Translation Memory SDK

Contents

[Translation Memory SDK 1](#_Toc461619994)

[Versions 2](#_Toc461619995)

[Overview 2](#_Toc461619996)

[The workflow for creating and distributing a plugin 2](#_Toc461619997)

[Translation Memory framework in memoQ 3](#_Toc461619998)

[Translation memory plugins 3](#_Toc461619999)

[Translation memory base classes and interfaces 3](#_Toc461620000)

[PluginDirectorBase 3](#_Toc461620001)

[IModule and IModuleEx interfaces 3](#_Toc461620002)

[IPluginDirector interface 3](#_Toc461620003)

[EngineBase 3](#_Toc461620004)

[IEngine 3](#_Toc461620005)

[SessionBase 3](#_Toc461620006)

[ISession 3](#_Toc461620007)

[Translation memory SDK sample application 4](#_Toc461620008)

[Implementation steps of a TM plugin 4](#_Toc461620009)

[Create the new class library 4](#_Toc461620010)

[The plugin director 4](#_Toc461620011)

[PluginDirectorBase members 5](#_Toc461620012)

[The engine component 7](#_Toc461620013)

[The session component 8](#_Toc461620014)

[The plugin options 15](#_Toc461620015)

[The configuration dialog 15](#_Toc461620016)

[Localization 16](#_Toc461620017)

[Implementation checklist 16](#_Toc461620018)

[Testing the sample plugin 16](#_Toc461620019)

[Testing the new plugins 21](#_Toc461620020)

[Testing in the sample application 21](#_Toc461620021)

[Testing in memoQ client 21](#_Toc461620022)

[Checklist to update a plugin for memoQ 8.0 22](#_Toc461620023)

# Versions

|  |  |  |  |
| --- | --- | --- | --- |
| Date | Version | Who | Change |
| July 15, 2015 | 1.0 | SZB | Initial version |
| Sept 17, 2015 | 1.01 | SZB | Concordance support added |
| Sept 14, 2016 | 2 | DÁ | Changes related to memoQ 8.0 |
| Mar 07, 2017 | 2.01 | NT | TargetSideSearch added |

# Overview

memoQ enables customers and 3rd party developers to be able to create translation memory plugins for **memoQ client**. This document describes the fundamentals of the translation memory framework, and provides a step-by-step guide for creating a new plugin.

Creating plugins using the SDK is supported from memoQ version 8.0.

The plugin framework since memoQ 8.0 is not compatible with plugins in previous versions of memoQ. This means that plugins developed for previous versions of memoQ will not work in memoQ 8.0 and newer versions. The plugins have to be adjusted and recompiled to target memoQ 8.0.

These plugins has to be developed for **.NET Framework 4.6.1** in **C# language**.

The TM SDK has a Visual Studio solution, which can be opened by Visual Studio 2013 or higher.

# The workflow for creating and distributing a plugin

Assuming CompanyA wants to create a new TM plugin, the steps for the recommended workflow are the following:

* CompanyA develops a new TM plugin using the TM SDK using C# language.
* CompanyA sends the source code of the plugin to Kilgray.
* Kilgray reviews the code and tests the functionality of the plugin. CompanyA performs fixes based on the review, if required.
* Kilgray compiles the source code of the TM plugin, signs the resulting DLL with its private key (unsigned plugins are not loaded by memoQ!), makes it part of the memoQ client installer. As a result, the TM plugin is distributed with the memoQ client installer from this point.
* The source code of the TM plugin becomes part of the memoQ code base at Kilgray.
* Information about bugs reported by customers are forwarded to CompanyA by Kilgray. CompanyA is responsible for fixings the bugs, bug fixes are reviewed by Kilgray.

This workflow is required to ensure that plugins meet the quality requirements of memoQ and do not jeopardize the stability of the entire product.

If your company does not want to distribute the TM plugin (only wants to use it internally), it is to be handled based on a different workflow. Please contact to Kilgray to agree on the details.

# Translation Memory framework in memoQ

The translation memory framework provides the possibility to use external translation memory services from memoQ. Kilgray delivers several built-in translation memory plugins with the memoQ client (MyMemory TM, TAUS TM etc.), but companies also have the possibility to write brand new translation memory plugins.

## Translation memory plugins

Every translation memory plugin should be a standalone .NET DLL, which has the following references:

* MemoQ.Addins.Common.dll
* MemoQ.TMInterfaces.dll

Please note that these are the sole memoQ assemblies that should be referenced.

This has changes in memoQ 8.0, please make sure to update the references!

These libraries contain all of the necessary classes for the plugins. **The usage of any other external libraries is not allowed in translation memory plugins.** If it is absolutely necessary consult it with the Kilgray.

## Translation memory base classes and interfaces

The memoQ application and the plugins can communicate with the help of a few base classes and interfaces. Every translation memory plugin should use the following abstract classes:

* PluginDirectorBase
* EngineBase
* SessionBase

### PluginDirectorBase

This is the base class of plugin directors. Each plugin director represents an entry point of a plugin. This class implements *IModule*, *IModuleEx* and *IPluginDirector* interfaces.

