.NET File Statistics 1.0 Component Specification

1. Design

***File Statistics provides a framework to collect statistic data from a subset of the file system and renders the format in configurable reporting style*.**

As a framework component the first question comes into mind when designing the component would be ***what aspects of the component need to be pluggable***. It's obvious that the algorithm to extract the statistical data as well as the way to render the data should be typically configurable. A third aspect that is factored out is the filtering logic to specify which subset of the files are processed by certain processor. Though this could be tied to the processor the design would not as flexible that way. Once the three aspects are cleared the strategy design patterns come into play.

Once we have the interfaces the next question would be ***how to glue them together***. The design finally chooses to use a non-sharable manager class because the use of the component involves multiple steps and result is aggregated when each file is processed. (The design does not prevent from using stateless reporting mechanism, like logging, though.) The manager also fulfills the task to instantiate the framework from config files for better configurability and for the use case of the command line interface. Each of the pluggable implementation has contract constructor to configure itself.

Once the framework is settled the rest of the design is to produce plug-ins to fulfill the requirements. The design should be relatively easy to understand.

1.1 Design Patterns

- **Strategy Pattern**: used for IFileMatcher, IFileProcessor and IStatsReporting and their respective implementations. This is essential for a framework component in order to provide maximum pluggability.
- **Template Method Pattern**: used for FileNameMatcher, FileCategoryMatcher and FileTypeMatcher to minimize redundancy for extensions.

1.2 Industry Standards

None

1.3 Required Algorithms

1.3.1 Counting Lines in C Style Source Codes

For this release we will include a naive C style source line counter. It provides moderately accurate figure with a relatively simple implementation. This does not necessarily work for some cases.

The design recommends a state machine implementation. Here is the transition table.

State	Input	Action
START	slash(/)	SLASH
	comma(;)	Increment line
	open brace({)	Increment line
	other	START
SLASH	star(*)	BLOCK_COMMENT
	slash(/)	LINE_COMMENT
	other	START
STAR	slash(/)	START
	start(*)	STAR

	other	BLOCK_COMMENT
SQ_BACKSLASH	any	SINGLE_QUOTE
DQ_BACKSLASH	any	DOUBLE_QUOTE
SINGLE_QUOTE	backslash(\)	SQ_BACKSLASH
	single quote(‘)	START
	other	SINGLE_QUOTE
DOUBLE_QUOTE	backslash(\)	DQ_BACKSLASH
	double quote(“)	START
	other	DOUBLE_QUOTE
LINE_COMMENT	any	LINE_COMMENT
	line feed	START
BLOCK_COMMENT	star(*)	STAR
	other	BLOCK_COMMENT

1.3.2 Command Line Interface

The command line interface should be powerful to expose all the functionality and flexibility to the user, which should still be easy to use (meaning most options should have well defined defaults).

Here is what we have got:

[-c config-file] [-n namespace] [-b base-path] [-r] [-o output-file] path [path2 path3 ...]

- ***-c config-file***

This specifies the configuration file to use. If the namespace is provided at the same time it will be loaded to the namespace in order to instantiate the component. Otherwise it is loaded to default namespace.

- ***-n namespace***

This specifies the namespace to use. If the config file is provided at the same time the config file is loaded to the namespace. Otherwise the namespace should be preloaded. If neither config file nor namespace is provided the component instantiates from the default namespace (assuming preload).

- ***-b base-path***

This specifies the base directory for the reporting. If this is provided all path are supposed to be relative to this path and relative paths are used for reporting. If not provided all path are supposed to be either absolute to relative to executing path, with reporting based on absolute path.

- ***-r***

This specifies whether the directories are processed in recursive mode. If not provided directories do not go into sub-directories. If no directory is processed this switch does not make any effects.

- ***-o output-file***

This specifies the output file which is either absolute or relative to executing path (does not rely on base-path). If not provided reporting is printed to console.

- ***path [path2 path3 …]***

This specifies a list of files or directories to process. If base directory is provided they are relative, otherwise absolute to relative to executing path. Files and/or directories can mix.

