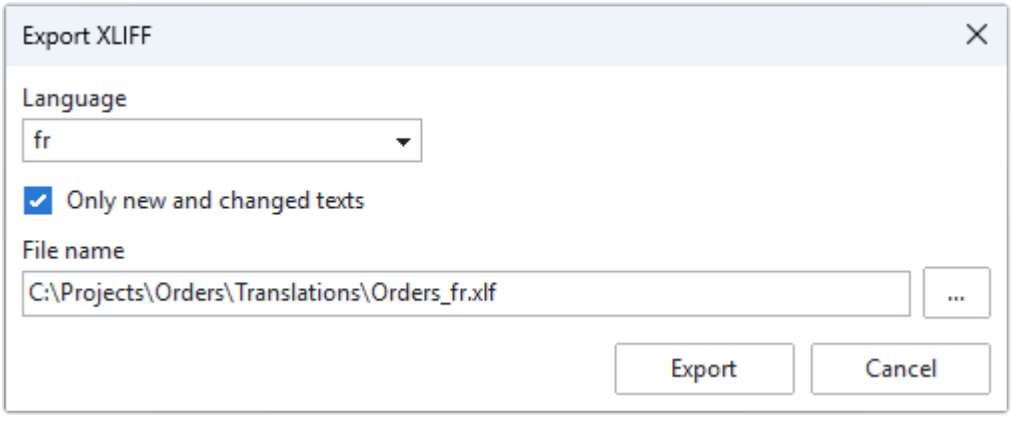


Figure 3.6. Exporting the open French texts for a translator.

Import XLIFF reads the file and opens the review list. The list has one row for every unit with the old text, the new text, the new state and the validation issues, and only the rows that are ticked are applied.

Kind of row	Meaning
Update	The translation differs from the one of the project. Ticked by default when it has no validation error.
Unchanged	The file holds what the project already has.
Unknown ID	The project has no item with that id.
Source mismatch	The translator worked on an older source text. The translation arrives with the state changed.

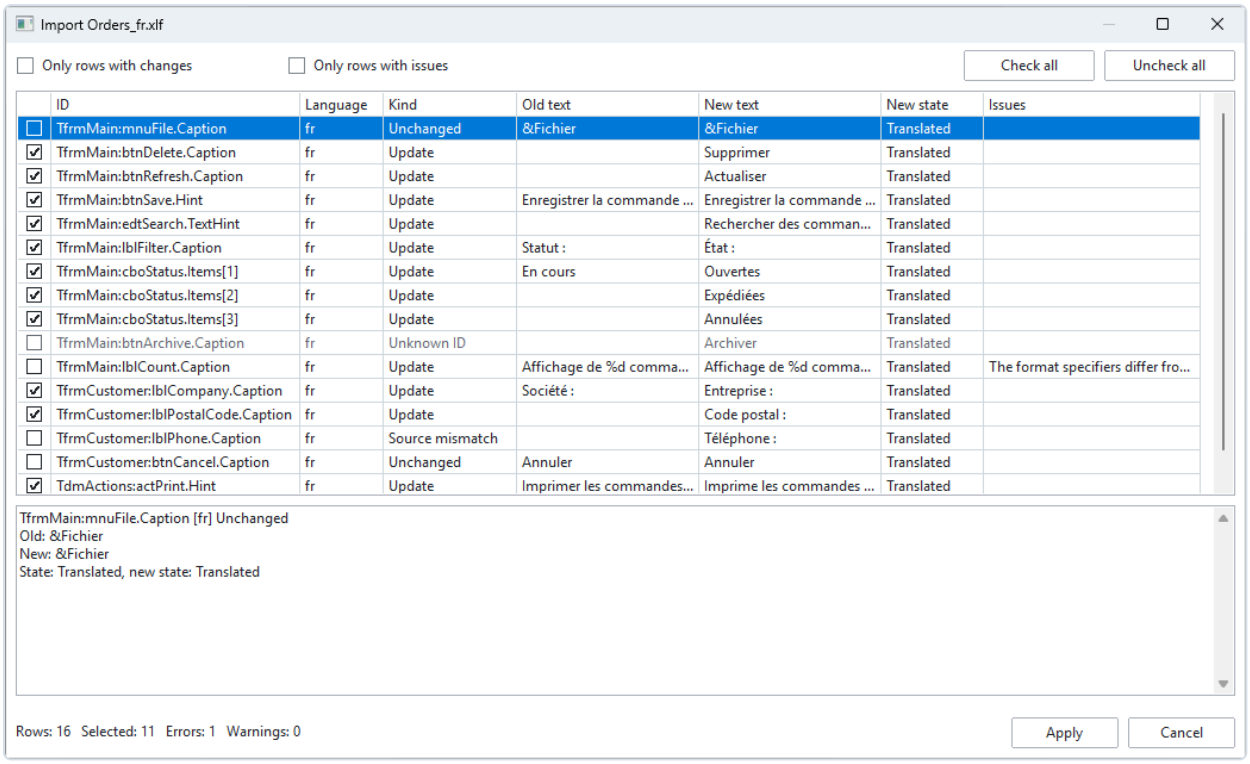


Figure 3.7. The review list of an import. The row with a format error is not ticked, and nothing is stored before Apply.

