CODE ANALYZER

Operational Concept Document

Zhe Li

zli89@syr.edu

9/14/14

CSE681 FALL2014

Table of Contents

1. Requirement 3
2. Executive Summary 4
2.1. Purpose 4
2.2. Potential Users 4
2.3. Basic Structure 4
2.4. Critical Issues 5
3. Introduction 5
3.1. Objective and key idea 6
3.2. Obligations 6
3.3. Organizing principles 6
4. Uses 6
4.1. Use Case Diagram 6
4.2. Actors 7
4.3. Description of uses by different actors 8
4.4. Extensions 10
5. Context Diagram 11
6. Application Activities 11
6.1. Main tasks and activities 11
6.1.1. Parse the command line 11
6.1.2. Retrieve file references, i.e. file set, recursively or non-recursively 11
6.1.3. For each file in file set, identify defined types, and save the defined type information 11
6.1.4. For each file in file set, calculate function size, cyclo complexity and a summary both for a single file and file set 12
6.1.5. For each file in file set, find all type relationship in the file 12
6.1.6. Display the analyzing results 12
6.1.7. Store analyzing results in XML file 12

6.1.8. Activity Diagram 12
6.2. Activities of Parser Facility 13
6.2.1. Open one file as semi expressions 13
6.2.2. Build the parser 13
6.2.3. Parse each semi expression 14
6.2.4. Display or Pack info 14
6.2.5. Activity Diagram 14
6.3. Activities of File Manager 14
6.3.1. Get all file names 15
6.3.2. Store filenames 15
6.3.3. Search all subdirectories 15
6.3.4. Activity Diagram 15
7. Partitions 16
7.1. Module Partitioning 16
7.1.1. Executive Module 16
7.1.2. File Manager Module 16
7.1.3. Command Line Process Module 17
7.1.4. Toker Module 17
7.1.5. SemiExpression Module 17
7.1.6. ActionAndRules Module 17
7.1.7. Repository Module 18
7.1.8. Parser Module 19
7.1.9. Analyzer Module 19
7.1.10. Display Module 19
7.1.11. XML storage Module 19
7.2. Package Diagram 20
8. Critical Issues 21

8.1. Invalid Input 21
8.2. Excessive file set 21
8.3. Unexpected changes during the operation 22
8.4. Non-writable path 22
9. Conclusion 22
Appendix – Execution Results 23
References 25

1. Requirement

This project develops a program for analysis of C# code. The analyzer, given a set of file references, finds all the types, their members, and relationships with other types. Further, the analyzer constructs two metrics for each member function: its size in lines of code, and complexity defined as the number of elements in its scope tree.

The Code Analyzer:

- Shall be implemented in C# using the facilities of the .Net Framework Class Library (FCL), version 4.5, as provided in the EECS clusters.
- Shall accept a specification of files using a path, file patterns, and options from the command line. The Analyzer shall provide 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.
- Shall provide output information that lists types defined within the file set and their function names.
- Shall display function sizes, and complexities for each function in each file, and a summary for all the files in the specified set.
- Shall provide 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.
- Shall display all output on standard output2. The analyzer shall provide an option, /X, which when present causes the output to also be written to a file in XML format.

2. Executive Summary

Code Analyzer serves as an effective tool used during the development process of a large software project, which will help different roles of users get familiar with the structure of the source codes by type analysis and other important issues of code evaluation such as cyclocomplexity. With the Code Analyzer, users can evaluate their code, which will make their code or products more efficient and robust.

2.1. Purpose

The purposes of this Code Analysis are:

- To scan C# code files in specific directory w/ or w/o recursively searching for analysis.
- To evaluate functions based on their size and complexity.
- To analyze the type-base relationship to give an overview of program structure.
- To store analysis results in file for further use.
- To help potential developer users get a quick understanding of the structure of a set of code.

2.2. Potential Users

Users can be Developers, Tester, Quality Assurance Analyst, Software Architect, Software Manager/Team leader, Maintenance Engineer and customer. Therefore, Code Analyzer is used throughout the whole software developing process.

2.3. Basic Structure

This project is basically a text file parser based code analyzing program. Generally, the code analyzer has following modules.

- Executive Module, a main controller of the program in charge of calling command line process to analyze the command line input and then based on the command line input to retrieve files calling file manager, then perform analyzing calling analyzer, finally call display module to show the analysis results. Additionally, it can perform more if new functionality is added with new modules are plugged in.
- File Manager Module, get all files based on the path and pattern and the command determining whether to retrieve files recursively with respect to subdirectories.
- Command Line Process Module, parse the command line option input following the executive name. Three options need to be parsed, whether to scan files recursively (/S), whether to save the results to XML(/X), whether to perform type relationship analysis or function evaluation analysis(/R).
- Toker Module, tokenizer split the file stream into tokens and identifies the each Token in the code

and output the tokens.