1.3.3 Xml Reporting

In order to provide better human readability and easy stylesheet support, the output xml uses dynamic element names for the stats result. The display name of the metric will be used to render the element node (which consequently requires well-formed name to be used but the framework does not validate over that). Refer to the sample output which should be pretty clear in structure. For readability it is advised to use pretty formatting for the output. When multiple filename, directory name or aliases are listed, alphabetical order should be used (case-insensitive).

1.4 Component Class Overview

1.4.1 Namespace TopCoder.File.Statistics

- **FileStatistics**: This is the main class of the user API. It provides the basic API, which includes maintaining a list of file processors, a reporting handler and controls the way to process the file. Files are processed one by one and the results are aggregated into the reporting. If you want to start a new reporting you should either explicitly set the reporting or get the old reporting to reset it. There are two ways to process the file: process it with the first matching processor and process it with each matching processor. Refer to ExtendedFileStatistics for advanced usage.
- **ExtendedFileStatistics**: This is the extended version of FileStatistics which supports some advanced operations including directory processing and reporting printing. The reason to separate this out is that we may not need to rely on the directory traversing algorithm incorporated here (which is recursive).
- **Main**: The command line entry point of the File Statistics utility. It uses ExtendedFileStatistics to fulfill the task. When argument is invalid help message should be printed.
- **IFileProcessor (interface)**: IFileProcessor defines the contract to process a file. One file processor defines an algorithm to produce statistical result for a certain class of files against certain metrics. (For instance count line numbers for a text file.) Multiple IFileProcessors are aggregated into FileStatistics, which can run in two modes (process with first matching processor and process with each matching processor). File processors are mapped with an alias. This alias is dynamically provided by the user and the reporting can be grouped against this alias.
- **IFileMatcher (interface)**: IFileMatcher defines the contract to match a certain file (in order for an attaching IFileProcessor to process the file).
- **IStatsReporting (interface)**: IStatsReporting interface defines the contract to aggregate and render the statistical report. Result is fed into this interface one by one and the file report is then generated. There is a way to reset (clear) the current aggregated results.
- **StatsResult**: StatsResult is container for multiple results, each of which are mapped from a Metric to a Long. This class wraps some map operations as well as provides a way to clone and aggregate results.
- **Metric**: Metric represents a measurement of the stats result. A metric is identified by its name. Metrics with the same name will be aggregated. This class overrides Equals() and GetHashCode() so that it can be used as map keys.

1.4.2 Namespace TopCoder.File.Statistics.Processor

- **FileProcessorBase (abstract)**: A base implementation that can be used for FileProcessors. It provides alias and matcher support, including loading them from configuration file. It serves TextLineCounter and CStyleLineCounter as well as can be generically used by future implementations.

1.4.3 Namespace TopCoder.File.Statistics.Processor.LineCounter

- **TextLineCounter**: TextLineCounter counts lines for text files. It supports a single metric of "Line Count".
- **CStyleLineCounter**: CStyleLineCounter counts lines for C style source files. This implementation can handle C, C++, C# or Java. It only takes braces and comma into

account so it would not be very accurate in most cases. Comments and string literals can be handled. It supports a single metric of "Line Count".

- **SimpleStateMachine**: This is a private static inner class to CStyleLineCounter that implements a state machine that can process the source code. Notice this is an implementation recommendation. Developer can choose to improve from this start point. Dropping this class is acceptable as long as the implementation of CStyleLineCounter is stateless (and the clarity and/or efficiency aspects are improved).