### IModule and IModuleEx interfaces

memoQ manages the MT, TM and TB plugins as individual modules. These interfaces provides some general functions for memoQ to be able to initialize, cleanup the modules and to be able to get general information about the modules.

### IPluginDirector interface

This is memoQ’s starting point to the plugin. memoQ instantiates one instance for each plugin at application startup, and this instance is used after this point when memoQ has to communicate with the plugin.

### EngineBase

This is the base class of plugin engines. This class implements *IEngine* interface.

### IEngine

An object implementing the *IEngine* interface is requested by memoQ for a particular language combination at project open with the help of the plugin director.

### SessionBase

This is the base class of plugin sessions. This class implements *ISession* interface.

### ISession

memoQ calls the object implementing this interface to perform the translation. A new session object is created on lookup and concordance requests. *ISession* objects are always created by *IEngine* objects.

# Translation memory SDK sample application

Kilgray has implemented a small application for the developers who would like to implement new translation memory plugins. Developers will be able to test their translation memory plugins with the help of this application.

You can see four projects if you open the *TM\_SDK* solution from the SDK:

* DummyTMInterface
* DummyTMPlugin
* DummyTMService
* TestClient

The sample application is implemented inside the TestClient project. The test client loads the plugins on startup, in this case the DummyTMPlugin, which contains the implementation of a sample translation memory plugin. The DummyTMService project contains a simple web service, which is used by the sample plugin. The DummyTMInterface project contains the interface of this service.

In the next section we’re going to see how to implement a brand new translation memory plugin.

# Implementation steps of a TM plugin

## Create the new class library

As mentioned above all plugins should be implemented as standalone libraries. To achieve this create a new Visual Studio project and the type of the project should be Class Library. Being done with the project creation mark the assembly with the *ModuleAttribute* attribute. Open the project’s *AssemblyInfo.cs* file, and insert the following line after the last line (change the name of the module and the plugin director class):

[assembly: Module(ModuleName = "Dummy TM", ClassName = "DummyTMPlugin.DummyTMPluginDirector")]

This attribute can be found in the *MemoQ.Addins.Common.Framework* namespace.

memoQ will check this attribute when it loads the translation memory assemblies. The module name field should be the name of the translation memory plugin and the class name should be the name of the plugin director class.

Now you have to set up the memoQ library references. The necessary DLLs are under the *References* folder.

After that, you have to set the project’s build path. Open the project properties and go to the ‘Build’ tab. Make sure that the ‘Configuration’ property is set to Debug, then set the ‘Output path’ to “..\TestClient\bin\Debug\plugins\” (without the quotes). The test client loads the plugins from that directory.

## The plugin director

This component is the entry point of the plugin. First of all you have to create a new **public** class inside the project. The naming convention is: **<plugin\_name>PluginDirector.cs**

This class should derive from the *PluginDirectorBase* class which can be found in the *MemoQ.TMInterfaces* namespace.

### PluginDirectorBase members

The class members:

* *environment* variable: the memoQ's application environment; e.g., to provide UI language settings etc. to the plugin.
* *IsActivated* property: you can tell here whether the plugin is activated or not.
* *Initialize* function: you can implement the plugin’s initialization logic here.
* *Cleanup* function: you can implement the plugin’s cleanup logic here.
* *PluginConfigured* property: you should return true here if the plugin has been set up correctly (e.g. account details, supported languages)
* *PluginEnabled* property: this property tells whether the plugin is enabled or disabled. The plugin should store this information among its own settings.
* *Environment* property: sets memoQ's application environment; e.g., to provide UI language settings etc. to the plugin.
* *IsLanguagePairSupported* function: you have to give back whether the plugin supports the given language pair or not. Do not call any service here, give back the result based on the saved plugin settings.
* *DisplayIcon* property: you should return here the icon of the TM plugin. This image will be displayed on the user interface where memoQ lists the available plugins.
* *FriendlyName* property: you should get back the plugin’s human readable name. It will be displayed on the user interface where memoQ lists the available plugins.
* *PluginID* property: you have to get back the plugin’s identifier here.
* *CopyrightText* property: you should return the plugin’s copyright information here; it will be displayed on the user interface where memoQ lists the available plugins.
* *ShowOptionsForm* function: memoQ calls this function when the user would like to configure your translation memory plugin. In this function you should display the configuration dialog of the plugin.
* *CreateEngine* function: this function has two input parameters, the source and the target language. Based on these languages you should instantiate and give back a translation memory engine here.
* *SupportsSegmentLookup* property: you can tell here whether the plugin supports segment lookup or not.
* *SupportsConcordance* property: you can tell here whether the plugin supports concordance or not.

The class is the following:

/// <summary>

/// Abstract base class of the TM plugin directors.

/// </summary>

public abstract class PluginDirectorBase : IModule, IModuleEx, IPluginDirector

{

/// <summary>

/// The memoQ's application environment; e.g., to provide UI language settings

/// etc. to the plugin.

/// </summary>

protected IEnvironment environment;

/// <summary>

/// Gets whether the plugin is activated.

/// </summary>

public abstract bool IsActivated { get; }

/// <summary>

/// Initializes the plugin.

