

Gaurav Bhasin  
 Software Modelling and Architecture  
 Jim Fawcett

**Operational Concept Document for Code Analyzer**

Contents

[**1. Introduction 2**](#_Toc398753398)

[**1.1 Executive Summary 3**](#_Toc398753399)

[**1.2 Specifications 3**](#_Toc398753400)

[**2. USERS OF THE SYSTEM 4**](#_Toc398753401)

[**3. USES OF THE CODE ANALYSER 5**](#_Toc398753402)

[**4. FUTURE DEVELOPMENT 6**](#_Toc398753403)

[**5. CONTEXT DIAGRAM 8**](#_Toc398753404)

[**6. ACTIVITY DIAGRAM 9**](#_Toc398753405)

[**7. MODULE DIAGRAM 11**](#_Toc398753406)

[**8. CLASS DIAGRAM 15**](#_Toc398753407)

[**9. DISPLAY OUTPUT 16**](#_Toc398753408)

[**10. CRITICAL ISSUES 17**](#_Toc398753409)

[**12. CONCLUSION 18**](#_Toc398753410)

[**12. REFERENCE 18**](#_Toc398753411)

# Introduction

Code Analysis is the process of analyzing the code of the application to discover, review, validate or verify certain properties of the application. This is useful during the process of development and for testing the application before it is put in production mode, especially for checking the size, complexity and security related aspects. Code analysis can be classified from several perspectives:

**1. What We Analyse:** We can analyse **source code** or **binary code** (byte code) of the application. Both of these categories have their pros and cons.

**2. How or when we analyse:** We can analyse the code **statically** without executing it or **dynamically** while the application is executed. Static analysis, being conservative, is prone to false positive, but it is exhaustive. On the other hand, dynamic analysis, being very accurate, may miss certain behaviour which is not manifested in any of the execution monitored.

**3. Purpose of the analysis:** We can look for **flaws**, like NULL pointer dereferencing or passing ASCII string instead of the UNICODE string. We can look for **aspects** of the code like building various graphs of dependencies or deducing conditions when recursion will happen. The analysis also brings forward the size, complexities and relationship between various packages, types and member functions used in the application.

Some of the metrics defined for code analysis at module level are as follow:

* Source Lines of Code (SLOC)
* Non-commented Source Lines of Code
* Comment Lines
* Number of Methods
* Decision Density
* Cyclomatic Complexity
* Maximum loop nesting
* Maximum conditional nesting
* Halstead measures (volume, difficulty, effort, operators and operands)
* Software Engineering Institute (SEI) Maintainability Index
* Class and File counts

## **Executive Summary**

Code Analyzer is the C# Application to analyze other C# applications in terms of size and complexity. This managed code analyzer finds all the types, their members, and relationships with other types. It constructs two metrics for each member function: its size in source lines of code (SLOC), and cyclomatic complexity defined as the number of elements in its scope tree.

The purpose of this code analysis is to find functions that may need attention because of their size and/or complexity.

Code Analyzer is a powerful tool to understand the structure and design of the code as it reveals the relationship and interactions between all the types. Thus giving a reference point for various groups of users such developers, quality assurance testers and managers. It provides the customizable GUI for different set of users to pull out the relative information about the code according to the need and responsibility of different users.

## **Specifications**

1. Code Analyzer is implemented in C# using the facilities of the .Net Framework Class Library (FCL), version 4.5.
2. It accepts a specification of files using a path, file patterns, and options from the command line. The Analyze provides a command line option /S indicating that all subdirectories rooted at the path are to be searched for files matching all supplied patterns. If there is no /S on the command line only the cited path will be searched.
3. It provides output information that lists types defined within the file set and their function

names.

1. The Analyzer display function sizes, and complexities (in terms of scope levels) for each function in each file, and a summary for all the files in the specified set.
2. It provides a command line option, /R, which, when present causes the analyzer to display

the relationships between all types defined in the file set, e.g., inheritance, composition, aggregation,

and using relationships instead of the function sizes and complexities.

