ALTO Studio
Audio Dialogue Tool

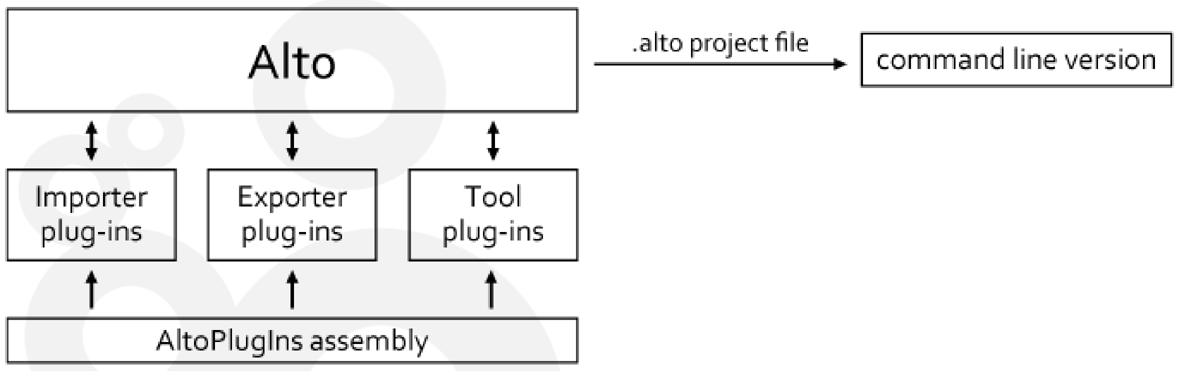
Integration notes for version 4.0

tsugi
Koshimura building, 3-5-8 Yoneyama
Chuo-ku, 950-0916 Niigata City, Japan
www.tsugi-studio.com

Introduction

Alto Studio can import dialogue from various sources: from game middleware projects (Audiokinetic's Wwise, Firelight Technologies' FMOD, CRI Middleware's ADX2 or Tazman Audio's Fabric) to Excel sheets, or directly from folders of audio files with various hierarchies and naming conventions.

However, it is sometimes necessary to interface with proprietary tools or dialogue databases. For this reason, Alto Studio also includes a plug-in system. This plug-in system makes it possible to write "Importer" plug-ins that allow the import of dialogue files, "Exporter" plug-ins that can save the data in a proprietary format and "Tool" plug-ins that can be used to extend the feature set of Alto Studio. Furthermore, the command line version of Alto Studio can be run from a script while using the same .alto project files than the GUI version.

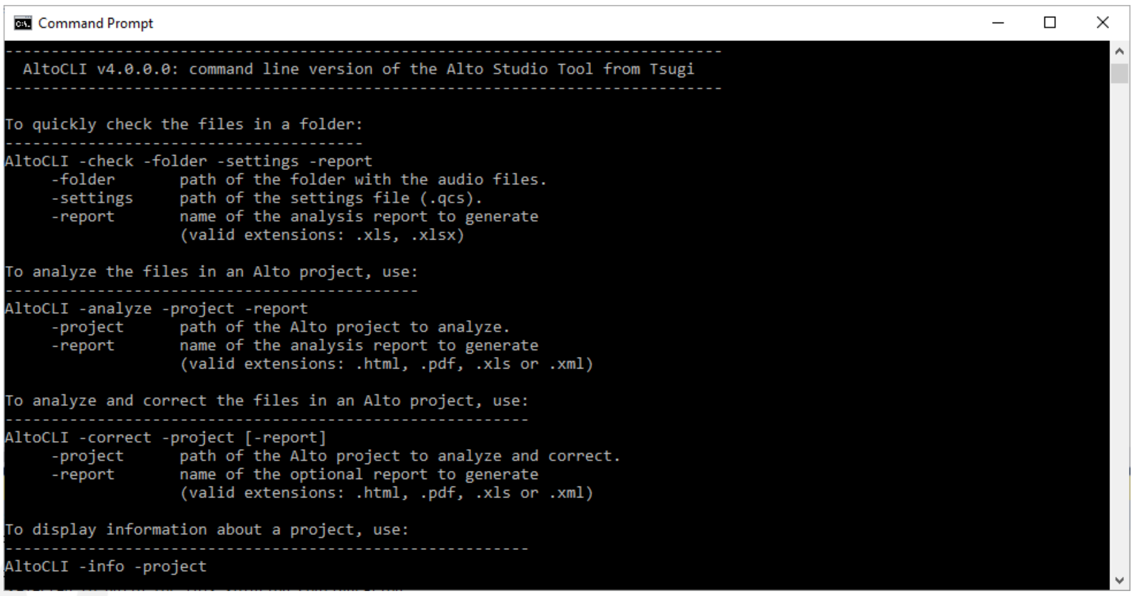


This documentation focuses on these ways to integrate Alto with your current build pipeline or tool chain.

Please note that the plug-in API and the options of the command line version are constantly extended based on the requests from our clients. Therefore, don't hesitate to contact us if you need an extra feature.

Command line version

You can integrate Alto Studio into your build pipeline by using the command line version, which is located in the same folder than the Alto Studio tool. The name of the executable is AltoCLI.exe. The command line version reads the same project or settings files than the GUI version. This makes it easy to configure a project and keeps the calling syntax simple.



```
Command Prompt
------------------------------------------------------------------------------
  AltoCLI v4.0.0.0: command line version of the Alto Studio Tool from Tsugi
------------------------------------------------------------------------------

To quickly check the files in a folder:
---------------------------------------
AltoCLI -check -folder -settings -report
     -folder       path of the folder with the audio files.
     -settings     path of the settings file (.qcs).
     -report       name of the analysis report to generate
                   (valid extensions: .xls, .xlsx)

To analyze the files in an Alto project, use:
---------------------------------------------
AltoCLI -analyze -project -report
     -project      path of the Alto project to analyze.
     -report       name of the analysis report to generate
                   (valid extensions: .html, .pdf, .xls or .xml)

To analyze and correct the files in an Alto project, use:
---------------------------------------------------------
AltoCLI -correct -project [-report]
     -project      path of the Alto project to analyze and correct.
     -report       name of the optional report to generate
                   (valid extensions: .html, .pdf, .xls or .xml)

To display information about a project, use:
---------------------------------------------------------
AltoCLI -info -project
```

