Localization of AOO proposal for new workflow

27 October 2012

Contents

Introduction

This document is a proposal for a new l10n workflow, the reasons to change the current workflow are simple:

- ✔ It puts us back in total control of the l10n process
- ✔ It removes the need to rely on partially broken or lost tools.
- ✔ It reduces the number of steps strings must go through to be translated and integrated.
- ✔ It automates a number of operations that have been manual so far.
- ✔ It allows to have a proper version control for translations.

The document describes an end state, of course it will be developed and committed in phases. As development progresses this document will be updated to describe all steps in the l10n workflow not only the technical parts.

There have been a discussion on file formats (.po versus .xliff), in order not to confuse the themes this document is written in a way that both file formats can be used, only the last chapter discusses advantages/disadvantages of the 2 formats.

There is also an ongoing discussion on how to integrate offline translation and offer better QA methods, as a direct consequence the document can be seen as two parts:

- Developer part, this part revolves around the build/commit process used by developers
- Translation part, this part revolves around handling of language files

Thanks to all those persons who contributed to enable me to write this document.

Overview

Localization, often abbreviated as l10n, defines the process to make a software package available in local languages, different to the language of the developer.

L10n is in more popular terms called “Localization of AOO”, or in very simple terms just “translation of AOO”. L10N defines the workflow which makes AOO available in local languages.

Localization is from the perspective of the involved people a multi-step process that involves a variety of tools and procedures. There different main types of people involved have quite different and to some extent conflicting views and requirements, therefore the workflow is a real “best of all worlds” approach.

It has been an objective to make as much as possible of the workflow automated, but there still remain a few manual steps.

The process is in short:

- Developers add messages to source files, which are automatically extracted and converted to language files,
- Language files are manually committed in SVN, and thereby available to translators,
- Translation work, regularly committed in SVN or held offline
- Translation review/commit
- Building language specific test versions with/without key recognition
- Release candidates for language specific AOO built on the SVN content.

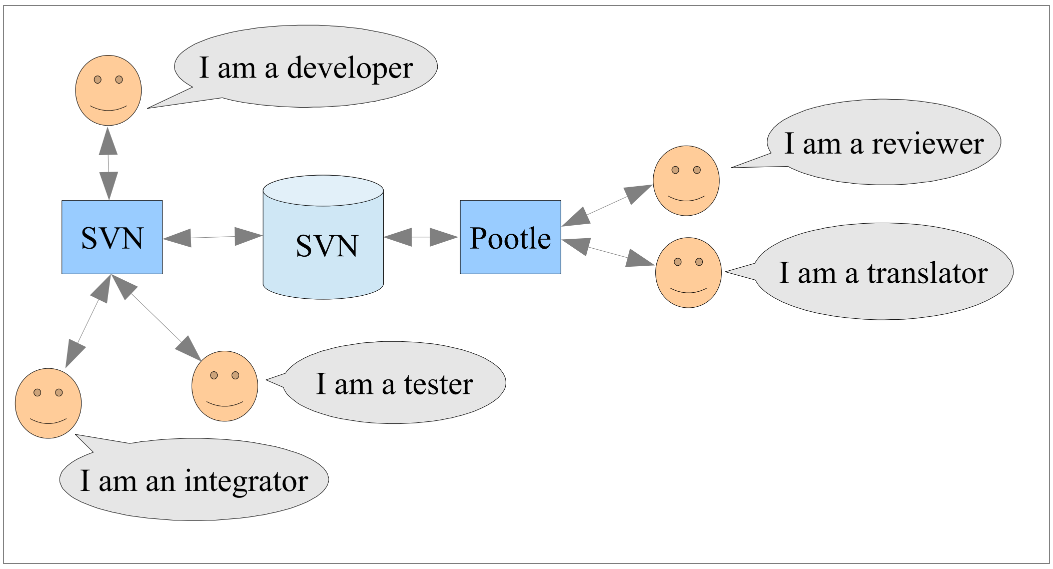
If you are looking for information about how to contribute translations then this page gives an overview.