1. It displays all outputs on standard output. The analyzer also provides an option, /X, which

when present causes the output to also be written to a file in XML

# 2. USERs of the System

Code Analyzer satisfies the needs of various group of users with different requirements. Main groups of users are as follow:

1. **Developers:**

Developers can use this code analyzer to keep track of size and complexity of their application code. It will help them to keep the complexity of the code compliance with the planned design. Code Analyzer also has the option to display the relationships between the types such as classes, structs, enums etc. Developers can thus use this output information to check whether the code development is as per the architectural design of the software. All the relationships such as Inheritance, Composition, Aggregation and Using between the types defined in the given source files should be as planned in the technical design documents and UML diagrams defined for the project. Thus, this managed code analyzer keep an eye on the planned progress of the project.

Also the scope analysis of the each function limits the complexity of routines during program development. Developers should try to limit the number of scopes in the functions reasonably and code analyzer helps them count the scope levels automatically.

1. **Quality Assurance Analysts:**

Code Analyzer provides a powerful tool for quality assurance of the software development in managed languages like C#. The final report generated by the analyzer is one stop shop for all the necessary information need by the QA team.

QA team may not need to understand the internal implementation of the code to make their test cases. This facilitates black-box testing and QAs can identify the interactions between various types

Just by seeing the output report. The report can also be generated on the basis of scope levels of various functions defined in each file which will give an insight on the different independent branch-paths in the code. This is necessary for determining the number of test cases necessary for thorough testing of a software module.

1. **Managers**

Managers/Higher management of the project developments can use the code analyser to identify the complexity of the application. This will help in effort estimation of the any modification or enhancement needed in the software. This automated code analysis will give manager more confident understanding of the existing code rather than relying on error prone manual analysis.

# 3. Uses of the code analyzer

Code Analyzer can be used for various purposes such as follow:

1. **Unit Testing of Code by Developer:**

Developers can use the code analyzer for building their unit test cases by identifying all interactions between the types defined and also evaluate their code by checking on unwanted relationships between the classes.

1. **Understanding and Evaluating the Dependency structure of the software**

The relationships identified by the Code Analyzer can be used to identify unwanted dependencies between the classes and help in modular design of the code by removing such tight coupling between classes.

1. **To understand the complexities of the functions/ methods and improve performance of the overall system by increasing the cohesion between modules**

Methods defined in the classes implement the business functionalities and process data to get the desired results. The cyclomatic complexity of the functions in terms of scope levels in them gives an idea of number of independent paths through source code. This provides a way to quantify the complexity of the program and can be kept minimal by checking on the number of scopes in a function.

1. **To understand the impact of enhancements on the overall code**

The relationships identified by the analysis of the program will increase the visibility of dependency in the source code. This will help in identifying the areas of source code which will need modification due to recent enhancements in the functionality of the program.

1. **Calculating software source metrics**

It helps in calculating various source code metrics such as Source line of code, class and function count in each source file, different namespaces used throughout the program.

# 4. future Developments

Code Analyzer is built of independent modules which leads to a scalable product. Future upgrades can be made by adding more packages to implementing new functionalities consuming the services from the existing modules.

Some of the enhancements that can be made to this analyzer are as follow:

1. **Text Analyzer:**

The parser module used in this application can be used to implement Text analyzer in future. Implementation of text analyzer will increase the utility of the Code Analyzer providing a search mechanism to look for a particular text through a set of files/directory provided.

1. **Implementing more Source code metrics:**

The current application only provides the mechanism to provide two metrics for member functions: its size in lines of code, and complexity defined as the number of

elements in its scope tree. There are other metrics to quantify source code like Maximum loop nesting, Maximum conditional nesting, Decision Density, Cyclomatic Complexity in terms of independent paths in the code. All these can be implemented in future versions of the Code Analyzer.

1. **Flexible Reporting Capabilities:**

The current application provides the output of the analysis to the standard output and an option to generate an XML file of the result. The reports in future should also be able to export the results in HTML file and to desired relational database.