The available commands are described below:

<u>Check</u>: check the properties of the audio files in a folder and generates a report. It is similar to the Quick Check feature from the tool and reads the same settings file. (.qcs).

```
AltoCLI -check -folder -settings -report
     -folder       path of the folder with the audio files.
     -settings     path of the settings file (.qcs).
     -report       name of the analysis report to generate
                  (valid extensions: .xls, .xlsx)
```

<u>Analyze</u>: analyze the files in an Alto Studio project and generate a report. This command is the equivalent of the Analysis or Compare features from the tool and reads Alto Studio project files.

```
AltoCLI -analyze -project -report
     -project      path of the Alto Studio project to analyze.
     -report       name of the analysis report to generate
                  (valid extensions: .html, .pdf, .xls or .xml)
```

Depending on the extension specified for the report's name, Alto Studio will generate an XML file, an Excel workbook, a PDF document, or HTML pages.

Correct: analyze and correct the files referenced by an Alto Studio project. This command is the equivalent of the Correct or Conform features from the tool and reads Alto Studio project files.

```
AltoCLI -correct -project [-report]
     -project      path of the Alto Studio project to analyze and correct.
     -report       name of the optional report to generate
                   (valid extensions: .html, .pdf, .xls or .xml)
```

The report generation is optional, and the type of report created depends on the extension specified for the report's name.

Info: display information about an Alto Studio project, such as the reference and localized languages and script information if any.

```
AltoCLI -info -project
     -project      path of the Alto Studio project to analyze and correct.
```

Voices: display the list of synthesis voices installed that are compatible with Alto Studio.

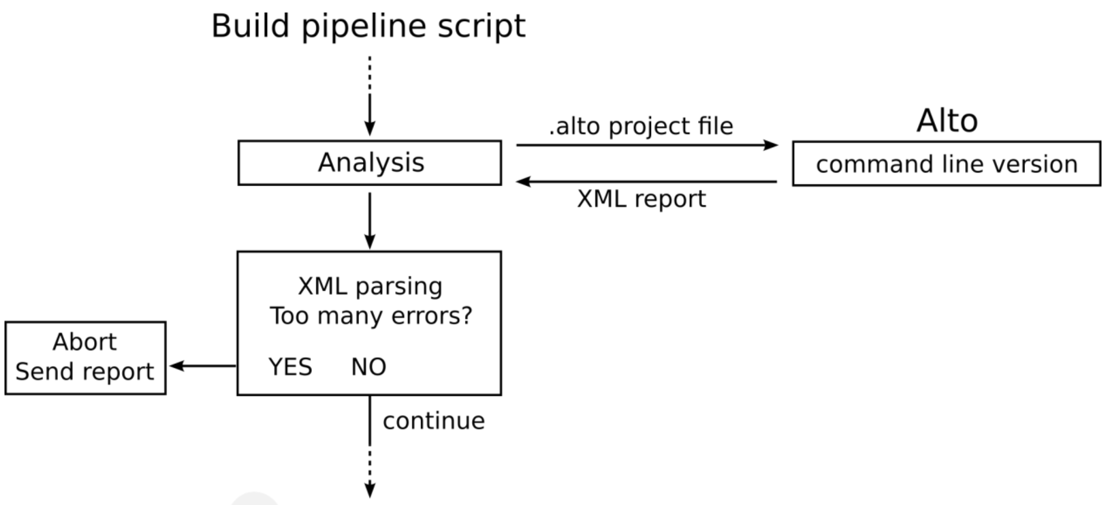
```
AltoCLI -voices
```

The names can be used for the -voice parameter of the next command.

Synthesize: synthesize dialogue using Windows text-to-speech synthesizer.

```
AltoCLI -synthesize -text -voice -output [-basename] [-volume] [-rate]
     -text         path of .txt file containing the dialogue lines.
     -voice        name of the voice to use e.g., 'Hortense'
     -output       path of the folder where to save the .wav files.
     -basename     optional base name of the .wav files (default = name of the voice).
     -volume       optional volume of the voice [1...100] (default = 100)
     -rate         optional rate of the voice [-10...10] (default = 0)
```

Please note that for the moment it is only possible to run one Alto Studio process at any given time. The figure below describes an example of integration.



Do not hesitate to contact us if your integration requirements are not yet met by the current version of AltoCLI. We can study the development of specific commands for your project.

Creating Alto Studio plug-ins

This section describes the steps required to create an Alto Studio plug-in. This is very easy and can usually be done in a couple of hours.

1 – Create a new Visual C# project of type "Class Library"

Please note that Alto being a C# application, plug-ins must be written in C# or any other managed language.

2 – Add a reference to AltoPlugIns

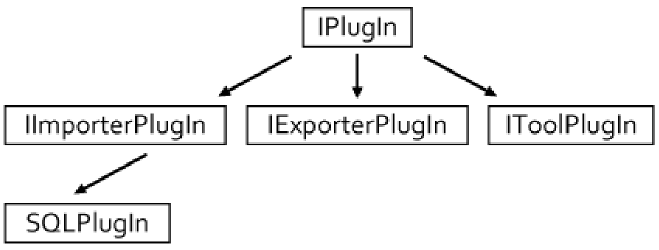
In the Solution Explorer, click on your project and select "Add Reference…" from the popup menu. Select the "Browse" tab and browse to the location of the AltoPlugIns.dll file (i.e. the root of the Alto Studio installation folder). This assembly contains the interfaces necessary to write a plug-in for Alto. You can now add the required namespace:

```
using AltoPlugIns;
```

3 – Derive your plug-in from the right interface

Alto Studio supports three types of plug-ins, which are all derived from the base `IPlugIn` interface. There are import plug-ins (`IImporterPlugIn`) that let you get dialogue files into Alto Studio, export plug-ins (`IExporterPlugIn`) to save the data in the format of your choice and tool plug-ins (`IToolPlugIn`) to add features to the software.