The same in code, for a build step or a tool of your own:

```
uses
  sgcLocalization_Engine, sgcLocalization_Project;

var
  oChanges: TsgcLocalizationChangeSet;
  i: Integer;
begin
  Localizer.ExportXLIFF('de', 'C:\Translations\MyApp.de.xlf');
  // ... the translator returns the file ...
  oChanges := Localizer.ImportXLIFF('C:\Translations\MyApp.de.xlf');
  try
    // take only clean rows
    for i := 0 to oChanges.Count - 1 do
      if oChanges[i].Issues.HasErrors then
        oChanges[i].Selected := False;
    Localizer.Project.Apply(oChanges);
    Localizer.SaveToFile(Localizer.ProjectFileName);
  finally
    oChanges.Free;
  end;
end;
```

AI translation gives proposals

The AI translation sends the open texts of a project to a language model and brings the answers back as proposals. A proposal is shown next to the translation, it never overwrites a translated or reviewed text, and it becomes the translation only when a person accepts it.

The localizer talks to the model through a provider component assigned to its `AIProvider` property. Two providers ship with the suite:

Provider	Needs	What it does
`TsgcUIAIChatProvider`	sgcWebSockets and the package `sgcUICoreWS`	Connects to ChatGPT (OpenAI), Claude (Anthropic), Gemini, DeepSeek, Ollama, Grok and Mistral through the `TsgcAIChat` component of sgcWebSockets assigned to its `Chat` property. The service, the model and the key are set on that component.
`TsgcUIAIEventProvider`	Nothing else	Hands every request to its `OnRequest` event. Use it with an HTTP client or a service of your own.

In the translation editor, choose **AI translate**. The dialog shows the provider, the target languages, the scope (the new texts, the changed texts, both, or the rows selected in the grid), the batch size and two optional texts: a glossary with the terms the model has to respect, and instructions such as the tone or the kind of application.

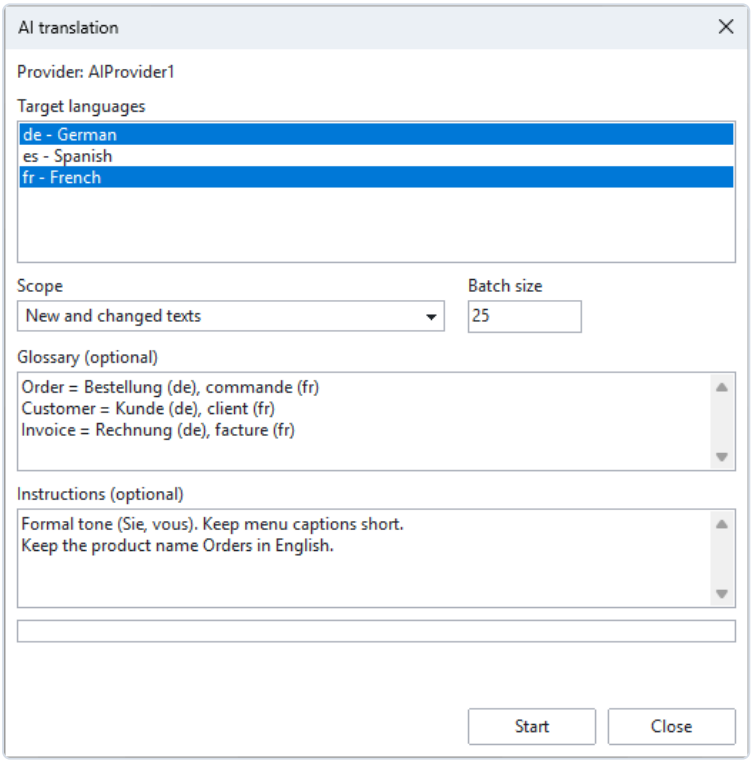


Figure 3.8. The AI translation dialog, with a glossary and instructions for the model.

The run works in batches, shows its progress and can be cancelled. When it finishes, the review list opens with one row for every proposal and its validation issues, and only the ticked rows are stored. Back in the grid the items are in the state proposal: **Accept** in the edit panel makes a proposal the translation, **Reject** drops it. The model is told to keep format specifiers, placeholders, accelerators and line breaks, and to return every plural form of the target language, and every answer is still checked by the validation.

> **Your texts leave the machine.** The texts of the user interface are sent to the service of the provider. Use a local model through Ollama when they must stay in house, and have a speaker of the language review the proposals before they ship. The state reviewed exists for that step.

Storage and deployment

The `Storage` property of the localizer decides where the texts are at run time:

Storage	Where the texts are	Use it when
`lstFile` (default)	In the file named by `FileName`, loaded when the application starts.	Translations should be updated, or languages added, without building the application again.
`lstEmbedded`	In the form file of the localizer, as a compressed compact copy of the project.	The application ships as a single executable, or runs on a mobile platform.

With `lstFile`, an absolute `FileName` is taken as it is. A relative name is searched at run time in this order, and the first file that exists wins:

1. The folder of the executable.
2. The data folder of the platform: the documents folder on Android and iOS, the `Contents/Resources` folder of the bundle on macOS.
3. The three parent folders of the folder of the executable, the nearest first. This finds the file of the project folder when the application runs from `Win32\Debug`.

With `lstEmbedded` the `.sgcloc` file stays the master that the translators work on, keep it in version control. The copy in the form file is written whenever the form is saved in the designer, so save the form of the localizer after the last change of the translations, then build. Demo 03 uses this storage.

A translation file never stops an application from starting. When the file is missing or damaged, the application starts in the source language and the event `OnProjectLoadError` tells the name of the file and the reason. The project can also be loaded by code, from a file, a stream or a resource of the executable:

```
// a file chosen by the application, in a data folder of the user for example
Localizer.LoadFromFile(vDataFolder + 'MyApp.sgcloc');
// a stream, from a database or a download
Localizer.LoadFromStream(oStream);
// an RCDATA resource linked into the executable
// MyApp.rc:  TRANSLATIONS RCDATA "MyApp.sgcloc"
Localizer.LoadFromResource('TRANSLATIONS');
```

A load at run time applies the active language again with the new texts, so translations can be updated while the application runs. No language DLLs and no registry entries are used: the runtime packages or units and the project are everything the application needs.