- SemiExpression Module, uses Token Module to construct a sequence of tokens which can naturally analyzed for code analysis.
- ActionAndRules Module, set the rules to the parser and define the actions according to the rules. Rules are used for detecting a specific syntax in a semi-expression, where actions affiliated to a specific rule are the analyzing actions when the rule is matched. Action will use repository module to store the analyzing results.
- Repository Module, used for storing the analyzing results and the temporary data when performing parsing.
- Parser Module, use semi expressions returned by SemiExpression module to perform all rules analysis one by one. It is the main controller of the parser facility and is the interface interacting with other modules.
- Analyzer Module, a major controller for file analysis who direct parser to analyze all files in the file set, including preprocessing the fileset by calling different parser. Additionally, analyzer has to build parser to meet different kinds of parsing needs.
- Display Module, retrieve data stored in repository and display them in standard format on console.
- XML storage Module, based on command line input, store analyzing results as XML format file.

2.4. Critical Issues

- Invalid command line input, illegal command line input should be handled.
- Performance, the total number of files should be concerned due to the control of analyzing efficiency and effectiveness. Multithread file analyzing is being considered.
- Unexpected change during parsing, a file whose name is stored in the fileset after searching is modified or deleted before being parsed. A mechanism to handle this situation should be considered.
- Failure to write output file, for example if the output file path is not writable, the writing will fail. A method to store analyzing results for backup should be explored.

3. Introduction

To develop software products effectively and efficiently, Code Analyzer is an essential tool for code evaluation. To help related users get quick understanding of the program structure, Code Analyzed is an indispensable tool for type based relationship analysis. From perspective of code evaluation, code

analyzer can show all function size and cyclocomplexity in code. For type based relationship analysis, code analyzer will give a map of relationship among all defined types. Moreover, the flexibility of handling file searching and result storing makes code analyzer more user friendly. Beyond a project, the code analyzer can be used during future software development career.

3.1. Objective and key idea

The objective of this tool is to show/store the structural information including type relationship and function size & complexity inside code files. The purpose of this code analysis is to find functions that may need attention because of their size and/or complexity. A secondary purpose is to help developers get a quick understanding of the structure of a set of code. The key idea is to achieve a code analyzer with a rules-and-actions based analyzing parser facility as the core and adding helper modules to the core, so that by customizing rules and actions, analyzing results will be stored in repository for further use, i.e. display or storing in XML file.

3.2. Obligations

The primary obligations of this tool would be

- To parse user command line input to ensure its validity and obtain user's command
- To provide a user-friendly way to control what files and what kind of information they want to analyze.
- To display the type relationship among all defined types for each type in a clear way
- To display the cyclomatic complexity and size of each function in each file and a summary for each file.
- To store the code information into the file for further use

3.3. Organizing principles

The organizing principles to achieve the main functionality using parser based analyzer module and separate other functionality into isolated modules which only communicate with the executive module.

4. Uses

4.1. Use Case Diagram

A use case is a list of steps, typically defining interactions between a role (known in UML as an "actor", an actor is who or what initiates the events involved in that task) and a system, to achieve a goal. Several user cases are depicted in Figure 1:

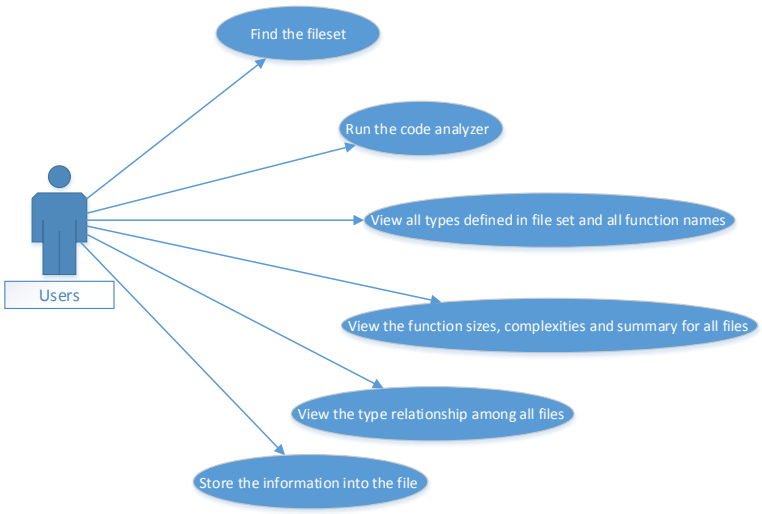


Figure 1 Use case diagram

1) Find the file to be analyzed

   User has to provide path and pattern and option that determines if the files to be analyzed should include the files in subdirectories of specific path.