To create an SQL importer plug-in for example, derive your class from the `IImporterPlugIn` interface.



4 – Add the plug-in attribute to your class

```
[AltoPluginAttribute(PlugInType.Importer)]
public class SQLPlugIn : ImporterPlugIn
{
}
```

This is how the Alto Studio plug-in manager detects that a DLL is an Alto-compatible plug-in. The attribute takes only one parameter, which is the type of the plug-in: `PlugInType.Importer`, `PlugInType.Exporter` or `PlugInType.Tool`.

5 – Implement the IPlugIn interface

All plug-in interfaces derive from the base `IPlugIn` interface that any Alto-compatible plug-in must implement.

```
public interface IPlugIn
{
    string Name { get; }
    string Description { get; }
    string Author { get; }
    Guid GUID { get; }
    Version Version { get; }
    PlugInType Type { get; }
    Image Logo { get; }
    bool HasEditor { get; }

    bool EditParameters();
    bool ValidateParameters( out string errorMessage );
}
```

Name, Description, Author
These properties are strings that help you describe your plug-in. They will be displayed by the user interface of Alto Studio. The name does not have to be the same name than the plug-in assembly itself.

GUID
A unique identifier must be assigned the plug-in. You can create a new GUID by selecting the Tools->Create GUID menu command in Visual Studio. Do not copy an existing GUID! In the case of importer plug-ins, the GUID and the version of the plug-in will be saved in the Alto Studio project if the dialogue lines were imported through that plug-in.

Version

This property will allow you to track several versions of a plug-in, which is especially useful when loading or saving data. The version of a plug-in is always displayed by Alto Studio when using it.

Type

The type of the plug-in is again `PlugInType.Importer`, `PlugInType.Exporter` or `PlugInType.Tool`.

Logo

The plug-in logo: if null, the Alto Studio's default plug-in logo will be displayed by the GUI.

HasEditor

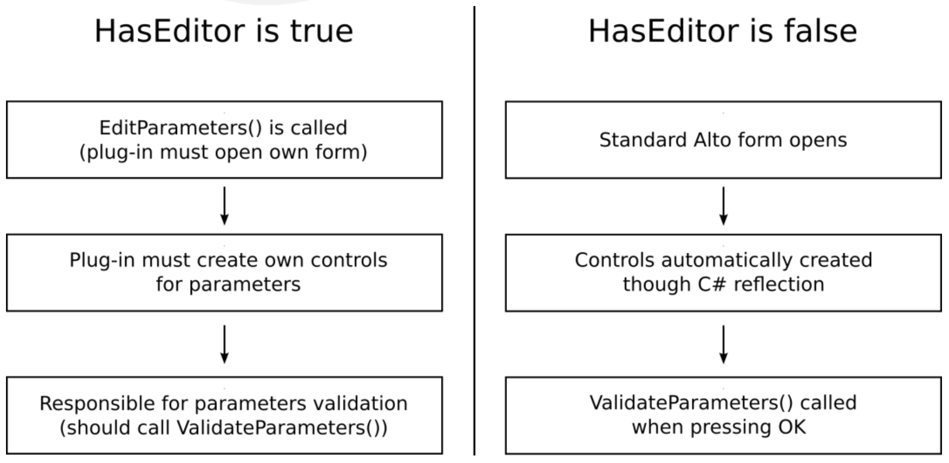
The `HasEditor` property should be set to true if you are planning on providing your own GUI to edit the parameters of the plug-in. In this case, the `EditParameters()` method will be called. If `HasEditor` is set to false, a default editor will be generated by Alto Studio based on the parameter definitions (i.e. using reflection).

EditParameters

Typically, this method should open a form and allow the editing of the parameters. It should return true if the parameters have been changed and false otherwise.

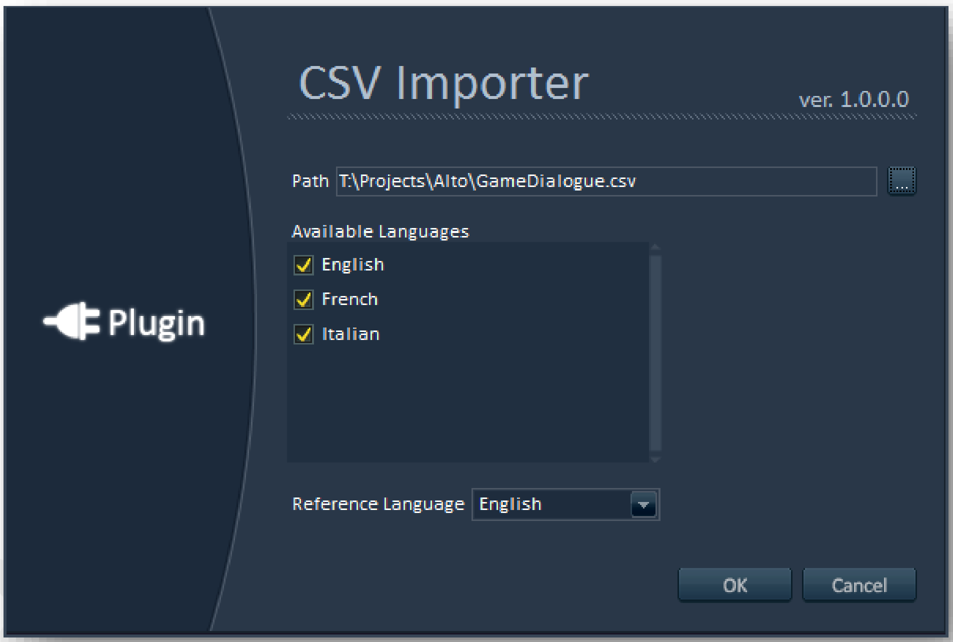
ValidateParameters

The `ValidateParameters` method is called when the OK button of the plug-in configuration form is pressed, before the form is closed. It is the responsibility of the plug-in to make sure that the parameter values are valid. If a parameter value is invalid, the method should return false and send a description of the problem in `errorMessage`. In that case, the message will be displayed by Alto Studio and the form will not be closed.