4. The QuickStart demos

The three QuickStart demos build on each other, and the standalone translation editor completes them. Open them in order, run each one, then read its section. Every section has the same parts: what you will build, the components on the form, the code, how it works, and a few changes to try.

Demo	Project	New ideas
01.Hello	`Hello.dpr`	The localizer, the project file, an ignore rule, `sgcSetLanguage`
02.SnippetsAndPlurals	`SnippetsAndPlurals.dpr`	`sgcT`, `sgcTC`, `sgcTN`, plural forms, `OnLanguageChanged`
03.LanguageCombo	`LanguageCombo.dpr`	The language combo, a frame, a data module with actions, a form created later, the embedded storage
01.TranslationEditor	`TranslationEditor.dpr`	The editor as a program, a localizer as a container, an AI provider event

> **Note.** The demos need sgcLocalization installed, so the components appear on the SGC Localization palette page and the `sgcUI\Source` folder is on the library path. They use standard VCL controls and only units of sgcUI, their translations are already in the project files, and they need no network, no server and no API key.

4.1 Demo 01: Hello

Folder `01.Hello`, project `Hello.dpr`.

What you will build

The smallest translated application there is. A form with a main menu, labels with accelerators, an edit, a combo box with three items, a check box and two buttons, with hints. Four buttons in a group box switch between English, German, Spanish and French. Nothing on the form marks a text as translatable: the localizer finds every published text by itself.

Components on the form

Component	Class	Properties you set
`Localizer`	`TsgcVCLLocalizer`	`FileName = 'Hello.sgcloc'`, `Languages`: `de`, `es`, `fr`. One item in `Rules` with `ComponentMask = 'btnLang*'`. `SourceLanguage` and `Storage` keep their defaults, `en` and `lstFile`.
`mnuMain`	`TMainMenu`	A File menu and a Help menu, the captions with accelerators.
`lblName`, `edtName`, `cboColor`, `chkRemember`, `btnSave`, `btnClose`	Standard controls	Captions, hints and the three `Items` of the combo box: the texts to translate.
`btnLangEN`, `btnLangDE`, `btnLangES`, `btnLangFR`	`TButton`	The language buttons. Their captions are the names of the languages in their own language.

The form file of the localizer is shown in section 1.2. The rule keeps the four language buttons out of the project, because the name of a language is not translated.

The code

The `uses` clause names the engine unit, for `sgcSetLanguage`, and the VCL binding unit, for the component:

```
uses
  SysUtils, Classes, Controls, Forms, StdCtrls, Menus,
  // sgc
  sgcLocalization_Engine, sgcLocalization_VCL;
```

The whole runtime code is one line for each button. The handlers of the Spanish and French buttons are the same with `es` and `fr`:

```
procedure TfrmHello.btnLangENClick(Sender: TObject);
begin
  { en is the source language: the texts of the form file come back }
  sgcSetLanguage('en');
end;

procedure TfrmHello.btnLangDEClick(Sender: TObject);
begin
  sgcSetLanguage('de');
end;
```

How it works

1. The form loads from its DFM. The localizer is the first one the application creates, so it becomes the default localizer.
2. `Storage` is `lstFile`, so the localizer loads `Hello.sgcloc`. It looks in the folder of the executable first and then in the three folders above it, which finds the file of the demo folder when the IDE writes the executable to `Win32\Debug`.
3. A click calls `sgcSetLanguage`. The default localizer replaces every text that the project holds for that language: the menu, the captions, the hints and the items of the combo box.
4. The selected item of the combo box stays selected, in the new language.
5. `sgcSetLanguage('en')` asks for the source language, and the texts of the form file come back.

The code never touches a caption. That is the pattern of every sgcLocalization application: the forms keep their source texts, and the localizer applies the language.

Try this

- Set `Language` of `Localizer` to `de` in the Object Inspector. Run again: the application starts in German. The designer still shows the source texts.
- Add a button with a caption and a hint to the form. Right click `Localizer`, choose **Scan this form**, then open **Translation editor...**: the two new items are there with the state New. Translate them and save.
- Add `it` to `Languages`, translate a few texts into Italian in the editor and add a fifth language button named `btnLangIT`. The rule already covers it.

4.2 Demo 02: Snippets and Plurals

Folder `02.SnippetsAndPlurals`, project `SnippetsAndPlurals.dpr`.

What you will build

A form whose right column is filled by code. A spin edit holds a count, and the labels next to it show a plain text, a formatted text, the word Open in two contexts and two plural texts that follow the count. Three buttons switch between English, German and Polish. Polish is there on purpose: it has four plural forms.

Components on the form

Component	Class	Properties you set
`Localizer`	`TsgcVCLLocalizer`	`FileName = 'SnippetsAndPlurals.sgcloc'`, `Languages`: `de`, `pl`. The rule `btnLang*`, and the event `OnLanguageChanged`.
`spnCount`	`TSpinEdit`	The count. Its `OnChange` event refreshes the texts.
`lblPlainCaption`, `lblFormatCaption` and the other caption labels	`TLabel`	Texts of the form file. The scan finds them.
`lblPlainValue`, `lblFormatValue`, `lblCommandValue`, `lblStateValue`, `lblPluralValue`, `lblDaysValue`	`TLabel`	Filled by code with the snippet functions.