1.4.4 Namespace TopCoder.File.Statistics.Matcher

- **AnyFileMatcher**: AnyFileMatcher is a trivial implementation that matches any file.
- **FileNameMatcher (abstract)**: FileNameMatcher is a template class that handles filename oriented matching logic.
- **ExtensionMatcher**: FileNameMatcher concrete implementation that matches one or more file extensions. The extensions are matched in a case-sensitive manner.
- **RegexMatcher**: FileNameMatcher concrete implementation that matches file names with one or more regular expression.
- **FileCategoryMatcher (abstract)**: FileCategoryMatcher is a template class that handles file category (text/binary) oriented matching logic. The purpose to still create subclasses for such minor functionality is about the convenience in configuration.
- **TextFileMatcher**: FileCategoryMatcher concrete implementation that matches text files.
- **BinaryFileMatcher**: FileCategoryMatcher concrete implementation that matches binary files.
- **FileTypeMatcher (abstract)**: FileTypeMatcher is a template class that handles file type oriented matching logic. File type is resolved by the Magic Numbers component.
- **TypeNameMatcher**: FileTypeMatcher concrete implementation that matches file types with one or more type names (as configured in Magic Numbers).
- **MimeMatcher**: MimeMatcher concrete implementation that matches file types with one or more mimes (as configured in Magic Numbers).

1.4.5 Namespace TopCoder.File.Statistics.Reporting

- **AbstractStatsReporting (abstract)**: This class provides an IStatsReporting implementation base. It aggregates results from each processor and classifies them based on single file, directory or alias. It also provides overall stats. It defers the actual rendering logic to concrete implementations (BasicStatsReporting and XmlStatsReporting).
- **StatsCollection**: StatsCollection is a generic container for mapping from String keys to StatsResult instances. It is used in AbstractStatsReporting to hold file, directory and alias classified mappings. The class provides basic manipulation methods for the mapping.
- **BasicStatsReporting**: BasicStatsReporting is a reporting implementation that renders user-friendly reports.
- **XmlStatsReporting**: XmlStatsReporting is a reporting implementation that renders XML reports. Refer to component specification for formatting of the XML. It is capable of using a stylesheet to transform the XML into user- oriented format (HTML, CSV, etc.)

1.5 Component Exception Definitions

1.5.1 Custom Exceptions

- **FileStatisticsException**: Framework exception which provides the extension base for all exceptions.
- **ConfigurationException**: Used to cover configuration related exceptions for all the constructors with a namespace or with a property.
- **FileMatchingException**: Encapsulates errors from the IFileMatcher interface to indicate a dependency error. If the file operation fails IOException is used instead.
- **FileProcessingException**: Encapsulates errors from the IFileProcessor interface to indicate an implementation specific error. If the file operation fails IOException is used instead.
- **StatsReportingException**: Encapsulates errors from the IStatsReporting interface to indicate an implementation specific error.

1.5.2 System Exceptions

- **ArgumentNullException**: Used wherever null argument is used while not acceptable.
- **ArgumentException**: Used wherever empty string argument is used while not acceptable. Normally an empty string is checked with trimmed result.
- **IOException**: Propagated from file operations.

1.6 Thread Safety

This component is not thread-safe. The current IFileMatcher and IFileProcessor implementations are thread-safe, which the rest part of the components is not thread-safe. Different thread should instantiate separate FileStatistics and IStatsReporting to fulfill concurrent tasks (but the IFileMatcher and IFileProcessor implementations can potentially be shared). This will not cause problem with the command line interface because everything is run in a single thread.

2. Environment Requirements

2.1 Environment

.NET 1.1

2.2 TopCoder Software Components

- **Configuration Manager 2.0**: used to support component configuration.
- **Magic Numbers 1.0**: used to examine file types. Notice this component also has the ability to distinguish text and binary files but unfortunately it is not exposed on API.
- **Command Line Utility 1.0**: used to process command line arguments.