The document has 8 parts:

- role definition,
- non-technical workflow overview,
- simplified data flow,
- detailed technical workflow walk-through,
- File formats used,
- Tools used,
- Temporary: Discussion on .po versus .xliff,
- Temporary: Project plan.

Actors and Systems

The l10n process can and should be viewed with respect to 4 different categories of people who access the process through different tools, but one common repository.



The picture illustrates the different type of people involved in the workflow, the process itself is explained later.

Note: this view only relates to the l10n procedure, the picture for the whole project is a lot more complex.

Developers

Developers construct the actual program with different programming languages (C++, Java, Python...) and will as part of the development embed english messages (errors, warnings …) in the source code and/or build UI containing english text. Developers are fluent in their language (C++, java, python etc.) but for sure not in all the native languages supported by AOO therefore localization is needed.

The development review process has a number of shortcomings:

- no check of the text (spelling, wording, consistency),
- reporting language "bugs" are different for english than other languages.
- We have en-GB, en-ZA but no en-US and no.

This is a theme that we should address outside the scope of this proposal, and of course keeping in mind developers do not have to be linguist.

Translators

Translators add texts in the local native language, relating (translating) to the original message. In a release there is a 1-n relation between the original message and the supported languages, where n is the number of supported languages.

It is not a requirement for the translators to know the different programming languages, because the texts are automatically made available in form of content files with the principal layout::

> <original english text> = <native language text>

Translators for a given language are normally working as a small team. The division of files among the team is highly different and handled by the team itself.

Language reviewer

In general one person of the team review the translations, collect all files and make them available using pootle. The pootle server includes tools that takes the mechanic part out of the review process

The reviewer must either be a commiter or work with one, in order to commit the files to SVN.

Integrators

Integrators have mainly administrative task, such as:

- committing the language files in SVN if not done by the developers,
- control the automatic nightly build process,

Integrators does in principle not need to have programming or translation knowledge, because they are basically doing administrative tasks.

Testers

Testers check the total system and do a quality assurance of the behaviour.

Testers need a deep knowledge of the behaviour of the system, but deep technical knowledge is not needed.

To help testing a special test build will be available, showing the keyId of strings. This build has the advantages and disadvantages:

- it makes it easy to verify that each message is tested
- it ruins the UI because of the text length.

System: SVN

The sub version server is the actual repository and all systems work directly on this server.

All source files, language files, glossaries, documents etc. are stored in SVN.

System: pootle server

The pootle server provides an dedicated environment for all translators to work in.

The pootle server uses SVN to store the language files, so a translation is immediately available for build.

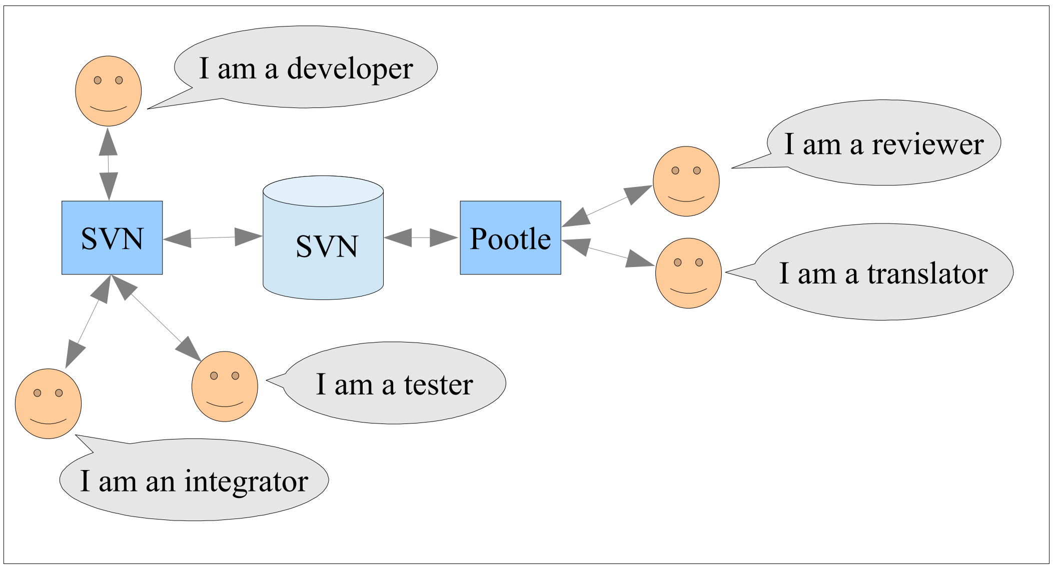
Offline translators use pootle to upload/download files for translation and to take status on ongoing work.