HasEditor is true
EditParameters() is called (plug-in must open own form)
Plug-in must create own controls for parameters
Responsible for parameters validation (should call ValidateParameters())
HasEditor is false
Standard Alto form opens
Controls automatically created though C# reflection
ValidateParameters() called when pressing OK

Importer plug-ins

This section describes the methods that must be implemented to create an importer plug-in and comply with the `IImporterPlugIn` interface. The CSV Importer is an example of a very simple importer plug-in. Examining its source code is a great starting point to learn how to build importer plug-ins.



Path T:\Projects\Alto\GameDialogue.csv

Available Languages
✓ English
✓ French
✓ Italian

Reference Language English

The `IImporterPlugIn` interface is defined like this:

```
public interface IImporterPlugIn : IPlugIn
{
    string ProjectFilterString { get; }
    bool IsValidProjectFile(string projectPath);

    bool Initialize(string projectPath);
    void Terminate();

    List<string> GetLanguages();
    string GetReferenceLanguage();

    List<string> GetReferencePaths();
    string GetLocalizedPath(string refFilePath, string language);
    List<string> GetLocalizedFiles(string language);

    void WriteSpecific(ref XmlTextWriter writer);
    void ReadSpecific(ref XmlNode parentNode);
}
```

ProjectFilterString
This property must return filter string to use in an open file dialog in order to load projects for the game audio middleware / tool / database we target.

IsValidProjectFile
This method is responsible for checking the validity of the file passed. It should return true is the file passed is in the expected format, and false otherwise.

Initialize
This method is only called once, when the plug-in is loaded (when Alto Studio is started). Any consequent initialization or resources allocation should be done here.

Terminate
This method is only called once, when the plug-in is released (when Alto Studio is closed). Any resources used by the plug-in should be freed here.

GetLanguages
This method must return the list of all the available languages in the project (including the reference language). There are no rules about the names of the languages, however they must be unique.

GetReferenceLanguage
This method must return the reference language. This is the language against which all comparisons will be done. The reference language must be part of the languages retuned by `GetLanguages`.

GetReferencePaths
This method must return the list of all the reference files in the project (i.e. all the files which are in the reference language).

GetLocalizedPath
This method must return the path of a localized file based on the path of the reference file and the name of a localization language. Unlike the other methods, `GetLocalizedPath` can be called a lot. During analysis, it is called for each file of the reference language times the number of localized languages which are being analysed. It is also called by various Alto Studio tools or GUI elements. If your projects handle large numbers of files, it is recommended to ensure that this method is optimized.

GetLocalizedFiles

This method must return the list of the paths of all localized files in a specific language in the project. Note that this may also be files that are present in a localized folder, have the right prefix and/or suffix, but do not have a corresponding reference file. This function is indeed used by the Comparison feature to detect extra files. If you do not intend to look for extra files, you can simply return an empty list.

WriteSpecific, ReadSpecific

Each importer plug-in has a few built-in controls by default: the first one is to browse to the project, the second one is the list of languages available in the project and the last one is a combo box to select the reference language. For example, the CSV Importer plug-in depicted above does not have any other parameters so that is all that is displayed and saved.

However, if more parameters are required, then Alto Studio needs to be able to save them in the project file (.alto) and read them again. This is achieved by using the `WriteSpecific` and `ReadSpecific` methods. These methods are called when an Alto Studio project is saved and loaded respectively.

Alto Studio saves its project files in XML format therefore the `WriteSpecific` method is passed a reference to a `XmlTextWriter` object and the `ReadSpecific` method is passed a reference to the root node (of type `XmlNode`). A very simple example of implementation would look like this:

```
void WriteSpecific(ref XmlTextWriter writer)
{
    writer.WriteStartElement("MyPlugInSpecific");

    writer.WriteStartElement("MaxNumberOfFiles");
    writer.WriteAttributeString("value", m_maxNumberOfFiles.ToString());
    writer.WriteEndElement();

    writer.WriteEndElement();
}

void ReadSpecific(ref XmlNode rootNode)
{
    XmlNode myPlugInNode = rootNode.SelectSingleNode(".//MyPlugInSpecific");

    XmlNode infoNode = myPlugInNode.SelectSingleNode(".//MaxNumberOfFiles");
    m_maxNumberOfFiles = Convert.ToUInt32(infoNode.Attributes["value"].Value);
}
```

Exporter plug-ins

This section describes the methods that must be implemented to create an exporter plug-in and comply with the `IExporterPlugIn` interface. The XML Exporter is an example of a very simple exporter plug-in. Examining its source code is a great starting point to learn how to build exporter plug-ins.

```
public interface IExporterPlugIn : IPlugIn
{
    string ProjectFilterString { get; }
    bool IsValidProjectFile(string projectPath);

    bool Initialize(string projectPath);
    void Terminate();

    bool Export(string refLanguage,
         List<string> locLanguages,Dictionary<string,List<string>> files);

    void WriteSpecific(ref XmlTextWriter writer);
    void ReadSpecific(ref XmlNode parentNode);
}
```

As you can see, the `IExporterPlugIn` interface is very similar to the `IImporterPlugIn` interface.

It also contains a `ProjectFilterString` property as well as a `IsValidProjectFile` method. There is also an `Initialize` and a `Terminate` method, which can be used to create a file and close it respectively.

