**Globalization Parser for C# Projects 1.0**

**Design Document**

1. **Introduction………………………………………………………………………………………………………….. 1**
   1. **Overview……………………………………………………………………………………………………….. 1**
2. **Users and their Characteristics……………………………………………………………………………… 1**
3. **Tool Features………………………………………………………………………………………………………… 2**
   1. **User Interface………………………………………………………………………………………………… 2**
   2. **Directory-Based Parsing…………………………………………………………………………………. 4**
      1. **Directory Analysis………………………………………………………………………………… 5**
   3. **Solution-Based Parsing………………………………………………………………………………….. 6**
      1. **Filter Projects………………………………………………………………………………………. 6**
      2. **Solution Analysis…………………………………………………………………………………. 7**
   4. **Starting the Parse………………………………………………………………………………………….. 8**
   5. **Parsing Category Selection……………………………………………………………………………… 8**
      1. **String Concatenation……………………………………………………………………………. 8**
      2. **String Comparison……………………………………………………………………………….. 8**
      3. **Functions returning type String……………………………………………………………. 9**
      4. **Hard Coded Strings………………………………………………………………………………. 9**
         1. **Inclusions…………………………………………………………………………………. 9**
         2. **Exclusions…………………………………………………………………………………. 9**
      5. **Date Formats……………………………………………………………………………………….. 9**
      6. **Decimal Formats………………………………………………………………………………….. 9**
   6. **Exporting……………………………………………………………………………………………………….. 10**
4. **Introduction**
   1. **Overview**

This document specifies the feature set of the Globalization Parser Tool specifically for C# Projects. The features are meant to aid developers detect common trouble areas in code that often result in localization errors. These common trouble areas inside code include string concatenation, string comparisons, functions that return strings, using hard-coded strings instead of utilizing an external resource file (.resx extension), date formats, and decimal formats.

1. **Users and their Characteristics**

The expected user for this tool is a developer seeking to prevent or fix globalization errors in an existing project.