L10n workflow (non technical) view

The workflow is designed for a high degree of parallel work fully automated, basically it handles developers, translators and testers as equals.

Even though the tools allow parallel work, the workflow will have a manually decided point of "string freeze" which signals that development cannot change strings, and translators have a fixed set of strings to translate. The "string freeze" can also been seen as a transition point, where the focus of the release process shift from development to QA, translation and bug fixing.

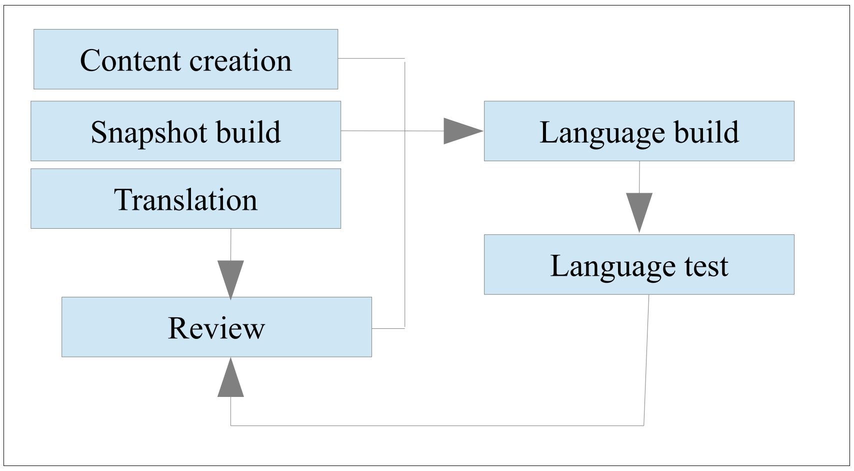
The workflow allows translators to start early on translations and thereby providing language testers with language versions parallel with the English version. In theory it should reduce the time needed between end of development and release date, and in praxis it allows early testing of language versions meaning a more stable and complete release.
Lets look at the components (using the picture from before)



The central and only repository is SVN. The SVN server is accessed by SVN utilities and Pootle. It is important to note that none of these programs have local storage (except for caching etc.), so there are NO copying of data!

The common repository is the key point to a parallel workflow.

The workflow consist of the following “steps”:



The phases **Content creation**, **Translation** and **Review** changes the language files:

- **Content creation**, create texts to be translated, but do NOT change text,
- **Translation**, translates existing text string, but do NOT create text,
- **Review**, changes translated text, but do NOT create text.

The phases **Snapshot build** and **Review** makes the texts available for build and translation.

Of course a text cannot be translated before it is created, so there are a natural flow from top to bottom seen for a single text string, but the process as a whole is truly parallel. The only sequences are:

- Translator → Review, which is a true quality gate,
- Language build → Language test, which is pure necessity.

Content Creation

Developers construct/develop new functionality or correct bugs/issues using different tools and programming languages. During the programming they may insert texts in the source files, this is done very differently depending on programming language and type of application (UI or error/information messages).

The texts are automatically extracted with every build and sent to a language staging area as part of the normal build process of a local directory.

When a developer build the complete AOO, all language files in all languages are automatically updated and ready for use solely on the local platform because nothing is committed automatically. The source files are NOT changed (contain only the original english entry).

When a developer build the complete AOO with language optionll language files in all languages are automatically updated and ready for use solely on the local platform because nothing is committed automatically. The source files are NOT changed (contain only the original english entry).