1. **Difference Spotter:**

The code analyzer could be scalable up to spot differences between highly identical files like updated versions of the same source code file.

1. **Identify Programming and design Issues:**

In a team environment of work culture, the same software is developed in parts by different developers ranging from few dozens to hundred. So, it will be highly useful if there is a testing algorithm to check main design rules of the project. The code analyzer could be upgraded to bring forward design and programming issue pertinent to a particular set of requirements. It can be customized to add new rules and policies to identify such faults in different source files developed by various programmers.

1. **Implement code analyzing for different file extensions**:

The current application analyses specifically the C# source files. In future, the modules for examining different file extensions such as .java, .xml, .cpp etc can be added.

1. **Compiling "little embedded languages"**

Code Analyzer can be upgraded to identify embedded language code and pass it to respective compiler engine. Like SQL queries are parsed in PL/SQL code and pass to SQL Engine to process the query. Also, like JavaScript embedded in HTML code can be differentiated using the concept of Code Analyzer.

# 5. COntext Diagram



*Figure 1 Context Diagram for Code Analyser*

Context Diagram gives a very high level view of the system. It identifies the major components of the system and give a brief overview of flow of information.

Code Analyzer in the diagram above is the main processing block which accepts input from the command line. The command line arguments contain information about:

* the file extensions which identify the set of files to be analyzed
* path of the directory to be searched
* options for specifying
  + - directory search : either all the sub-directories of the supplied path is searched for the given file extensions or just the cited path is to be searched
    - type of analysis : either relationships between the types to be displayed or the function sizes and complexities needs to be displayed

Analyzer then pulls up the files from file system and process the managed code for various code metrics and pushes the analysis report to standard output/XML File.

# 6. activity Diagram



Figure Activity Diagram for Code Analyzer

List of Tasks as mentioned in the Activity diagram above:

1. Parse command line arguments for file extensions, path and options.
2. Identifying set of files to be analyzed based on the path and option provided.
3. Parse each file for tokens.
4. Group tokens into grammatically meaningful semi-expressions.
5. Identify types and their functions in the list of source code files
6. Analyze functions for their line count and scope levels
7. Analyze types for relationship between them

# 7. Module Diagram



The above diagram represents the calling relations between the packages those were identified to be a part of Code Analyzer. These packages divide the code analyzer into independent modules designed to process a set of activities on given inputs and pass the result to the next package. Such division of software into cohesive packages leads to the scalability of the software for future enhancements as mentioned in the section above.

**Executor**

This is the starting point of Code Analyzer application. Executor is the central hub which co-ordinates and communicates between other packages. This package owns all the component instances

**ParseCL**

ParseCL is the module which parses the command line arguments to retrieve file extensions, directory path and options. Then it passes back this parsed information back to the executor for further processing.

**File Manager**

Executor passes the path directory, file pattern information and option from ParseCL package to File Manager package. On the basis of option and patterns, File Manager creates a set of files that needs to be analyzed. If option ‘/S’ is present then File manager recursively search for the required patterns in all the sub-directories of given directory path. Otherwise, it searches only in the cited path.

This package further uses Navigator module to implement the functionality to navigate through all the sub-directories in a given directory. File manager composes full path names of the files matching the patterns and send it back to executor as list of strings.

**Analyzer**

This package receives a list of files from the Executor. Analyzer passes each file to Parser to recognize all the types and, functions with their line count and number of scope levels into the repository. If the user has given the option /R, the Analyzer will then traverse the repository to detect all the relationships amongst the types such as Inheritance, Composition, Aggregation and Using.

**Parser**

The Parser package uses the Semi and Tokenizer packages to collect token sequences for analysis. It is essentially a container of rule detectors. Parser simply feeds the current token sequence to each rule in an internal rule container. When that is complete it requests another token sequence and repeats until there is nothing more to collect.

Each rule has a collection of actions. When the rule is satisfied by a token collection it invokes its actions. It is up to the action to decide what to do with the information contained in the token sequence when its rule fires. Each action is handed a reference to a data repository to store and retrieve information to carry out its task.