1. **Tool Features**
   1. **User Interface**

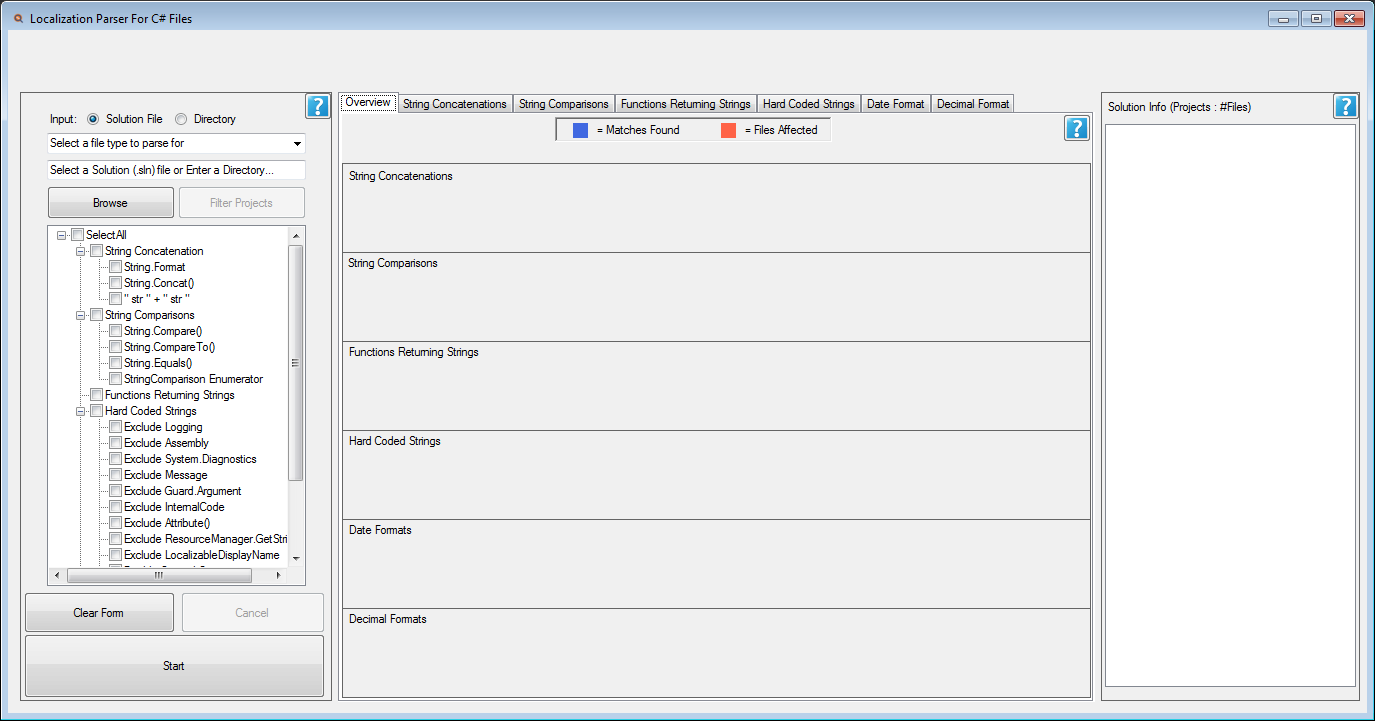


Figure 1 The Overview UI in its initial state



Figure 2 The Overview UI in its active state

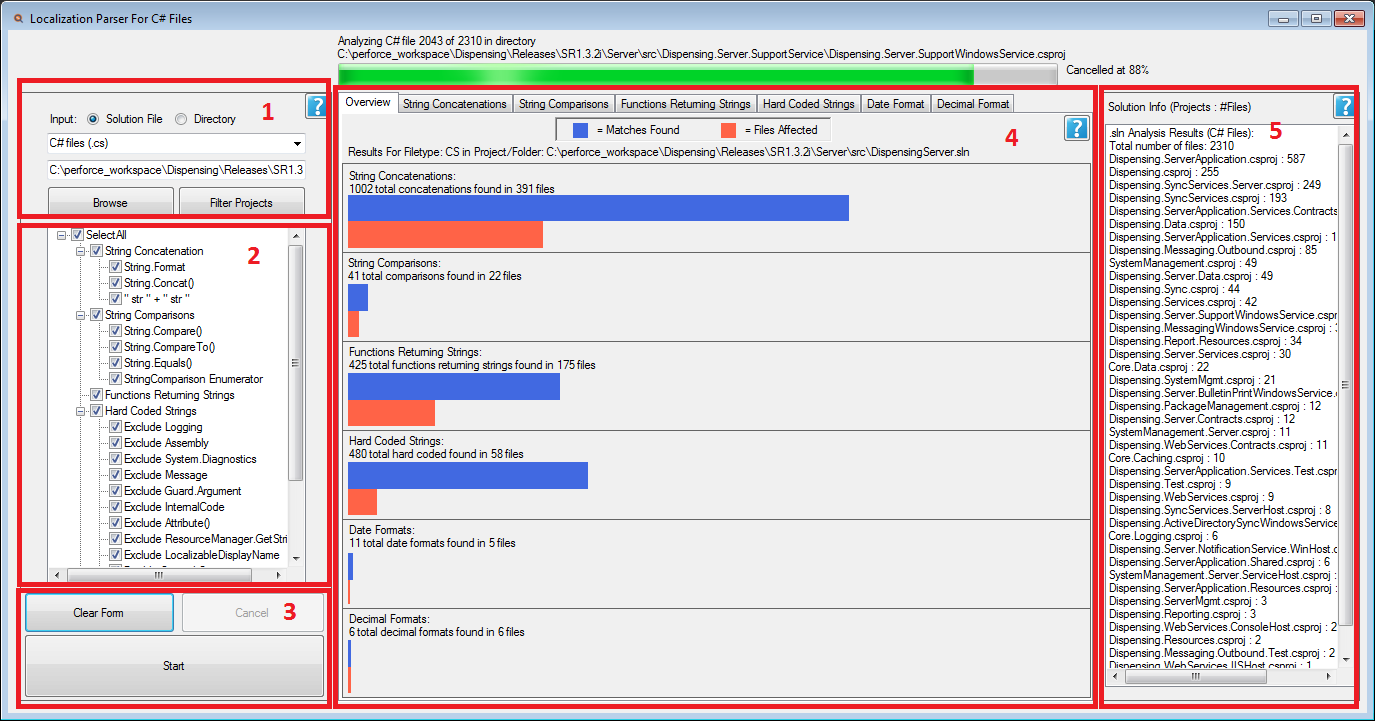


Figure The Elements that make up the Overview layout

1. Area to Input Directory or Solution.
   1. Radio buttons limit input to either a Solution File (.sln) or Directory path
   2. Drop down menu allows selection between the different supported file types:
      1. C# Files
      2. CSHTML Files
      3. Javascript Files
   3. Using the **Browse** button, the solution file or directory path may be selected
      1. If the input is a solution file, the **Filter Projects** button allows the user to filter select CS Project files (.csproj) associated with that solution file.
2. Parsing Category Filters and SubFilters
   1. The available default filtering options differ depending on which file type is chosen. For example, “Functions Returning Strings” will be disabled for CSHTML file parsing.
3. Buttons for clearing the form, cancelling the parse, and beginning the parse
4. The Overview layout displays the results for each parsing category using bar graphs. Each parsing category has a separate panel featuring the following:
   1. Blue bar graph representing the number of matches found in files
   2. Orange bar graph representing how many files contain matches
   3. Panels may be clicked to view a category’s Results layout (**Figure 4**)
   4. Users may also click on the overhead tabs to navigate to a specific category
5. Info pane that displays data regarding the chosen Directory/Solution (**Figure 5**)