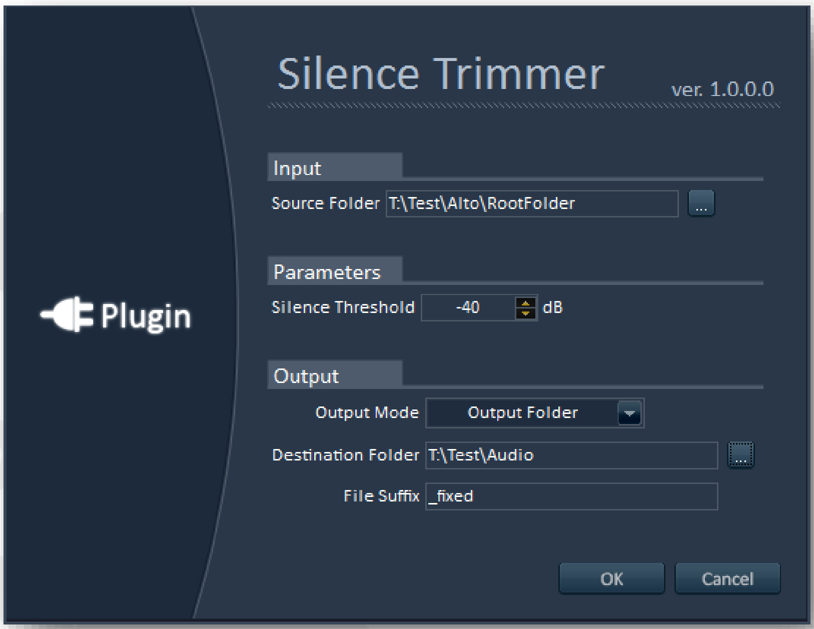
The difference comes from the `Export` method. This method is used by Alto Studio to pass all the dialogue files information to the exporter plug-in, so that it can write it or send it as needed.

The `Export` method has 3 parameters:

- The first one is a string containing the name of the reference language.
- The second parameter is a list of strings, each of them containing the name of one of the localized languages.
- Finally, the third parameter is a dictionary whose key is the path of an audio file in the reference language, and whose value is a list of string containing the paths of the corresponding audio files in the different localized languages.
  Note that the paths of the localized files follow the order of the languages in the list of the second parameter.

Tool plug-ins

This section describes the methods that must be implemented to create a tool plug-in and comply with the `IToolPlugIn` interface. The Silence Trimmer is an example of tool plug-in. It removes silence at the beginning and at the end of dialogue files. Examining its source code is a great starting point to learn how to build tool plug-ins. In addition, the Silence Trimmer plug-in takes advantage of the automatic GUI generation feature of Alto Studio by using C# attributes for its parameters (see next section).



Source Folder T:\Test\Alto\RootFolder
Silence Threshold -40 dB
Output Mode Output Folder
Destination Folder T:\Test\Audio
File Suffix _fixed

The `IToolPlugIn` interface is defined like this:

```
public interface IToolPlugIn : IPlugIn
{
    bool RequiresProject { get; }
    bool Initialize();
    void Terminate();
    bool Execute(AltoServerCommands commands, out bool bDialogueModified);
}
```

RequiresProject
This property describes if the tool plug-in requires an Alto project to be loaded to work. For example, The Silence Trimmer plug-in allows you to specify an input folder from which the files will be processed. It is therefore independent from the Alto project. If you start Alto and no project is loaded, you will still be able to access the Silence Trimmer tool. However, if this flag is set to true for a plug-in, the corresponding entry in the tool menu will be greyed until a project is loaded.

Initialize

This method is only called once, when the plug-in is loaded (when Alto Studio is started). Any consequential initialization or resources allocation should be done here.

Terminate

This method is only called once, when the plug-in is released (when Alto Studio is closed). Any resources used by the plug-in should be freed here.

Execute

This method does the actual work when the OK button is pressed in the tool's window. It receives a set of delegate functions (see the description of `AltoServerCommands` below) that allow the plug-in to use Alto Studio internal functions and to access the current project's data. The method must return true if the processing was successfully completed or false otherwise. In addition, `bDialogueModified` must be set to true by the plug-in if it changes the data of the dialogue files in the project and therefore the analysis must be redone. If it is set to true, Alto Studio will display a message to warn the user.

The `AltoServerCommands` class provides methods to get information about the current project and to trigger functions in Alto Studio. In future versions, we may also give access to the current report and analysis settings. Please let us know what your company needs and we will do our best to add it.

```
public class AltoServerCommands
{
    public StartPlayback StartPlayback;
    public StopPlayback StopPlayback;

    public GetSampleData GetSampleData;
    public UpdateSampleData UpdateSampleData;
    public WriteSampleData WriteSampleData;

    public GetReferenceLanguage GetReferenceLanguage;
    public GetLanguages GetLanguages;
    public GetReferenceFiles GetReferenceFiles;
    public GetLocalizedFile GetLocalizedFile;

    public GetMetaTags GetMetaTags;
    public SetMetaTags SetMetaTags;
    public GetScriptInformation GetScriptInformation;
    public SetScriptInformation SetScriptInformation;

    public CalculateLKFS CalculateLKFS;
    public CalculateLRA CalculateLRA
    public CalculatedBTP CalculatedBTP;
    public CalculateRMS CalculateRMS;
    public GetSpectrum GetSpectrum;
    public GetSpectralBands GetSpectralBands;
    public GetPitchEnvelope GetPitchEnvelope;
    public GetAmplitudeEnvelope GetAmplitudeEnvelope;
    public CorrectLKFS CorrectLKFS;
}
```

