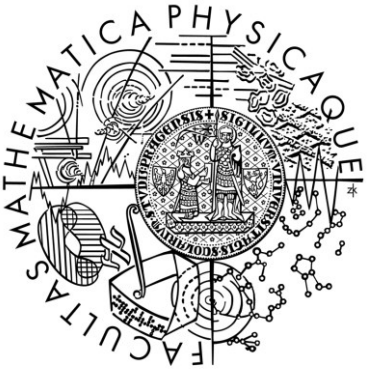
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Ondřej Štumpf

Localization Plugin for Visual Studio

Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Jakub Malý

Study programme: Computer Science

Specialization: General Computer Science

Prague 2013

I would like to thank my supervisor, RNDr. Jakub Malý, for numerous pieces of advice and great patience that significantly helped me to create this thesis. The same gratitude belongs to my family for their huge support.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In…..... date..... signature

Název práce: Lokalizační plugin pro Visual Studio

Autor: Ondřej Štumpf

Katedra / Ústav: Katedra softwarového inženýrství

Vedoucí bakalářské práce: RNDr. Jakub Malý, Katedra softwarového inženýrství

Abstrakt: Mnoho aplikací se v dnešní době vyvíjí s úmyslem nasadit je po celém světě a umožnit každému uživateli výběr zobrazení v různých jazycích. Proces přizpůsobování aplikace více kulturním prostředím se nazývá lokalizace. Programovací jazyky lokalizaci do jisté míry podporují, ale tato podpora není zdaleka dokonalá a může významně zpomalit a zkomplikovat vývoj aplikace. Cílem tohoto projektu je vyřešit tento problém v prostředí .NET Framework vytvořením pluginu do Microsoft Visual Studia, který by vývojářům usnadňoval rutinní úkoly spojené s lokalizací. Podporované jsou nejběžnější programovací jazyky a technologie .NETu, tj. C#, Visual Basic .NET a ASP .NET. Tento plugin umožňuje extrahovat řetězce ze zdrojových kódů, vybrat ty, které se mají lokalizovat a přesunout je do souborů s resources. Reverzní operace, tj. nahrazení odkazu v kódu skutečnou hodnotou, je rovněž podporována. S udržováním pouze relevantních dat v souborech s resources vývojářům pomáhá nový editor těchto souborů, který s nimi umožňuje provádět i další pokročilé operace.

Klíčová slova: lokalizace, Visual Studio, soubory s resources, řetězec

Title: Localization Plugin for Visual Studio

Author: Ondřej Štumpf

Department / Institute: Department of Software Engineering

Supervisor of the bachelor thesis: RNDr. Jakub Malý, Department of Software Engineering

Abstract: Many applications nowadays are intended to be used worldwide, supporting several languages in user interface. The process of making the application aware of multilingual environments is called localization. The programming languages used to develop such applications provide certain support for localization, but it is not perfect and may significantly complicate the development. This project aims to solve the problem in the .NET Framework environment by creating a plugin for Microsoft Visual Studio, which helps developers with routine tasks associated with localization. The most common .NET programming languages and technologies are supported, i.e. C#, Visual Basic .NET and ASP .NET. The plugin makes it possible to extract localizable string literals from source code (offering help with selecting only the relevant ones) and move them to specified resource files and vice versa. Also, editor of resource files is provided, helping developers maintain only relevant content in them and perform advanced operations with the resource files.

Keywords: localization, Visual Studio, resource file, string

Contents

1 Introduction 1
1.1 Project Goals 2
1.2 Thesis Structure 3
2 Analysis 4
2.1 Resource Files 4
2.2 Working with Resources – Issues 9
2.3 Visual Localizer Solution 10
2.4 Code Model 11
2.4.1 Visual Studio API 11
2.4.2 String Literals in Code 11
2.4.3 References to Resources in Code 15
2.5 Visual Studio Extensibility 16
2.5.1 Macros 16
2.5.2 Add-ins 17
2.5.3 VSPackages 17
2.5.4 VSPackage Structure 18
3 Architecture 19
3.1 Visual Localizer vs. VLlib vs. VLTranslat 19
3.2 Registration 20
3.3 Object Model 21
3.3.1 Commands 21
3.3.2 Tool Windows 25
3.4 Code Explorers 26
3.4.1 C# and VB .NET 27
3.4.2 ASP .NET 28
3.4.3 Determining Attribute's Data Type 30
3.5 Code Lookupers 31
3.5.1 String Literals 32
3.5.2 References to Resources 33
3.5.3 Namespace Resolution 34
3.6 Tool Grids 35
3.7 Localization Probability 36
3.7.1 Custom Criteria 36
3.7.2 Common Criteria 38
3.7.3 Calculation 39

3.8 Commands Execution 40
3.9 ResX Editor 40
3.9.1 Structure 41
3.9.2 The “Others” Tab 43
3.9.3 Looking up References 43
3.10 VLTranslat 45
4 User Interface 47
4.1 Installation 47
4.2 Move to Resources Command 47
4.3 Batch Move to Resources Command 49
4.3.1 Filter 51
4.4 Inline Command 52
4.5 Batch Inline Command 53
4.6 Global Translate Command 53
4.7 ResX Editor 55
4.7.1 Overview 55
4.7.2 Adding Resources 56
4.7.3 Removing Resources 56
4.7.4 Modification of Existing Resources 57
4.7.5 Translation 57
4.7.6 Inlining 58
4.7.7 Embedded vs. Linked Resources 58
4.7.8 Synchronization 58
4.7.9 Merging 59
4.8 Settings 59
5 Evaluation 61
5.1 Testing 61
5.2 Localization Criteria Optimization 61
6 Unresolved Issues 67
7 Prospective Enhancements 69
8 Similar Tools 70
9 Conclusion 71
10 Bibliography 72
11 Attachments 75

1 Introduction

Users of computers worldwide have long ago grown accustomed to the comfort of applications communicating with them in their native language. Almost every application nowadays, even a small one, provides several language mutations users can choose from. Usually, this choice does not affect only displayed texts, but also other culture-specific conventions, like number format, date and time format, direction of the text, currency etc. The process of making applications aware of multi-lingual environments is called *localization*.

Most of the modern programming languages provide some kind of support for localization. The common approach is to separate culture-neutral code from culture-specific output data which are often placed in separate files. In Java, ".properties" files and Bundle classes [1] are used. In Ruby on Rails [2], YAML format [3] and the "t" helper [4] accomplish the same. In .NET, ResX files along with special tools provided by the Visual Studio work almost identically. This enables developers to focus on the application logic and leave the culture-specific parts on translators, who, on the other hand, have no need to learn the programming language.

All these environments share not only the advantages of the approach, but they also share the same risks. First of all, one must be sure that the application code contains no culture-specific data. When developing the application from scratch, we need to be very disciplined and dutiful and place all culture-specific data to dedicated files. Skilled developers are able to do that, but it is not very effective. However, when we completed the application and now we face a request to localize it, we are forced to read all the code, find whatever must be localized and move it elsewhere. There is no way of knowing whether we did a thorough job and did not omit something.

Moreover, it is almost impossible to keep the resource files "clean". They may contain texts that are not referenced at all, they may miss data, they may contain duplicates etc. Also, there are usually more mutations of the same file, one for each supported language. These files must be kept synchronized, which can become a very irritating task. And again – there is no way of knowing whether we did not make a mistake.

1.1 Project Goals

This project, called "Visual Localizer", aims to solve these problems in the .NET Framework environment. It offers developers intuitive and easy-to-use tool that becomes part of their IDE (Integrated Development Environment) and makes development of localized applications much more simple, elegant and efficient. Specifically, the project goals are as follows:

- Create a tool for looking up string literals in the source code and moving them to selected resource files, producing a reference in the code instead. All string literals will be correctly recognized according to the programming language syntax. The tool will also be able to handle basic string concatenation (especially in Visual Basic .NET – see 2.4.2).
- Support also the reverse operation – looking up references to resources in the source code and replacing them with hard-coded string literals. From now on, we will call this operation *inline*.
- Provide an editor of resource (ResX) files, much more user-friendly than the default Visual Studio editor and enhanced with many useful features (transfer between embedded and linked resources, inlining resources…)
- Aid the developers maintain only relevant content in resource files by keeping track of references to a resource
- Synchronization of resource files for various languages will be supported in the editor
- Examine if and how could well-known translation services (Google Translate [5], Microsoft Translator [6] and MyMemory [7]) be employed in order to automatically translate resource files
- Provide consistent support for Visual Studio projects written in C#, Visual Basic .NET (Windows Forms) and ASP .NET (websites and web applications)
- Our tool will be compatible with Visual Studio 2008, Visual Studio 2010 and Visual Studio 2012

1.2 Thesis Structure

The thesis begins with an analysis of the environment we will be working in – we will examine resource files structure, how to explore source code and how to integrate our extension into the Visual Studio environment. Chapter 3 describes the Visual Localizer project code model and other back-end functionality. The following chapter is dedicated to the description of the project from user's point of view. In Chapter 5 we will prove that our tool can be used in real-life projects – we will demonstrate that on several randomly picked open source projects.

Following two chapters describe issues that were not solved during the development and how could Visual Localizer be extended in the future. Finally, we will briefly examine tools with similar functionality as Visual Localizer.

2 Analysis

As it was mentioned above, the .NET Framework employs the same localization strategy as many other programming environments. This strategy is built on separating the application logic from the user output. Particularly in .NET, resource files (ResX files, files with the ".resx" extension) are used to store static, non-code data such as strings, images, sounds, icons etc. We will discuss the format of these files, the way their resources are referenced and how the localization itself is performed in Section 2.1.

Next we will examine current situation in which developers find themselves when dealing with resources and the localization process in .NET. We will describe ordinary tasks and problems and the solution Visual Localizer offers.

In the following sections we will elaborate selected parts of syntax of all supported programming languages, focusing on what is crucial for Visual Localizer's architecture, i.e. string literals and references to resources. Finally, we will show how Visual Studio can be extended and which of the several possible ways is the most suitable for our purpose.

2.1 Resource Files

The .NET Framework uses ResX files to store resource data. A ResX file is actually a XML (Extensible Markup Language [8]) file with specific schema which contains key-value entries, each entry representing one resource. The schema of ResX files is directly included in every resource file, using the XML Schema [9] definition language. Basic structure of a ResX file is shown in the following example:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <!-- Microsoft ResX Schema -->
  <xsd:schema id="root" xmlns=""
      xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:msdata="urn:schemas-microsoft-com:xml-msdata">
    <xsd:import
     namespace="http://www.w3.org/XML/1998/namespace" />
    <xsd:element name="root" msdata:IsDataSet="true">
      <!-- ... schema ... -->
    </xsd:element>
  </xsd:schema>
```

```
  <resheader name="resmimetype">
    <value>text/microsoft-resx</value>
  </resheader>

  <resheader name="version">
    <value>2.0</value>
  </resheader>
  <resheader name="reader">
    <value>System.Resources.ResXResourceReader,
      System.Windows.Forms, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089</value>
  </resheader>
  <resheader name="writer">
    <value>System.Resources.ResXResourceWriter,
     System.Windows.Forms, Version=2.0.0.0, Culture=neutral,
     PublicKeyToken=b77a5c561934e089</value>
  </resheader>

  <data name="key1" xml:space="preserve">
    <value>Value of key1</value>
  </data>
  <data name="key2" xml:space="preserve">
    <value>Value of key2</value>
  </data>

  <assembly alias="System.Windows.Forms" name="System.Windows.Forms,
     Version=2.0.0.0, Culture=neutral, PublicKeyToken=b77a5c561934e089" />

  <data name="photo" type="System.Resources.ResXFileRef,
     System.Windows.Forms">
    <value>Resources\Images\photo.png;System.Drawing.Bitmap,
     System.Drawing, Version=2.0.0.0, Culture=neutral,
     PublicKeyToken=b03f5f7f11d50a3a</value>
  </data>
</root>
```

The header of the file specifies the MIME (Multipurpose Internet Mail Extensions [10]) type, the version and what tools were used to generate the file. The resource entries themselves are represented by the `<data>` elements; the resource file from the example above contains two string resources named `key1` and `key2` and one image resource named `photo`, linked to the file "Resources\Images\photo.png". As the ResX header suggests, we do not need to create the files by ourselves. .NET provides the `ResXResourceReader`[1] class which parses the resource file and enumerates the resource entries located within; the `ResXResourceWriter`[2] class on the other hand handles generation of the resource file from the specified list of resources.

[1] http://msdn.microsoft.com/cs-cz/library/system.resources.resxresourcereader.aspx
[2] http://msdn.microsoft.com/en-us/library/system.resources.resxresourcewriter.aspx

The resource key specified in the `name` attribute of a `<data>` element must be unique within the file; the value of the resource entry must be either a string, a strongly typed value (see 3.9.2), a link to a resource file (*linked resource entry*) or the resource file itself (in a text representation of its binary data – *embedded resource entry*) [11]. In case of linked resources, the linked file itself must be a part of the project (i.e. built into the project assembly).

ResX files hold resources, but do not provide any convenient way of referencing them from code. This is a job for `ResXFileCodeGenerator`, a custom tool[3] shipped with Visual Studio, responsible for creating a strongly typed *designer class* used to access the resources. There is also `PublicResXFileCodeGenerator` – the only difference is that while `ResXFileCodeGenerator` generates `internal` classes, `PublicResXFileCodeGenerator` generates `public` classes. This can be useful when creating a DLL library that exposes resources to its users. The custom tool creates a special class (*designer class*) which resides in its own file (*designer file*). This file contains only valid C# or VB .NET source code. For each resource entry in the ResX file there is a similarly named property in the designer class.

Since there is no such rule that resource keys have to be valid identifiers for the current programming language, it is sometimes impossible to name the property the same as the resource key. In these situations the `ResXFileCodeGenerator` tool replaces invalid characters in the resource key with underscores and uses other techniques to create a valid identifier from it. This may of course lead to a case when two resource keys are modified to be represented by the same property – in that case, none of the properties is created in the designer class and the resource is therefore inaccessible. [12]

The example below shows a simple designer class, generated by `PublicResXFileCodeGenerator` from the resource file we discussed earlier:

[3] Although the term "custom tool" sounds very general, in the Visual Studio context it possesses a very concrete meaning; it is a tool that is run on certain project files when the project is built, creating a new project file as output [43].

```
namespace Namespace1 {
  using System;

  [global::System.CodeDom.Compiler.GeneratedCodeAttribute(
       "System.Resources.Tools.StronglyTypedResourceBuilder",
       "2.0.0.0")]
  [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
  [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]
  public class Resource1 {
     // initialization and more cultures-related properties here
      public static string key1 {
          get {
              return ResourceManager.GetString("key1",
                     resourceCulture); } }
      public static string key2 {
          get {
              return ResourceManager.GetString("key2",
                     resourceCulture); } }
      public static System.Drawing.Bitmap photo {
          get {
              object obj =
                     ResourceManager.GetObject("photo",
                          resourceCulture);
              return ((System.Drawing.Bitmap)(obj)); } }
   }
}
```

The class provides developers easy access to all resources located in the resource file, simply by using the static property of the class, for example:

```
string s = Namespace1.Resource1.key1;
```

All Visual Studio projects share this approach; i.e. they use ResX files to store data and custom tools to generate designer classes from them. However, there are certain differences:

- In C# projects, the designer file is not at all different from other source code files – it is a part of the solution, visible in the Solution Explorer etc.
- Although the designer file is created completely the same way in VB .NET projects, it is not visible in the Solution Explorer.
- There are two basic kinds of ASP .NET projects – a website project and web application project. The latter behaves identically as the previous cases. Resource files in websites, on the other hand, are different – first of all, they must be located in the `App_GlobalResources` subfolder in order to be reference-able. They also do not use the `ResXFileCodeGenerator` custom tool, but what is called `StronglyTypedResourceBuilder`. It also

creates a designer class, but unlike `ResXFileCodeGenerator`, this class does not belong to any customizable namespace, but to a standard `Resources` namespace, which in conclusion means no two resource files can share a name.

One of the primary goals of resource files is to enable localization. Normally, the resource files have names such as "Resource1.resx", with the designer class thus named `Resource1`. This variant will be hereafter called a *culture-neutral file*. However, if we name the resource file "Resource1.cs.resx", it becomes a culture-specific file for the Czech localization. No designer class is generated for this file (although a blank designer file is generated - curious). In such situations, the .NET Framework supports one culture-neutral file and many culture-specific files with a corresponding name. To every culture-specific resource file there should be exactly one culture-neutral "parent". As a result, we always reference the culture-neutral file and leave the .NET to select proper culture-specific version, if such is present (based on the `Thread.CurrentThread.CurrentUICulture` property).

The following example shows three resource files – one culture-neutral and two corresponding culture-specific files, Czech and German. The code block below describes their usage in a localized application.

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <!-- Resource1.resx - culture-neutral -->
  <!-- ... -->
  <data name="helloWorld" xml:space="preserve">
    <value>Hello, World!</value>
  </data>
</root>
```

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <!-- Resource1.cs.resx - culture-specific - Czech -->
  <!-- ... -->
  <data name="helloWorld" xml:space="preserve">
    <value>Ahoj, světe!</value>
  </data>
</root>
```

```
<?xml version="1.0" encoding="utf-8"?>
<root>
  <!-- Resource1.de.resx - culture-specific - German -->
  <data name="helloWorld" xml:space="preserve">
    <value>Hallo, Welt!</value>
  </data>
</root>
```

```
/* this will pick up the value according to the regional and culture
settings from the OS user settings, let's assume the user works in English
environment */
string def = Resource1.helloWorld; // contains "Hello, World!"