The code

One method sets every text that code builds:

```
procedure TfrmSnippetsAndPlurals.UpdateTexts;
var
  vCount: Integer;
begin
  vCount := spnCount.Value;
  { a plain text }
  lblPlainValue.Caption := sgcT('This text is set in code.');
  { a text with Format arguments. A translation may change their order with
    index specifiers such as %1:d }
  lblFormatValue.Caption := sgcT('Selected %d of %d',
    [vCount, spnCount.MaxValue]);
  { the same word in two contexts }
  lblCommandValue.Caption := sgcTC('command', 'Open');
  lblStateValue.Caption := sgcTC('state', 'Open');
  { plural forms: the count selects the form, the arguments fill it }
  lblPluralValue.Caption := sgcTN('%d file', '%d files', vCount, [vCount]);
  lblDaysValue.Caption := sgcTN('%d day left', '%d days left', vCount,
    [vCount]);
end;
```

It is called when the form is created, when the count changes and when the language changes:

```
procedure TfrmSnippetsAndPlurals.spnCountChange(Sender: TObject);
begin
  UpdateTexts;
end;

procedure TfrmSnippetsAndPlurals.LocalizerLanguageChanged(Sender: TObject);
begin
  { the texts of the form file are already switched, the snippets follow }
  UpdateTexts;
end;
```

This is how two of the snippets are stored in the project file, the word Open in the context command and the plural text with its German and Polish forms:

```
{"id":"snippet:command|Open","hash":"04422C9A","rev":4,"t":{
  "de":{"text":"Öffnen","state":"reviewed","hash":"04422C9A","origin":"manual"},
  "pl":{"text":"Otwórz","state":"reviewed","hash":"04422C9A","origin":"manual"}}},
{"id":"snippet:|%d day left","plural":"%d days left","hash":"4D7C1624","rev":7,"t":{
  "de":{"forms":{"one":"Noch %d Tag","other":"Noch %d Tage"},
    "state":"reviewed","hash":"4D7C1624","origin":"manual"},
  "pl":{"forms":{"one":"Pozostał %d dzień","few":"Pozostały %d dni",
    "many":"Pozostało %d dni","other":"Pozostało %d dnia"},
    "state":"reviewed","hash":"4D7C1624","origin":"manual"}}}
```

In the file every item is on one line; the lines are wrapped here to fit the page.

How it works

1. The localizer loads the project before `FormCreate` runs, so the first call of `UpdateTexts` already finds the snippets.
2. `sgcT` looks up the text without a context. With arguments it formats the translation like `Format`.
3. `sgcTC` looks up the same word, Open, in two contexts. German answers with *Öffnen* for the command and with *Geöffnet* for the state.
4. `sgcTN` takes the count, asks the plural rule of the active language for its category and returns that form. In Polish the counts 1, 2, 5 and 22 give three different forms.
5. A switch translates the texts of the form file first and fires `OnLanguageChanged` last. The handler calls `UpdateTexts`, and the snippets follow.

The snippets were added to the project with **Add snippet** in the translation editor, the plural ones with their plural text.

Try this

- Switch to Polish and move the count through 1, 2, 3, 4, 5, 21 and 22. Watch the ending of the noun.
- Remove the `OnLanguageChanged` handler in the Object Inspector and switch the language: the left column changes, the right column keeps the old language until the count changes.
- Add a label and set it with `sgcT('Ready.')`. It stays English in every language until you add the snippet in the editor and translate it.
- Open the plural snippet in the editor with a Polish cell selected: the edit panel shows four tabs, one, few, many and other.

4.3 Demo 03: Language Combo

Folder `03.LanguageCombo`, project `LanguageCombo.dpr`.

What you will build

A small application with everything a real project has: a main form, a frame with an address that is used twice (billing and shipping), a data module with an action list, and a detail form that is created when the user asks for it. A language combo at the top switches all of them. The translations are embedded in the form file, so the executable runs without a `.sgcloc` file next to it.

Components on the form

Component	Class	Properties you set
`Localizer`	`TsgcVCLLocalizer`	`FileName = 'LanguageCombo.sgcloc'`, `Storage = lstEmbedded`, `Languages`: `de`, `es`, `fr`.
`cboLanguage`	`TsgcVCLLanguageCombo`	`Localizer = Localizer`. Nothing else.
`gbBilling`, `gbShipping`	`TGroupBox`	Each hosts one instance of the frame `TfraAddress` (unit `uLanguageComboFrame`) with labels, edits and a country combo box.
`btnDetails`, `btnAbout`	`TButton`	`Action` points to `actDetails` and `actAbout` of the data module `dmLanguageCombo`. The captions come from the actions.

The combo and the localizer in the form file:

```
object cboLanguage: TsgcVCLLanguageCombo
  Left = 104
  Top = 24
  Width = 184
  Height = 23
  Localizer = Localizer
  TabOrder = 0
end
object Localizer: TsgcVCLLocalizer
  FileName = 'LanguageCombo.sgcloc'
  Storage = lstEmbedded
  Languages.Strings = (
    'de'
    'es'
    'fr')
  ProjectData = {...}
end
```

`ProjectData` is the compact copy of the project. The designer writes it when the form is saved; you never edit it.

The code

The main form has no code at all. The data module holds the two action handlers:

```
uses
  // sgc
  sgcLocalization_Engine,
  // demo
  uLanguageComboDetail;

procedure TdmLanguageCombo.actDetailsExecute(Sender: TObject);
var
  oForm: TfrmLanguageComboDetail;
begin
  { the form frees itself when it is closed }
  oForm := TfrmLanguageComboDetail.Create(Application);
  oForm.Show;
end;

procedure TdmLanguageCombo.actAboutExecute(Sender: TObject);
begin
  ShowMessage(sgcT('The active language is %s.', [sgcLanguage]));
end;
```

How it works

1. The combo lists the source language and the target languages of its localizer by their native names. The items are never stored in the form file.
2. A selection sets `Localizer.Language`. No event handler is needed.
3. The frame is translated once for its class, `TfraAddress`, and both instances follow. The selected country of each combo box stays selected.
4. The captions of the two buttons come from their actions. The items of the project are `TdmLanguageCombo:actDetails.Caption` and `TdmLanguageCombo:actAbout.Caption`, and the translated actions pass their texts to the buttons.
5. The detail form is created after the switch. `AutoTranslate` translates it before it is painted, so it opens in the active language. A detail form that is already open changes with the others.
6. `sgcLanguage` returns the active language for the message of the About action.