The definitions of these delegates are as follows:

```
delegate void PlaybackFinished();
delegate bool StartPlayback(string path,PlaybackFinished callback);
delegate void StopPlayback();

delegate float[] GetSampleData(string path,out UInt16 bitDepth, out float
sampleRate,out UInt16 channels);
delegate bool UpdateSampleData(string path,float[] data, UInt16 bitDepth, float
sampleRate, UInt16 channels);
delegate bool WriteSampleData(string path, float[] data, UInt16 bitDepth, float
sampleRate, UInt16 channels);

delegate string GetReferenceLanguage();
delegate List<string> GetLanguages();
delegate List<string> GetReferenceFiles();
delegate string GetLocalizedFile(string referencePath, string language);

delegate List<string> GetMetaTags(string path);
delegate bool SetMetaTags(string path, List<string> metaTags);
delegate bool GetScriptInformation(string path, out string text, out string
character, out string actor);
delegate bool SetScriptInformation(string path, string text, string character,
string actor);

public delegate float CalculateLKFS(float[] data, float sampleRate, uint channels,
ChannelMapping channelMapping, out float[] momentaryLoudness);
public delegate float CalculateLRA(float[] data, float sampleRate, uint channels,
ChannelMapping channelMapping, out float[] shortTermLoudness);
public delegate float CalculatedBTP(float[] data, float sampleRate, uint channels,
ChannelMapping channelMapping);
public delegate float CalculateRMS(float[] data, uint channels, ChannelMapping
channelMapping);
public delegate float[] GetSpectrum(float[] data);
public delegate float[] GetSpectralBands(float[] data,float sampleRate, out float[]
centerFreq,out float[] bandwidth);
public delegate float[] GetPitchEnvelope(float[] data,float sampleRate, float
blockDuration, float minPitch, float maxPitch, float noiseThreshold);
public delegate float[] GetAmplitudeEnvelope(float[] data,float sampleRate);
public delegate void CorrectLKFS(ref float[] data,float samplerate,uint channels,float
targetLKFS, out bool clipping);
```

Playback commands

StartPlayback
This command uses Alto's audio engine to start the playback of a file. The plug-in can receive a notification when the playback is finished if a callback function of type `PlaybackFinished` was passed as an argument.

StopPlayback
This command stops the playback of a file started with `StartPlayback`.

Audio files commands

GetSampleData
This command gets the sample data of an audio file (as well as its number of channels and its sample rate).

UpdateSampleData
This command updates the sample data of an audio file (as well as number of channels and sample rate).

WriteSampleData
This command writes a new audio file given its sample data, bit depth, number of channels and sample rate.

Languages and file paths commands

GetReferenceLanguage
This command gets the name of the reference language of the project.

GetLanguages
This command gets the names of all the languages in the project (including the reference language).

GetReferenceFiles
This command gets all the paths of the reference files in the project.

GetLocalizedFile
This command gets the path of a localized file given the path of a reference file and a language.

File information commands

These commands allow for the acquisition and update of information associated with audio files. Any update is done at the project level (in case your plug-in needs to process this type of information). However, it is not saved in the original files or the project itself.

GetMetaTags
This command gets the meta tags associated with an audio file.

SetMetaTags
This command updates the meta tags associated with an audio file.

GetScriptInformation
This command gets the script data (text of the dialog, character's name, voice actor) that is associated with an audio file.

SetScriptInformation
This command updates the script data (text of the dialog, character's name, voice actor) that is associated with an audio file.

Analysis and correction commands

Some of these commands require information about the channels mapping, which is passed as a value from the enum below:

```
public enum ChannelMapping
{
    DTS,        // L R Ls Rs C LFE
    ITU,        // L R C LFE LS Rs
    Film,       // L C R Ls Rs LFE
}
```

CalculateLKFS
This command calculates the integrated - or program - loudness (LKFS) following the EBU R-128 specifications. It also returns the momentary loudness. If the file is shorter than 400 ms it is extended to 400 ms and the original signal is looped. This ensures that we can calculate the loudness even for short signals while following the specifications.

CalculateLRA
This command calculates the loudness range (LRA) following the EBU R-128 specifications. It also returns the short-term loudness. If the file is shorter than 3 seconds it is extended to 3

seconds and the original signal is looped. This ensures that we can calculate the loudness range even for short signals while following the specifications.

CalculatedBTP
This command calculates the true peak value (dBTP) following the EBU R-128 specifications.

CalculateRMS
This command calculates the Root Mean Square (RMS) over the whole signal.

GetSpectrum
This command calculates the spectrum of a mono signal and returns the magnitudes. It essentially performs a Fast Fourier Transform (FFT). Therefore, an error will occur if the size of the signal passed is not a power of two (you can pass the signal with zeroes if needed).

GetSpectralBands
This command calculates the loudness in equidistant spectral bands (ERB) of a mono signal. The spectrum is divided in 31 spectral bands.

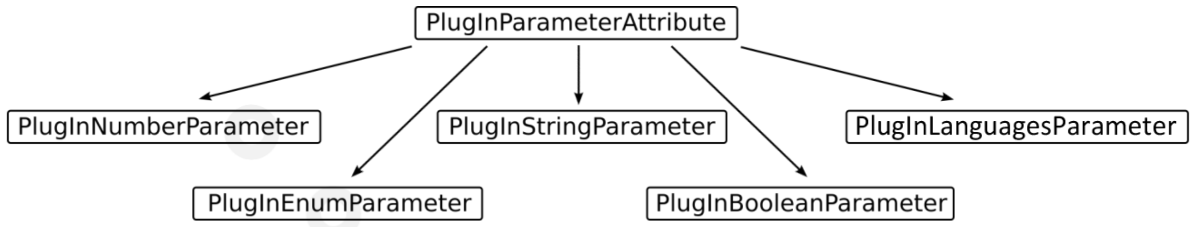
GetPitchEnvelope
This command extracts the pitch envelope of a mono signal. The signal is divided in blocks of 20 ms and an array is returned with a pitch estimation for each block. If the pitch could not be estimated or the signal was silent in a block, a value of 0.0f is returned for that block.

GetAmplitudeEnvelope;
This command extracts the amplitude envelope of a mono signal. It returns an array of the same size than the original signal containing the envelope.

CorrectLKFS;
This command calculates the current loudness of a signal and corrects it if the value is different from the specified LKFS target. A flag is also returned that indicates if clipping occurred during the correction.