// this line explicitly changes the UI culture of the executing thread
Thread.CurrentThread.CurrentUICulture=CultureInfo.GetCultureInfo("cs-CZ");
string cs = Resource1.helloWorld; // contains "Ahoj, světe!"

Thread.CurrentThread.CurrentUICulture=CultureInfo.GetCultureInfo("de-DE");
string de = Resource1.helloWorld; // contains "Hallo, Welt!"

Thread.CurrentThread.CurrentUICulture=CultureInfo.GetCultureInfo("pl-PL");
string pl = Resource1.helloWorld; // contains "Hello, World!"
// since Polish resource file was not found
```

The editor of ResX files provided by Visual Localizer is aware of all discussed configurations and is able to correctly work with all of them.

2.2 Working with Resources – Issues

The .NET Framework and Visual Studio work together to make it possible to work with resource files. However, it is still the developer himself who must populate the resource files with data, reference them from code and as the development continues, maintain only relevant content in these files.

Suppose for example the developer has a string literal in the source code and wants to move it to a resource file. What most developers do is that they copy the string to the clipboard, open the resource file, create a new string entry (having to create a new unique resource key), save and close the file, delete the string literal from the code, insert a reference to the newly created resource and possibly add an import statement. It is a very long and dull process even when performed once; batch processing (i.e. moving all strings from the source code file to a resource file) is almost an impossible thing to do.

Now let us assume the developer made an error and moved to the resource file something he did not mean to. Undoing the operation is almost as hard as actually performing it; of course, using the "undo" command in the source code document will remove the reference and insert the string literal back, but that would leave an unused resource in the resource file. The file would have to be opened, the resource entry deleted and the file saved again. Moreover, this only works if the developer spotted his error right after he made it, otherwise he could not use the undo command.

Resource files can contain not only strings, but also other resources like images, icons, sounds etc. Developers want (or should want) to maintain only relevant content in the resources files, i.e. every resource should be at least once referenced. This is important when working with string resources (the project should be clean) but even more significant for media resources – the sound resources for example tend to be quite large and no one wants an unnecessarily large application.

Moreover, as it was already mentioned, .NET enables localization by following a simple resource files naming convention. Ideally, the culture-neutral file and its culture-specific mutations share all string resources keys (differing in values), while media resources are present only in the culture-neutral file. However, there is no support for this kind of synchronization in the default ResX editor. Even such a simple operation as merging two resource files together may become an irritating task, since the Visual Studio editor does not support selecting all resources in the file at once. It is possible to solve this by copying plain XML text between the files, but since they may contain binary data (embedded resources – see 2.1), it is not very comfortable.

2.3 Visual Localizer Solution

Visual Localizer aims to provide a tool that would ease all the ordinary tasks discussed in the previous section. I have separated the tasks in two categories – first, resource files editing-related tasks, which include keeping track of references to each resource entry, merging and synchronization of the ResX files. This will be solved by the custom editor of ResX files (see 3.9), which will support all these operations.

The second category consists of tasks that handle the data transfer between the source code and a resource file. This includes moving a specific string literal to a resource file (creating a reference on its place) and the *batch* version of this, i.e. scanning whole set of source code files for potentially localizable strings and moving them to specified resource files. The reverse operation, i.e. replacing the resource reference with an actual resource value is also supported. Both of these operations closely cooperate with the Visual Studio undo manager, making it possible to reverse any of these operations as one atomic action.

It is clearly necessary for Visual Localizer to work closely with the source code. We must be able to correctly recognize string literals (for moving) and

references to resources (for inlining and ResX editor reference tracking). We will discuss the Visual Studio API support for this in the following section.

2.4 Code Model

In this section we will discuss the way Visual Localizer works with source code files. We will describe the support we can expect from the Visual Studio API and then focus on the functionality we have to create ourselves.

2.4.1 Visual Studio API

Visual Studio offers the `EnvDTE.DTE` [13] object, which is a basic entry point plugins use to interact with the hosting instance of Visual Studio. Through this object we can obtain instances of `FileCodeModel` [14] for any C# or VB .NET source code file. The `FileCodeModel` object exposes the source code in an object model, enabling us to look up the code without actually having to parse it ourselves. Unfortunately, this level of abstraction works only on the levels above methods – we can only access the plain text of the method, not its parsed content.

However, we also want to support ASP .NET projects, which consist not only of C# or VB .NET code files, but mostly of AspX files [15]. The format of these files resembles XML, but with few extra features. Unfortunately, we cannot obtain `FileCodeModel` instance for AspX files.

To sum up – we need to implement our own way of "parsing" methods' bodies in order to extract string literals and references to resources. Also, we need to create our own parser of the AspX code with the same functionality. To do this, we must be able to recognize the syntax of the programming languages; we will discuss this topic in the following two sections.

2.4.2 String Literals in Code

In C#, string literals are marked using the double quotes (") in code. For the sake of correct string recognition, we need to consider the following [16] [17]:

Strings can contain *escape sequences*. These are predefined sequences of characters that are interpreted differently – for example, the "\n" sequence is interpreted as the newline character. A special case of an escape sequence is escaping

a double quote – when preceded with a backslash, it loses its meaning as an end of a string. Also, it is possible to insert any character by specifying its hexadecimal Unicode value.

```
string s1 = "First line.\nSecond line.";
string s2 = "Hello,\"World\"!";
string s3 = "A\x123B";
```

Comments can be used in code. The code that is commented out should be skipped – we need to correctly recognize the "//" and "/*" commenting styles. The first marks the rest of the line as comment, the latter is a multiline comment – marks everything between "/*" and "*/" [4].

```
int a; // this is ignored by the parser
/* all of this
   is also ignored */
```

When the character before the opening quote is the "at" character (@), the string literal is called a *verbatim string*. It differs from a common string in the way escape sequences are handled. In fact, verbatim strings do not interpret escape sequences at all, so "\n" would not be interpreted as the newline character, but as its literal content. The only escape sequence supported by verbatim strings is the escape sequence for double quote – two sequential double quotes are interpreted as one double quote, not the end of string.

```
string s = @"First line.\nNOT a second ""line"".";
```

Common string literals can span only a single line, while verbatim strings can span multiple lines.

Not every string literal occurrence can be safely replaced with a reference to a resource. For example, fields or variables with the `const` modifier must be initialized with compile-time evaluable expressions, which a reference to a resource is not. In

[4] Actually, C# supports a third kind of comments, declared with "///". These are called *documentation comments* and are used by certain tools to generate documentation output. However, they will be correctly recognized since they also begin with "//", just as standard line comments.

this category also belong string literals used in attributes or as default values for methods' arguments[5].

```
const string s = @"This cannot be reported as a localizable string.";
```

Visual Basic .NET (VB) uses the same basic principles as C#, but differs in certain syntax aspects:

Escape sequences are not supported. The way every string literal is treated is very similar to verbatim strings in C#, only without the leading "@" character and no support for multiline strings. The only way to insert a control character into the string is to use string concatenation, as shown in the following example. Visual Localizer can detect these situations and replaces most used cases with actual characters. More about this can be found in Section 3.5.

```
Dim s1 As String = "First line.\nNOT a second ""line"".";
Dim s2 As String = "First line." & vbNewLine & "A second line."
```

VB does not support multiline comments. Instead, single quotes (') behave the same way as "//" in C# – the remaining text on the line is marked as comment and ignored by the compiler. The single quote can be substituted with the `REM` (remark) statement.

```
Dim a As Integer
' line of comment
REM also a line of comment
```

While both C# and VB have their string literals clearly designated, ASP .NET[6] is not so straightforward. The AspX file format is based on XML and is used by the ASP .NET engine to generate HTML [18] code, which is then sent to the browser. Therefore, localizing an AspX document is similar to localizing HTML – everything that is eventually displayed to the user must be first handled by the localization mechanism. This may mean element attributes values, but also plain text nodes within the document. Moreover, files may contain blocks of either C# or VB code, so we must find a way to correctly handle such content. To sum up - what we

[5] This feature is available since .NET 4.0.

[6] Despite a similar name, ASP is something very different from ASP .NET [41]. In this project we will always be working with ASP .NET.

have called “string literal” so far may mean several things in the context of ASP .NET and Visual Localizer must be able to handle all such cases. The example below shows all potential sources of localizable strings – a page directive, an ASP .NET element, HTML element, plain text and C# or VB .NET code block:

```
<%@ Page Language="C#" Title="Localizable" %>
<asp:Button runat="server" Text="Also localizable"/>
<input type="button"  Text="Also localizable"/>
Plain text - can localize
<% string s = "Can localize this"; %>
```

There are two kinds of comments in ASP .NET code:

- Standard HTML-style comment, called *client-side comment* in this context and marked with “<!--” and “-->”. Text within these tags is processed by the server engine and passed to the browser.
- *Server-side comment*, marked with “<%--“ and “--%>”. Text within these tags is not processed by ASP .NET at all and neither is it sent to the browser.

We will stick to the rules of the Microsoft AspX parser and thus report string literals within client-side comments as potentially localizable and we will omit string literals within server-side comments the same way we omitted content of C# or VB code comments.

The way escape sequences are treated is based on the origin of a string. If it is C# or VB string from embedded code block, respective rules apply. On the other hand, when dealing with a string coming from element’s attribute, the only way how to escape characters is to use HTML entities.

```
<asp:Literal runat="server" Text="Hello, &quot;World&quot;!"/>
```

We already mentioned elements’ attributes’ values as a possible source of localizable data. However, not every attribute can be localized – only those which actually represent a string property can be safely replaced with a reference to a resource. In this project we use *.NET Reflection* [19] along with the configuration taken from “web.config” [20] files to determine attribute’s data type. The whole type resolution mechanism is described in Section 3.4.3.

2.4.3 References to Resources in Code

Also when looking up references to resources in code, there are several things we have to take into account. First, we need to skip the code that it commented out the same way we did when looking up string literals. Second, we need to correctly resolve the target of the reference. This may not be as easy as it seems, as shown in the following C# example:

```
using Application.ResourcesA;
using rc = Application.ResourcesC;

namespace Application {
    namespace ResourcesB {
        class Class1 {
            string s1 = Application.ResourcesA.Resource1.Key1;
            string s2 = Resource1.Key1;
            string s3 = rc.Resource1.Key1;
        }
    }
}
```

Clearly `s1` gets initialized by the value of the `Key1` property from the `Resource1` class belonging to the `Application.ResourcesA` namespace. However, determining the qualified name of initializer of `s2` requires more work – first we must find out whether the `ResourcesB.Resource1` class does exist and only if not we can assume the qualified name should be `Application.ResourcesA.Resource1.Key1`. Moreover, the .NET has concept of *aliases*, which can be used in case of name conflicts as shown in the initializer of `s3`. Omitting the `rc` alias in the initializer would cause the `Application.ResourcesB` class to be used, not the `Application.ResourcesC` class as it was intended. The qualified name resolution algorithm is the same for all considered programming languages and will be discussed in more detail in Section 3.5.3. [21]

C# and VB .NET also differ in defining statements. While in C# each statement is terminated with semicolon (;) and therefore can span multiple lines, VB's statements are implicitly terminated by the newline character. If we want to combine statements on multiple lines, we need to use the line-joining character – underscore. Being able to recognize where line-joining occurred is important for looking up references. [22]

```
string s = Resources   .
       Resource1.
       Key1; // this is a valid statement in C#

Dim s2 As String = My.Resources. _
       Resource1. _
       Key1
     '   in   VB   .NET,   underscore   must   be   used   in   order   to
     '   join lines
```

In ASP .NET we must also deal with various kinds of references. There is a simple reference in a code block:

```
<% string s = Resources.Resource1.Key1; %>
```

Reference in an output statement:

```
<%= Resources.Resource1.Key1; %>
```

Reference in an output statement with HTML encoding (since .NET 4.0):

```
<%: Resources.Resource1.Key1; %>
```

Reference in an expression:

```
<%$ Resources:Resource1,Key1 %>
```

Visual Localizer is able to parse all these options and inline them if requested.

2.5 Visual Studio Extensibility

So far we have discussed the functional issues of Visual Localizer. In this section, we will focus on how to integrate the Visual Localizer extension into the Visual Studio IDE.

Visual Studio can be extended on three basic levels, which differ in complexity and capabilities of the extension – we will briefly describe every one of them. Finally, we will focus on the extension type most suitable for Visual Localizer. [23]

2.5.1 Macros

Macros in the Visual Studio work in a very similar way to the way they do in the Microsoft Office [24]. Users can record or code their own macros in Visual Basic .NET and distribute them; there is even no need for the Visual Studio SDK (Software Development Kit). However, this approach has certain disadvantages –

first of all, there is no way of distributing the macro in a binary form. That is, anyone will be able to see the source code of the macro, which is rather inconvenient in the commercial environment. Second, the scope of macros is very limited – no new functionality can be introduced, it can only automate often repeated tasks.

2.5.2 Add-ins

Add-ins represent another level of the Visual Studio extensibility. They cooperate with the hosting Visual Studio environment much more closely than macros, which gives developers a much more powerful tool. Add-ins can access Visual Studio object model (the `EnvDTE.DTE` object), which serves as an entry point through which the add-ins communicate with the hosting environment. They can be written in C#, VB .NET or C++; all these options are equivalent. Also they can be distributed in a binary form, making them an acceptable choice for commercial developers.

New GUI elements like tool windows, toolbars and settings pages can be created using add-ins. It is possible to handle various Visual Studio events and access solution items. Custom commands can be created and added as menu items in the Visual Studio menus and context menus.

To create an add-in, one must have the Visual Studio SDK installed. This SDK among other things extends Visual Studio with new project types and creates an experimental registry hive used during debugging.

Unfortunately, add-ins do not make it possible to create custom file editor, Visual Localizer therefore cannot be implemented as an add-in.

2.5.3 VSPackages

"*VSPackages are software modules that extend the Visual Studio integrated development environment (IDE) by providing UI elements, services, projects, editors, and designers.*" [25]

VSPackages are the most powerful of Visual Studio extension options. In fact, the Visual Studio itself consists of hundreds of VSPackages. As developers of a VSPackage, we have the same options and possibilities as the developers of Visual Studio. Both macros and add-ins are subsets of VSPackages – anything that can be achieved in a macro or an add-in can also be achieved in a VSPackage. This also means that Visual Studio SDK must be installed to develop a VSPackage.

Specifically, we can create custom file editors, new project types, debuggers or even support for new programming languages. VSPackages can also communicate with each other or force other packages to be loaded.

Thanks to these advantages, Visual Localizer is implemented as a VSPackage.

2.5.4 VSPackage Structure

A VSPackage is basically a DLL library written in C#, VB .NET or C++ that gets loaded into the Visual Studio environment. The core of every VSPackage is a class implementing the `IVsPackage` interface (or derived from the `Package` class) – from now on, we will call this the *package class*. The package class is extremely important for two reasons:

- Its `Initialize()` method is run whenever the package is loaded, enabling the VSPackage to perform startup tasks.
- It can be decorated with many attributes with special meaning for Visual Studio. These attributes are used in the registration process (Section 3.2) and specify what services the VSPackage will consume, what services it will offer for other VSPackages to use, when the package should be loaded etc.

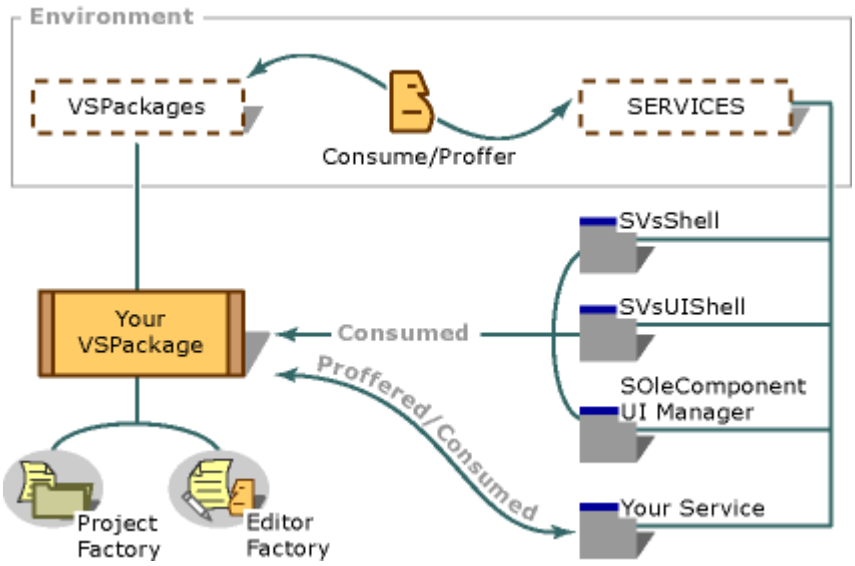


Figure 1: Schema of the Visual Studio architecture [26]

3 Architecture

In the following sections we will describe how various features of Visual Localizer are implemented. We will begin with separating the code into isolated libraries which other developers can take advantage of. Next we will focus on the Visual Localizer itself – how it gets registered in the Visual Studio, what object model does it use and how its commands work. Then we will discuss the "localization probability" feature, which is fairly unique and helps developers quickly localize even very large projects. Finally, we will describe how the ResX editor is implemented, particularly focusing on the reference-tracking feature.

3.1 Visual Localizer vs. VLlib vs. VLTranslat

While certain functionality of Visual Localizer is fairly unique and would hardly be reused anywhere else except the project itself, many other functions could be useful for large amount of VSPackage developers. I have decided to extract this general, easily reusable functionality to a separate code library called "VLlib". It basically contains three kinds of objects:

- Wrappers for often used yet inconvenient Visual Studio API objects and functions. Since Visual Studio makes heavy use of COM (Component Object Model [27]) and Visual Localizer is coded in C#, there are certain issues concerning data type conversions, error codes vs. exceptions etc. Using such API in C# is not at all impossible, but can be rather uncomfortable.
- Functionality not specific for Visual Localizer, for example the template class for custom file editors, classes derived from `DataGridView` enhanced with new features or the parser of AspX files.
- Methods and classes surprisingly missing in the Visual Studio API. For example, such seemingly trivial task as utilizing the Output Window in Visual Studio is not supported. I had to create my own attribute class used by the `RegPkg` utility in the registration process (see 3.2) to achieve this. Another example may be the undo stack of opened documents – despite the name "stack", its API lacks the method that would pop the element from the top.

We will describe VLlib functions in more detail whenever we use them in Visual Localizer.

There is one more code library in the solution, called "VLTranslat". It exposes API for using translation services of Google Translate [5], Microsoft Translator [6] and MyMemory [7]. This library is not specific for Visual Studio extensions development and can be used in any .NET project. We will talk about VLTranslat library in a dedicated Section 3.10.

3.2 Registration

Registration is the process during which an existing installation of the Visual Studio is notified about a new VSPackage[7]. In fact, installation of a VSPackage consists only of two steps:

1. Copy all files required by the VSPackage (libraries, resources etc.) to a directory on the hosting computer.
2. Inform the hosting Visual Studio environment about the new VSPackage, specifying when it should be loaded.

The first step is fairly straightforward – user specifies the target directory in the installer GUI and the files are copied. The directory can be shared by all versions of Visual Studio in which the package is to be registered. The seconds step takes place in the Windows registry and requires more work – we need to use the `RegPkg` [28] tool (shipped with the Visual Studio SDK). This tool is given the file in which the *package class* (see 2.5.4) is located, extracts data from its attributes and creates output in the registry format. The output of `RegPkg` defines new registry keys and values which are necessary for our VSPackage to be correctly loaded in the Visual Studio environment. However adding the data to the registry is not enough – to complete the registration process, following command must be executed on the hosting computer: [29]