Try this

- Set `DisplayMode` of the combo to `ldmEnglishName` or `ldmCode`, and `IncludeSource` to False.
- Open two detail forms, then change the language in the combo: every window follows.
- Set `Storage` back to `lstFile`, build, and copy the executable alone to another folder. It starts in English, because the project file is not found. Copy `LanguageCombo.sgcloc` next to it and start it again.
- Translate a text in the editor, save the project and save the form. With `lstEmbedded` the save of the form is what brings the new text into the next build.

4.4 The standalone translation editor

Folder `Demos\Localization\Delphi\01.TranslationEditor`, project `TranslationEditor.dpr`.

What you will build

The translation editor as a program of its own, for translators who do not have the IDE. It opens the `.sgcloc` file named on the command line or chosen in a file dialog, and shows the window of figure 3.3. Give the executable and the project file to a translator, and take the project file back.

Components on the form

Component	Class	Properties you set
`Localizer`	`TsgcVCLLocalizer`	A container for the project that is opened. Its `AIProvider` points to the event provider.
`AIProvider`	`TsgcUIAIEventProvider`	The event `OnRequest`, the place to call an AI service.
`btnOpen`, `btnEditor`, `lblFile`, `OpenDialog`	Standard controls	Open a project, show the editor again, the name of the open file.

The code

```
function TFormMain.Editor: TsgcLocalizationEditorForm;
begin
  { the editor is a form without a form file, created over the localizer.
    This form owns it and shows it non modal. sgcLocalizationShowEditor is
    the modal call for an application that only needs a dialog. }
  if FEditor = nil then
    FEditor := sgcLocalizationCreateEditor(Self, Localizer);
  Result := FEditor;
end;

procedure TFormMain.OpenProject(const aFileName: string);
begin
  { loads the file into the localizer and remembers the name for Save }
  Editor.OpenProject(aFileName);
  lblFile.Caption := aFileName;
  Editor.Show;
end;

procedure TFormMain.FormCreate(Sender: TObject);
begin
  { the localizer is a container for the project, it never translates this tool }
  Localizer.ApplyToApplication := False;
  { and the Scan command never reads the form of the tool into a project }
  sgcLocalizationRegisterSkippedClass(TFormMain);
  { TranslationEditor.exe MyApp.sgcloc }
  if (ParamCount > 0) and FileExists(ParamStr(1)) then
    OpenProject(ParamStr(1));
end;
```

How it works

1. `ApplyToApplication := False` makes the localizer a container: it loads, edits and saves a project, but it never translates the tool itself and it is not the default localizer.

2. `sgcLocalizationCreateEditor` returns the editor form without showing it, for an editor that is not modal. `sgcLocalizationShowEditor` is the modal call.
3. The Scan button of the editor is disabled in the tool, because the tool has no application forms to scan. Scan from the IDE or from the application that owns the forms.
4. **AI translate** ends in `AIProviderRequest`, once for every batch of texts, in a worker thread. The demo is not connected to a service and raises an error that the editor reports. Chapter 7 shows how to connect one.
5. `FormCloseQuery` passes the question to the editor, which asks to save a project with changes.

Try this

- Run it and open `00.QuickStart\01.Hello\Hello.sgcloc`. Change a German text, save, and run the Hello demo again.
- Choose **Export XLIFF** for French, open the file in a text editor and look at one unit. Then choose **Import XLIFF** with the same file: every row of the review list is Unchanged.
- Choose **Validate** after you removed the `&` from a translated menu caption. The Validation tab reports the accelerator.
- Choose **Mark version**, give the name `1.0`, add a snippet, and filter with **New or changed since**.

> **Tip.** An editor compiled with Delphi 7 to 2007 edits the languages that the code page of the system can show. Build the tool with a Unicode Delphi for every other script. The project file is the same for every compiler.

5. Main components

The SGC Localization palette page is short, and one component covers a typical first application: the localizer. This chapter introduces the components, then the global functions you meet in code, and finally the notes for FireMonkey, Lazarus and C++Builder.

Every component has its own help topic, named after the class, such as `TsgcVCLLocalizer.htm`. Each topic lists the properties, methods and events with their types and defaults.

Reference table

Class	Unit	What it is for	Key members
`TsgcVCLLocalizer`	`sgcLocalization_VCL`	The localizer of VCL and LCL applications	`FileName`, `Storage`, `SourceLanguage`, `Languages`, `Language`, `ActiveLanguage`, `Rules`, `AIProvider`, `ScanApplication`, `TranslateRoot`, `LoadFromFile`, `SaveToFile`, `ExportXLIFF`, `ImportXLIFF`, `OnLanguageChanged`
`TsgcVCLLanguageCombo`	`sgcLocalization_VCL`	The language list for the user	`Localizer`, `DisplayMode`, `IncludeSource`, `RefreshLanguages`, `OnSelect`
`TsgcFMXLocalizer`	`sgcLocalization_FMX`	The localizer of FireMonkey applications	The same as `TsgcVCLLocalizer`
`TsgcFMXLanguageCombo`	`sgcLocalization_FMX`	The FireMonkey language list	`Localizer`, `DisplayMode`, `IncludeSource`, `UpdateLanguages`, `OnChange`
`TsgcLocalizationProject`	`sgcLocalization_Project`	The items, translations, marks and rules, reached through `Localizer.Project`	`AddSnippet`, `Apply`, `Validate`, `Mark`, `ItemsSinceMark`, `CountByState`
`TsgcLocalizationAITranslator`	`sgcLocalization_AI`	The AI translation in code, the component behind the dialog	`Project`, `Provider`, `TargetLanguage`, `Scope`, `BatchSize`, `Glossary`, `Instructions`, `Execute`, `ExecuteAsync`, `Cancel`
`TsgcUIAIEventProvider`	`sgcUICore_AI`	An AI provider answered by your own code	`OnRequest`
`TsgcUIAIChatProvider`	`sgcUICoreWS_AI`	An AI provider over the AI components of sgcWebSockets	`Chat`