/// </summary>

public abstract void Initialize(IModuleEnvironment env);

/// <summary>

/// Cleans up the resources used by the plugin if needed.

/// </summary>

public abstract void Cleanup();

/// <summary>

/// Gets whether the plugin is configured.

/// </summary>

public abstract bool PluginConfigured { get; }

/// <summary>

/// Gets or sets whether the plugin is enabled.

/// </summary>

public abstract bool PluginEnabled { get; set; }

/// <summary>

/// Sets memoQ's application environment; e.g., to provide UI language

/// settings etc. to the plugin.

/// </summary>

public virtual IEnvironment Environment

{

set { environment = value; }

}

/// <summary>

/// Gets if the plugin supports the provided language combination. The strings

/// provided are memoQ languages codes.

/// </summary>

public abstract bool IsLanguagePairSupported(string srcLangName,

string trgLangName);

/// <summary>

/// Gets a 48x48 display icon to show in memoQ's Tools / Options. Black is the

/// transparent color.

/// </summary>

public abstract Image DisplayIcon { get; }

/// <summary>

/// Gets the friendly name to show in memoQ's Tools / Options.

/// </summary>

public abstract string FriendlyName { get; }

/// <summary>

/// Gets the plugin's non-localized name.

/// </summary>

public abstract string PluginID { get; }

/// <summary>

/// Gets the copyright text to show in memoQ's Tools / Options.

/// </summary>

public abstract string CopyrightText { get; }

/// <summary>

/// Shows the plugin's options/about form

/// </summary>

public abstract void ShowOptionsForm(Form parentForm);

/// <summary>

/// Create an TM engine for the supplied language pair.

/// </summary>

public abstract IEngine CreateEngine(string srcLangName, string trgLangName);

/// <summary>

/// Indicates whether the plugin supports the segment lookup.

/// </summary>

public abstract bool SupportsSegmentLookup { get; }

/// <summary>

/// Indicates whether the plugin supports concordance.

/// </summary>

public virtual bool SupportsConcordance { get; }

}

## The engine component

The memoQ calls the plugin director’s *CreateEngine* function to get back a translation memory engine for a language pair. memoQ uses this engine to perform the lookups and concordance. The engine component should derive from the *EngineBase* class which can be found in the *MemoQ.TMInterfaces* namespace. The naming convention is: **<plugin\_name>Engine.cs**. The base class has the following members:

* *SmallIcon* property: memoQ displays this icon under lookup results when a TM hit is selected from this plugin.
* *CreateSession* function: memoQ calls this function to be able to perform the lookup or concordance. You should instantiate and return a session object here. This session will not be used in multi-threaded way.
* *SetProperty* function: this function sets an engine-specific property, for example subject matter area.
* *Dispose* function: in this function you should release the allocated resources.

The class is the following:

/// <summary>

/// Abstract base class of the TM plugin engines.

/// </summary>

public abstract class EngineBase : IEngine

{

/// <summary>

/// Gets a small icon to be displayed under translation results when a

/// TMPlugin hit is selected from this plugin.

/// </summary>

public virtual Image SmallIcon { get { return null; } }

/// <summary>

/// Creates a session for translating segments. Session will not be used in

/// a multi-threaded way.

/// </summary>

public abstract ISession CreateSession();

/// <summary>

/// Sets an engine-specific custom property, e.g., subject matter area.

/// </summary>

public virtual void SetProperty(string name, string value)

{ }

/// <summary>

/// Disposes the resources used by the engine if needed.

/// </summary>

public abstract void Dispose();

}

## The session component

This component is responsible for lookup and concordance. It should derive from the *SessionBase* class. The naming convention is: **<plugin\_name>Session.cs**. The base class has the following members:

* *Lookup* function: the function should return an array of *TMPluginHit* objects. This function has four parameters:
  + *Segment* segment: The segment to be translated.
  + *Segment* precedingSegment: The preceding segment.
  + *Segment* followingSegment: The following segment.
  + *int* matchThreshold: The minimum match rate of hits.
* *Concordance* function: the function should return a *TMPluginConcordanceResult* object. This function has two parameters:
  + *string[]* texts: The array of texts sent for concordance.
  + *TMPluginConcordanceSettings* settings: The settings of the concordance operation.
* *Dispose* function: in this function you should release the allocated resources.

The class is the following:

/// <summary>

/// Abstract base class of the TM plugin sessions.

/// </summary>

public abstract class SessionBase : ISession

{

/// <summary>

/// Performs a lookup.

/// </summary>>

public abstract TMPluginHit[] Lookup(Segment segment,

Segment precedingSegment, Segment

followingSegment, int matchThreshold);

/// <summary>

/// Disposes the resources used by the engine if needed.

/// </summary>

public abstract void Dispose();

/// <summary>

/// Performs a concordance.

/// </summary>>

public virtual TMPluginConcordanceResult Concordance(string[] texts,

TMPluginConcordanceSettings settings);

}

The *Lookup* function should work with *Segment* objects. You can use the *PlainText* property of the to get the content of the actual segment as a string, or work with any of the public methods available in this class.