```
devenv.exe /setup /nosetupvstemplates
```

[7] There is a new version of the registration process, using the VSIX format [42] to deploy VSPackages. However, this option is available since Visual Studio 2010 and therefore is unacceptable for usage in Visual Localizer.

The installer must therefore first copy the files, update the registry with the `RegPkg` provided values and finally, call the `/setup` command with "devenv.exe" being located in the Visual Studio installation directory. The same steps must be executed on the VSPackage uninstall. In Visual Localizer, the "VLSetupFinalizer" library is responsible for executing the `/setup` command for each version of Visual Studio, in which the user asked Visual Localizer to be registered.

3.3 Object Model

3.3.1 Commands

Commands are objects responsible for looking up code searching either for string literals or references to resources. There are two approaches to this task – we may want to work either with one string or reference ("ad-hoc command") or we may want to process whole set of files ("batch command"). The hierarchy of ad-hoc commands is displayed on Figure 2; the hierarchy of batch commands is shown on Figure 3. Also, while the "batch commands" are only responsible for looking up data, passing them to an appropriate tool window (see 3.3.2) when finished, the "ad-hoc commands" handle the whole process – collect the data, display a dialog if necessary and perform the operation.

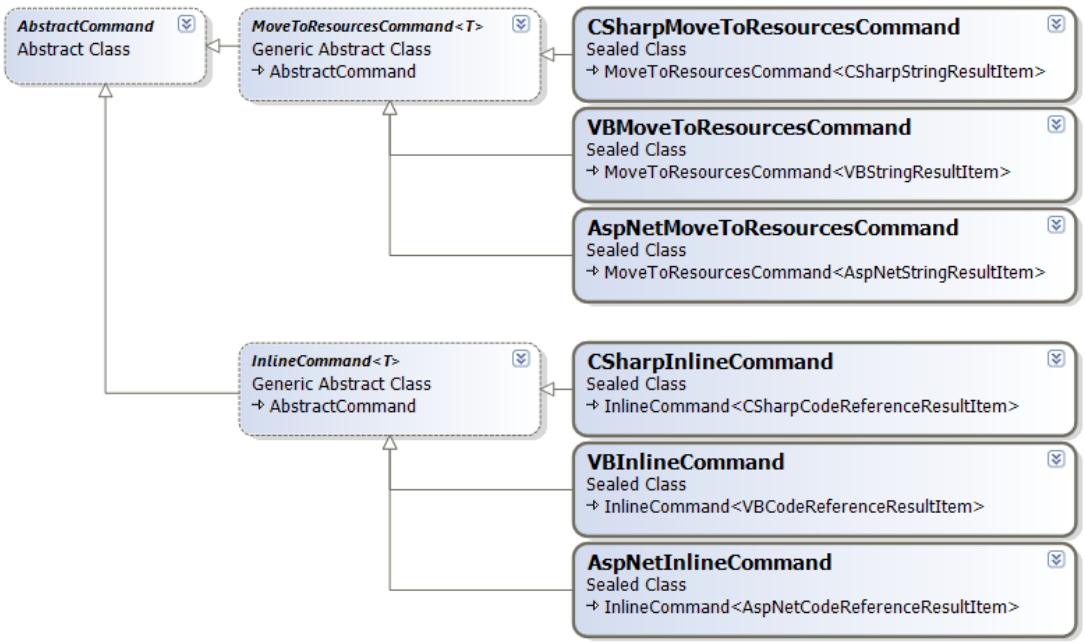


Figure 2: Class diagram of ad-hoc commands.

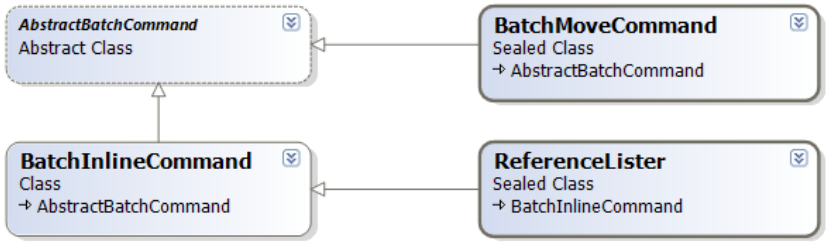


Figure 3: Class diagram of batch commands.

The purpose of the `BatchMoveCommand` and `BatchInlineCommand` is clear – they are invoked when corresponding menu item is clicked. The `ReferenceLister` command, on the other hand, is never invoked directly by the user. It is used by the ResX editor to collect references to all resources in currently edited file. It differs from the `BatchInlineCommand` particularly in the way read-only files are handled; while the `BatchInlineCommand` ignores them (their references cannot be inlined), the `ReferenceLister` includes them in the search.

Nevertheless, all commands share a need to parse the source code. Objects providing this functionality will be hereafter called *code explorers* and *code lookupers*. The *code explorers* are given a list of source code files and use their `FileCodeModel` to recursively break down to methods' bodies (or variables' initializers). At that point, *code lookupers* take over control – they are passed the plain code of a method from the code explorer and return list of either string literals or references to resources located in the method. We will call any such occurrence a *result item*. The workflow we just described is displayed in Figure 4.

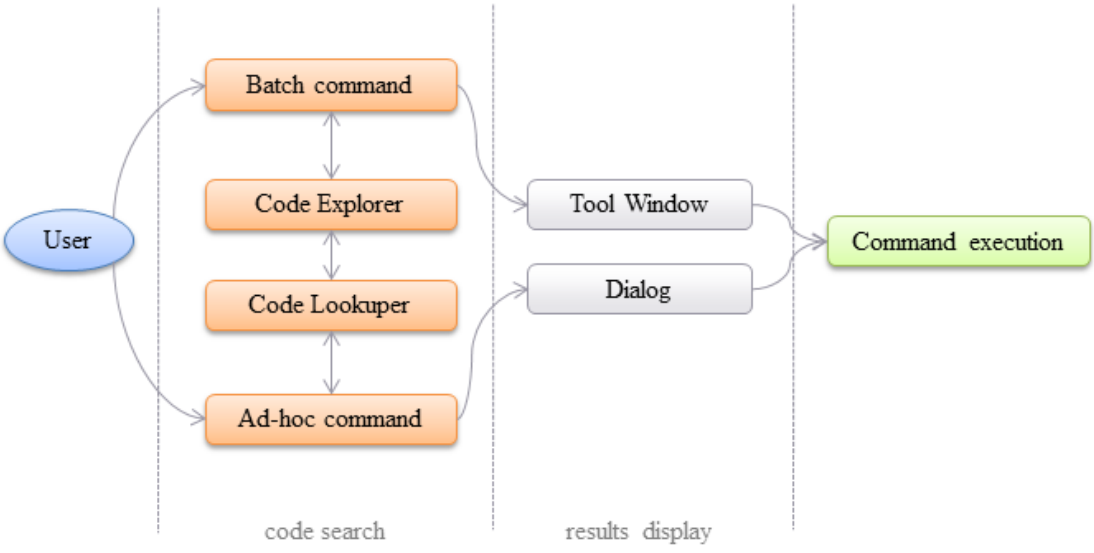


Figure 4: Commands workflow.

Since both C# and VB .NET code files can be accessed through the `FileCodeModel`, the corresponding code explorers (`CSharpCodeExplorer` and `VBCodeExplorer`) make heavy use of it and differ only in small details. However, as mentioned earlier, there is no `FileCodeModel` for AspX files. Therefore, the `AspNetCodeExplorer` uses the AspX parser implemented in VLlib to extract data from the source files. We will describe this parser in Section 3.4.2.

As mentioned above, the *code lookupers* work directly with the source code of a method or a variable's initializer. However, they are used in one more case – when parsing code block's text in ASP .NET. The AspX code explorer passes found blocks of code to the corresponding code lookuper the same way as if it was method's body content.

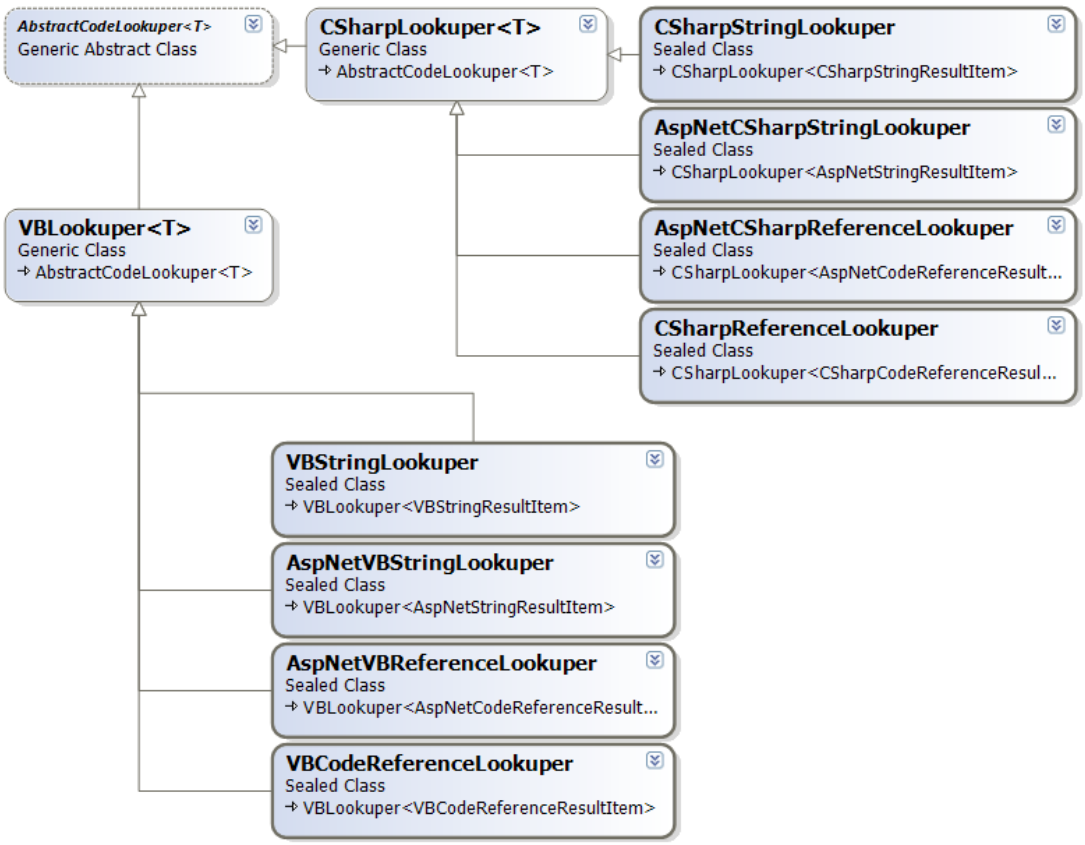


Figure 5: Class diagram of code lookupers.

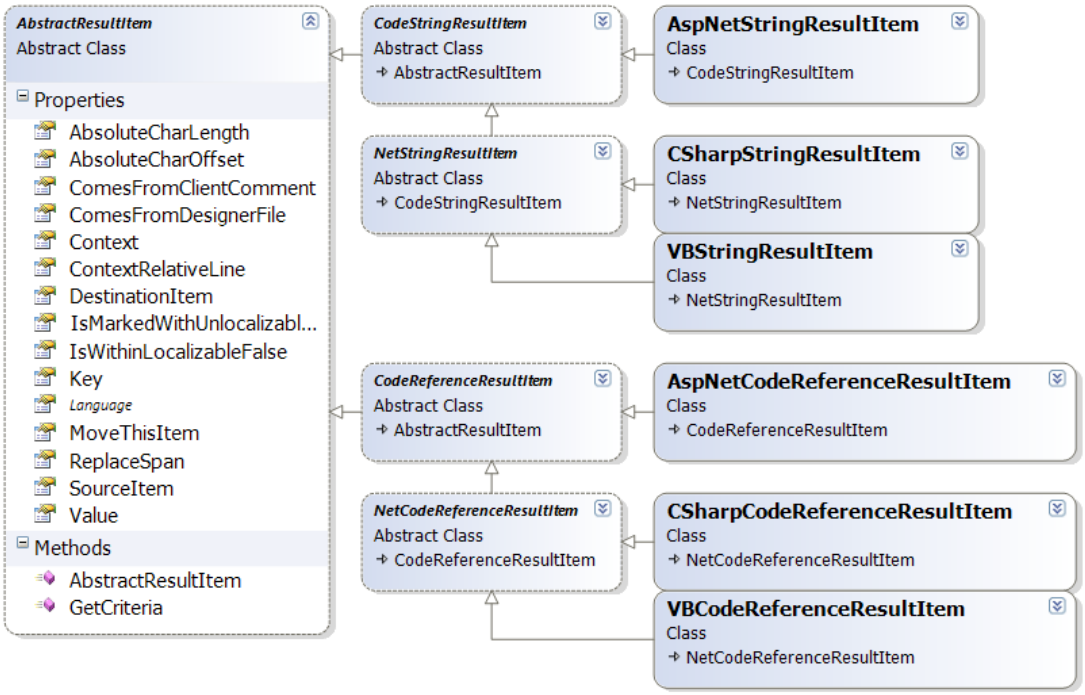


Figure 6: Class diagram of result items.

As seen from the diagrams above, a result item can represent either a string literal or a reference to a resource. Each result item contains:

- Position (line and column numbers, absolute char offset)
- Information about the code block in which it was found (namespace, class and method or variable names) or element (in case of ASP .NET)
- Resource key and value (either taken from the resource file in case of a reference or value of the string literal)
- Namespaces that affect the result item (those that are imported and those in which the result item is located)
- Detailed information about the object – whether it is a verbatim string, whether it comes from code block decorated with the `[Localizable(false)]` attribute, whether it comes from AspX element etc.
- Data specified later by the user in a tool window – the destination resource file, whether the item should be processed at all etc.

3.3.2 Tool Windows

The term "tool window" is used in the Visual Studio environment for framed containers which contain various functions and helpers, usually with a relation to the currently edited files. For example, everything listed in the "View" menu is a tool window.

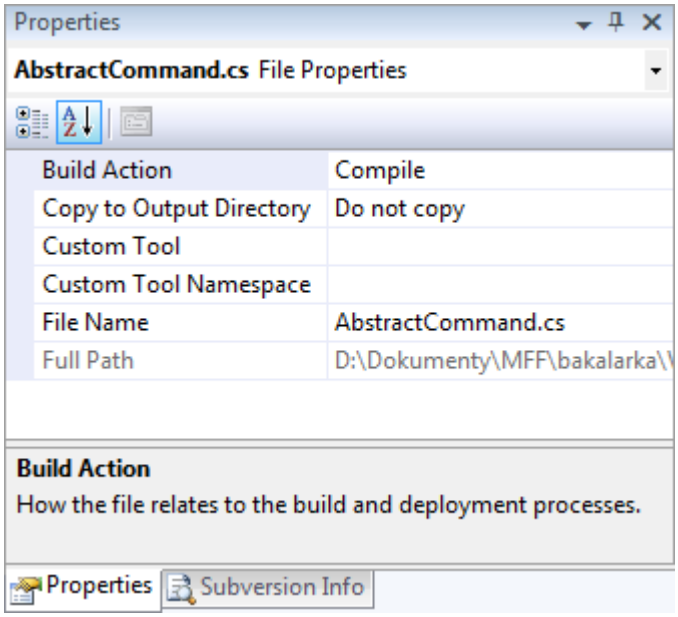


Figure 7: The Properties tool window in Visual Studio 2008.

The "batch" commands use tool windows as an intermediary phase to display found elements (result items) to the user. The tool windows share a generic base class whose type parameter is the type of the control that handles inner content; the frame and the toolbars are handled by the Visual Studio.

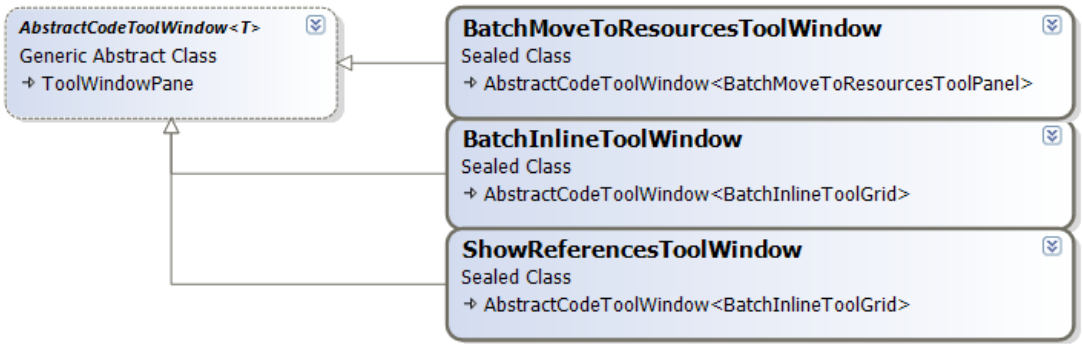


Figure 8: Class diagram of tool windows.

The class diagram points out one significant difference – while a grid is the only content of the `BatchInlineToolWindow`, the `BatchMoveToResourcesToolWindow` contains `BatchMoveToResourcesToolPanel`. This panel consists of `BatchMoveToResourcesToolGrid` and the panel displaying filter options. The `ShowReferencesToolWindow` is used by the ResX editor to display the list of references to selected resource entries. It is similar to `BatchInlineToolWindow`, but it is simpler – contains no toolbar and except for highlighting the result item in code supports no actions.

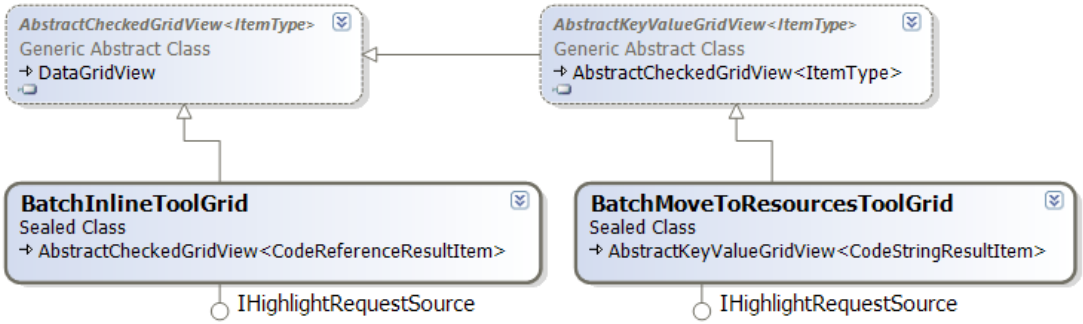


Figure 9: Class diagram of grids.

Both `AbstractCheckedGridView` and `AbstractKeyValueGridView` are located in VLlib. The first represents a grid with a checkbox column, whose header also behaves as a checkbox, making it possible to un-check all rows at once. The `AbstractKeyValueGridView` adds the validation of keys functionality, which will be discussed later in Section 3.6.

3.4 Code Explorers

The code explorers are responsible for examining given code file and decomposing it into methods' bodies. Because of the differences in handling strings and resource references (see 2.4.2 and 2.4.3) between the individual programming languages, we have to provide a distinct code explorer for every supported language.

3.4.1 C# and VB .NET

Instance of `FileCodeModel` can be obtained for any successfully compiled C# or VB .NET source code file. This interface makes it much easier to process the code since it contains the `CodeElement` objects, which represent a separate block of code, for example an import statement, class definition, field definition etc. These objects are nested in each other, creating an intuitive impression of a field being defined within a class, a class being defined within a namespace etc. Each `CodeElement` contains useful information such as name, modifiers (`private`, `const`), attributes and others. Unfortunately, it is not possible to access the whole code this way – `CodeElements` are not created for methods' bodies, making us use our own resources (*code lookupers*) to parse their content.

The code explorers recursively iterate through code's `CodeElements` and look for texts that can be passed to a code lookuper, i.e. texts possibly containing either string literals or references to resources. Specifically, such texts can origin from[8]:

- Method's body
- Property's getter and setter bodies
- Field initializer

While we can safely assume any method's body is a possible source of reference-able string literals, fields are different. If we want to be sure we can safely replace the string literal initializer with a reference to a resource, we must (and the code explorers do) check the following:

- The field must not have the `const` modifier
- If the field belongs to a `struct`, it must have the `static` modifier

The behavior of the code explorers is common for both C# and VB .NET, except one thing – C# lacks the concept of *modules*. A module is similar to a standard class with the `static` (or more precisely `Shared`) modifier (cannot create instances), with all its members being also `static` [30]. Modules are included in the

[8] Actually, strings can also be found elsewhere, as mentioned in 2.4.2. But neither of these cases is relevant for us, since these string literals cannot be replaced with a reference to a resource without producing a compilation error.

code explorer's search as well since they can contain both localizable string literals and references to resources.

3.4.2 ASP .NET

When attempting to parse AspX documents, we are not provided any help from the Visual Studio or any other API. We cannot use any XML parser, since AspX is not a valid XML (the "<%" tags are not supported). Moreover, our requirements for any such parser are quite steep – we need to parse whole content, correctly recognize code blocks, get accurate position of every element, attribute, output block and even plain text parts.

I found no library providing such functionality – so I have written my own. It is inspired by the SAX technology [31] in the XML. The parser (`VisualLocalizer.Library.AspX.Parser`) is given an implementation of the `IAspxHandler` interface, through which is its implementer informed about currently processed blocks and elements (so called *event-driven XML API*). The `IAspxHandler` interface provides all methods necessary for our project:

- `OnElementBegin(ElementContext)`
- `OnElementEnd(EndElementContext)`
- `OnPageDirective(DirectiveContext)`
- `OnCodeBlock(CodeBlockContext)`
- `OnOutputElement(OutputElementContext)`
- `OnPlainText(PlainTextContext)`

In the context of Visual Localizer, `AspNetCodeExplorer` implements `IAspxHandler`, so whenever it is necessary to process (explore) an AspX file, the parser is run on the file's plain text content and `AspNetCodeExplorer` therefore receives notifications about every significant object in the file.

When a code block or an output element occurs, the `AspNetCodeExplorer` passes its contents to a proper *code lookuper,* which has no need to know the code originated from AspX file – it is standard C# or VB .NET code[9]. There is an interesting feature (or bug?) in the Microsoft implementation of the AspX parser - since the parser itself does not understand the content within a code block,

[9] There are actually two ways of creating a code block within AspX document. The first one uses "<%" tags, the second <script runat="server"> elements.

determining the end of it may become a tricky task. For example, what if the code block looked like this:

```
<% string fakeEnd = "%>"; %>
```

Strictly speaking, the first occurrence of "%>" is located within a string, so it should be ignored and the code block should end with the second occurrence of this tag. However, the official Microsoft AspX parser accepts any occurrence of "%>", being within a string or not, as the end of the code block. Visual Localizer therefore uses the same behavior.

Another topic to discuss is the actual programming language in which the code block is written. The first option how to determine it is to check the code-behind file of the AspX document – that should be a valid C# or VB .NET source file and therefore its `FileCodeModel` could be used to determine the language. However, it is possible to create an AspX document with no code-behind file, which leaves us with the last option – the `Language` attribute in the `Page` or `Control` directive. Visual Localizer uses the value of this attribute to determine the programming language of the code blocks located in the document.

When a non-empty plain text block is reported, the *result item* is created right away.

When an element or a page directive is processed, the `AspNetCodeExplorer` creates *result items* from its attributes. However, this may lead to an error – imagine the following element:

```
<asp:Button runat="server" ID="b1" BackColor="Aqua" Text="Hello!"/>
```

Clearly not every attribute can be safely replaced with a reference to a resource and some of those which can are not likely to be localized. The `BackColor` attribute for example will hardly be a subject to localization. Creating a reference of any kind instead of the `runat` and `ID` attributes values would even cause a compile error.

Visual Localizer attempts to determine the data type of the attribute before it is reported as a *result item*. Only those attributes whose data type is `string` are listed in the result. The whole process is explained in the following section.

Let us assume for now that we are able to determine the attribute's data type. The fact that it is `string` does not guarantee that when replaced with a reference, no compilation error will occur. The `ID` attribute's type in the example above is `string`; however it cannot be replaced with a reference because `ID` has a special meaning for the compiler. Therefore Visual Localizer contains hard-coded list of attributes whose type is `string` yet they should not be reported as result items – this list contains only `ID`, `Name` and `runat` attributes, but it can be extended using "localization criteria" discussed in Section 3.7.

3.4.3 Determining Attribute's Data Type

Our task is clear – we are given the element's name (possibly with a prefix), its attribute's name and we want to determine the attribute's data type. This is of course relevant only for ASP .NET elements; standard HTML attributes are always treated as strings. ASP .NET elements are defined as classes with attributes being their properties. What we therefore need is to find the class defining the element and get the type of the property with the same name as the attribute.

First let us examine how exactly ASP .NET elements are defined. This is achieved in a configuration file called "web.config" which is by default located in the project's directory. To be more accurate, the configuration files form a hierarchy – in every project's subfolder there can be a configuration file that affects only source files in its directory and subdirectories, possibly overriding inherited settings. The complete hierarchy looks like this: [32]

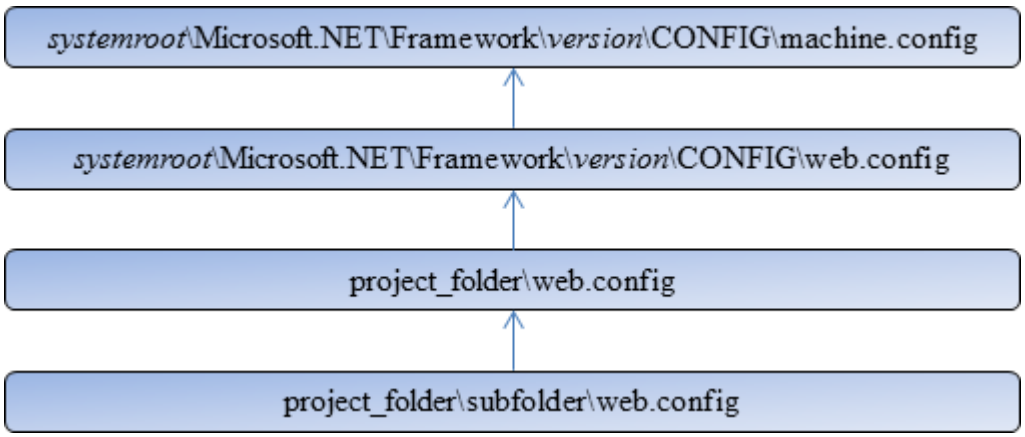


Figure 10: Hierarchy of the "web.config" configuration files.

The configuration files among other things contain section in which elements are defined, i.e. mapped to their defining classes. There are two ways how this can be done:

- assembly, namespace and prefix is specified
- tag name, prefix and a source file is specified

While the first approach can be useful for example when using third party libraries in which elements are defined, the latter is more convenient when declaring custom element types. Visual Localizer is able to process both cases.

Back to our initial task – we know the element's name, prefix and attribute's name and want to determine its data type. To solve this, the `WebConfig` class was created in VLlib. It loads the configuration files according to the hierarchy above and determines where to look for the element's definition, using tag prefix as a key to look up the right definition. If the result of this operation is an assembly, we can safely use *.NET Reflection* [19] to get the type – first the assembly is loaded using the `Assembly.Load()` method and then the `Assembly.GetType()` method is used to determine type of the property with name corresponding to the attribute's name.

If the element is a custom element created by a user (i.e. definition source file was specified), we cannot use reflection since it would be extremely slow[10]. Instead we use `FileCodeModel` to explore the definition file, looking for appropriate property. If such is found, we can extract its type thanks to its `CodeElement`.

Although all intermediate results are cached and reused whenever possible, the process of determining the data type of an attribute can be considered slow – therefore it can be turned off in the Visual Localizer settings (Section 4.8).

3.5 Code Lookupers

The code lookupers are responsible for looking for string literals or references to resources in a C# or VB .NET code.

[10] And unreliable as well – the project could be in a phase when it cannot be compiled and therefore the output assembly cannot be created.

3.5.1 String Literals

Recognizing string literals in C# is fairly straightforward – we only need to omit comments and correctly parse escape sequences to determine the string's end (see 2.4.2). In VB .NET however, the situation is slightly more complicated, paradoxically because its string literals are simpler and do not support escape sequences at all. If we need to create a string literal including control characters, in VB .NET we have no other choice but:

```
Dim s As String = "A" & vbCr & "B" & vbTab & ControlChars.Quote & "C"
```

This is equivalent to the C# code:

```
string s = "A\rB\t\"C";
```

Visual Localizer is aware of this issue and is able to concatenate such string expressions into a single *result item*, inserting control characters into the correct places. Specifically, these forms of control character input are supported (specified either with their namespace or not):

- `ControlChars`[11] class
- `Constants`[12] fields
- `Chr` and `ChrW` functions[13]

Whenever a code lookuper finds a *result item*, it attempts to concatenate it with the previous one by examining the code between them. If any of supported cases occurs (or their combination), the result items are merged, inserting respective control characters. This method has one flaw – it cannot detect control characters being added to the beginning or to the end of the string, because there is no other *result item* that would handle the merging. Any solution of this problem would cause the lookuper to work much more slowly; however, there is a simple workaround – place an empty string literal after (or before) the control character. That way it gets merged with the rest of the *result item*.

---

[11] http://msdn.microsoft.com/en-us/library/microsoft.visualbasic.controlchars.aspx
[12] http://msdn.microsoft.com/en-us/library/microsoft.visualbasic.constants.aspx
[13] http://msdn.microsoft.com/en-us/library/613dxh46.aspx

3.5.2 References to Resources

When looking for references to resources, a different approach is used, because unlike string literals, a reference is not clearly separated from the rest of the code. As the first step, all resource files that could possibly be referenced from the code are found. This includes resource files within the project itself and also resource files generated by the `PublicResXFileCodeGenerator` in the referenced projects. Then all possible references are collected and customized Aho-Corasick [33] algorithm is used to look them up in the code. Basic support for this functionality is implemented in the `Trie` class in VLlib.

There are few issues to solve when using this approach, all of them caused by the fact that the reference in the code may differ from the "official" reference created from the *designer class* (see 2.1) name and the resource key. The first thing that can cause this is the fact the resource key itself does not have to be a valid identifier for the programming language and therefore it cannot be part of a reference. If such situation occurs, the custom tool generating the *designer class* modifies the key by replacing invalid characters in order to create a valid identifier which can then be used in a reference. Visual Localizer uses the same algorithm to make the resource key valid and therefore the right references are looked up.

References in code may contain arbitrary whitespace around the dots separating the namespace from the class, the class from the key etc. In VB .NET, the underscore character can even occur within a reference as a line joining character. Both of these issues are solved quite easily, since whitespace can only occur on very specific places.

Finally, we need to resolve the qualified name of the reference. The process is described in the following section.

3.5.3 Namespace Resolution

Let us consider the following scenario:

```
using Application.ResourcesA;
using rc = Application.ResourcesC;

namespace Application {
    namespace ResourcesB {
        class Class1 {
            string s1 = Application.ResourcesA.Resource1.Key1;
            string s2 = Resource1.Key1;
            string s3 = rc.Resource1.Key1;
        }
    }
}
```

As it was already mentioned, the algorithm looks up references in the form of *DesignerClass.ResourceKey*. Although the namespace in which the designer class belongs is known, it cannot be part of the search expression since the reference in the code does not have to contain it.

Considering the example above, the algorithm contains the same search expression (`Resource1.Key1`) three times, but with three different meanings. Visual Localizer uses the same algorithm as Microsoft's compiler to determine in which namespace the reference's target belongs [21]:

1. If the reference is a qualified name and its type exists, we have found a result and quit.
2. If the reference contains an alias, we try to find its definition and create a qualified name from it; if the type exists, we have a result.
3. Otherwise we try to match the reference to namespaces declared in the same scope, starting at the innermost.
4. If no such type exists, we try to match it to the imported namespaces. If we find exactly one match, we have a result; otherwise we report an ambiguity error.

Project's `CodeModel` property is used to determine whether the specified type exists. All C#, VB .NET and web application ASP .NET projects have their `CodeModel` always available; however ASP .NET websites' code model behaves rather non-deterministically. Therefore, when creating a reference in a website project, a qualified name is always used to avoid name conflicts.

Moreover, in VB .NET is the situation slightly more complicated thanks to the `My` feature [34]. `My.Resources` contains all resource keys located within the implicit "Resources.resx" file. However, `My.Resources` is not a type; it is just a compiler's shortcut aiming to ease programmer's work. This makes `CodeModel` fail to report the type as existing. The only solution for this is to replace `My.Resources` part of a reference with the actual namespace and designer class – only then it can be found in the code model.

3.6 Tool Grids

So far we have discussed the process of searching for string literals and references to resource in code. When this process is completed, we need to display the results to a user – in "batch commands", this is a task for the tool grids, located in their tool windows[14].

The `BatchInlineToolGrid` is quite simple; it only supports sorting and checking and unchecking rows in order to remove irrelevant results, no values in the grid can be edited.

The `BatchMoveToResourcesToolGrid` on the other hand enables user to edit resource key and value as well as the destination resource file. Several issues are introduced by this functionality – we need to check for duplicate keys and we need to validate the key names to make sure we do not produce compilation error after the command executes.

First we need to check whether the destination resource file contains the resource key in the grid. If not, we can continue with the duplicate keys check. Otherwise, resource values are compared (the one in the grid and the one in the resource file). If these values are equal, there is no need to report an error – the existing key is simply referenced, no new resource entry is created.

The duplicate keys check algorithm compares two keys in the grid at each step and either leaves them, or marks them as conflicted. It is only run if both keys successfully passed the first test (in the paragraph above).

---

[14] "Ad-hoc commands" are much less demanding and do not need to be discussed in further details; the inline command actually does not display any intermediate results and performs the operation right away, while the move to resources command displays a simple dialog for just one *result item*.

1. Compare the keys, ignoring case. If they are not equal, they cannot be in conflict.
2. If they are equal, check their destination resource files. If they are not the same, the keys are not in conflict.
3. If both keys and their destination files are equal, compare resource values. If they are different, the keys are in conflict; otherwise they are not – the first occurrence adds a new resource entry and the second one only references it.

3.7 Localization Probability

Hardly all strings in source code are subjects to localization. Sometimes, something else than user output is saved in the variable with the string data type, for example files paths, SQL queries, identifiers or IP addresses. Ideally, the "batch move" command would be able to differentiate between strings that get eventually displayed to a user (and therefore should be localized) and strings that are only used to store values (and should be omitted). The perfect behavior cannot be achieved simply because there is no characteristic that deterministically distinguishes the two groups of string literals in code. Therefore Visual Localizer contains only heuristic approximation of such algorithm, called "localization probability".

"Localization probability" (LP) is displayed as a percentage value in the second column of the "batch move" tool window grid. It represents a suitability of the string literal for localization and was designed to quickly eliminate all string literals which seem not to contain culture-dependent value. By default, all string literals have 50% LP and certain criteria are used to decrease or increase the value.

3.7.1 Custom Criteria

To calculate LP, concept of "localization criteria[15]" is used. In this section, we will describe the more general variant of localization criteria, called "custom criteria". In the following section, we will discuss "common criteria", which are simpler and are based on "custom criteria". Since Visual Localizer enables

[15] Single form „criterion"

customization of both “custom” and “common” criteria (see Section 4.3.1), using them may rapidly increase effectiveness of the “batch move” command.

A custom localization criterion consists of three parts and takes the form:
If <<predicate>> (<<target>>) then <<action>>

To explain them, we will use following example:
*If the name of the class in which the string literal belongs matches “^.*Abstract.*$”, the LP should be lowered by 20.*

The first part of the criterion is the *target*, i.e. what is tested. In the example, “class name” is the target. Visual Localizer provides the following set of predefined targets:

- String literal value
- The name of the namespace the string literal was found in (if any)
- The name of the class the string literal was found in (if any)
- The name of the method the string literal was found in (if any)
- The name of the variable the string literal was found in (if any)
- The element name, if the string literal comes from AspX element or AspX plain text
- The element prefix, if the string literal comes from AspX element or AspX plain text
- The attribute, if the string literal comes from AspX element
- The line of code on which the string literal was found

The second part of localization criterion is the *predicate*, i.e. condition that the *target* must satisfy in order to pass the criterion. In the examples above “matches (regular expression)” is the predicate. These are the predefined predicates supported by Visual Localizer:

- Be null
- Contain no letters (e.g. “127.0.0.1”)
- Contain no whitespace
- Contain only capital letters and symbols (e.g. “COLUMN_ID”)
- Match specified regular expression
- Not match the specified regular expression

The last part of the criterion is the *action*, i.e. how should LP be modified, if the *predicate* is satisfied. In the example above, "lower the LP by 20" is the action. The list of available actions is as follows:

- Force localize – string literals matching the criterion are given a 100% LP
- Force NOT localize – string literals matching the criterion are given a 0% LP
- Set value – an additional value from -100 to +100 must be supplied, representing the likeliness of localizability of the string literal matching the criterion. Positive values raise LP, negative values make it smaller.
- Ignore – is equal to setting the value to 0

Clearly there may be situations in which various criteria are in conflict; we will discuss this as well as the exact calculation of the LP in Section 3.7.3. When Visual Localizer is installed, it contains several predefined custom criteria that were designed to cover most cases and fit most developers' needs; therefore it is possible to take advantage of the "localization probability" right away without creating the criteria by hand. More about these default settings can be found in Section 5.2.

3.7.2 Common Criteria

While the "custom criteria" consist of *the target*, *the predicate* and *the action*, in "common criteria" the target and the predicate are merged together and cannot be customized. That is, there are several predefined common criteria – they cannot be deleted, no common criteria can be added and only the *action* part can be edited. For example, these are some the common criteria available:

- String literal is a verbatim string
- String literal comes from [Localizable(false)] block
- String comes from ASP .NET plain text

The complete list can be viewed and customized in the Visual Localizer settings page.

3.7.3 Calculation

As it was already mentioned, one of the two purposes of the "localization criteria" is to calculate the "localization probability". Their second purpose (filter in the "batch move" tool window) is discussed in 4.3.1. In this section we will discuss the concrete algorithm for calculating LP.

If two localization criteria affecting one string literal are in conflict (i.e. one says the string must be localized while the other claims the opposite), following rules are successively applied:

1. If at least one criterion has the "Force localize" action, the string literal is given 100% LP.
2. Otherwise, if at least one criterion has the "Force NOT localize" action, the string literal is given 0% LP.
3. Otherwise the values are combined using specific formula. If no criterion affects the string literal, it is given 50% LP.

The "specific formula" mentioned in the third step presumes that several localization criteria (*LC* in the formula) affect the string literal, neither of which has the "Force localize" or the "Force NOT localize" action.

$$LP = \left(0.5 + \tan^{-1}\left(\frac{\sum_{LC} LC.Weight}{50}\right) \cdot \frac{1}{\pi}\right) \cdot 100$$

When creating the formula, I used the inverse tangent function, which has the two most desirable properties – its infinite limits are finite numbers and the function is increasing. This function is given a sum of localization criteria weights; the rest of the formula merely adjusts the range of the function so that the result would be a value between 0 and 100. If no criterion affects the string literal, i.e. the sum of `LC.Weight` is 0, the formula returns 50.

3.8 Commands Execution

Once we displayed the data to a user and let them modify it, we can execute the actual command, which either replaces string literals with references or vice versa. The implementation of this process depends on the state of the file from which the result item origins.

If the file is closed, all its contents are loaded into a memory buffer and the necessary modifications are performed. Once completed, the file is written back to disk. To optimize this process, the result items are first sorted according to their position in the file. If we did not do this, we would have to adjust items' position after each replacement, since the buffer's content would change.

If the file is opened, Visual Studio API functions are used to modify document's buffer. Moreover, a new undo unit is added to the document's undo stack, enabling thus to revert the operation as one atomic action. For the same reasons as with a closed file, the result items are replaced starting at the end of the file.

When executing the "move to resources" command, the string literals are unescaped (i.e. escape sequences are replaced with actual characters) before they are added to a resource file. Similarly when executing the "inline" command, the resource value is first modified to be correctly inserted into the code, without producing a compile error.

3.9 ResX Editor

The editor of ResX files provided by Visual Localizer can be almost completely separated from the commands functionality we discussed so far.

The first thing that must be pointed out is that implementing custom editor is not at all an easy task and the Microsoft's documentation does not make this process much simpler. Even though Visual Studio provides a graphical wizard for this task, what this wizard produces (almost 4,000 lines of code with doubtful meaning) can hardly be used in an actual development. [35]

3.9.1 Structure

Since I found the default editor created by the Visual Studio wizard unsuitable, I created a generic base class for editors in Visual Studio called `AbstractEditor` and located in VLlib. This class does the hard work of implementing the interfaces Visual Studio requires from every file editor. These interfaces take care of saving and loading file's buffers, registering the document's view in the context of other opened files and processing Visual Studio commands. The `AbstractEditor` reduces this to basic and expectable tasks, wrapping the COM-based interfaces to a single object with clearly defined methods.

The `ResXEditor` class is derived from the `AbstractEditor` and provides functionality specific for resource files, such as loading and saving the files using `ResXResourceReader` and `ResXResourceWriter`. Commands issued from the Visual Studio (such as copy, paste, open file…) are recognized and passed to the underlying GUI objects to be executed. The GUI itself is represented by the `ResXEditorControl` class. This class paints the editor's toolbar and the tab control, whose tabs correspond to the types of resource content. It is also responsible for distributing the resources among these tabs on file load and collecting them back on file save.

The tabs can be separated into two categories – the first category is used to display media resources and is based on the `ListView` class. Images, icons, sounds and files in general are handled by these tabs. The second category of the tabs displays strings and other strongly typed resources. These two tabs use grids to display and edit content. The complete class diagram is demonstrated on Figure 11.

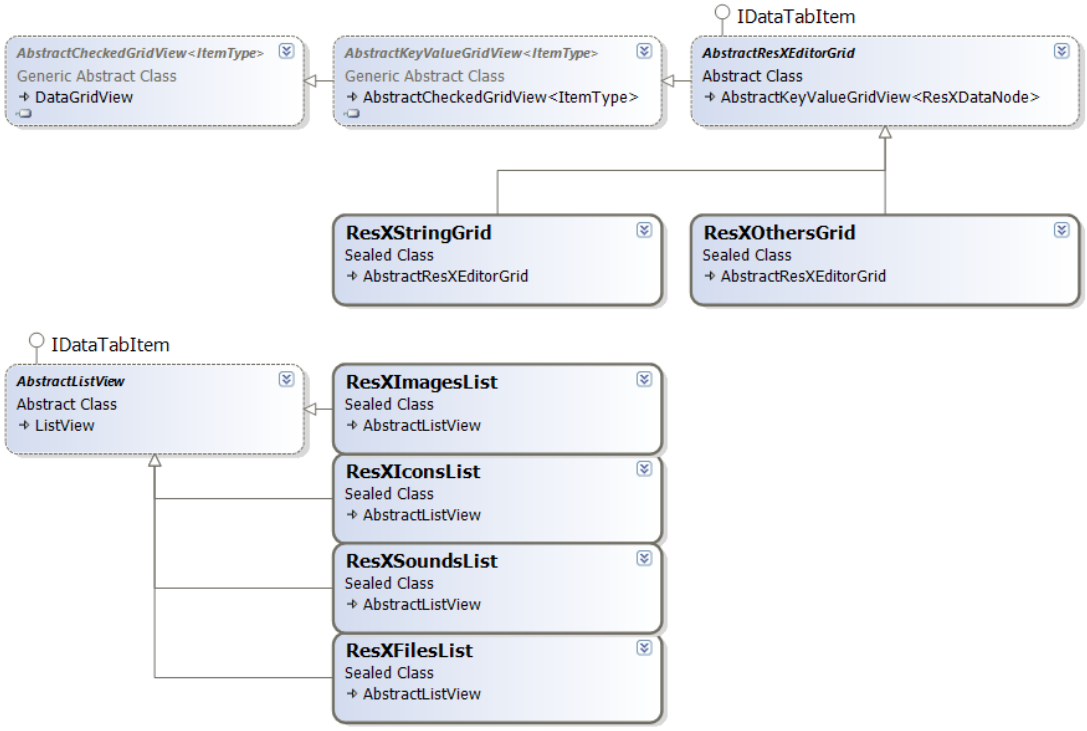


Figure 11: Diagram of the ResX editor structure.

The following schema shows the relation between various ResX editor components. Since the hard work of implementing Visual Studio interfaces is done by the `AbstractEditor` class, the `ResXEditor` is actually quite simple – it only contains methods for loading and saving files in the ResX format and maps the Visual Studio commands to their handlers in `ResXEditorControl`.

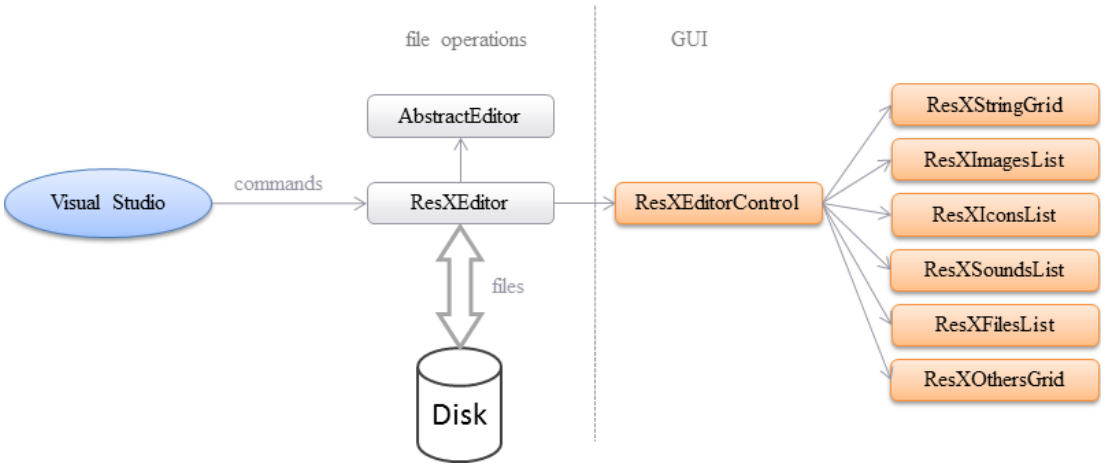


Figure 12: Data and commands flow in the editor of ResX files.

Implementation of most of the editor's functions is fairly straightforward. The key name conflict resolution system is the same as in tool grids (Section 3.6); `ResXStringGrid`, `ResXOthersGrid` and `BatchMoveToResourcesToolGrid` actually share a base class which provides this functionality.

We will now focus on the two features of the ResX editor which deserve further attention – the grid of strongly typed resources (`ResXOthersGrid`) and the mechanism of collecting references to resources stored in the file.

3.9.2 The "Others" Tab

All the tabs in the ResX editor window have clearly defined content – strings belong into the "Strings" tab, images and icons are stored in the tabs with corresponding names. Both linked and embedded file resources not fitting into any of the previous categories are stored in the "Files" tab.

However, the ResX file format makes it possible to store content of any .NET type. Any such resource entries are stored in the "Others" tab, which displays the full name of the type along with its value. Visual Localizer uses the same algorithms for managing this kind of content as the .NET Framework; particularly, `TypeConverters` are used to convert the actual strongly typed resource value to/from its string representation in the grid. Since the user can manipulate both with the resource type and its value, additional validating mechanisms are employed to ensure data integrity – it is therefore not possible to store such a resource value, which cannot be converted to the corresponding object. See Section 4.7.4 for further details.

3.9.3 Looking up References

An important part of the editor is the `ReferenceLookuperThread` – this thread runs periodically in background with a low priority, using almost the same algorithm as the "batch inline" command to look for the references to the resource entries. The interval in which is the thread run can be customized in the Visual Localizer settings and is 10 seconds by default. This periodical checking of code means that at every moment, a list of references to each resource entry is known; of course there is a slight delay caused by the lookuper thread interval. Several functions take advantage of this fact – first, the references count is displayed directly in the editor. The "Show references" function (available in the context menu) displays the list of references so

that user can find them in code. Also, when a resource key is renamed, all references to this key are changed in their code files as well.

As we discussed in Section 3.3.1, the code-exploring commands (and therefore the `ReferenceLookuperThread` as well) use `FileCodeModel` to examine source code files. Although this works perfectly in C#, VB .NET and even ASP .NET web application projects, the behavior of `FileCodeModel` in ASP .NET websites is slightly different. Specifically, `FileCodeModel` instance cannot be obtained for code-behind files of AspX documents without opening them. This rather unpleasant feature (bug?) has only one possible solution – open the files in the background, without displaying the window to the user.