The localizers

`TsgcVCLLocalizer` holds the translation project, scans forms, frames and data modules, and applies the active language to them. Its roots are the forms and the data modules of `Screen`, with the frames found inside them. The first localizer created is the default localizer that the global functions use. The localizer components and the language combos are never scanned or translated themselves.

Property	Default	What it does
`SourceLanguage`	`en`	The language the forms are written in.
`Languages`	empty	The target languages, one tag per line: `de`, `pt-BR`.
`Language`	empty	The requested language. Setting it at run time switches the application. `auto` takes the language of the system when the project has it.
`FileName`, `Storage`	`lstFile`	Where the project is, see section 3.12.
`AutoTranslate`	True	Forms, frames and data modules created after the switch are translated automatically.
`AutoBiDi`	True	A switch sets the application to right to left for `ar`, `he`, `fa` and `ur`.
`SyncUILanguage`	True	A switch also sets the language of the built in texts of the sgcUI controls.
`Rules`	empty	The ignore and include rules of a scan.
`AIProvider`	none	The provider component of the AI translation.

The events are few: `OnLanguageChanged` after a switch, `OnScanProperty` to accept or refuse a text during a scan, `OnMissingSnippet` and `OnTranslate` for snippets, and `OnProjectLoadError` when the project file is missing or damaged at startup.

The language combos

`TsgcVCLLanguageCombo` is a drop down list of the languages of its localizer. A selection sets `Localizer.Language`, and the combo follows every language switch made elsewhere. `DisplayMode` chooses what the list shows: `ldmNativeName` (default) the name of the language in the language itself, `ldmEnglishName` its English name, `ldmCode` its tag. `IncludeSource`, True by default, makes the source language part of the list.

```
procedure TfrmMain.AddItalian;
begin
  Localizer.Languages.Add('it');
  // the list is filled again after a change by code
  Combo.RefreshLanguages;
end;
```

`TsgcFMXLanguageCombo` is the FireMonkey twin. Its display modes are `fldmNativeName`, `fldmEnglishName` and `fldmCode`, and the method that fills the list again is `UpdateLanguages`.

The global functions

Function	Unit	What it does
`sgcSetLanguage(aLanguage)`	`sgcLocalization_Engine`	Sets `Language` of the default localizer.
`sgcLanguage`	`sgcLocalization_Engine`	`ActiveLanguage` of the default localizer, an empty text without one.
`sgcT`, `sgcTC`, `sgcTN`	`sgcLocalization_Engine`	The snippet functions.
`sgcDefaultLocalizer`, `sgcSetDefaultLocalizer(aLocalizer)`	`sgcLocalization_Engine`	The default localizer: the first one created, or the one assigned.
`_`, `_n`	`sgcLocalization_Short`	Short names for `sgcT` and `sgcTN`, in their own unit. Add it only where the short names are wanted.
`sgcLocalizationShowEditor(aLocalizer)`	`sgcLocalization_Editor_VCL`	Opens the translation editor as a modal window.
`sgcLocalizationRegisterSkippedClass(aClass)`	`sgcLocalization_Scan`	Excludes a component class or a whole form class, with its descendants, from scanning and translation.

FireMonkey, Lazarus and C++Builder

FireMonkey. The steps are the ones of VCL with `TsgcFMXLocalizer` and `TsgcFMXLanguageCombo`, in Delphi XE6 or later. The translation editor at design time is the same window. On a mobile platform set `Storage` to `lstEmbedded`, so the translations travel inside the form file and no extra file is deployed. From Delphi 10.1 Berlin a form created later is translated before its first paint. In older versions a form that is created and shown in one go may paint once in the source language, and one line avoids it:

```
procedure TfrmOptions.FormCreate(Sender: TObject);
begin
  // only needed before Delphi 10.1 Berlin
  if sgcDefaultLocalizer <> nil then
    sgcDefaultLocalizer.TranslateRoot(Self);
end;
```

Lazarus. The LCL uses the components with the VCL names from the package `sgcLocalizationLazarus`. One difference matters: **Scan project** covers the forms that are open in the designer. Open the forms of the project before the scan, or scan the running application with `Localizer.ScanApplication` and save the project. Lazarus strings are UTF-8, so every target language works without a matching system code page.

C++Builder. The design time steps are the ones of Delphi. The header of the form includes the units of the localizer, and the code calls the same functions:

```
// uMain.h
#include "sgcLocalization_Engine.hpp"
#include "sgcLocalization_VCL.hpp"

// uMain.cpp
void __fastcall TfrmMain::mnuGermanClick(TObject *Sender)
{
  // every open form changes at once, and the forms created later follow
  sgcSetLanguage("de");
}

void __fastcall TfrmMain::mnuSaveClick(TObject *Sender)
{
  // sgcT is overloaded, pass a String
  String vText = sgcT(String("The file was saved."));
  ShowMessage(vText);
}
```

6. Common problems

These are the problems that come up most often in a first translated application. Each entry says what you see, why it happens and how to fix it.