2.3 Third Party Components

None

3. Installation and Configuration

3.1 Package Name

TopCoder.File.Statistics
TopCoder.File.Statistics.Processor
TopCoder.File.Statistics.Processor.LineCounter
TopCoder.File.Statistics.Matcher
TopCoder.File.Statistics.Reporting

3.2 Configuration Parameters

Parameter	Description	Values
processors	Multiple sub-properties each specify an IFileProcessor instance to be aggregated in the FileStatistics.	Property container required
reporting	Specifying an IStatsReporting instance to be used with FileStatistics.	Property container optional, XmlStatsReporting is instantiated if not specified
matchall	Flag to indicate whether file is processed with each matching processor or the first matching processor.	“yes”/”true”/”on” is translated to true, other values translated to false. optional, false if not specified
matcher	Specifying an IFileMatcher instance to be used with the associating IFileProcessor.	Property container optional, AnyFileMatcher is instantiated if not specified
classname	Used within containers to specify the classname to instantiate.	Fully qualified class name. required
alias	Used in processor container to specify the processor alias.	Non-empty name. required
stylesheet	Used in XmlStatsReporting to apply on the raw XML generated.	Valid file path. optional, no XSLT applied if not specified.
extensions	Used in ExtensionMatcher to specify the file extensions to match.	Multi-valued. Empty is allowed. required
patterns	Used in RegexMatcher to specify the regular expressions to match filenames.	Multi-valued. Valid regular expression. required
types	Used in TypeNameMatcher to specify the file type names to match (which are configurable in Magic Numbers).	Multi-valued. Non-empty. required
mimes	Used in MimeMatcher to specify the file mimes to match (which are configurable in Magic Numbers).	Multi-valued. Non-empty. required

3.3 Dependencies Configuration

Magic Numbers require configuration.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.

- Execute ‘nant test’ within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Follow configuration instructions.

4.3 Demo

4.3.1 Configure File Statistics

```
// create a file statistics instance
FileStatistics statistics = new FileStatistics();

// create file matchers
IFileMatcher matcher1 = new AnyFileMatcher();
IFileMatcher matcher2 = new ExtensionMatcher(“txt”);
IFileMatcher matcher3 = new RegexMatcher(Pattern.compile(“pattern”);
IFileMatcher matcher4 = new TextFileMatcher();
IFileMatcher matcher5 = new BinaryFileMatcher();
IFileMatcher matcher6 = new TypeNameMatcher(“Java”);
IFileMatcher matcher7 = new MimeMatcher(“text/plain”);

// create file processors
IFileProcessor processor1 = new TextLineCounter(“Text File”, matcher1);
IFileProcessor processor2 = new CStyleLineCounter(“Source File”, matcher6);

// create stats reporting implementation
IStatsReporting reporting1 = new BasicStatsReporting();
IStatsReporting reporting2 = new XmlStatisReporting(“stylesheet/xml_to_html.xsl”);

// manipulate file processors
statistics.AddFileProcessor(processor1);
statistics.AddFileProcessor(processor2);
IFileProcessor removed = statistics.RemoveFileProcessor(“Source File”);
IFileProcessor query = statistics[“Text File”];
IList processors = statistics.AllFileProcessors;
statistics.ClearAllFileProcessors();

// manipulate reporting
IStatsReporting original = statistics.Reporting;
statistics.Reporting = reporting2;

// manipulate match all flag
if (!statistics.IsMatchAll) statistics.IsMatchAll = true;
```

4.3.2 Process File

```
// process single file
statistics.ProcessFile(new FileInfo(“test_files/a/p.txt”));
statistics.ProcessFile(new FileInfo(“test_files/b/q.java”));

// process directory
statistics.ProcessDirectory(new DirectoryInfo(“test_files/a”));
statistics.ProcessDirectory(new DirectoryInfo(“test_files”), true);
```

4.3.3 Generate Report

```
// obtain report
string report = statistics.Reporting.GenerateReport();

// output report
statistics.PrintReport();
statistics.PrintReport(new FileInfo(“test_files/output.html”));

// reset reporting
statistics.Reporting.Reset(new FileInfo(“test_files”));
```

4.3.4 Command Line Interface

```
// specify configuration file
-c conf/FileStatistics.xml a.txt b.java
// specify namespace
-n com.topcoder.file.statistics a.txt b.java

// specify base directory
-b test_files a.txt b.java

// recursively process sub-directory
-r test_files

// specify output file
-o test_files/output.html a.txt b.java
```

5. Future Enhancements

Provide useful file matchers and file processors.