The "batch move" and "batch inline" commands work this way – when they come across a file which has no use `FileCodeModel`, they open it in a hidden editor window, process the code model and then close it. The opening of the files however causes flickering of the mouse cursor; this is not a problem for the "batch move" and "batch inline" commands, since the other work they do causes flickering as well. However, if the same solution was employed by the ResX editor, it would cause the mouse flicker every few seconds, which is unacceptable.

Therefore the ResX editor uses following procedure to handle the issue:

1. On editor startup (file opening), explore the whole code, open files on background if necessary. This will cause some flickering, but since the whole editor window is being painted at that time, it is acceptable. Close the hidden windows when finished.
2. If a file without `FileCodeModel` (and therefore closed) is found during the later periodical runs, freeze the result items originating from this file (i.e. do not remove them from the list obtained in step 1) and add the file to the list of "frozen" files.
3. The only way how could references in the problematic files change is when the files are edited, i.e. opened, saved and closed. Visual Localizer is notified about the file close event; it checks the list of "frozen" files from the step 2 and if the file is found in this list, the ResX editor is notified to update the references from this file on next run of the `ReferenceLookuperThread`. This is again done by opening the file in a hidden editor window.

3.10 VLTranslat

The VLTranslat library is not dependent on any other part of Visual Localizer or Visual Studio. It provides an interface through which well-known translation services can be easily utilized. The functionality of this library is used by the "Global Translate" command (see 4.6) and the translate function in the ResX editor (see 4.7.5). Translation services of Microsoft, Google and MyMemory are implemented – we will briefly describe the differences between them.

The Microsoft Translator [6] web service is passed the source language, the target language and the text to translate in a request header. It returns a well-formed XML result, which is then parsed using `XmlDocument` parser. To make this work, a special ID must be passed along with request parameters – this ID identifies the user making the request and can be obtained be registering on Microsoft's website[16]. The registration is free of charge as well as translating up to 2,000,000 characters a month – larger volumes are charged. Visual Localizer enables user to set their ID in the settings page.

Google provides API to comfortably access its Translate service [5], however, usage of this API costs money. I therefore decided to use directly the web service, which returns response formatted in JSON (JavaScript Object Notation [36]). Since the schema of the response is quite simple, no third party library is used to parse it – a simple JSON parser is included directly in VLTranslat.

MyMemory [7] is probably the least known of implemented translation services – unlike Microsoft and Google which use machine-translation algorithms, MyMemory uses human translators and multi-lingual web content to match sentences in one language to the same sentence in a second language. It does not require any registration. Like Microsoft, it returns XML based responses, which are parsed by VLTranslat using `XmlDocument`.

Especially when using these services from the ResX editor, the text that is being translated may contain format placeholders ({n}). The translation services however may have problems with them; sometimes they put all of them at the end of the text, sometimes they reorder them and worst of all – they may render the string invalid by inserting whitespace around the brackets. Any of this is highly undesirable behavior, therefore it is possible to turn on the "optimize" function (in Visual

[16] https://datamarket.azure.com/dataset/bing/microsofttranslator

Localizer settings) which temporarily replaces the placeholders with random numbers and puts them back after the translation. Usually, the translation services do not attempt to modify numbers.

4 User Interface

In the following sections we will discuss Visual Localizer's features from user's point of view. We will describe each command and editor's function and how they can be customized using Visual Localizer settings page.

4.1 Installation

The Visual Localizer extension is shipped as MSI (Microsoft Windows Installer) package. Since the installer modifies Windows registry in the HKLM[17] hive, it must be run with administrator privileges. Also, .NET version at least 3.5 is required. All instances of Visual Studio should be first terminated before running the Visual Localizer installer.

In the first step, the installer lets user pick the versions of Visual Studio in which the package should be registered. Visual Studio 2008, Visual Studio 2010 and Visual Studio 2012 are supported; because of the Microsoft policy, it is not possible to install any extension to the Express editions of Visual Studio.

Finally, installation directory is specified. No other actions are required; the installer handles the registration process described in Section 3.2 by itself. If the installation is completed successfully, Visual Localizer appears in the list of Visual Studio installed products, available in Help/About.

4.2 Move to Resources Command

The "move to resources" command is invoked from the code window context menu. It assumes user clicked on a string literal in code – if this assumption fails or the string literal cannot be moved for other reasons (`const` modifier, for example) an error is displayed.

Otherwise a dialog pops up, allowing user to specify details of the operation. The resource key and value can be modified as well as destination resource file. Inputted data are validated after every change – presence of the resource key in the

[17] HKEY_LOCAL_MACHINE

selected resource file is checked and depending on their resource values, either an error is displayed or an option to reference or overwrite the resource entry is offered.

Certain resource files in the list may have "(internal)" written next to them – this indicates that the resource file is located in another project and its designer class has the `internal` (or `Friend`) modifier. Referencing such resource file is possible, but will result in a "protection level" error of the compiler.

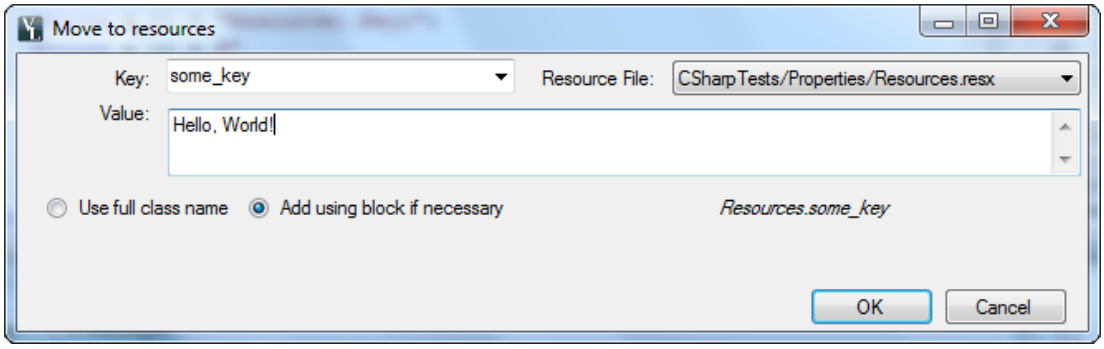


Figure 13: The dialog displayed during "move to resources" command.

The dialog also offers selection of the *namespace policy* – a choice between using a qualified reference name (includes namespace) or using an unqualified name and adding an import statement if necessary. However, user's choice may be overridden by Visual Localizer if producing such reference would cause an error – for example, if the new unqualified reference would in the context of the current code block actually point to something else. A namespace resolution algorithm described in Section 3.5.3 is used to detect such situations.

While in C# and VB .NET the newly created reference has just one possible form, in AspX code it must be decided between several options. If the referenced string literal is located in a website project, a resource expression ("<%$") is used. Otherwise, standard output element ("<%=") is produced. Moreover, if the string literal is actually a plain text between AspX elements, the reference is wrapped in the `<asp:Literal>` element.

Pressing "OK" in the dialog replaces the string literal with the reference and adds this operation as an atomic unit to the undo stack of the document.

4.3 Batch Move to Resources Command

When invoked from the code context menu, the command processes the active document. If the Solution Explorer's menu was used, all selected project items are recursively processed.

The command examines the specified code files, looking for localizable string literals. It ignores the code that is commented out, skips empty strings and also those string literals that cannot be referenced (`const` modifier for example) are ignored. A list of *result items* is created from the strings that were found. Each result item contains data such as position, whether the string literal is a verbatim string, whether it comes from a designer file etc. These data are utilized for calculating the "localization probability" of the string literal (Section 3.7) and also in the filter panel of the "batch move" tool window (Section 4.3.1).

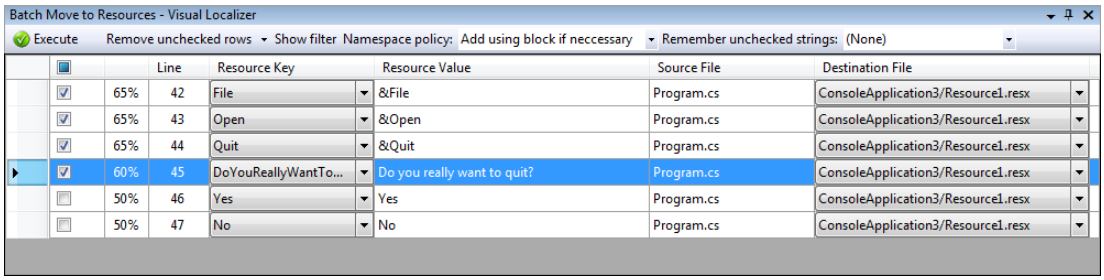


Figure 14: The tool window for the "batch move" command.

The list of result items is then displayed to the user in a tool window, docked to the bottom of the Visual Studio window by default. In the following paragraphs we will describe features of the "batch move" tool window and how it can be used to make the batch-processing quick and effective.

Selecting only relevant string literals to be moved to resource files is achieved using checkboxes in the first column. To make this easier, the grid can be sorted by any of its columns and also, the grid's context menu contains option to check/uncheck all currently selected rows.

The destination resource file, the key and value can be specified (data are validated whenever changed and potential name conflicts, discussed in Section 3.6, are displayed). Again, the grid's context menu can be used to choose the destination resource file for all currently selected rows. The options available in the menu are intersection of possible destination files of all selected rows.

When a grid's row is double-clicked, corresponding string literal is highlighted in its (newly opened) code window.

The *namespace policy* can be customized. Either fully qualified references are created or unqualified references are used and import statements are added if necessary, using exactly the same behavior as the "move" command in Section 4.2.

Also, the unchecked string literals policy can be selected. By default, the unchecked rows of the grid are ignored – however they can also be marked in the source code so that future "batch move" command would ignore them. A special comment is used for this purpose, called "VL_NO_LOC". If this option is used, the comment will be inserted before every unchecked string literal in the grid. When the "batch move" command processes the document in the future, thusly marked string literals will appear in the grid, but can be easily removed using a *filter*. However, since neither VB .NET nor ASP .NET support multiline comments necessary for this operation, this option is effective only in C# source code.

The unchecked rows from the grid can be whenever removed for better clarity (and possibly returned back). Corresponding buttons in the toolbar are used to achieve this.

The *filter* can be used to select only relevant string literals for localization. This will be discussed in the following section, as well as the "localization probability" displayed in the second column.

When the "Execute" button is pressed, all checked string literals are replaced with references in their respective source code files and corresponding resources are added to the resource files. For each such replacement a new undo unit is placed up the undo stack of the document[18]. If the "VL_NO_LOC" policy was selected, the "/*VL_NO_LOC*/" comment is inserted before every unchecked C# string literal; also this action creates a new undo unit.

All the files used in the tool window (either as source files of the string literals or resource files) are locked while the window is displayed. This is necessary to ensure data integrity – the result items contain absolute positions of the strings in their source files, so modifying the file may cause the operation to fail.

[18] This is of course possible only for opened files.

4.3.1 Filter

The purpose of the filter is to remove irrelevant string literals from the grid and leave only those that are really a subject to localization. The filter works with a set of "localization criteria". In Section 3.7 we described the localization criteria in detail and we also explained the way "localization probability" is calculated; in this section, we will focus on how they can be edited and how they are utilized by the filter.

Localization criteria can be customized in the Visual Localizer settings page, accessible through Tools/Options. The "common criteria" are displayed in the grid, which enables user to edit the *action* part of the criterion. The "custom criteria" can be added using the "Add" button at the bottom of the settings page – a full set of properties (i.e. *target*, *condition* and *action*) must be specified to safely save the settings.

Using these options, user can create a very specific set of criteria to make batch localization much easier process. As it was mentioned above, the criteria are also used to calculate "localization probability". More specifically, when a new result item is added to the "batch move" tool window, its LP is calculated and if it is at least 50%, the row is by default checked. User can then use LP as an indicator whether the string literal should be localized (for example sort the grid according to this column and check only rows with certain LP).

Moreover, the set of criteria specified in the settings page is copied and displayed in the filter panel above the grid. This panel contains all the criteria as well as option to modify their *actions* just for this instance of "batch move" tool window. Any changes made in this panel are instantly reflected in the LP of the grid's rows. To make this even more effective, more *actions* are available in the filter panel combo boxes:

- Check the rows matching the criterion
- Uncheck the rows matching the criterion
- Check the rows matching the criterion and remove the other (unchecked) rows
- Remove the rows matching the criterion

Any rows removed during these operations can be returned back to the grid using the "Restore unchecked" button in the tool window's toolbar. Changing the

action of a criterion in filter panel will not be saved – it is only valid for this instance of the tool window.

Even when the tool window is already displayed, user can open the Visual Localizer settings page and edit (add, remove) the criteria. More precisely, changing the action of the criteria will not have any immediate effect – the criteria used in the filter panel are *copy* of the ones in the settings. However, one can add new custom criteria and those will be immediately displayed in the filter panel, ready to be used.

Visual Localizer contains several custom criteria by default. These criteria were created to fit most developer's needs and project conventions. More about this can be found in Section 5.2.

4.4 Inline Command

As well as the "move to resources" command, the inline command is invoked from the code context menu. It assumes user clicked on a reference to resource that should be replaced with a hard-coded string literal taken from the resource file. If such assumption fails, i.e. there is no result item on the place user clicked, an error is displayed.

Otherwise, the reference is replaced and corresponding undo unit is added to the document's undo stack. The resource entry itself stays unmodified. Even if there are culture-specific versions of the resource file, the inline value is always taken from the culture-neutral one.

Since the value of the resource stored in the resource file may contain special characters, such as double quotes or newlines, Visual Localizer escapes these characters with respect to the current programming language syntax before inserting the string into the code.

4.5 Batch Inline Command

The “batch inline” command can be invoked either from the code context menu (the active document is processed), or from the Solution Explorer’s context menu, in which case all selected project items are recursively processed.

The command explores the code, looking for references to resources that can be replaced with hard-coded string literals. The same rules as in the “inline” command are applied. All found results are displayed on the “batch inline” tool window, which is similar to the “batch move” version, but offers fewer options since it is likely to be used less often. No value in the grid can be edited; it is only possible to check/uncheck the references that will be replaced with hard-coded strings.

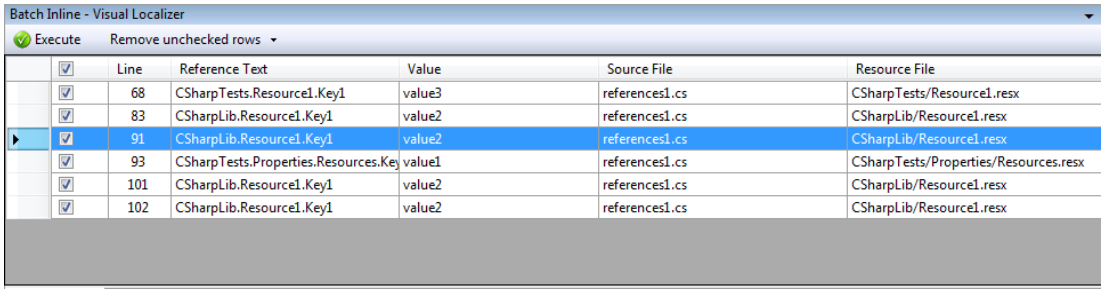


Figure 15: The tool window used in the "batch inline" command.

When the “Execute” button is pressed, all checked rows are replaced in their respective source code files, creating a new undo unit if possible (the file is opened). The “Remove unchecked rows” and “Restore unchecked rows” buttons work the same as in the “batch move” tool window.

4.6 Global Translate Command

Unlike the commands discussed above, the “global translate” command does not work with source code files. Its purpose is to enable user to perform batch translation of selected resource files. Invoked from the Solution Explorer’s context menu, the command scans the selected project items, looking for ResX files.

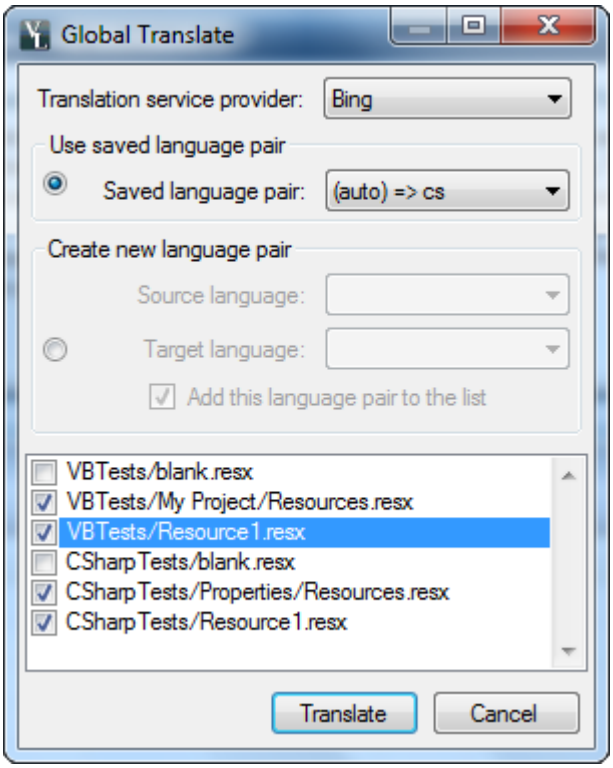


Figure 16: The "global translate" command dialog.

All found resource files are displayed in a dialog, in which other details of the operation can be specified. It is necessary to select the source and target language for the translation – either a saved language pair (created either in the settings page or the ResX editor) can be used, or a new one can be created and optionally saved along with the others. Also the translation service must be specified, keeping in mind that using Microsoft Translator requires valid *AppID* (see Section 3.10) to be inputted in the Visual Localizer settings page (if the *AppID* field is empty, the Bing service will not even appear in the list of available translation services).

When the “Translate” button is clicked, selected translation service is used to translate all string resources in the checked resource files. If the resource file is opened in the moment of performing this operation, new undo units are added to its undo stack as appropriate.

4.7 ResX Editor

Visual Localizer provides custom editor of ResX files, which is more comfortable to work with and offers more functions than the default Visual Studio editor. The editor is set as default for opening ".resx" files during the installation of Visual Localizer.