Nothing is translated

The application stays in the source language

What you see. `sgcSetLanguage('de')` runs, and every text stays as it was.

Why. Most often the project file was not found. A relative `FileName` is searched in the folder of the executable, then in the data folder of the platform, then in three parent folders. The current folder is not used. A missing or damaged file is ignored at startup, so the application still starts.

Fix. Check `Localizer.ProjectLoaded` and `Localizer.ProjectFileName`, which returns the full name that is used. Handle `OnProjectLoadError`, which receives the file name and the reason. Then deploy the `.sgcloc` file next to the executable, or use `lstEmbedded`.

The texts are in the project, but the application shows the source

What you see. The editor shows a text for the language, the application does not use it.

Why. Only texts in the state translated, reviewed or changed are shown. New items and proposals that were not accepted show the source text.

Fix. Accept the proposals, or type the translations, and save the project. With `lstEmbedded`, also save the form of the localizer and build again.

The language of the system is not used

What you see. `Language` is `auto`, and the application starts in the source language.

Why. With `auto` the language of the system is taken only when it, or its base language, is one of the languages of the project.

Fix. Check `Localizer.ActiveLanguage` and the `Languages` list.

sgcSetLanguage does nothing at startup

What you see. A call in the project file or in an `initialization` section has no effect.

Why. `sgcSetLanguage` works on the default localizer, the first one created. It does nothing before a localizer exists.

Fix. Call it after the form or data module of the localizer was created, or set the `Language` property at design time.

Single texts and forms

One text is not translated

What you see. A form changes language, one caption stays.

Why. There are four usual causes. The text was changed by code before the switch, and a switch only replaces a value that still holds the source text or the text of the previous language. The component was created in code with an owner other than the form, so no scan found it. The control takes its text from an action, and the item is on the action. Or the translation fails the format check, `%s` against `%d` for example, and the source text is shown in its place.

Fix. Build a text that code changes with a snippet function and refresh it in `OnLanguageChanged`. Set the texts of components created in code with `sgcT`. Translate the caption of the action. Run **Validate** to find the format errors.

A snippet function returns the source text

What you see. `sgcT('Ready.')` returns Ready. in every language.

Why. The snippet is not in the project, or the text in the code and the text of the snippet differ in a space, a dot or the case. The text is the key and it is compared exactly. A snippet added with a context is not found by `sgcT`, which uses no context, and the other way round.

Fix. Add it with **Add snippet**, or collect the missing ones with `OnMissingSnippet`. A snippet used with `sgcTN` needs its plural text.

One form is not translated

What you see. Every form follows the language except one.

Why. A form or data module that is not owned by `Application` and is never shown, a data module with report titles created with a nil owner for example, is not seen by the localizer. The same happens to every form created later when `AutoTranslate` is False.

Fix. Call `Localizer.TranslateRoot` for it after it was created:

```
oReport := TdmReport.Create(nil);
try
  Localizer.TranslateRoot(oReport);
  oReport.Print;
finally
  oReport.Free;
end;
```

Characters show as question marks

What you see. A Russian or Japanese translation shows as `????` in an application built with Delphi 7 to 2007.

Why. These compilers are ANSI. A language written in a script that the code page of the system does not hold cannot be shown by those applications.

Fix. Use a system with the matching code page, or a Unicode compiler. To edit such a language, use the standalone translation editor built with a Unicode Delphi.

The layout flips to right to left

What you see. After a switch to Arabic or Hebrew the forms are mirrored.

Why. This is `AutoBiDi`, which sets the direction that these languages are read in.

Fix. Nothing, when that is what you want. Set `AutoBiDi` to False to control the direction yourself.

Scanning

The scan finds texts that should not be translated

What you see. A code, a product name or a sample value is listed in the editor.

Why. It is a published text with letters, and no rule excludes it.

Fix. Add an ignore rule for the class, the component name, the property or the value, or decide in `OnScanProperty`. Items that were already found become obsolete at the next scan of their form and are hidden.

A scan at run time finds texts that code has set

What you see. An item holds a customer name or a counter as its source text.

Why. A scan reads the values the components hold at that moment, so a label filled in `FormCreate` arrives with the text of the code.

Fix. Scan at design time, or build such a text with a snippet function.

Scan project in Lazarus misses forms

What you see. Only some forms of the project have items.

Why. In Lazarus the command covers the forms that are open in the designer.

Fix. Open them first, or scan the running application with `ScanApplication`.

The combo does not show a language added by code

What you see. `Localizer.Languages.Add('it')` ran, the combo still lists the old languages.

Fix. Call `RefreshLanguages` of `TsgcVCLLanguageCombo`, or `UpdateLanguages` of `TsgcFMXLanguageCombo`.

Translators and AI

An XLIFF file is refused

What you see. Import XLIFF reports an error and opens no review list.

Why. The file must be UTF-8, a UTF-16 file is refused. It must also name its target language.

Fix. Ask the translator to save the file as UTF-8 with the target language set. Rows of the kind Unknown ID mean that the tool of the translator changed the ids of the units, or that the file belongs to another project.

AI translate does nothing

What you see. The run ends at once, or with an error.

Why. No provider is assigned to `AIProvider`, the provider is not complete, or the scope has no items because everything is already translated.

Fix. A `TsgcUIAIChatProvider` needs its `Chat` component with a valid key, a `TsgcUIAIEventProvider` needs an `OnRequest` handler. Choose the selected rows as the scope to ask for other texts. Remember that proposals never replace a translated or reviewed text: they wait in the Proposal field of the edit panel until they are accepted.