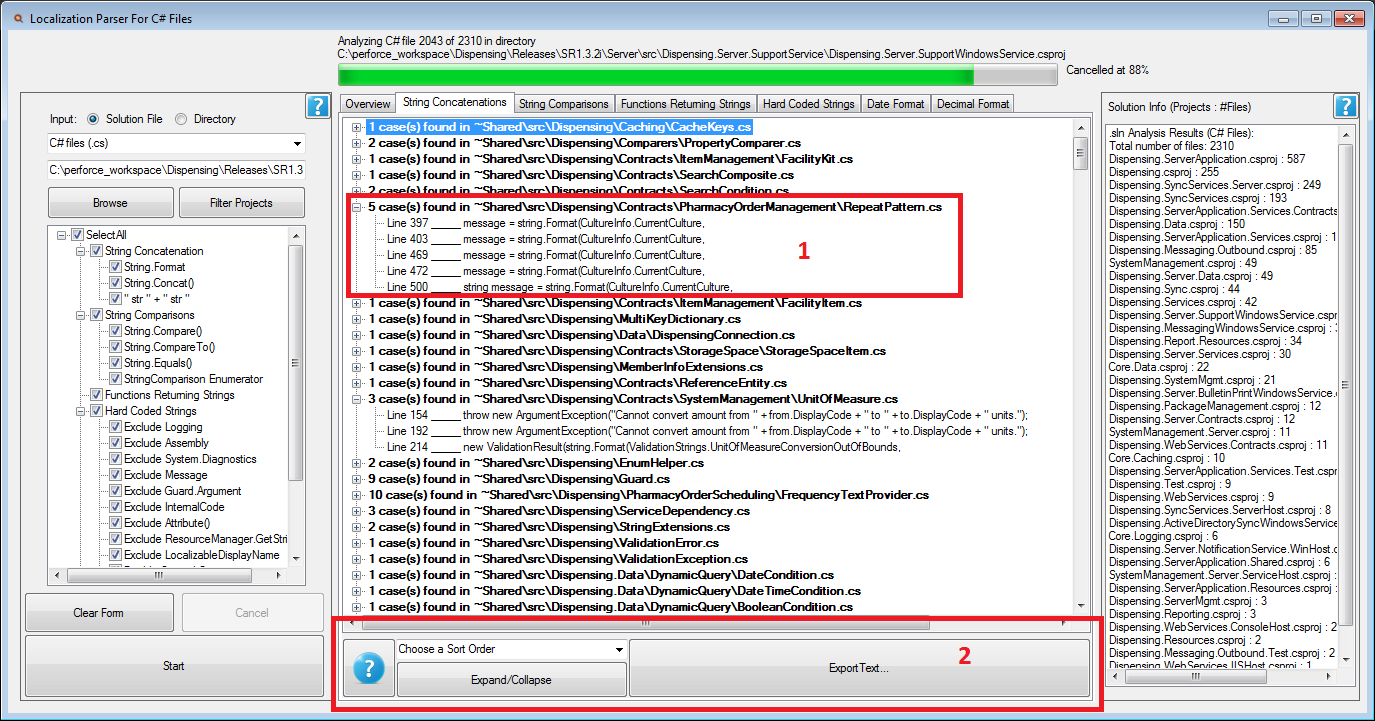


Figure Every Category has their own Results View

1. Results are listed in hierarchy.
   1. Each result listed is represented as an expandable/collapsible node with a parent and a child.
   2. The parent node details the number of matches found within a file and the name of the file.
   3. The child node lists the line number of the code the match took place in followed by the code excerpt.
2. The bottom panel contains the following features:
   1. A **Help** button for giving a brief overview of the currently selected parsing category
   2. A drop down menu allowing users to sort the results in the following ways:
      1. By occurrence (Default)
      2. By Number of Matches (Descending)
      3. By Number of Matches (Ascending)
   3. An **Expand/Collapse** button to expand or collapse all nodes in the results view
   4. An **Export Text** button that exports the results to a plain text file. The sorting of the results in the export file will match how the nodes are sorted in the Results View.
   5. **Directory-Based Parsing**

The user may input a Directory that ideally contains C# projects and files to parse for globalization errors (**Figure 3 Area 1**)**.**

* + 1. **Directory Analysis**

Upon selection of the specified Directory, the info pane at the top left of the UI will fill with the following data (**Figure 5**):

* The total number of files in the directory and all subdirectories
* The type of file (by file extension) and the number of times the file type occurs in all subdirectories
* The file types and their frequency will list in descending order

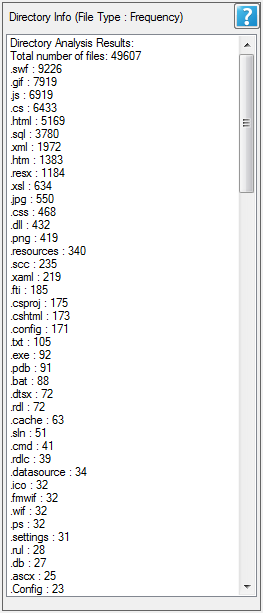


Figure 5 Info pane shows different results depending on whether Directory or Solution is selected

* 1. **Solution-Based Parsing**

The user may input a Solution file (.sln extension) to parse only C# files belonging to a specific Solution, which may contain multiple Projects (.csproj extension).

* + 1. **Filter Projects**

Once a Solution is selected, another window will appear (**Figure 6**) that will allow the user to choose which of the Projects associated with that Solution to parse. By default, all Projects are selected.

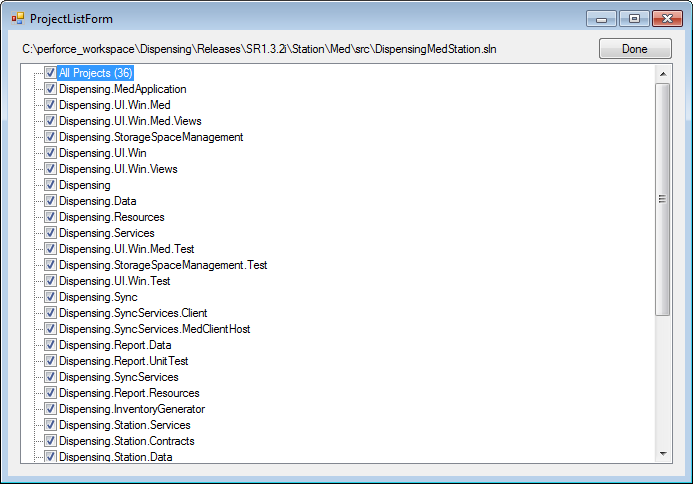


Figure 6 Filter which Projects within the Solution to parse

* + 1. **Solution Analysis**

Upon selection of the specified Projects, the info pane (**Figure 7**) will fill with the following data:

* The total number of C# files in all Projects
* The name of the Project and the number of C# files associated with that Project (listed in descending order)

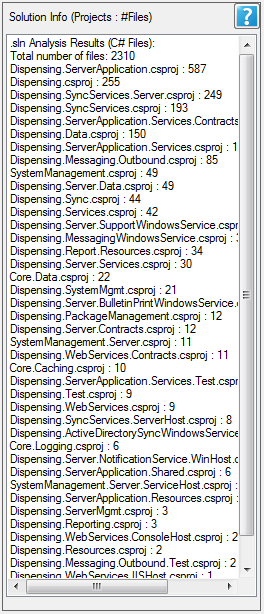


Figure 7 The Info pane when Projects in a Solution are selected

* 1. **Starting the Parse**

The **Start** button (**Figure 3, Area 3**) is used to begin the procedure once the Directory/Solution is selected and at least one Parsing Category is chosen (See **3.5 Parsing Category Selection**). During the parse, a progress bar will appear to inform the user of the status of the operation (**Figure 8**). It displays the current file being parsed and the total number of files to parse. The parse may be cancelled at any time via the **Cancel** button (**Figure 3 Area 3**). A window will pop up to inform the user of the cancellation and the progress bar completion percentage will change from percentage number to “Cancelled at \_%”**.**

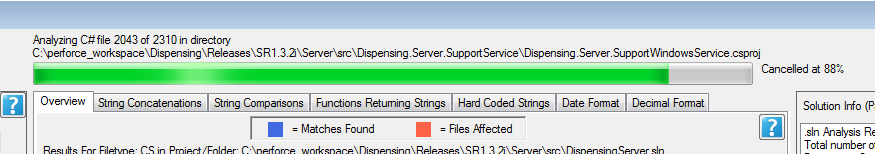


Figure 8 Progress Bar

* 1. **Parsing Category Selection**

In order to start the parsing of the Directory/Solution, the user must select at least one or more parsing categories. All categories may be selected at the same time. Each category aims to find specific instances of common errors that occur in the localization process. All detections are done via **Regular Expressions**.

* + 1. **String Concatenation**

String Concatenation is the act of combining multiple strings into one in code. Selecting this feature displays every instance of combining strings within all C# files by searching for use of .NET string-combining function **String.Format**, **String.Concat** and the use of concatenation operation (**+**) in combination with double quotes (**“”**). If Javascript files are selected, only **String.Concat** will be a selectable sub filter.

* + 1. **String Comparison**

String comparison is the act of comparing the values of two strings. Comparisons are based on the sort order of each character, and thus differ between locations and must be addressed for localization. Selecting this feature displays every instance of comparing strings by searching for use of .NET string-combining functions **String.Compare**, **String.CompareTo**, **String.Equals**, and also the **StringComparison** Enumerator, which is used to specify culture, case, and sort rules used by the previously mentioned .NET string comparison functions**.** This filtering option is only supported with C# files. If Javascript or CSHTML files are the selected file types, this filter and all associated sub filters will be disabled.

* + 1. **Functions returning type String**

This feature searches for Functions that return type String. The Regular Expression used searches for the word “**string**”as the return type for patterns matching method signatures. The pattern matching accounts for additional and optional modifiers that may be found in the method signature (i.e. abstract, sealed, static) and creates matches based solely on the return type. This filtering option is only supported with C# files. If Javascript or CSHTML files are the selected file types, this filter and all associated sub filters will be disabled.

* + 1. **Hard Coded Strings**
       1. **Inclusions**

This feature searches for strings directly embedded into the code. As much as possible, strings should not be explicitly placed in code and instead be transferred over to an external Resource file. The Regular Expression used matches **alphabetic characters and spaces enclosed in an open and closed double quote**.