If any exception occurred during the translation you have to throw a *TMPluginException* object. You have to use the *TMPluginException* class to wrap the original exception.

The *TMPluginHit* class is the following:

public class TMPluginHit : IFuzzyHit, ITextPairProvider

{

public TMPluginHit();

public TMPluginHit(TMPluginEntry tmPluginEntry, int matchRate,

int freckleFlags = 0);

public TMPluginHit(TMPluginEntry tmPluginEntry, int matchRate,

int freckleFlags, int fragmentStart, int fragmentLength);

/// <summary>

/// True if a match is below the lookup match threshold, and retained only b/c

/// it's a fragment hit

/// </summary>

public bool BelowFuzzyThreshold { get; }

/// <summary>

/// Represents the hit in a human readable form

/// </summary>

public string Description { get; }

/// <summary>

/// True if result is a fragment hit

/// </summary>

public bool FragmentHit { get; }

/// <summary>

/// For fragment hits, the length of the fragment range the lookup segment's

/// text

/// Warning! This a plain text position, tags must be taken into consideration

/// when used as Segment index

/// </summary>

public int FragmentLength { get; }

/// <summary>

/// For fragment hits, the start of the fragment range in the lookup segment's

/// text

/// Warning! This a plain text position, tags must be taken into consideration

/// when used as Segment index

/// </summary>

public int FragmentStart { get; }

/// <summary>

/// Returns the plugin entry’s source segment

/// </summary>

public Segment GetText1 { get; }

/// <summary>

/// Returns the plugin entry’s target segment

/// </summary>

public Segment GetText2 { get; }

/// <summary>

/// The last time the entry producing the hit was modified

/// </summary>

public DateTime LastModified { get; }

/// <summary>

/// The hit's match rate: an integer value between 0 and 100

/// </summary>

public int MatchRate { get; set; }

/// <summary>

/// Minor differences detailed for 95-99% matches; or null if this info is not

/// available (legacy hits)

/// </summary>

public NearExactFreckles NearExactFreckles { get; }

/// <summary>

/// The unique identifier of the plugin

/// </summary>

public string PluginID { get; set; }

/// <summary>

/// The name of the plugin

/// </summary>

public string PluginName { get; set; }

/// <summary>

/// Role of the hit

/// </summary>

public Roles Role { get; }

/// <summary>

/// The hit’s source segment

/// </summary>

public Segment SourceSegment { get; }

/// <summary>

/// The hit’s target segment

/// </summary>

public Segment TargetSegment { get; }

/// <summary>

/// Name of user that last modified the entry producing the hit

/// </summary>

public string UserName { get; }

/// <summary>

/// Change fragment length (needed when trimming spaces etc. during fragment

/// assembly)

/// </summary>

public void ChangeFragmentLength(int newFragmentLength);

/// <summary>

/// Create a deep-copy clone of object

/// </summary>

public IFuzzyHit Clone();

/// <summary>

/// Returns the entry producing the hit

/// </summary>

public TMPluginEntry GetEntry();

/// <summary>

/// Represents the plugin hit in a human readable form

/// </summary>

public override string ToString();

}

The *TMPluginEntry* class is the following:

public class TMPluginEntry : ITMOrCorpusEntry

{

public TMPluginEntry();

public TMPluginEntry(Segment source, Segment target);

public TMPluginEntry(Segment source, Segment target,

Segment pre, Segment foll);

public TMPluginEntry(Segment sourceSeg, Segment targetSeg,

DateTime modified, string username, string subject, string domain,

string clientID, string projectID);

public TMPluginEntry(Segment sourceSeg, Segment targetSeg,

DateTime modified, string username, DateTime creationDate,

string creatingUser, string subject, string domain, string clientID,

string projectID, string documentName, Segment precedingSeg,

Segment followingSeg);

/// <summary>

/// The client’s unique identifier

/// </summary>

public string ClientID { get; set; }

/// <summary>

/// Name of user that created the entry

/// </summary>

public string CreatingUser { get; set; }

/// <summary>

/// The time the entry was created

/// </summary>

public DateTime CreationDate { get; set; }

/// <summary>

/// Represents the entry in a human readable form

/// </summary>

public string Description { get; }

/// <summary>

/// The document that contains the entry

/// </summary>

public string Document { get; set; }

/// <summary>

/// The domain of the entry

/// </summary>

public string Domain { get; set; }

/// <summary>

/// The following segment of the source segment

/// </summary>

public Segment FollowingSegment { get; }

/// <summary>

/// The last time the entry was modified

/// </summary>

public DateTime Modified { get; set; }

/// <summary>

/// The preceding segment of the source segment

/// </summary>

public Segment PrecedingSegment { get; }

/// <summary>

/// The project’s unique identifier

/// </summary>

public string ProjectID { get; set; }

/// <summary>

/// The source segment

/// </summary>

public Segment SourceSegment { get; }

/// <summary>

/// The subject of the entry

/// </summary>

public string Subject { get; set; }

/// <summary>

/// The target segment

/// </summary>

public Segment TargetSegment { get; }

/// <summary>

/// Name of user that last modified the entry

/// </summary>

public string UserName { get; set; }

/// <summary>

/// Create a deep-copy clone of object

/// </summary>

public TMPluginEntry Clone();

/// <summary>

/// Represents the entry in a human readable form

/// </summary>

public override string ToString();

/// <summary>

/// Represents the entry in a human readable form

/// </summary>

public string ToString(string separator);

}

The *TMPluginConcordanceResult* class is the following:

public class TMPluginConcordanceResult

{

/// <summary>

/// The result items of the concordance operation

/// </summary>

public readonly TMPluginConcordanceResult.TMPluginConcordanceItem[] Results;

/// <summary>

/// The total number of results. (Not the size of the Results array!