When a developer build the complete AOO with a specific language option. All language files in all languages are automatically updated and ready for use solely on the local platform because nothing is committed automatically. The source files are NOT changed (contain only the original english entry). Furthermore the resource manager is instructed to start AOO in the choosen language.

Sharing the language files poses no problem, since the developer works on offline.

Sharing the commit responsibility for the language files poses no problem, because by nature the usage is divided:

- Developers are the only ones who create strings (add keys) to the language files
- If a developers remove a text (key), the line is marked in the language file, and the translator can remove the line.
- If a developer changes a text (key), it is considered a new text (key), and the translators must copy the translation from the old key (unused) to the new key.

In order for the developer to test a change, a complete build of AOO is needed, therefore all language files are created. Each main directories correspond to exactly one language file, so when a developer commits the source files, it is easy also to commit the language files, in order to make them available for translation. If the developer does not commit language files it will happen as part of a snapshot build.

Snapshot build

At regular intervals (more frequently as the release date comes closer) a snapshot build is made. The differences from a normal build to a snapshot build are:

- the revision is marked in order to be able to rebuild exact the same revision, independently of what new items has been committed,
- it is being made available for download on openoffice.org in all languages,
- it includes build in a "key" version in all languages, allowing testers to see the string keys.

The snapshot build consist of the following steps:

- extract all language files,
- manually commit language files, not already committed (would normally be done by the developers),
- mark revision, which includes all languages,
- build all language versions,
- build all language versions in "key" mode (for testers),
- publish on openOffice.org.

With this method a snapshot build is just as complete as a release, containing both source code and language files. This is a good starting point for translators, since it is consistent and can be tested.

Translation

There are (based on 1 file for each subdirectory in main):

- 8 help files, corresponding to the product parts (writer, calc...),
- 46 UI/message files correspond to the directories in main (source tree), this will be reduces further (with a possibility to combine directories) on basis of feed back from the translation teams.
- 1 glossary file

to be translated in (at the moment) 112 languages, given a total of 6.048 files to be translated (a big reduction compared to the old process which had 450 files a total of 50.400 files).

Alone the number signals the amount of effort required and the need for an efficient process and the requirement for a highly automated QA process.

The files to translate are:

8 help files			
sbasic.<type>	scalc.<type>	schart.<type>	sdraw.<type>
shared.<type>	simpress.<type>	smath.<type>	swriter.<type>

46 UI files			
accessibility.<type>	avmedia.<type>	basctl.<type>	basic.<type>
chart2.<type>	connectivity.<type>	crashrep.<type>	cui.<type>
dbaccess.<type>	desktop.<type>	editeng.<type>	extensions.<type>
filter.<type>	forms.<type>	formula.<type>	fpicker.<type>
framework.<type>	instsetoo_native.<type>	javainstaller2.<type>	mysqlc.<type>
officecfg.<type>	padmin.<type>	readlicense_oo.<type>	reportbuilder.<type>
reportdesign.<type>	sc.<type>	scaddins.<type>	sccomp.<type>
scp2.<type>	sd.<type>	sdext.<type>	setup_native.<type>
sfx2.<type>	shell.<type>	starmath.<type>	svl.<type>
svtools.<type>	svx.<type>	sw.<type>	swext.<type>
sysui.<type>	sysui.<type>	ucbhelper.<type>	uui.<type>
vcl.<type>	wizards.<type>	xmlsecurity.<type>	

<type> is used in a generic sense, see later for the discussion about file formats. The number of directories might change for two reasons:

- it is decided to combine directories (the tool offers it),
- developers use text strings in non-listed directories.

The tight relationship directory – language file, makes it easy for developer and translator to work together while still maintaining the possibility for a team to split the translation on a file basis.

More importantly, when a developer make a change it is immediately available to the translator (provided the developer also commit the language files), this allows very fast turn around times in cases like bug fixes.

Additional there is one glossary file available for each language, which should be used for generic terms (e.g. Cancel) in order to secure a consistent translation. If the glossary is used the translation will automatically be controlled against the glossary during the next build.