A text cannot be set to reviewed

What you see. The state goes back to translated, or **Mark as reviewed** reports skipped rows.

Why. The text has a validation error, and an error blocks the state reviewed.

Fix. Read the issues of the text in the edit panel and correct the translation.

7. Where next

You have a translated application, a language combo and snippets with plurals. This chapter lists the other demos, shows how to connect an AI service, and points to the help.

The other demos

Folder	What it holds
`Demos\Localization\FMX\00.QuickStart`	The three QuickStart demos for FireMonkey: `01.Hello`, `02.SnippetsAndPlurals` and `03.LanguageCombo`.
`Demos\Localization\Lazarus\00.QuickStart`	The three QuickStart demos for Lazarus.
`Demos\Localization\Lazarus\01.TranslationEditor`	The standalone translation editor as a Lazarus project.
`Demos\Localization\CBuilder\00.QuickStart\01.Hello`	The Hello demo for C++Builder.

Connecting an AI service

The AI translation needs a provider component in the `AIProvider` property of the localizer. There are two ways to get one.

With sgcWebSockets. Drop a `TsgcAIChat` of sgcWebSockets (palette page SGC AI) and a `TsgcUIAIChatProvider`, set the `Chat` property of the provider to the `TsgcAIChat`, and set `Localizer.AIProvider` to the provider. Choose the service with the `Provider` property of the `TsgcAIChat`, where the model and the key are set as well: OpenAI for ChatGPT, Anthropic for Claude, Gemini, DeepSeek, Grok, Mistral, or Ollama for a model on your own machine. The same in code:

```
uses
  SysUtils, Classes,
  // sgcWebSockets
  sgcAI, sgcAI_Chat,
  // sgc
  sgcLocalization_VCL, sgcUICoreWS_AI;

procedure UseClaude(aLocalizer: TsgcVCLLocalizer; const aApiKey: string);
var
  oChat: TsgcAIChat;
  oProvider: TsgcUIAIChatProvider;
begin
  oChat := TsgcAIChat.Create(aLocalizer.Owner);
  oChat.Provider := aicpAnthropic;
  oChat.ChatOptions.ApiKey := aApiKey;
  oProvider := TsgcUIAIChatProvider.Create(aLocalizer.Owner);
  oProvider.Chat := oChat;
  aLocalizer.AIProvider := oProvider;
end;
```

Without sgcWebSockets. Drop a `TsgcUIAIEventProvider` and answer its `OnRequest` event with the text of your own service. The request carries the instructions for the model and the texts to translate as JSON:

```
uses
  sgcUICore_AI;

// The handler runs in the thread that executes the request, a worker thread
// for the translation editor, so it must not touch the user interface.
procedure TdmMain.ProviderRequest(Sender: TObject; aRequest: TsgcUIAIRequest;
  var aAnswer: string);
begin
  aAnswer := MyService.Ask(aRequest.SystemMessage, aRequest.UserMessage);
end;
```

With a provider in place, **AI translate** works in the editor at design time, in your application and in the standalone tool. The component behind the dialog, `TsgcLocalizationAITranslator`, does the same in code, for a build step that proposes the new texts of every release:

```
uses
  sgcLocalization_Engine, sgcLocalization_Project, sgcLocalization_AI;

procedure TfrmMain.ProposeGerman;
var
  oAI: TsgcLocalizationAITranslator;
  oChanges: TsgcLocalizationChangeSet;
begin
  oAI := TsgcLocalizationAITranslator.Create(nil);
  try
    oAI.Project := Localizer.Project;
    oAI.Provider := Localizer.AIProvider;
    oAI.TargetLanguage := 'de';
    oAI.Scope := lasNewAndChanged;
    oAI.Glossary := 'Invoice = Rechnung';
    oChanges := oAI.Execute;
    try
      // the rows are proposals: Apply stores them next to the texts
      Localizer.Project.Apply(oChanges);
      Localizer.SaveToFile(Localizer.ProjectFileName);
    finally
      oChanges.Free;
    end;
  finally
    oAI.Free;
  end;
end;
```

> **Keep your API key out of the source.** Read it from a setting, an environment variable or a key store at run time, never commit it to version control, and remember that requests to a hosted model are billed to the account of the key.

Help topics

The sgcLocalization help is part of the sgcUI help at https://www.esegece.com/help/sgcUI, in the Localization section. These topics are the best continuation of this guide:

Topic	What it covers
Introduction	What sgcLocalization does and where to start.
Architecture, Installation, Platforms	The units and layers, the packages, and the compilers and frameworks.
Quick Start VCL, FMX, Lazarus, C++Builder	The five steps for each framework and compiler.
Scanning and ignore rules, Snippets and plurals	The scan in detail with the full built in list, and the texts of the code.
Translation editor, Incremental translation, Validation	Every command of the editor, the states and marks, and the validation rules.
XLIFF exchange, AI translation	The exchange with translators and the proposals of a language model.
Runtime language switch, Project file and deployment	What a switch does, the format of the file, the storage and a deployment checklist.
Troubleshooting	More problems and their causes.
One topic per component	`TsgcVCLLocalizer`, `TsgcVCLLanguageCombo`, `TsgcFMXLocalizer` and `TsgcFMXLanguageCombo`, with every property, its type and its default.

On the website

- https://www.esegece.com/products/sgclocalization/: the product page, with the features and the downloads.

Getting help

If you are stuck, write to info@esegece.com or open a ticket in the helpdesk of your account at https://www.esegece.com/. It helps to include your Delphi version, your sgcUI version and edition, and the demo or a small project with its `.sgcloc` file that shows the problem.

Enjoy translating your application.

---

sgcLocalization and sgcUI are products of eSeGeCe. https://www.esegece.com