/// For example if the limit was set to 5, but the plugin has found 10 items,

/// this should be set to 10)

/// </summary>

public readonly int TotalResultCount;

/// <summary>

/// The constructor of the class.

/// </summary>

public TMPluginConcordanceResult(int totalResultCount,

TMPluginConcordanceResult.TMPluginConcordanceItem[] results);

/// <summary>

/// This class represents a single item found by the concordance operation.

/// </summary>

public class TMPluginConcordanceItem

{

public readonly TMPluginConcordanceResult.TMPluginConcordanceRangeInfo[]

ConcordanceTextRanges;

public readonly TMPluginEntry Entry;

public TMPluginConcordanceItem(

TMPluginConcordanceResult.TMPluginConcordanceRangeInfo[]

concordanceTextRanges,

TMPluginEntry entry);

}

/// <summary>

/// This class represents a range information for a single concordance item.

/// </summary>

public class TMPluginConcordanceRangeInfo

{

/// <summary>

/// The length of the text sent for concordance in the source segment

/// of a TM entry

/// </summary>

public readonly int Length;

/// <summary>

/// The starting index of the text sent for concordance in the source

/// segment of a TM entry

/// </summary>

public readonly int Start;

public TMPluginConcordanceRangeInfo(int start, int length);

}

}

The *TMPluginConcordanceSettings* class is the following:

/// <summary>

/// This class represents the settings of the concordance operation.

/// </summary>

public class TMPluginConcordanceSettings

{

/// <summary>

/// Case sensitivity in concordance is enabled or not.

/// </summary>

public readonly bool CaseSensitive;

/// <summary>

/// The maximum number of items returned by the concordance operation.

/// </summary>

public readonly int Limit;

/// <summary>

/// Numeric equivalence in concordance is enabled or not.

/// </summary>

public readonly bool NumericEquivalence;

/// <summary>

/// The sort options for the concordance operation.

/// </summary>

public readonly TMPluginConcordanceSettings.SortInfo SortOptions;

/// <summary>

/// A list of texts used to filter the results of concordance.

/// </summary>

public readonly List<string> TargetFilterStrings;

/// <summary>

/// Whether the search is done on the target side.

/// </summary>

public readonly bool TargetSideSearch;

public TMPluginConcordanceSettings(

TMPluginConcordanceSettings.SortInfo sortOptions, bool caseSensitive,

bool numericEquivalence, int limit, List<string> targetFilterStrings,

bool targetSideSearch);

/// <summary>

/// Primary sort column options.

/// </summary>

public enum PrimarySortColumn

{

Prefix = 0,

Middle = 1,

Suffix = 2,

Source = 3,

Target = 4,

Meta = 5,

}

/// <summary>

/// Secondary meta sort column options.

/// </summary>

public enum SecondaryMetaSortColumn

{

User = 0,

ModifTime = 1,

}

/// <summary>

/// Secondary prefix/suffix sort column options.

/// </summary>

public enum SecondaryPrefixSuffixSortColumn

{

First = 0,

Last = 1,

}

public class SortInfo

{

public SortInfo(TMPluginConcordanceSettings.PrimarySortColumn sortColumn,

TMPluginConcordanceSettings.SecondaryMetaSortColumn metaSortColumn,

TMPluginConcordanceSettings.SecondaryPrefixSuffixSortColumn

prefixSuffixSortColumn,

bool ascending);

/// <summary>

/// The sorting is in ascending or descending order.

/// </summary>

public bool Ascending { get; set; }

/// <summary>

/// Secondary meta sort column.

/// </summary>

public TMPluginConcordanceSettings.SecondaryMetaSortColumn MetaSortColumn

{ get; set; }

/// <summary>

/// Secondary preffix/suffix sort column.

/// </summary>

public TMPluginConcordanceSettings.SecondaryPrefixSuffixSortColumn

PrefixSuffixSortColumn { get; set; }

/// <summary>

/// Primary sort column.

/// </summary>

public TMPluginConcordanceSettings.PrimarySortColumn SortColumn

{ get; set; }

}

}

The *TMPluginException* class is the following:

[Serializable]

public class TMPluginException : UserException

{

public TMPluginException(string message, string englishMessage,

Exception innerException = null)

: base(message, englishMessage, innerException)

{ }

public TMPluginException(SerializationInfo info, StreamingContext context)

: base(info, context)

{ }

}

You can use the first constructor to instantiate a *TMPluginException*. It is important to fill the *message* parameter with localized text, because memoQ displays this message under the translation grid as lookup error. See localization details later.