**Semi-Expression**

A semi-expression is a sequence of tokens that have meaning for code analysis. Semi provides a

class, CSemiExp, which extracts these token sequences according to the following rules:

1. A semi-expression is a sequence of tokens that ends with the terminal characters ‘{‘, ‘}’, ‘;’, or ‘\n’ if the previous tokens contain ‘#’.

2. Instances of the CSemiExp class collect semi-expressions from strings or file streams.

3. A semi-expression is usually exactly the right amount of code to make a decision about a grammatical construct. The open and closed braces, ‘{‘, ‘}’delimit changes in scope. And the ‘;’ terminates declarations and expressions. # … \n defines a compiler directive.

**Tokenizer**

Tokenizer package provides a CToker class that supports reading words, called tokens, from a string or a file stream. Tokens are constructed using rules that are appropriate for code analysis:

1. Transitions from alphanumeric to whitespace or punctuator characters, and back again, are treated as token boundaries.

2. Quoted strings and comments are returned as single tokens.

3. Some punctuators, that have special meaning in code, are returned as single character tokens, even if they are embedded within a longer string of punctuators.

4. Newlines, ‘\n’, are returned as tokens, but all other whitespace is discarded.

5. CToker reports all newlines it has process through its property lineCount.

**Rules and Actions**

RulesAndActions package contains all of the Application specific code required for most analysis tools.

It defines the following four rules which each have a grammar construct detector and also a collection of IActions:

- DetectNameSpace rule

- DetectClass rule

- DetectFunction rule

- DetectScopeChange

Three actions - some are specific to a parent rule:

- Print

- PrintFunction

- PrintScope

The package also defines a Repository class for passing data between actions and uses the services of a ScopeStack, defined in a package of that name.

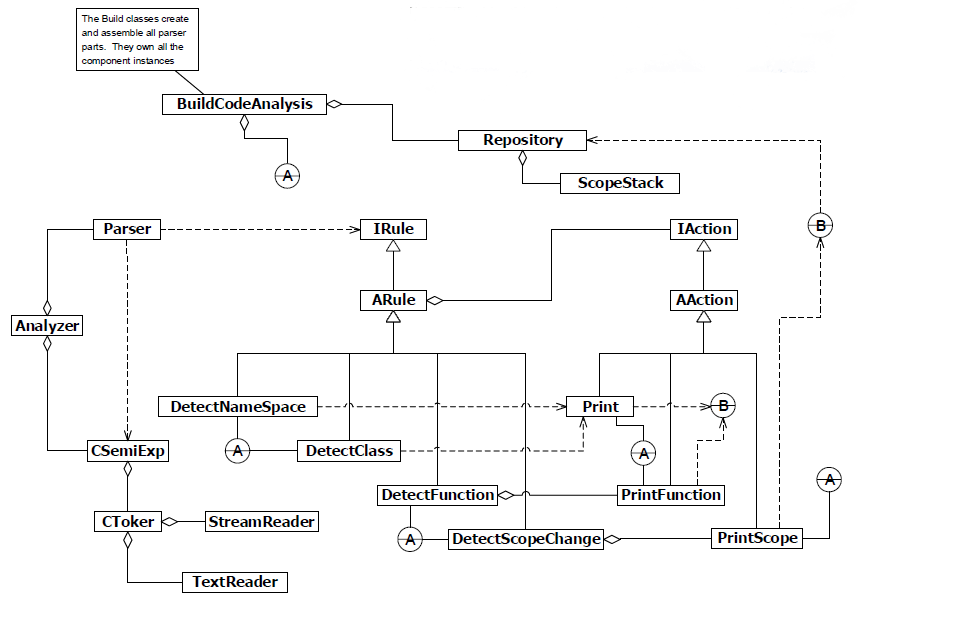
**Repository**

This is package which will store the information provided by the parser (via rules and actions) and the analyzer. This package uses a data structure to save the data gathered by the analysis. Finally it will pass the data to display package for further actions.