* + - 1. **Exclusions**

The goal is to find all strings in the code that will be seen by the end user. To that end, the program actively tries to avoid any strings used for backend logging and diagnostics, so lines that contain the words “**Logger**”, “**Assembly**”, “**Message**”, “**guid**”, and “**System.Diagnostics**” are ignored. Likewise, C# projects with a UI tend to dynamically generate code for form components using double quoted strings and may dilute the results, so exclusion also include “**InternalCode**”, “**Attribute**”, “**ResourceManager.GetString**”, “**LocalizableDisplayName**”, “**Guard.Argument**”. Hard coded sql statements such as “WHERE” and “FROM” are also a concern. Thus, a sub filter is included which has the option to only match alphabetic characters and spaces enclosed double quotes such that the pattern indicates the **presence of more than one word.** This increases the likelihood that the matching string is one that the user will see. For example, a line containing the string “Application” would not match, but “Med Application” and “Failed to dispose” will produce a match.

* + 1. **Date Formats**

This feature searches for instances of hard-coded date format. Date formats vary by region, and thus should not be explicitly formatted in source code. The pattern searches for two to four consecutive “**y**” characters followed by a slash **/** or a dash **–** character. For example: “yy-” and “/yyyy” would both produce matches.

* + 1. **Decimal Formats**

This feature searches for instances of hard coded Decimal Formats. Decimal formats vary by region, and thus should not be explicitly formatted in source code. In C#, numeric formats are hard-coded using the combination of numbers and pound characters to specify how many numbers are to be shown before or after a decimal. Thus, the Regular Expression pattern searches for one or more pound characters (**#**)followed by a period (**.**) or comma (**,**) and then finally any number 0-9 or another pound character (**#**). For example: “###.0” will produce a match. It also searches for the reverse of the pattern, with numbers first followed by the periods, commas, and pound characters. For example: “0.####” will produce a match. For Javascript files, Decimal formatting is done via functions **ToFixed()** and **ToPrecision()**. These sub filter options are only available to Javascript file types and not C# files or CSHTML.

* 1. **Exporting**

The User may choose to export all output to an external Text file (.txt extension) **(Figure 9**) by clicking on the **Export** button **(Figure 4, Area 2**). The output text contains the same content and formatting as the output textboxes **(Figure 4, Area 1**). The name of the File defaults to the name of:

“**Directory/Solution\_Parsing Category\_Output\_CurrentDate.tx**t”

For example, exporting the results of a String Concatenation parse on the Solution DispensingMedStation.sln taken on 4/29/2016 at 10:13:21 will output the file:

“**DispensingServer.sln\_String Concatenations\_20160429\_101321.txt**”

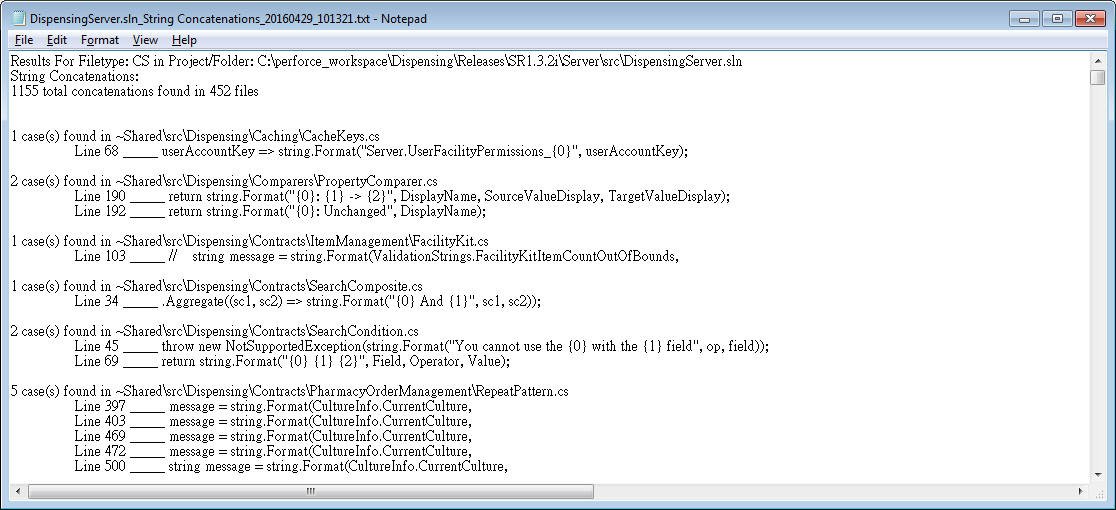


Figure 9 Exported Text File