Exposing plug-in parameters

Alto Studio will interpret any property marked with an attribute derived from `PlugInParameterAttribute` as a plug-in parameter. If the plug-in does not provide a graphical user interface, Alto Studio will use these parameters to create a settings window for the plug-in. Currently, five types of plug-in parameter attributes are available, depending on the type of the parameter.



PlugInNumberParameter

This attribute is used to specify any parameter whose value is a number (it can be integer or floating point).

```
public PlugInNumberParameter(string name, string description, string group, UInt16
line, string unit, float def, float min, float max,bool bInteger)
```

The `name`, `description` and `unit` are self-explanatory. They will be used by the tool to display the parameter. Most importantly, the default (`def`), minimum (`min`) and maximum (`max`) values of the parameter are to be specified. A boolean (`bInteger`) specifies if the parameter is an integer or not. The `group` name is used by Alto Studio to create headers under which parameters from the same group will be located. The `line` number is used to force several control on a same line.

Here is for example the parameter definition for a fade-in duration (floating point value):

```
private float m_fadeIn = 0.10f;

[PlugInNumberParameter("Fade in", "Duration of the fade in", "Audio Parameters", 1,
"s", 0.1f, 0.0f, 100000.0f, false)]
public float FadeIn
{
	set { m_fadeIn = value; }
	get { return m_fadeIn; }
}
```

If a GUI is automatically generated for your plug-in, a *NumericUpDown* control will be created for this type of parameter.

PlugInEnumParameter

This attribute is used to specify a parameter which can take discrete values among a set.

```
public PlugInEnumParameter( string name, string description, string group, UInt16
line, string unit, Int16 def, string[] valueNames )
```

As for the number parameter attribute, the `name`, `description` and `unit` are required, the `group` name and `line` number as well. `valueNames` is an array of strings that contains all the possible values of the parameter in textual format. `def` is the index of the default value in that array.

Here is the parameter definition for a fade-in curve:

```
public enum LevelCurve
{
	Linear,
	 Exponential,
	 HalfSine,
}


private LevelCurve m_curveIn = LevelCurve.Linear;

[PlugInEnumParameter("Fade in curve", "Level curve to use to do the fade in.",
"Audio Parameters", 2, "", 1, new string[] { "Linear","Exponential","HalfSine" })]
public LevelCurve FadeInCurve
{
	set { m_curveIn = value; }
	get { return m_curveIn; }
}
```

If a GUI is automatically generated for your plug-in, a *ComboBox* control will be created for this type of parameter.

PlugInStringParameter

This attribute is used to specify a parameter whose value is a string.

```
public PlugInStringParameter(string name, string description, string group, UInt16
line, string def,StringType stringType)
```

In addition to the usual `name`, `description`, `group` and `line` of the parameter - no `unit` needed this time – the type of the string (`stringType`) as well as its default value (`def`) must be specified.

The string type is defined like this:

```
public enum StringType
    {
        String,
        FileReadPath,
        FileWritePath,
        FolderPath,
    }
```

`String` is a regular string: it can be the name of an object, a comment, a copyright etc… `FileReadPath`, `FileWritePath`, and `FolderPath` all correspond to a path. If a GUI is to be automatically generated for the plug-in, these string parameters will have an extra browse button ("…") in addition to the regular TextBox, therefore allowing the selection of a file or a folder. The difference between `FileReadPath` and `FileWritePath` is that the first one requires the selection of an existing path (similar to when you want to read a file) whereas the second one accepts a new path (similar to when you want to write a file).
Here is an example of usage of this attribute:

```
private string m_file = "";

[PlugInStringParameter("File", "File which will receive all the stripped
metadata.", "Export",1, "", StringType.FileWritePath)]
public string File
{
      set { m_file = value; }
      get { return m_file; }
}
```

PlugInBooleanParameter

This attribute corresponds to a boolean value and is represented by a checkbox when a GUI is automatically created by Alto Studio.

```
public PlugInBooleanParameter( string name, string description, string group,
UInt16 line, bool def )
```

Its only specific parameter is the default boolean value. Here is an example of definition:

```
bool m_bOpenEditor = false;

[PlugInBooleanParameter("Open Editor", "Open the project in editor after export",
"Export", 2, false)]
public bool OpenEditor
{
	set { m_bOpenEditor = value; }
	get { return m_bOpenEditor; }
}
```

PlugInLanguagesParameter

This attribute is used to specify a parameter whose value is a list of strings. In the GUI, it is displayed as a checked listbox populated with the names of the languages in the projects. The user can include or exclude languages by clicking on the checkboxes. The list of string will then contain the names of the languages selected.

```
public PlugInLanguagesParameter(string name, string description, string group, UInt16
line, bool bIncludeReference)
```

In addition to the usual `name`, `description`, `group` and `line` of the parameter, the `bIncludeReference` boolean determines if the reference language must also be included in the languages displayed by the listbox.

Here is an example of definition:

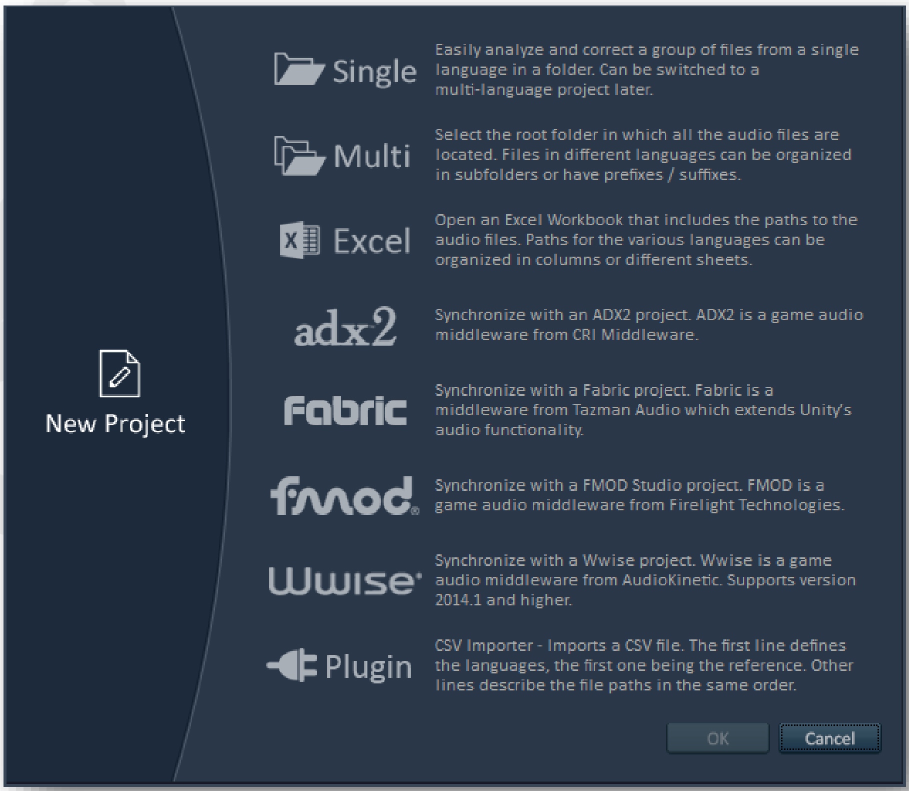
```
List<string> m_selectedLanguages = new List<string>();

[PlugInLanguagesParameter("Languages", "Select the languages you want to check",
"Languages", 0, false)]
public List<string> SelectedLanguages
{
	set { m_selectedLanguages = value; }
	get { return m_selectedLanguages; }
}
```

Testing a plug-in

To test your plug-in in Alto Studio, simply copy the plug-in DLL in the folder where the Alto Studio executable is located. If it has been marked correctly with the `AltoPluginAttribute`, it will be automatically recognized and loaded when Alto Studio starts.

If you didn't specify an editor for it but marked your parameters appropriately, a GUI will be automatically generated for it when you call it.

Importer plug-ins can be accessed through the "New project" command. They will appear in the list of project types you can create. The name of the plug-in, followed by its description appears in the window. If you select it and press OK the window of the importer plug-in will be displayed.



Easily analyze and correct a group of files from a single language in a folder. Can be switched to a multi-language project later.

Select the root folder in which all the audio files are located. Files in different languages can be organized in subfolders or have prefixes / suffixes.

Open an Excel Workbook that includes the paths to the audio files. Paths for the various languages can be organized in columns or different sheets.

Synchronize with an ADX2 project. ADX2 is a game audio middleware from CRI Middleware.

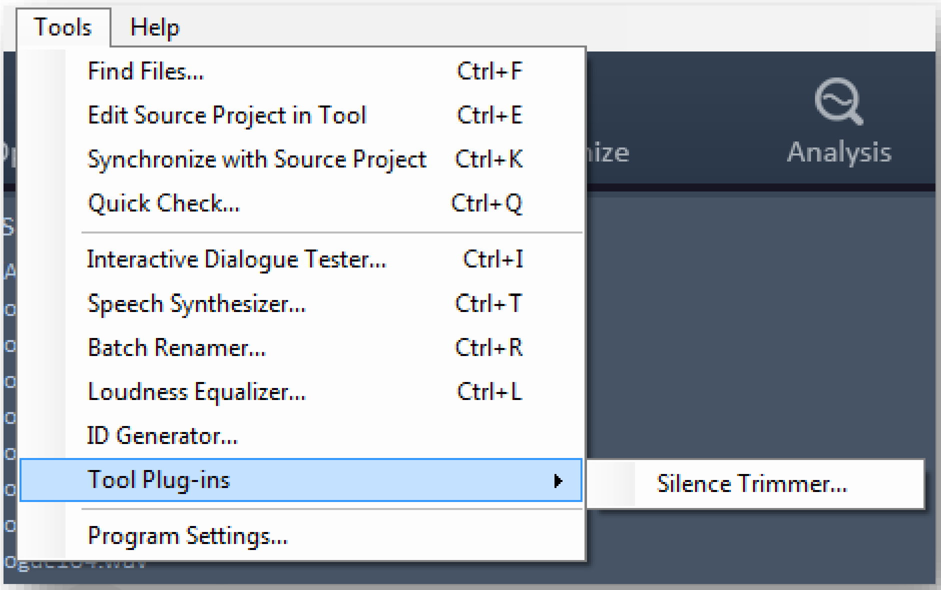
Synchronize with a Fabric project. Fabric is a middleware from Tazman Audio which extends Unity's audio functionality.

Synchronize with a FMOD Studio project. FMOD is a game audio middleware from Firelight Technologies.

Synchronize with a Wwise project. Wwise is a game audio middleware from AudioKinetic. Supports version 2014.1 and higher.

CSV Importer - Imports a CSV file. The first line defines the languages, the first one being the reference. Other lines describe the file paths in the same order.

Similarly, all the exporter plug-ins and the tool plug-ins found in Alto Studio's root folder are added to the "Export Project…" window and to the "Tools" menu respectively. When the tool command in the "Tools plug-ins" submenu is selected, the user interface of the plug-in (either automatically generated or customized) is displayed.


Tools
Help
Find Files... Ctrl+F
Edit Source Project in Tool Ctrl+E
Synchronize with Source Project Ctrl+K
Quick Check... Ctrl+Q
Interactive Dialogue Tester... Ctrl+I
Speech Synthesizer... Ctrl+T
Batch Renamer... Ctrl+R
Loudness Equalizer... Ctrl+L
ID Generator...
Tool Plug-ins
Program Settings...
Silence Trimmer...
Analysis

Please note that all plug-ins are detected and loaded by the Alto Studio plug-in manager when the software is started. A plug-in cannot be hot-swapped, or added to Alto Studio while it is already running. You will have to restart Alto Studio for the new plug-in to become available.