**Display**

This package receives the instruction of displaying data from the executor. It receive data from the repository and display the output on screen. If the user has specified the option /X then the output will be display in XML format.

# 8. CLass Diagram



The prototype of Class Diagram for the Class Analyzer is shown above. These classes/types are logically grouped in the discussed packages to implement each independent task for code analysis. They are organized in the following packages:

Executor Package: BuildCodeAnalysis

FileManager: TextReader and StreamReader

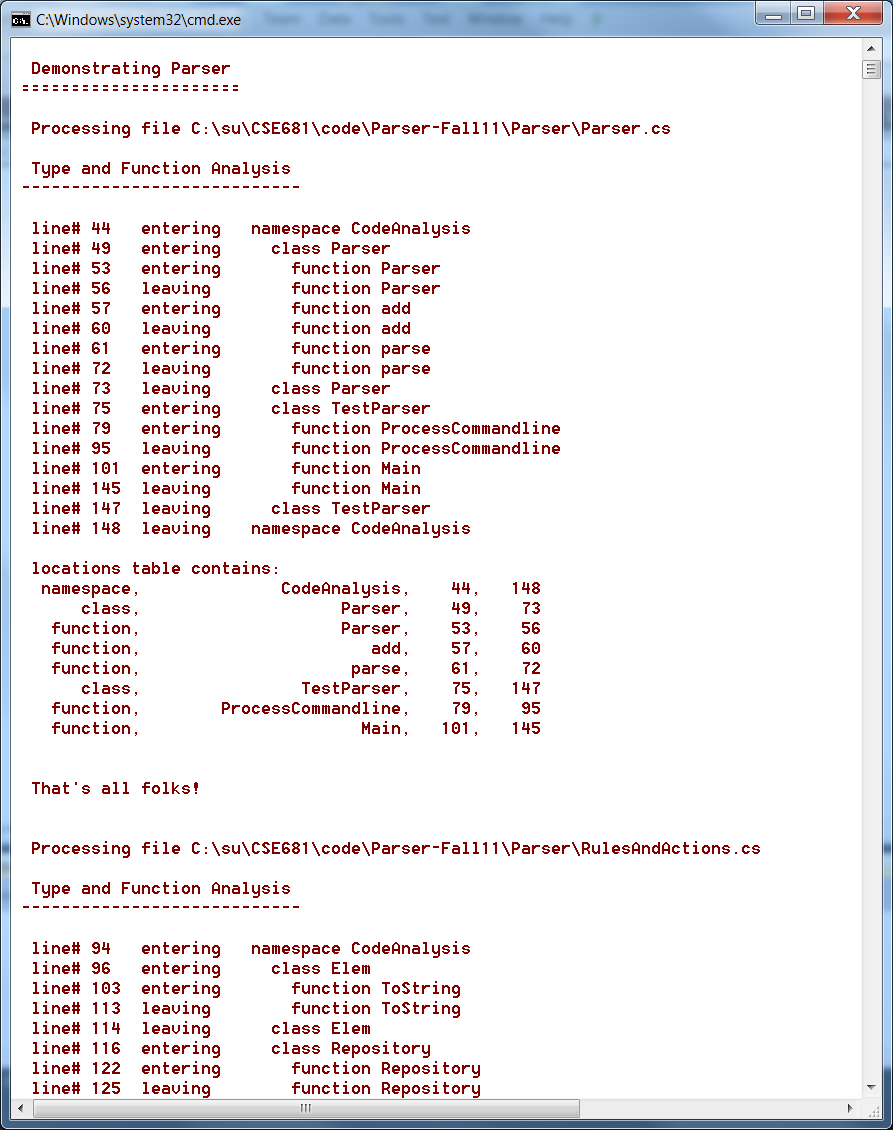
Analyzer: Analyzer

Parser: Parser, IRule, IAction

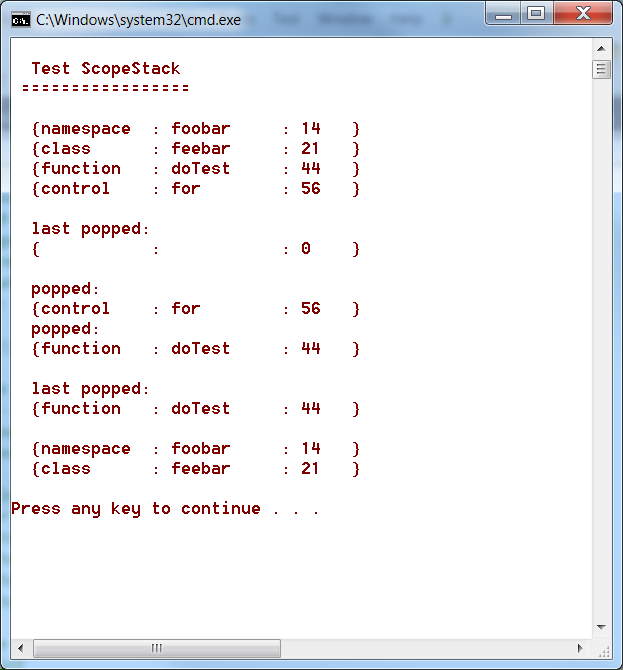
Rules and Actions: DetectNameSpace, DetectClass, DetectFunction, DetectScopeChange, Print, PrintFunction, PrintScope

# 9. display output

The prototype output for type finding process and scope analysis of function is designed as follow. This output window lists the types defined in the file and functions enclosed in each type. It also gives a preview of structure of the location table where all information is stored.



Scope Analysis functionality is tested by following output window which shows all the scopes defined in a file.



# 9. Critical issueS

Code Analyzer Application involves some critical tasks which needs to be implemented carefully. Otherwise, the whole application may break or respond in unexpected way. Amongst them few of the critical issues which may arise and are needed to be taken care of are:

1. **To calculate the complexity (No. of scopes) of all the functions:**

**Proposed solution:** To count the number of pairs of open and close braces.

**Issue:** Some scopes in the code are anonymous which are represented by just pair of braces. They are neither of a loop, conditional branching, or any function. Thus, the ideal way is to check for opening and its corresponding closing brace to identify a scope rather than on number of named constructs.

1. **To deal with function overloading:**