## The plugin options

You have to create a class to be able to store the settings of the plugin. The naming convention is: **<plugin\_name>Settings.cs**. This class should be responsible for:

* Save and load the settings to/from the following location: <IModuleEnvironment. PluginSettingsDirectory>\<PluginName>Settings.xml
* Give back the settings for the other components of the plugin.
* Tell whether the plugin is configured properly based on the actual settings.

This class should give back only the saved or the default settings. Do not call any services from this class directly, this class should be a simple entity class which can save and load itself.

memoQ does not use this class directly, therefore there is no interface related to this component.

## The configuration dialog

The plugin should have a configuration interface, where the user will be able to set up the plugin. You have to create a dialog with the proper user interface elements. This dialog will be displayed by the plugin director’s ShowOptionsForm function. The naming convention is: **<plugin\_name>OptionsForm.cs**. The requirements are the following:

* This dialog should be initialized based on the existing plugin settings. If there are no saved settings yet, initialize the dialog to the default settings.
* Do not allow to save the settings until all of the mandatory parameters were not configured correctly.
* If the user modifies the settings, collect the modifications in the memory, and save them only when the user OKs the dialog.
* Do not call any long operations from the user interface thread. Do this call in background threads.

## Localization

The third-party translation memory plugins will be localized by the Kilgray. The *IEnvironment* interface provides the *GetResourceString* function for the developers to be able to get localized texts from the translation memory environment.

All textual information which appear on the graphical user interface should be localized. Therefore the developer of the plugin has to provide the list of these strings for Kilgray. This list should contain key-value pairs. The key has to uniquely identify the string value. You will be able to use this localized texts inside your plugin with the help of the *GetResorceString* function by simply pass the key of the required text to the function. Apart from this the function has an other parameter which is the *pluginId*. This parameter should be the unique identifier of the translation memory plugin. It’s recommended to place this identifier as a public constant into the *PluginDirector* class.

It is possible that the *GetResourceString* function gives back null or an empty string. In this case the plugin should use its own default strings.

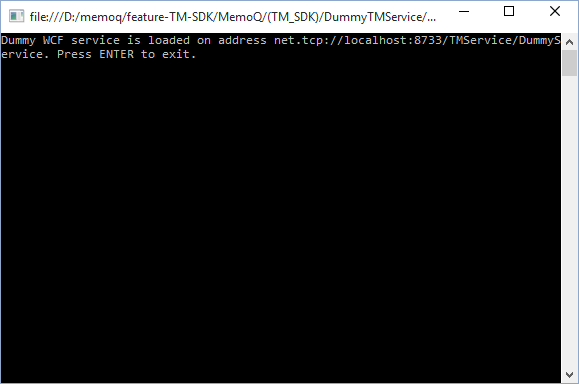
# Implementation checklist

If you are done with the implementation of the translation memory plugin, you have to check:

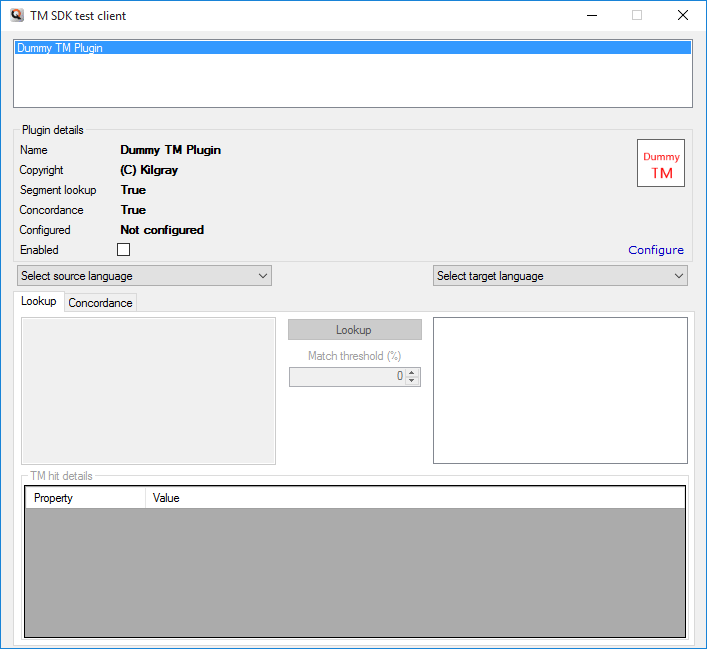
* The implementation is in a single class library, which contains references to the necessary memoQ libraries. The class library is written in C#.
* The class library’s *AssemblyInfo.cs* contains the *ModuleAttribute* attribute.
* The class library’s build path is set to the correct directory.
* There is a plugin director component, which derives from the *PluginDirectorBase* class and properly implements the necessary functions and properties.
* All allocated resources are properly disposed in the plugin director.
* There is an engine component, which derives from the *EngineBase* class and properly implements the necessary functions and properties.
* All allocated resources are disposed correctly in the engine.
* There is a session component, which derives from the *SessionBase* class and properly implements the necessary functions and properties.
* The *TMPluginException* class is used to wrap the original exceptions occurred during the translation.
* All allocated resources are disposed correctly in the session.
* There is a settings class, which is able to save and load itself to/from a proper location.
* The settings class is a simple entity class, does not call any services, and simply gives back the saved or the default settings.
* There is a configuration dialog, where the user is able to configure the plugin.
* The user cannot save the settings until all of the mandatory parameters were not configured correctly.
* The dialog collects the user modifications in the memory and saves only when the user OKs the dialog.
* The dialog does not call any blocking service in the user interface thread; it has to use background threads.
* Call the translation service only during the configuration process or during lookup and concordance operations. In all other cases use the stored plugin settings to give back the plugin information (for example the supported languages of the plugin).