2) Run the code analyzer

   User has to user command line to run the program

3) View all types defined in file set and all function names

   User has to list all types defined in file set and all function names

4) View the function sizes, complexities and summary for all files

   User has an option to see function sizes, complexities and summary for all files.

5) View the type relationship among all files

   User has an option to see type relationship among all files

6) Store the information into the file

   User has an option to save analysis result to XML file.

4.2. Actors

Users play an important role in the whole software developing process. A robust product has to satisfy multiple kinds of users' needs. Following is the list of potential users.

- Developer

  Developer is the person concerning facts of the software development process. Their work

includes researching, designing, developing software.

- Tester

  Testers write test plan and test case to test the functionality of various modules in project. The details they are writing in the test scenarios based on the understanding of testing modules.

- Quality Assurance Analyst

  Quality assurance analyst monitor the software engineering processes and to ensure quality.

- Software Architect

  Software Architect is in charge of the structure of a big program. Refactoring is one of important work to Software Architect. When the software product is finished and comes into the debugging stage, Software Architect may overview their code and makes some refactoring to make the product simpler and robust.

- Software Manager/Team leader

  Software manager or a team leader has to monitor the whole design process and test process. Also the developing progress is one of software manager's concern.

- Maintenance Engineer

  Maintenance Engineer usually finds errors in the products by themselves or from customers. They may report this problem to the developer or fix it by themselves.

- Customer

  Customers are the buyers after software is developed and tested.

4.3. Description of uses by different actors

- Developer

  When a new developer joins a new project, they will meet thousands of files and probably get lost in the countless code. Learning structure is very important and should be the first step for development. Putting the file into the analyzer, he will know the defined types' relationship with each other as well as total number of the modules, the number of the classes and functions. He will know the function size and logic structure of each file. He will understand which file is complicated by looking at complexity. All these information are output into a structural xml file which is convenient to them for a future review.

  When the software is being developed, the experienced developer always look back to review the structure of written code, evaluate the operation of the software. The developer may find size of some function he wrote before is too large involving too many activities. Hence, it can be divided into several smaller ones. Moreover, if the cyclomatic complexity is high, they may reconsider the

structure of this function.

- Tester

  Testers will write the test scenarios to test the software, which must be based on the fully understanding of the projects. The tool may save a lot of time for testers to read the codes, helping them gain an insight of the code. Also, testers can use Code Analyzer to find out the testing bugs belong to which lines or in which functions or classes and report the fails and give a clear feedback to the developers.

- Quality Assurance Analyst

  It is difficult for Quality Assurance Analyst to get an overview of projects. But with the code analyzer, Quality Assurance Analyst can evaluate the project and estimate the quality of projects. They can see the number of package, classes and functions. They can see the complexities of each function which are good indicators when evaluating the robustness and efficiency of the project. They can analyze type dependency complexity based on type relationship analysis. When they find some problem occurred related to the functionality of the projects, or any code file offense coding regulation such as big cyclocomplexity of function or unexpected circular type dependency, they can also give the feedback to the test team or developer.

- Software Architect

  Suppose the type dependency construct a circle, software architect may consider refactoring to break the circular dependency to make the structure maintainable and clear. When there is a need that the current project needs to add a new feature. The changes brought by the new feature shouldn't affect the current project two much. Code analyzer surely will help the Software Architect save a lot of time to make the new feature adjust the current projects by reviewing the structure of projects and scale of it.

- Software Manager/Team Leader

  At beginning phase of software development, not all class implementation is achieved. But the manager can still use code analyzer to see type relationship, even the relationship map is incomplete. As development progress, the relation map get closer to complete, the manager can monitor the progress of the project by periodically seeing the relationship map and each function details. Moreover, the manager can monitor the development quality through the functionality of function evaluation of code analyzer.

- Maintenance Engineer

  When the project is finished and delivered to the customer, Maintenance Engineer should support the project update based on understanding the structure of the project. With the Code Analyzer, they quickly understand the projects and know what will happen after update. If some problems occur, they can quickly locate the problem with the help of code analyzer and find what

problem is, for the code analyzer obtains the type relationship structure and function information. And with the output file information, they can give the accurate feedback to the customer.

- Customer

  Generally, the customers do not know much about the software programming. By using Code Analyzer, customers can have a clear view about the structure of the software. They can have the whole structural understanding about the software, make sure the functions they needed have been realized or not, and obtain a clear type relationship graph. They are able to think as a developer and put their own idea into the products by giving feedback to developing team.

4.4. Extensions

Considering the type relationship analysis, several extensions for this project can be proposed for improvement in the future version.

- Graph based type relationship monitoring

  Now since the requirement only requires demonstrating the relationship between two types. Thus using a two point format can achieve storage such as A inherits from B, B compose C. Furthermore, obviously, the type relationship can store in a graph structure, which each type is a vertex and the relationship between two types is an edge. Different kind of relationship can be distinguished by using edge name or attributes. Using graph structure can help users visually understand the structure of the program and find strong component to eliminate circular dependency.