4.7.1 Overview

The editor's window consists of a toolbar and six tabs, each corresponding to a possible resource value content type. Each resource entry is identified by the resource key, which must be unique within the file and is used when referencing the resource entry from code. If the value of the resource entry is *string*, it is placed in the "Strings" tab, which contains a grid similar to the one used in the "batch move" tool window. Images (i.e. bitmaps), icons and sounds are placed in corresponding tabs; the rest of the file types is placed in the "Files" tab. The last mentioned resources will be commonly referenced as "media resources". Since the ResX file format enables to store strongly typed data (integers, structures etc.), the ResX editor uses the "Others" tab to store such content.

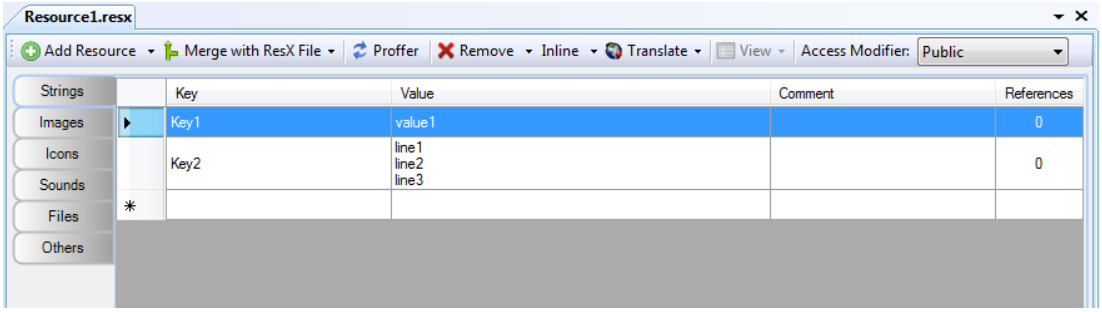


Figure 17: GUI of the ResX editor.

The editor enables user to perform all kinds of operations with the resource entries. Some of the operations can be invoked from the editor's toolbar; all of them are available in the context menu. Almost all of these operations can be undone using Visual Studio undo stack for the edited file. When performing an irreversible operation, a confirmation dialog is always displayed. We will describe available functions in the following sections.

4.7.2 Adding Resources

There are several ways of adding a new resource entry to the editor. First, the "Add Resource" button on the toolbar can be used to select files that will be linked to the resource file. The button also enables user to create a new string (by adding a row to the strings grid) or a new image file.

Second, if the clipboard currently contains list of file paths (e.g. from Windows file explorer or the Solution Explorer), the list can be pasted into the editor – the files will be sorted into appropriate tabs as necessary. The same effect can be achieved via the "drag'n'drop" operation – again, list of file paths is the desirable content. Finally, when working with string or strongly typed resources, the tab-separated text format is used to place data to the clipboard. Thus, it is for example possible to copy data to and from Microsoft Excel.

When adding a new file resource to the resource file which is not a part of the solution, the new resource is *embedded* rather than *linked* (see Section 4.7.7 for differences between linking and embedding).

4.7.3 Removing Resources

When removing a string or embedded resource, the corresponding entry is just deleted from the resource file. However, there are three versions of the remove process for linked media resource entries.

First, the entry is removed from the resource file, but the referenced file stays untouched as a part of the project.

Second, the entry is removed and the referenced file is excluded from the project. The file remains on disk.

And finally, the entry is removed and the referenced file deleted from disk. This operation cannot be undone for obvious reasons.

4.7.4 Modification of Existing Resources

When editing string or "other" resources, the value and the comment can be modified directly in the grid. To edit the comment of a media resource, a context menu on the resource item must be invoked and corresponding dialog selected.

When editing a resource from the "Others" tab, not only a resource value but also a resource type can be specified. Clearly, these two values must match – it is not possible to save a resource with invalid value. Unlike the resource value, the resource type is not edited directly in the grid, but using a special dialog which gets invoked whenever user attempts to edit the respective cell. In this dialog, the type is specified by its assembly and its full name. The list of available assemblies consists of assemblies currently loaded in the current application domain [37]. This feature is another big advantage of the Visual Localizer ResX editor – the default VS editor does not enable to add this kind of resources and not even change their types.

If the resource key is changed, the editor performs "pseudo-refactoring" of the source code – every reference to the resource is updated to point to the new key. Since the editor runs the reference-searching thread in periods, it would be easy to create an error this way – if two successive key renames would be performed, the thread would not have enough time to catch up and the references would be improperly renamed. Therefore, when renaming the references, their new position and text is updated right away.

4.7.5 Translation

Values of the resource entries in the strings grid can be translated using Microsoft Translator (requires *AppID* to be specified in the settings), Google Translate or MyMemory. Necessary argument for this operation is the language pair – a combination of source and target language. These can be added either on the settings page or using the "New language pair" button available in the translation providers menu.

All currently selected rows of the strings grid are translated; for each modifying action there is a new undo unit added to the undo stack.

4.7.6 Inlining

The “Inline” action invoked from the editor works the same as the “batch inline” command. However, once a reference is inlined, it cannot be re-found, therefore it is not possible to undo this action. Another reason for this is that the source code files may have been edited after the inline operation, so any existing result items are rendered invalid.

4.7.7 Embedded vs. Linked Resources

There are two ways of adding a media resource entry to a ResX file. The first way is called the *embedded* resource – the ResX file actually contains the media resource, in a text representation of its binary data. The second way, called a *linked resource*, stores only a reference to the media resource in the ResX file [11]. The Visual Localizer-provided ResX editor can work with both representations of data and is also able to transfer between them – the respective commands can be found in the context menu.

Also, while it is possible to open a linked media resource (via double-click), an embedded resource cannot be opened since there is no file for the editor.

By default, when adding a media resource to the ResX file which is a part of the solution, the resource entry is created as *linked*. If the ResX file stands alone, the resource is *embedded*.

4.7.8 Synchronization

The goal of the synchronization feature is to help developers keep multiple versions of resource files for various cultures synchronized, i.e. the resource files should contain the same resource entries (same keys) and differ only in values. The resource files structure and variants are discussed in Section 2.1.

When editing a culture-neutral resource file, the “Proffer” button is available in the toolbar. Clicking this button causes Visual Localizer to search the file’s directory and collect all culture-specific versions of this file. These files are then updated, i.e. presence of each resource key from the culture-neutral file is checked and added if not present.

When editing a culture-specific resource file, the button says “Update” and does almost the same, but only for the currently edited file. That is, its culture-neutral version is found (if exists) and missing resource entries are added.

The culture-specific resource files usually contain only *string* resources (the media resources are usually culture-independent and placed only in the culture-neutral file). Therefore the synchronization feature only works with string resources and completely ignores all media resources.

While all commands and editor functions so far ignored the letter case of the resource keys, the synchronization commands are case-sensitive. This is because if a key in a culture-specific file and its equivalent in the culture-neutral file differed (even only in case), .NET would not be able to match them and use them in translations.

4.7.9 Merging

The merging feature is a simpler version of the synchronization feature; another resource file is selected and its data are copied to the current file, even if already present. Unlike synchronization, this feature works with whole resource files, i.e. string resources and strongly typed resources as well as media resources.

4.8 Settings

Visual Localizer settings are completely integrated in the Visual Studio settings model. Thus, it is possible to edit the Visual Localizer settings among all others in the “Options” page. Also, when importing or exporting settings from Visual Studio, Visual Localizer settings are included in the process, making it possible to backup or transfer the settings as required.

There are two categories of settings; each of them has its own subpage in the settings window. First, the editor settings:

- “Reference update interval” – specifies interval in which the `ReferenceLookuperThread` is periodically run.
- “Invalid key name policy” – this affects both editor and the “move to resources” commands. It specifies the action to take when user inputs an invalid identifier for a resource key.

- "Bing AppID" – it is not necessary to specify this value, however without it, using Microsoft Translator service is not possible.
- "Optimize format strings for translation" – determines whether the format placeholders in strings are temporarily replaced with numbers when using a translation service. This issue is discussed in Section 3.10.
- "Language pairs" – the list of translation language pairs, used whenever working with translation services.

The second category of settings concerns only the "batch move to resources" and the "batch inline" tool window, since it specifies settings for the filter panel and the result items grid. How the filter works has already been explained in Section 4.3.1, now we will describe only the two remaining options:

- "Show context column" – the tool window grids may contain additional column providing the context for the result item, i.e. a few lines of code around the string literal or the reference to resource. Displaying this column may render the grid slightly confusing; therefore the column can be hidden.
- "Determine types of attributes in ASP .NET elements" – use .NET Reflection and `FileCodeModel` to filter out those string literals which are located within an attribute with other data type than string. The data type resolution system is discussed in Section 3.4.3.

All changes made in this settings page and confirmed are effective immediately, i.e. all open editors, tool windows etc. are revalidated and updated with the new settings.

5 Evaluation

In the previous chapters we described what Visual Localizer does and how it is achieved. This chapter is dedicated to proving that Visual Localizer works correctly and can be used in real-life projects.

5.1 Testing

Visual Localizer is quite a complex project and as such it should be tested. However, only certain parts of the application are suitable for automated tests, the other parts heavily work with GUI and therefore it is easier to test them by hand. All the automatic tests are located in the "VLUnitTests" project.

Classic unit tests are rather unsuitable for testing Visual Localizer, because they usually test separate methods or single classes. However, almost all classes and methods in Visual Localizer require context of a running Visual Studio instance. Almost no method in Visual Localizer can be used as standalone, without any others. There are exceptions of course and for these, classic unit tests were created – the AspX parser for example, or the VLTranslat library.

For the "(batch) move to resources" command, "(batch) inline" command and their execution, tests simulating user behavior were created. These tests are configured to be run in the context of Visual Studio (the "VLUnitTestContextSolution") and they test the command as a black box – they run it on known data and compare the results with the expected ones. This way, all result item data can be verified.

There are several issues concerning this – the testing methods have to programmatically control the testing Visual Studio instance, which seems to non-deterministically reject this kind of access. Therefore, tests must be run in smaller batches and in case of failed test, it must be differentiated between the error in the test and error in the tested command. Particularly `FileCodeModel` seems to be having problems when accessed from the testing environment.

5.2 Localization Criteria Optimization

In this section we will examine the "batch move to resources" command from another point of view. We have already discussed the "localization probability" (see 3.7) and how can filters be used to effectively select only relevant strings for

localization (see 4.3.1). Now we will focus on the default settings for the criteria, which is essential for LP to be useful at all.

Our aim regarding this part of the thesis was to ease the process of localization of those projects where a large code base already exists and the source codes were not written with localization in mind, i.e. they contain a lot of hard-coded strings in the source files. The developer, who wants to localize such project, must look through all the source files, examine the strings and identify those, which should be localized and move them to resources. Visual Localizer addresses this problem with its "batch move" command and "localization probability" characteristic. The "batch move" command not only moves strings to resources, but it should also help the user to distinguish the strings which should be localized from the rest. In this subsection, I will evaluate Visual Localizer on several real projects to see how it meets its goal.

I used several random projects from Codeplex[19] to examine their code and see what values string variables usually contain. The goal of this was to create default set of criteria that would take advantage of usual conventions, assigning high localization probability to strings that are most likely subjects to localization and vice versa.

Following patterns are recognized to lower the LP:

- String literal comes from a code block decorated with `[Localizable(false)]`
- String comes from client-side comment of AspX code
- String was previously rejected in the "batch move" tool window and is now prefixed with the VL_NO_LOC comment (Section 4.3)
- SQL queries (strings starting with SELECT, UPDATE, DELETE…)
- URLs (string starting with "http://", "https://" or "ftp://")
- ASP .NET project relative paths (strings starting with "~/" or "~\")
- IPv4 and IPv6 addresses, email addresses
- CamelCase strings (contain no whitespace characters and words are separated with different case of letters – identifiers are often named this way)
- References ("word.word.word")

[19] https://codetextbox.codeplex.com/
https://scintillanet.codeplex.com/
http://xmlexplorer.codeplex.com/
https://naudio.codeplex.com/
https://prakashjha.codeplex.com/

- Too short strings (less than 3 characters)
- CSS styles (in ASP .NET `<style>` element)
- Strings used (probably) as names (“object.Name = <string>”)

Visual Localizer considers the patterns above as probably non-localizable, but that does not mean they should stay hard-coded in all cases. For example the email addresses and IP addresses should be located in some kind of settings file or configuration file – but that is a matter of good coding style and not a subject of our work, we focus on strings which are subjects to localization.

The following patterns increase the localization probability:

- Multiline strings
- Strings containing whitespace characters
- Strings containing formatting placeholders ({0})
- Menu strings containing “&”
- String containing national characters (for example ‘ř’ in Czech)

Visual Localizer uses formula described in Section 3.7.3 to merge all these patterns into a single probability value. This value should serve as a clear indicator of whether the string should be localized. During the testing, I analyzed the source code and manually decided, whether each string should be localized or not (this is part of the work Visual Localizer aims to eliminate). Secondly, I used the default criteria and let Visual Localizer assign LP. For each string, there are four possible outcomes:

- *Correctly localized*, i.e. the computed LP was *above* the threshold for a string that *should* be localized
- *Incorrectly localized*, i.e. the computed LP was *above* the threshold for a string that *should not* be localized
- *Correctly rejected*, i.e. the computed LP was *below* the threshold for a string that *should not* be localized
- *Incorrectly rejected* , i.e. the computed LP was *below* the threshold for a string that *should* be localized

The results are shown on the following graph:

	Strings count	Code size	Cor. loc.	Incor. loc.	Cor. rej.	Incor. rej.
NAudio	1987	2.8 MB	50.5%	5.9%	42.9%	0.7%
Prakashjha	623	167 kB	14.4%	23.0%	60.4%	2.2%
CodeTextBox	101	126 kB	5.9%	41.6%	52.5%	0.0%
ScintillaNET	1157	1.3 MB	18.0%	36.0%	46.1%	0.0%
XmlExplorer	571	380 kB	12.8%	12.1%	75.1%	0.0%

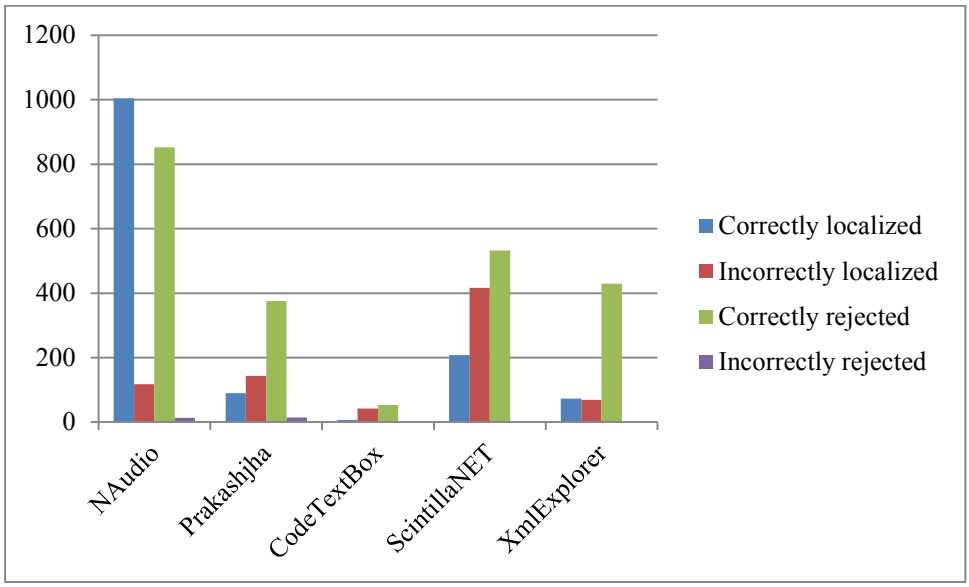


Figure 18: Graph displaying effectiveness of the LP.

As we can see from the graph, the default localization criteria generally give satisfactory results. The number of incorrectly rejected string literals is very low; even zero for the last three projects. On the other hand, the number of string literals that should not be localized but their LP claims they should ("incorrectly localized") is higher; this is caused by the fact that the default criteria cannot be too specific to handle all cases. Also, I believe it is better to have a higher number of incorrectly localized strings than the number of incorrectly rejected. The user can uncheck the incorrectly localized strings in the "batch move" tool window or he can manipulate the threshold of LP. He can also inline the incorrectly localized string easily when creating the language specific files. Omitting a string ("incorrectly rejected") is a

bigger issue – the string remains hardcoded in the source file and may be overlooked until a human localization tester reports that the program displays a non-localized string.

The most common "mistakes" of localization criteria are as follows:

- Font names ("Microsoft Sans Serif")
- Names of XML elements, attributes and database columns (almost all of these have 50% LP)
- Product names