# Testing the sample plugin

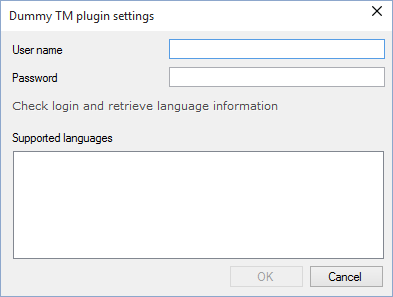
If you would like to test the existing sample plugin open the *TM\_SDK* solution in Visual Studio, and **set the *TestClient* and *DummyTMService* projects as startup projects** (multiple startup projects are to be set), build the solution and start debugging. The *DummyTMService* runs as a console application and emulates a TM service:



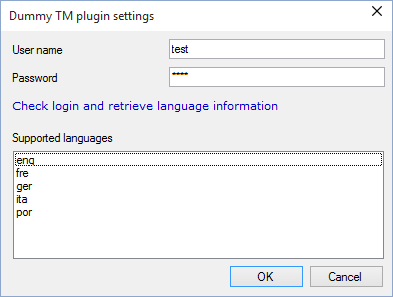
The *TestClient* emulates the memoQ client: it loads and uses the TM plugins from the ‘./bin/Debug/plugins’ directory:



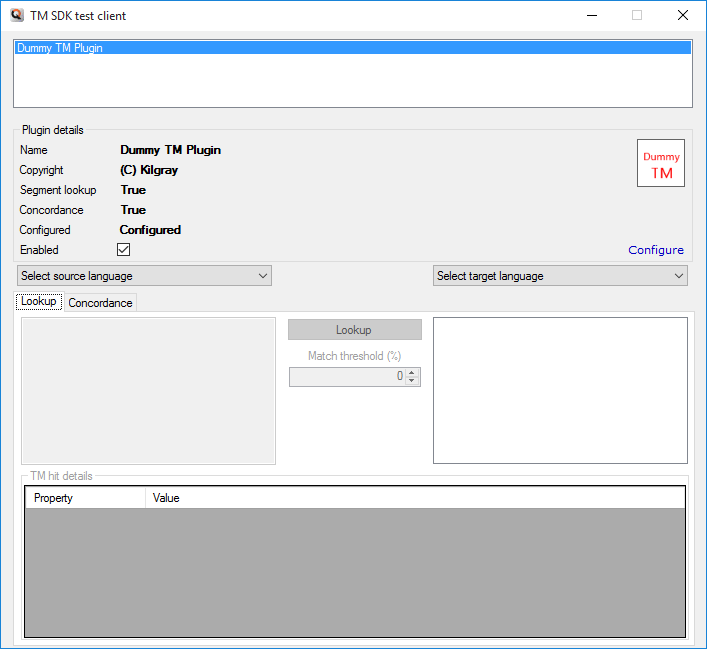
Currently there is only one TM plugin registered. The properties of the dummy plugin are in the *Plugin details* section. To be able to perform lookup or concordance operations with the plugin, configure it, because the plugin is not configured yet. Click on the *Configure* link.



The above dialog allows you to setup the plugin. Write something into the *User name* field, write something else into the *Password* field and click on the *Check login and retrieve language information*. In this case an error dialog appears, because the sample plugin allows the logins only if the user name and the password are the same. Now try to write the same strings into the user name and the password fields, and click on the check link again. Now the supported languages will appear after a few seconds:



Click on the *OK* button and enable the plugin inside the *Plugin details* section.



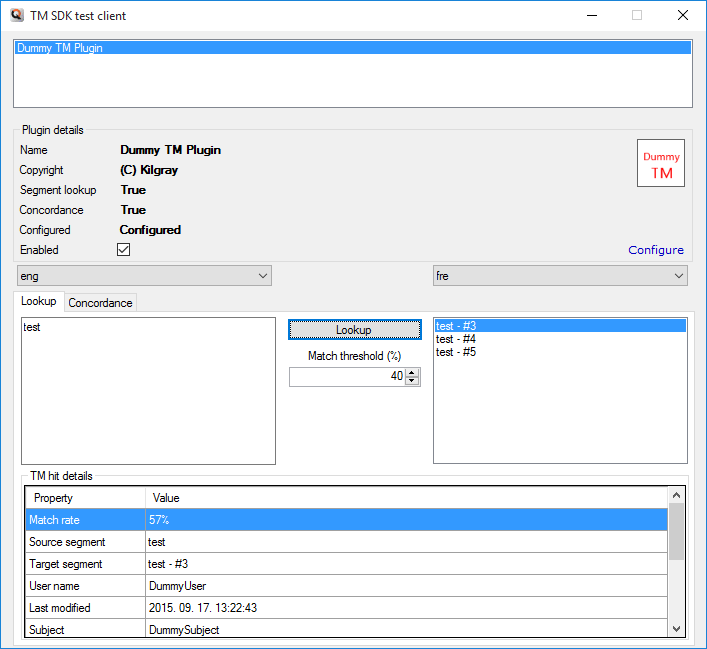
Select a language pair to be able to translate something. If you select a language pair which is not supported by the plugin, you will get the following error message: “This language pair is not supported by the selected plugin.”