Since the language files are stored in SVN, the translator can just as the developer backtrack changes.

Translation workflow

The bulk of translators work offline, and many are primarily interested in translation and local themes. Many want to have easy access to translation without having to concern themselves with SVN and other technical details. Pootle (extended version) provides mechanisms to provide possibilities for download/upload of set of files (they still have to be committed manually).

The translators downloads the files of interest from pootle. The translator can do that either anonymously or with an id. If they do it with an id, the files are given that status as being in translation (by id).

The translation itself is carried out with local tools like poEdit or virtaal. It is important for efficiency reasons that all changes are marked "fuzzy", since this allows the reviewer to work more efficiently.

It is important to note that translators have a free choice of when to start translation. There are 3 point to initiate translation:

- **"bleeding edge"**, using the newest SVN entries. This cannot be recommended (except for critical isolated bug fixing situations). With the "bleeding edge", there are NO guarantee that the text string keys will remain stable.
- **"snapshot build"**, this guarantees at least the AOO is build-able and use-able (even though some functions might not work). It is possible but unlikely that text strings (keys) change, but very likely that new strings are added at a later point in time. This stage is recommend for early translation work, especially because the snapshot build also provides the language reviewers with the possibility to test the translations.
- **"freeze string"** period, which is defined to be after all direct development is finished (not including QA), the translator has a stable set of text strings (keys) to work on. Developers will probably still solve QA problems, but the set of text strings remain constant. A small word of precaution: this is also the time where the pressure to release the product is highest, and per definition the translator is at the end of the development process and therefore prone to the highest pressure in order to keep deadlines.

A combination of translation based on **"snapshot build"** and **"freeze string"** seems to be the most efficient method. Use **"snapshot build"** to e.g. translate "draw" because development is finished but wait with "writer".

Once the translator has finished translating the downloaded files, they should be uploaded to the pootle server. The upload stores the files locally (WITHOUT commit) while waiting for a review process. This can be done on a file by file basis or in sets, depending on how the teams choose to work.

When the files are uploaded to the pootle server, the Reviewer takes over the responsibility.

Language reviewer

The language reviewer takes care of the team, splitting the files for translation and collecting them. The language reviewer will often act as a non-official head of the team being the contact point to the rest of the AOO community.

Once the translators have translated the files, they will be reviewed. It is important that the reviewer do NOT take active part in the translation but have “new” eyes. The purpose of the review are to:

- secure that the terms in the glossary are used throughout,
- secure the language is fluent and modern (today in many western languages english terms like “computer” or “internet” is accepted and used.
- Secure (with the help of pootle) that everything is translated syntactically correct, so it can be used the process following.

The reviewer (normally being a committer) can choose to either make changes directly on the pootle server or ask the team member to do make the changes and upload a now file.

The reviewer must remove the “fuzzy” mark, to acknowledge to translation and finally commit the file, to make it available for language builds.

Language build

Developers and integrators can any time do a language build with the newest translations, there are no manual steps needed. However it is advised to check with the language reviewers before starting the process.

During the build the language files will be split into the parts needed by the source files.

The language build are done in 3 different parts:

- Normal installation set, basically the resource manager is “told” which language to use during setup.
- Language packs, these are files that can be added to an existing installation.
- Test installation set, this set is only interesting for a tester, all text strings in the UI are shown with ID, in order for for the tester to check that all ID have been tested. WARNING this set cannot be used to check the UI layout since the key changes the actual layout.

Language test

When a language version is built, the UI test tools will also be localized and as a consequence a test suite can be executed in the local language.