The following graph displays average LP calculated for the categories of string literals in tested projects.

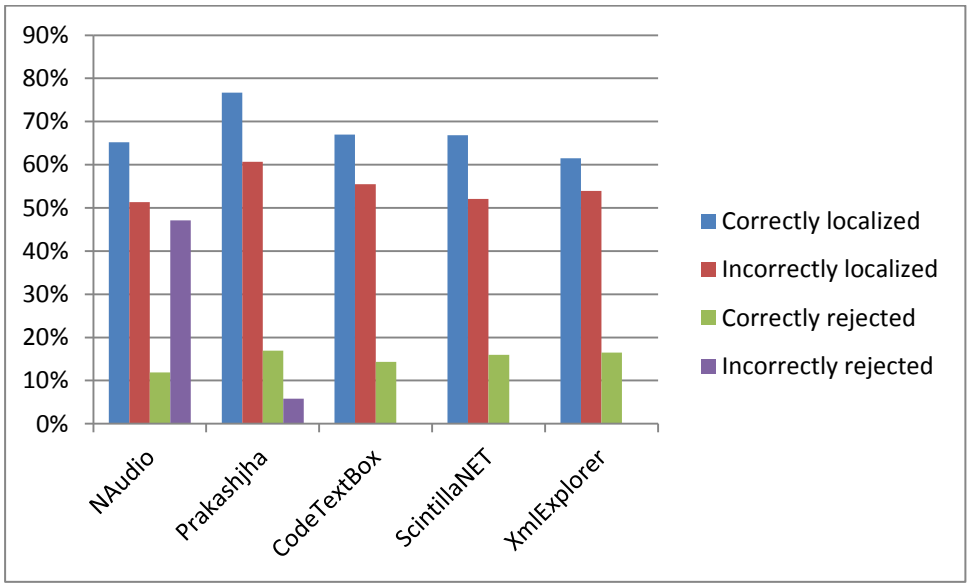


Figure 19: Localization probability calculated for various projects.

As expected, the line between correctly localized and incorrectly localized strings is quite thin. Also, incorrectly rejected strings seem to either obtain almost 0% LP or they almost pass the 50% limit.

In the previous experiment we only worked with the default set of localization criteria (and yet we obtained satisfying results). However, localization criteria are customizable, as described in 3.7 and 4.3.1. In the following experiment I focused on the *ScintillaNET* project, which had the highest number of incorrectly localized strings. I examined the project more closely and added 5 more criteria, which are specific for this project:

- Strings that appear in the `case` statement must not be localized
- Strings that are first arguments for the `GetAttribute()`, `SelectNodes()`, `SelectSingleNode()` or `Equals()` methods must not be localized
- Strings that are arguments for the `Group[]` indexer must not be localized
- Strings that seem to be a HTML entity must not be localized
- Literal content "value" and "key" must not be localized

This is what a project developer would do if he was using Visual Localizer. The results are demonstrated on the following graph:

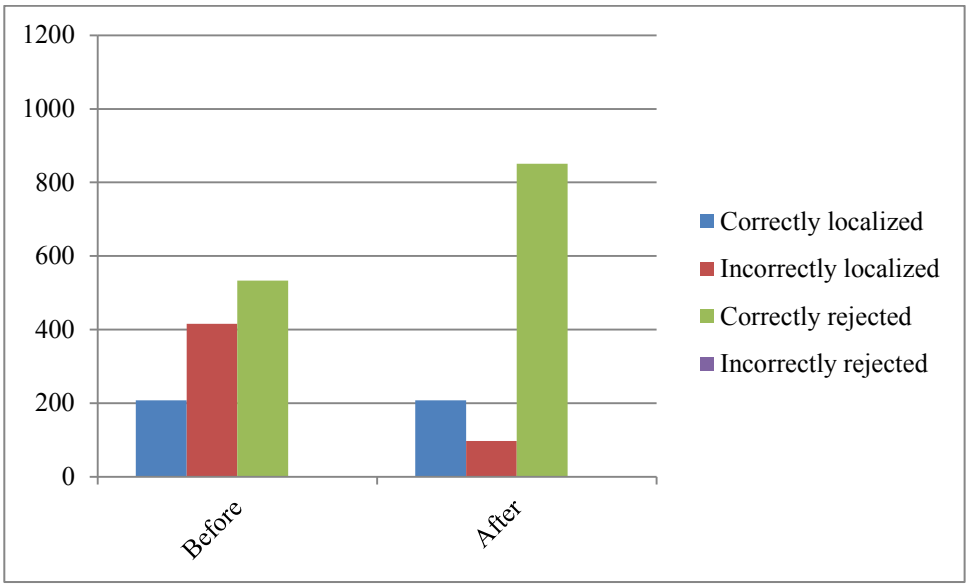


Figure 20: Graph demonstrating the effect of custom localization criteria on the ScintillaNET project.

The first set of values corresponds to the previous experiment, where only default localization criteria were used. In the second one, 5 project-specific criteria to eliminate the incorrectly localized strings were added. The number of correctly localized and incorrectly rejected strings remains unchanged, since the newly added criteria are dedicated to eliminating the incorrectly localized strings (false positives). Clearly, the new criteria worked – large amount of false positive results (319 to be accurate) were remarked as correctly rejected.

The experiments have confirmed that even large projects can be easily localized using Visual Localizer and the localization criteria. If the default set of localization criteria is not enough, custom criteria can be added making project localization even more effective.

6 Unresolved Issues

While developing Visual Localizer, I came across several issues I decided not to solve. None of these issues is mission critical, quite the opposite – they can be all considered as minor issues. In this chapter, we will point them out.

We have already mentioned ASP .NET website projects as kind of problematic. In Section 3.9.3 we discussed the missing `FileCodeModel`; websites however also employ different way of referencing other projects. Normally, when developers need to reference a code library which is located in another project of the same solution, they simply add the project to the "References" virtual folder of the referencing project. Visual Localizer (particularly the "batch inline" command and the ResX editor) uses these references to look for resource files, since they could be referenced from the code. However, websites' "References" folder contains directly the output assemblies of the referenced projects – therefore it is impossible to unambiguously track down the corresponding project and search it for resource files.

Visual Localizer is aware of the `Localizable(false)` attribute; filter discussed in Section 4.3.1 can be used to eliminate string literals affected by this attribute from the localization process. The attribute is searched for using the `FileCodeModel`, which provides the qualified name of the attribute and string values of its arguments. Therefore, we are only able to obtain the literal content of the `Localizable` attributes. .NET however allows any compile-time evaluable expression to be passed as an argument to an attribute, for example combination of constant `bool` fields. Visual Localizer does not resolve these expressions and only literal "false" is accepted, because there is no support for the resolution from the API and it does not seem like a serious problem.

The editor of ResX files provided by Visual Localizer is set as the default editor for the ".resx" files during the installation process. Any ResX file is then opened in our editor; however, Visual Studio contains one more way of accessing resources. In the "Properties" menu of certain project types there is a tab called "Resources" which when clicked opens the project default resource file[20] – but in the Visual Studio default editor. In C# projects, this does not trouble us since we can open the resource file from Solution Explorer as well and in this case, our editor is used. In VB .NET

[20] In C# projects, the Properties/Resources.resx is default; in VB .NET projects, My Project/Resources.resx is the default resource file.

however, the default resource file is not displayed in the Solution Explorer, although it exists on disk. This file therefore cannot be opened in our editor. I tried to find a way to make the “Properties” page use the Visual Localizer editor, but it seems to be such an unusual request that it is not supported by the Visual Studio API.

Finally, the last issue concerns combo boxes in the “batch move to resources” tool window. Since they are a part of Visual Studio-handled toolbar, they are defined in special files using the VSCT format [38], which is based on XML. Their definition must contain (among other things) the label text that gets displayed on the left side of the box and their width in pixels. This width is absolute and can be changed neither by user nor even in the code at all. In Visual Studio 2008, this width represents the width of the combo box and its label as well; in later versions, the width only affects the combo box, which leads to needlessly wide combo boxes, taking up too much space. Since there is no other way of setting the width rather than in VSCT files, this issue seems to be unsolvable.

7 Prospective Enhancements

The features that are currently implemented in Visual Localizer already make it a practical and useful tool. However, there are several features that would make it even better and since the hard work has already been done, implementing most of them should not be that difficult.

More programming languages and project types could be supported, i.e. Visual Localizer could look up string literals and references to resources in more types of source code – for example the Razor syntax for ASP .NET [39].

The editor of ResX files could support some kind of management of various culture-specific versions of the resource file, possibly displaying a complex grid with all available language mutations for each resource entry. Also, the synchronization feature discussed in Section 4.7.8 could be taken to a next level – for example, it would not be possible to add or remove resource entry from a culture-specific file and only values could be edited. The synchronization could then be done automatically on the background.

The "move to resources" commands could more closely operate with the resource file content, checking for duplicate (or even similar) values of string resources. The ResX editor could do the same, checking other ResX files in the project for resources with similar values, but different keys.

Finally, there could be a way how to easily manage language mutations of a referenced string resource from code. For example, when hovering a mouse over a reference to resource in the code window, a tooltip could be displayed with all available language mutations of the resource, with support for editing.

8 Similar Tools

As it was mentioned earlier, localization of applications became a common task for developers. The following tools aim to ease this task the similar way as Visual Localizer – in fact, some of this project's features were inspired by these tools.

ReSharper[21] is a robust plugin for Visual Studio that provides much more functions than just a localization support. In fact, it overrides a lot of Visual Studio functionality (for example *IntelliSense*), which leads to considerably larger system resources consumption.

Focusing on the localization feature, ReSharper can move a string to a resource file (with respect to the `Localizable` attribute), inline the reference back to hard-coded string literal and safely delete a resource. No kind of batch processing is supported – it only looks for the same strings as the one just being localized.

ReSharper is a commercial tool with quite steep pricing model – personal license for both C# and VB .NET support costs €179. This, combined with extensive resources requirements and lack of batch processing functions, makes ReSharper unsuitable when only localization support is required.

Resource Refactoring Tool[22] is an open source project, also providing "move to a resource" functionality. However, it is very limited – not only does it work only in Visual Studio 2005 and 2008, it does not support ASP .NET in any way, but also the implementation itself seems to be rather buggy. For example multiline strings in C# seem to be problem, also declaring a field `const` does not stop the tool from creating a reference in its initializer, thus producing a compile error.

When searching for a tool that would provide a better editor of ResX files, I came across **ResX Localization Studio**[23] (commercial), **Zeta Resource Editor**[24] (open source) and **RESX Manager**[25] (open source). Neither of these tools is an extension of the Visual Studio, they work as standalone applications, providing good support for human translators that edit the resource files. However, they do not provide any solution to our problem – keep the resource files clean and well-organized.

---

[21] http://www.jetbrains.com/resharper/features/internationalization.html
[22] https://resourcerefactoring.codeplex.com/
[23] http://resx-localization-studio.net/
[24] http://www.codeproject.com/Articles/16068/Zeta-Resource-Editor
[25] https://resxmanager.codeplex.com/

9 Conclusion

The goal of this project was to create practical and useful plugin for Visual Studio that would aid developers with localization of their .NET applications. After examining the localization process in .NET, we found out that the essential parts of this process are transferring localizable strings to and from resource files and also developers' endeavor to maintain only relevant content in these files. Dedicated commands were created to support the first, custom editor of ResX files to support the latter.

The commands were designed to be intuitive and effective. Combined with the powerful filter and eligible default settings of the localization criteria, they make it possible to localize whole project or solution in a few clicks.

The editor of ResX files keeps track of references to every stored resource, offering developers clear overview of which resources are relevant. In general, the Visual Localizer editor represents much stronger tool than the default editor provided by Visual Studio.

A consistent support for the most popular .NET programming languages – C# and Visual Basic .NET – is provided, as well as for Microsoft's leading technology for creating web content, ASP .NET. Visual Localizer correctly recognizes specific attributes of these languages and also differences between corresponding Visual Studio projects.

Effectiveness of the tool was tested on real projects. We verified that Visual Localizer can be easily used to localize even very large projects. We also examined currently available tools that share the specialization with Visual Localizer and we can state that Visual Localizer is the most suitable for the purposes of localization.

Several releases of Visual Localizer were uploaded to the project's homepage http://visuallocalizer.codeplex.com during the development. I received positive feedback from the users of all the releases and implemented many features the users suggested. By May 9, 2013, 622 downloads were made.

10 Bibliography

[1] Java Internationalization. *Java Tutorials.* [Online] 2013. [Cited: April 4, 2013.] http://docs.oracle.com/javase/tutorial/i18n/index.html.

[2] Ruby on Rails. *Ruby on Rails.* [Online] 2013. [Cited: April 4, 2013.] http://rubyonrails.org/.

[3] YAML Specification. *YAML.org.* [Online] September 29, 2009. [Cited: April 16, 2013.] http://yaml.org/spec/.

[4] Rails Internationalization (i18n) API. *Rails Guide.* [Online] 2013. [Cited: April 4, 2013.] http://guides.rubyonrails.org/i18n.html.

[5] About Google Translate. *Google Translate.* [Online] 2013. [Cited: April 4, 2013.] http://translate.google.com/about/.

[6] About Microsoft Translator. *Microsoft Translator.* [Online] 2013. [Cited: April 4, 2013.] http://www.microsoft.com/en-us/translator/.

[7] About MyMemory. *MyMemory.* [Online] 2013. [Cited: April 4, 2013.] http://mymemory.translated.net/doc/.

[8] Extensible Markup Language 1.0 (Fifth Edition). *W3C.* [Online] 2008. [Cited: April 15, 2013.] http://www.w3.org/TR/xml/.

[9] XML Schema 1.1. *W3C.* [Online] 2004. [Cited: April 16, 2013.] http://www.w3.org/XML/Schema.

[10] MIME Media Types. *Internet Assigned Numbers Authority.* [Online] 2013. [Cited: April 22, 2013.] http://www.iana.org/assignments/media-types.

[11] Linked and Embedded Resources. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/ht9h2dk8(v=vs.100).aspx.

[12] Adding and Editing Resources. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/7k989cfy.aspx.

[13] DTE Interface. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/envdte.dte.aspx.

[14] FileCodeModel Interface. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/envdte.filecodemodel.aspx.

[15] ASP .NET Web Page Syntax Overview. *MSDN.* [Online] 2013. [Cited: April 15, 2013.] http://msdn.microsoft.com/en-us/library/k33801s3.aspx.

[16] String Data Type. *MSDN.* [Online] 2013. [Cited: April 7, 2013.] http://msdn.microsoft.com/en-us/library/thwcx436.aspx.

[17] String literals. *MSDN.* [Online] 2013. [Cited: April 7, 2013.] http://msdn.microsoft.com/en-us/library/aa691090.aspx.

[18] HTML 4.01 Specification. *W3C.* [Online] 1999. [Cited: April 16, 2013.] http://www.w3.org/TR/html401/.

[19] Reflection in the .NET Framework. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/f7ykdhsy.aspx.

[20] ASP .NET Configuration Files. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/ms178684.aspx.

[21] Unqualified Name Resolution. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/aa711882.aspx.

[22] How to: Break and Combine Statements in Code. *MSDN.* [Online] 2013. [Cited: April 10, 2013.] http://msdn.microsoft.com/en-us/library/ba9sxbw4.aspx.

[23] Spectrum of Visual Studio Automation. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/9b54865a.aspx.

[24] Introduction to Macros. *MSDN.* [Online] 2013. [Cited: April 16, 2013.] http://msdn.microsoft.com/en-us/library/office/bb220916.aspx.

[25] VSPackages. *MSDN.* [Online] 2013. [Cited: April 8, 2013.] http://msdn.microsoft.com/en-us/library/bb166424.aspx.

[26] Visual Studio Development Environment Model. *MSDN.* [Online] 2013. [Cited: April 9, 2013.] http://msdn.microsoft.com/en-us/library/bb165114.aspx.

[27] Component Object Model. *MSDN.* [Online] 2013. [Cited: April 23, 2013.] http://msdn.microsoft.com/en-us/library/windows/desktop/ms680573.aspx.

[28] RegPkg Utility. *MSDN.* [Online] 2013. [Cited: April 9, 2013.] http://msdn.microsoft.com/en-us/library/vstudio/bb707481.aspx.

[29] Installing VSPackages. *MSDN.* [Online] 2013. [Cited: April 9, 2013.] http://msdn.microsoft.com/en-us/library/bb165366.aspx.

[30] Module Statement. *MSDN.* [Online] 2013. [Cited: April 10, 2013.] http://msdn.microsoft.com/en-us/library/aaxss7da.aspx.

[31] Simple API for XML (SAX). *DeveloperWorks.* [Online] 2013. [Cited: April 10, 2013.] http://www.ibm.com/developerworks/xml/standards/x-saxspec.html.

[32] ASP.NET Configuration File Hierarchy and Inheritance. *MSDN.* [Online] 2013. [Cited: April 10, 2013.] http://msdn.microsoft.com/en-us/library/ms178685.aspx.

[33] Dori, Shiri and Landau, Gad M. Construction of Aho Corasick Automaton in Linear Time for Integer Alphabets. [Online] [Cited: April 10, 2013.] http://cs.haifa.ac.il/~landau/gadi/shiri.pdf.

[34] My Reference. *MSDN.* [Online] 2013. [Cited: April 11, 2013.] http://msdn.microsoft.com/en-us/library/8ffec36z.aspx.

[35] Editors. *MSDN.* [Online] 2013. [Cited: April 10, 2013.] http://msdn.microsoft.com/en-us/library/bb166580.aspx.

[36] Introducing JSON. *JSON.* [Online] 2013. [Cited: April 23, 2013.] http://www.json.org/.

[37] Application Domains. *MSDN.* [Online] 2013. [Cited: May 1, 2013.] http://msdn.microsoft.com/en-us/library/cxk374d9.aspx.

[38] VSCT XML Schema Reference. *MSDN.* [Online] 2013. [Cited: April 18, 2013.] http://msdn.microsoft.com/en-us/library/vstudio/bb165416.aspx.

[39] ASP .NET MVC 3 Razor. *MSDN.* [Online] 2013. [Cited: April 14, 2013.] http://msdn.microsoft.com/en-us/vs2010trainingcourse_aspnetmvc3razor.aspx.

[40] Sandcastle Help File Builder. *Codeplex.* [Online] 2012. [Cited: May 1, 2013.] https://shfb.codeplex.com/.

[41] Choudhary, Chiranjiv. Differences between ASP .NET and ASP. *CodeProject.* [Online] December 18, 2006. [Cited: April 7, 2013.] http://www.codeproject.com/Articles/16703/Differences-between-ASP-NET-and-ASP.

[42] VSIX Deployment. *MSDN.* [Online] 2013. [Cited: April 9, 2013.] http://msdn.microsoft.com/en-us/library/ff363239.aspx.

[43] Nesteruk, Dmitri. Custom Tools Explained. *Codeproject.* [Online] November 26, 2008. [Cited: April 23, 2013.] http://www.codeproject.com/Articles/31257/Custom-Tools-Explained.

11 Attachments

The following content can be found on the enclosed CD:

A. Source projects

Visual Studio 2008 solution containing all source codes of Visual Localizer and its libraries, located in the `\src\VisualLocalizer` subfolder.

B. Testing solution

Solution used by unit tests to emulate running VS environment is located in `\src\VLUnitTestsContextSolution`.

C. Installation file

MSI package that performs all installation and registration tasks, found in `\install`.

D. Generated documentation

Documentation files generated by the Sandcastle tool [40], each corresponding to a source project. These files are stored in the `\doc\sandcastle` folder.

E. Electronic version of this thesis

The PDF file located in `\doc\thesis`.