Select a supported language pair and write something into the source textbox and click on the *Lookup* button. The possible translations will appear after a few seconds in the target list.

You can also set the minimum match rate of hits with the *Match threshold* control.

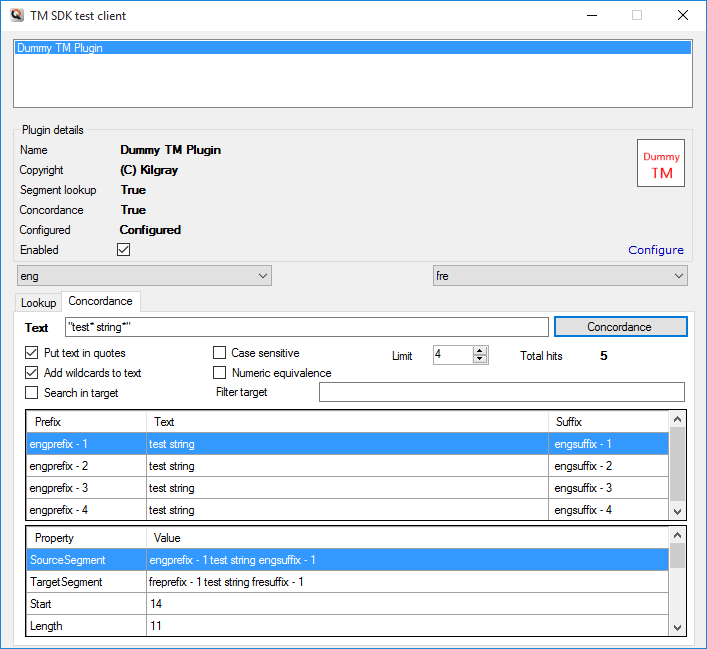
If you write more than one line in the source textbox the lines will be concatenated, separated with spaces.

If you select a TM hit from the list on the right the hit’s details are being shown under ‘TM hit details’.



To perform concordance operation change the tab to Concordance. In the textbox you can write the text you want to send for concordance. Under that you can configure the concordance operation with the checkboxes. You can also set a limit for the results of the operation.

If you select an item from the upper list, the item’s details will be shown in the lower list.



# Testing the new plugins

## Testing in the sample application

If you would like to test your translation memory plugin, you only have to set the build path correctly as mentioned above, the test client will load your plugin automatically.

If the plugin is implemented correctly your plugin will be listed on the main form of the test client. If you select the plugin from the list, you can see the general information of your plugin in the *“Plugin details”* box. If you click on the *“Configure”* link, you can set up your plugin. You will be able to test the translation if the plugin is configured and it is enabled. Select the source and the target languages (if you choose a language pair which is not supported by your plugin, a message will appear), and you can perform lookup or concordance operations as mentioned above. If the textbox contains multiline text, the lines will be concatenated. If there was any exception during the translation a message box will appear.

## Testing in memoQ client

You can also test your translation memory plugin in the memoQ client. First copy your plugin dll file into the *Addins* folder in the installation folder of memoQ client. By default memoQ does not load unsigned plugins. To enable loading your plugin you have to create an XML file named ClientDevConfig.xml in the *%programdata%/MemoQ* folder with the following content:

<?xml version="1.0" encoding="utf-8"?>

<ClientDevConfig>

<LoadUnsignedPlugins>true</LoadUnsignedPlugins>

</ClientDevConfig>

Now memoQ will load your plugin if it was implemented correctly.

# Checklist to update a plugin for memoQ 8.0

Given an existing plugin and its codebase the following steps describe the process to update the library to make it compatible with memoQ 8.0

* Target .NET 4.6.1 or newer version.
* Remove all memoQ codebase references, except for MemoQ.TMInterfaces.dll, and add MemoQ.Addins.Common.dll as reference.
* Update the AssemblyInfo.cs file, replace the namespace of MemoQ.Common.Framework to MemoQ.Addins.Common.Framework (for ModuleAttribute attribute).
* Update the implementation class of the IPluginDirectory interface to match the slightly changed interface.
* Update the implementation class of the SessionBase/ISession interface to match the new interface. All required entity classes (such as Segment) are now in MemoQ.Addins.Common assembly.
* Compile the library and fix compilation errors. In most only cases namespaces have to be fixed.
* Test the new plugin with the test client part of the SDK, or memoQ 8.0.
* Deploy the new plugin.