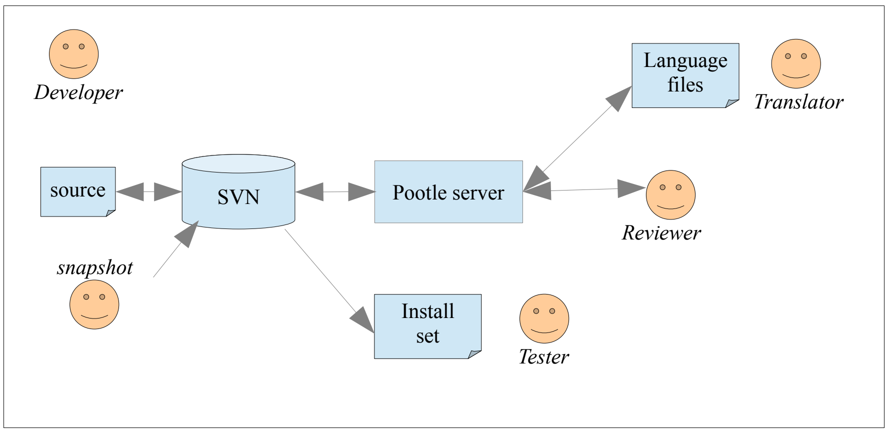
Furthermore a AOO that shows the keys will be provided.

The QA process for languages still needs to be refined. Today it depends a lot on the local teams, these team must be freed from the burden of the ordinary test.

Local teams should create test cases important for their language, like e.g. sort sequences or spelling control. Once created these test cases must be executed automatically and the findings handled by our QA team (independently of language).

Simplified data flow

As seen from the diagrams the flow is very natural and without manual steps:



The **“source”** files reside in the local developer directory until manually committed.

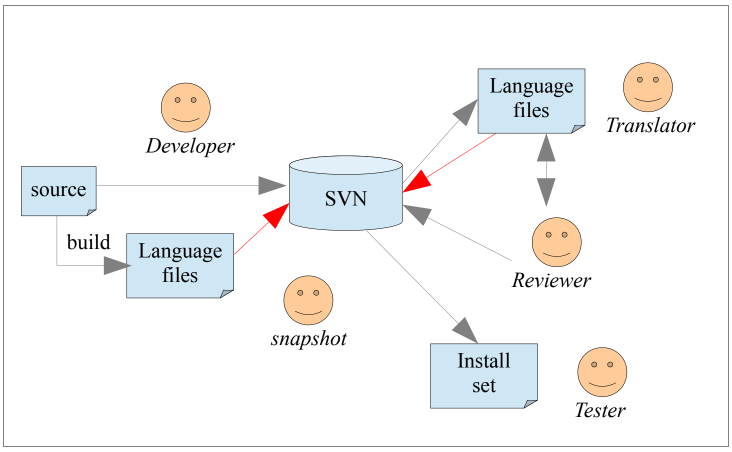
The **“language files”** reside in pootle, until committed manually by the reviewer.

SVN is the nucleus where we keep everything and can backtrack all changes. Even though not obvious from the drawing a developer and a translator are completely equal !! if the developer has status as “contributor” a “committer” is needed as intermediary just like the Reviewer.

But more importantly the data flow show clearly how automatically the process is, even though we have manually QA clearance gates (committing to SVN).

The simple data flow is THE most important point for a robust workflow, and also a guarantee that it can be expanded with future demands.

The flow of a single file can be viewed as:



The red arrows indicate optional commit points.

L10n workflow (developer) walk-through

This chapter is identical to L10n workflow (non technical) view but seen from a technical view showing actual commands, names of files and directories as well details of the tool behaviour.

Content Creation

Developers write text that needs to be localized. In principle the texts can be kept in files with any extension since most compilers are quite large in that respect. However the programming guidelines should secure that only defined extensions are used.

The extraction tool handles the following extensions:

Extensions scanned for text

Files	Extension	Desription
814	.hrc	header for resource files
98	.properties	java property files
1.040	.src	source for resource files
15	.tree	help files
53	.xcd	xml files only in postprocess
314	.xcs	xml file for java
1.365	.xcu	xml files for UI
4.543	.xhp	AOO help files
1	.xrm	xml readme file
8.243	out of 438.189	

The makefile/build contains a target **genLang**, and these extensions are globally associated with the target just like .cpp is associated with .o (and served by the C++ compiler). However the developer can in the local makefile overwrite the default. This makes sense in directories that contain files of these types with text that should not be translated.