**Issue:** Overloaded functions are those which have the same name and class but have different parameters. Thus to represent them differently in the repository can be an issue.

**Proposed Solution:** To store the parameters and there types in the repository. Should an extra column in the repository to store this information.

1. **To retrieve the complete filenames that are matching the provided file pattern:**

**Proposed Solution:** To use the inbuilt functions of the SystemIO.Directory

**Issue:** Searching all the files in the directory and its sub directory for matching name as the pattern, as well as forming a list of the complete name of the file can be a critical issue. This can dealt by using the functions of the directory class in SystemIO namespace.

1. **To handle errors arising out of special cases:**

**Issue:** Error Handling can be a very critical issue as the system might break in case of an unhandled error. Possible forms of error can be file-not-found, file not opening, invalid path, wrong arguments, etc.

**Proposed solution:** Effective exception handling cases should be made in order to resolve any such issues. Code should stop after displaying a proper error message.

# 10. Conclusion

Code Analyzer is a powerful tool to analyze source code without manual intervention. The analysis report can bring forward some major issues which could not have been easily identified until later stages of development. Thus, it provides a patrol run for software development pointing out malicious defects in the code. This document discussed the component of the Code Analyzer and their interaction among each other. It also threw light on expansion of this product by exploring the new areas of application of same concept and re-usable modules. Finally, we also discussed the critical issues we could face in implementation of software architecture of Code Analyzer. If these critical issues are not considered seriously, then it might have resulted in a serious design flaw which may go undetected while implementation and it may be a serious bug in the final product.

# 11. References

1. [*http://stackoverflow.com/*](http://stackoverflow.com/)
2. *Software Modelling And Architecture – Course Website by Dr. Jim Fawcett*