Build target **genLang** is served by the tool **generateLanguage** (source is located in main/l10ntools). **generateLanguage** extract the messages directly, only in case of Java a sub-process is started. With the standard association the tool will only run once, and not for each file, by overwriting the default it is up to the developer if the tool should run for each file or once (the tools handles multiple filenames as parameter).

If the developer decided to use other extensions, **generateLanguage** has a parameter –**useExt** that overwrites the file extension.

Assuming the developer is in a local directory (e.g. main/l10ntools) and calls:

```
build
```

the code will be built and one language file (<directory name>.xx) will be created. The language file will be placed in a staging area (extras/l10n/staging), think of it as an object file ready for linking.

In order to build a complete AOO, following command is used

```
build --all
```

This command will update all native language files, based on the generated language files.

To manually update all the native language files (only applicable after AOO has been completely build), use:

```
cd extras/l10n
build
```

Once AOO has been built once, language versions can be built.

The following command insert all languages in the source files for the project:

```
build --withLang
```

To make an installation set with a specific language, use:

```
Build –all --withLang=”da”
```

Note: When doing a commit, it is optional to commit the updated language files in. If it isn't done the texts are not available to the translators before next snapshot build.

Snapshot build

A snapshot build is very much like a release build, it creates an install-set.

The process of a snapshot (automated in a script) is:

```
build --all
```

Then manually commit any changed native language files, thereby making sure that both source and native language files are at the same level. The revision of SVN is the snapshot revision.

Build the actual snapshot, prepared for languages:

```
build –all --with-lang
```

Or build the actual snapshot, for all languages:

```
build –all --with-lang=ALL
```

The snapshot build is important for both translator and tester, since it is a checkpoint where code and native language files are guaranteed to fit each other.

Translation / review

In order for the translator to work according to the workflow, the pootle server needs to be adapted, see tools section.

Language build

To prepare all sources with all languages, use:

```
build –all –with-lang
```

To generate an install-set for one or more specific languages, use:

```
build –all –with-lang="..."
```

To generate an install-set for all languages, use:

```
build –all –with-lang=ALL
```

The “--all” will make a build in extras/l10n which will first update all language files based on the extractions (in staging area) from the source and then secondly process all resource files.

Resource files (src files) are processed when the other modules are built. The original src files contain strings only for en_US in lines that look like

```
Text [en_US] = "...";
```

generateLanguange adds the missing languages by adding lines like

```
Text [de] = "...";
```

By default all (available) languages are added not just the ones given to configure's --with-lang switch. The augmented src files are placed in <module>/<platform>/misc/... These are then aggregated into some srs files in <module>/<platform>/srs/. In a (or several) following step(s) the srs files are aggregated into res files, one for each language.

The resulting res files are delivered to main/solver and become part of the installation sets. Multi-language versions contain res files for more than one language.

At runtime the ResMgr class from the tools module is responsible to use the resource files of the currently selected language whenever a string is requested (as is the case for e.g. all button texts and in general for all text visible in the GUI.)

Language test

The snapshot build generates installation-sets or language packs for all languages, allowing tester to install a full AOO and test it manually.

At a later stage, the current automated test tools will be adapted to use different languages, in order for testers to run an automated UI test.

At a later stage, the testers will be provided with special installation-sets, where the message keys are visible, allowing the testers to have a check list with all messages and work their way through the list in a formalized way.

Apart from testing the UI, the testers need to write test cases specific for a language like sort sequences.

File Formats

Quite a number of different file formats are involved in the localization process. The following list is not complete and may be inaccurate:

Extension	Desription
.hrc	header for resource files
.properties	java property files
.po	contains the translated strings from a .pot file. Used on the pootle server.
.pot	created by gettext from source files. Contains strings that need translation. Not used by OpenOffice except as part of the pootle server update.
.res	created by transex3 from .srs files.
.sdf	used to store localized/localizable strings and their origins. Comparable to .po files.
.src	source for resource files Most strings used in the GUI are defined in .src files.
.srs	Made by rsc (which calls rscpp and rsc2) from multiple src files with *all* language strings included.
.tree	help files
.xcd	xml files only in postprocess
.xcs	xml file for java
.xcu	xml files for UI
.xhp	AOO help files
.xliff	a format with the same usage of .po, but it has more functionalities and is standardized.
.xrm	xml readme file

Tools

A small set of tools are involved in the localization process. They mostly extract strings from source files and merge the translated strings back in, or transform between different data formats.

The following list show the tools used in the workflow:

tool	description
build	Standard build tool
generateLanguage	Integrated in build to extract text and generate resource files
updateLanguage	Integrated in build to update native language files
jpropex	Called from generateLanguage to translate .properties files
pootle server	Standard translation server, with extensions for AOO

generateLanguage

This tool is normally hidden in the makefiles. It is a command line tool, sources are in l10ntools.

The tool operates in two modes, extract/generate. Each mode is described in the following, but the commonalities are described below.

The tool takes a number of parameters:

- **--extract**, used to generate a language file from one or more source files,
- **--update**, used to update source files from a language file,
- **--useExt**, used to overwrite the file extension,
- **--langFile**, used to name the langauge file,
- **--sourceFiles**, used to pass a file list

Emphasis has been on speed, therefore the tool require all files on the command line (for a directory) in order to only handle any file once.

--extract

When extracting text from sources, each file given on the command line is scanned according to extension, and the texts are added to the language file. The language file is of the same type that translators use for translation, but it is still an intermediary file in the sense that it only contains the extracted text.

--extract

With this switch, the directory extras/l10n/source is searched for language files with the given name, the files are loaded into a translation memory. Then each file given on the command line is updated with the contents of all native language files (of course only for the texts in this file).

UpdateLanguange

This tool is normally hidden in the makefiles. It is a command line tool, sources are in l10ntools.

The tool is only used to build the directory extras/l10n.

The tool takes the generated language files kept in the staging area, and merges them with the native language files. Remark the native language file is only written if there are updates !!

pootle server

The standard pootle server lack a couple of features needed, these must either be extended in the pootle server project, or as local addons:

Mark ownership of files, by download

When a file/file-set is downloaded, the files should be marked locally in the pootle server, with timestamp, and who has the file (comment field ?)

This enables the team coordinator to track offline files/work.

Download fileset

It should be possible to mark files for download, and then get them as a set, instead of having to download each single file.

Upload fileset

It should be possible to upload a fileset (use same name convention as stored in pootle), instead of having to upload each single file.

Who has a file

It should be possible to see who has downloaded a file and when it was done.

Markings

Each file should have the following marks:

Lent out, to be reviewed, to be tested, Ok

Each message should have the following marks:

Suggestion, to be reviewed, to be tested, Ok

SVN capabilities

The pootle server should have a easy way for a team coordinator (also committer) to refresh the database from SVN and an easy way to commit file changes (using the committer userId)

Review feature

The review feature of pootle should have 2 extension:

- Possibility to ignore a violation of e.g. bracket control on a single text
- Check word/term usage for consistency (is Cancel translated identically?)

The goal of the review must be that there are none warnings.

Temporary: Discussion on .po versus .xliff

There seems to have been big discussions on file format, the above described workflow DOES NOT depend on the file format.

However there are advantages and short commingles of both formats:

.po

This is a very simple format, even though it is not recommend to edit with vi/notepad. It is widely used.

Advantages

- We have it today, and the offline translators have tools like poEdit installed

Disadvantages

- It provided no standard facility to store originating file names, this must be done in a comment.
- It provided no facility for status of the translation (like: not-translated, not-reviewed...)

.xliff

This is a simple XML format, and can be edited with a standard xml editor even though it is not recommended. It is used more or less solely for translation.

A recommended multi-platform openSource editor is e.g. virtaal from the translation toolkit.

Advantages

- It can store originating file names as an XML tag.
- It has a status tag of the translation (like: not-translated, not-reviewed...)
- It is supported by pootle server

Disadvantages

- Offline translators need to get used to a new tool