- File dependency analysis

  Based on the type dependency analysis, dependency analysis can be extended to file scope. With file dependency, we can obtained the basic file metadata, once a file has been changed, all files depending on it should be retested.

- Graphic User Interface(GUI)

  Since the code analyzer is based on C#, it is easy to design a GUI using WPF encapsulating code analyzer to give user a better human-machine interaction and a better visual comprehension. For example, for previous extension, the graph based type relationship monitoring, the GUI can show the graph directly which help users have an intuitive understanding.

- Multi-thread handling with the files

  Due to the large amount of analyzed files, multi-thread mechanism is under consideration. Delegate or Thread can be used to concurrent program the analyzing process. It is noted that value access program must be handled such as using Thread Synchronization Mechanism. In this way, we turn time factor to system performance vector, that is, we exploit the abundant ability of system to save time running our analyzing program.

5. Context Diagram

This diagram Figure 2 Context Diagramshows how the code analyzer interacts with the external environment. The code analyzer mainly interact with the console application in this project. The code analyzer needs the input from the command line. And when the code analyzer gets the results concerning the code information, the output will also show in the console application.

Besides, the code analyzer will interact with the file system. The path and pattern are given to the file manager module, and this module will use these info and exploit file system to search files. For saving to XML file module, the code analyzer must have the permission to read and write the files through the file system. Basically, whichever LINQ to XML or DOM encapsulates the writing process into member function.

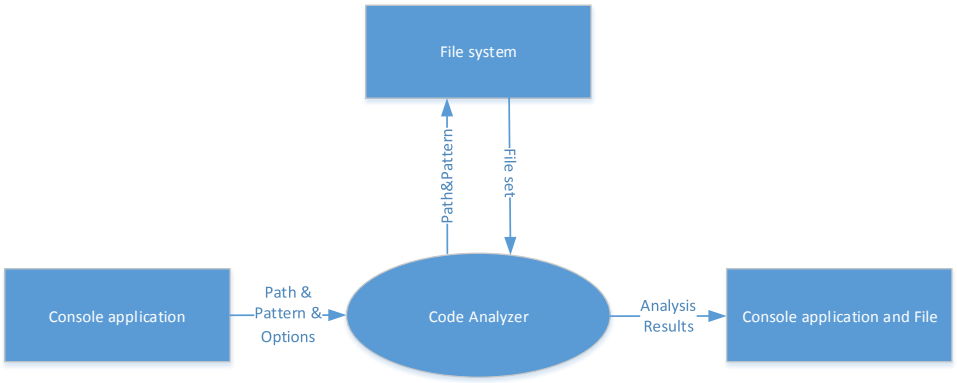


Figure 2 Context Diagram

6. Application Activities

6.1. Main tasks and activities

This project can be separated into following tasks

6.1.1. Parse the command line

The program will parse the user input to separate the command line input into path, pattern, and options.

6.1.2. Retrieve file references, i.e. file set, recursively or non-recursively

Based on the option /S and path, pattern, the program should retrieve files in the path with specific pattern. If /S option is entered by the user, the program should retrieve file in both path and all subdirectories under the path recursively.

6.1.3. For each file in file set, identify defined types, and save the defined type information

If /R option has been chosen, the program has to analyze type relationship. First of all, the program should identify all defined types in all files in file set.

6.1.4. For each file in file set, calculate function size, cyclo complexity and a summary both for a single file and file set

If /R option has not been chosen, the program has to analyze function info for each file including function size, cyclomatic complexity and for each file, a summary need to be concluded. Finally, the summary for file set will be concluded.

6.1.5. For each file in file set, find all type relationship in the file

After all defined types are identified, called preprocessing, based on identified types, the program will analyze type relationship for each file. Finally, the type relationship analyzing results will be stored inside program.

6.1.6. Display the analyzing results

Whichever kind of analysis user chose, the program will display the analyzing results.

6.1.7. Store analyzing results in XML file

If /X option has be chosen, the analyzing results will be stored to XML file. If not, the program will come to the end.

6.1.8. Activity Diagram

Basically, these tasks are concluded from the activity of the executive module. Thus this program can be described using activity diagram as Figure 3 shows, we can see at the beginning of the code analyzer running, it will parse command line first. Then base on whether there is /S command, search files with different ways. Then if the user requires to take a relationship analysis (/R) it will parse the file set twice, one for identifying all defined types, the other for finding all type relationship. If user requires to take a function evaluation analysis, it will analyze each file to give related info. After analyzing, it will display the result. Finally based on the user command /X, it will save the result into XML file or not.

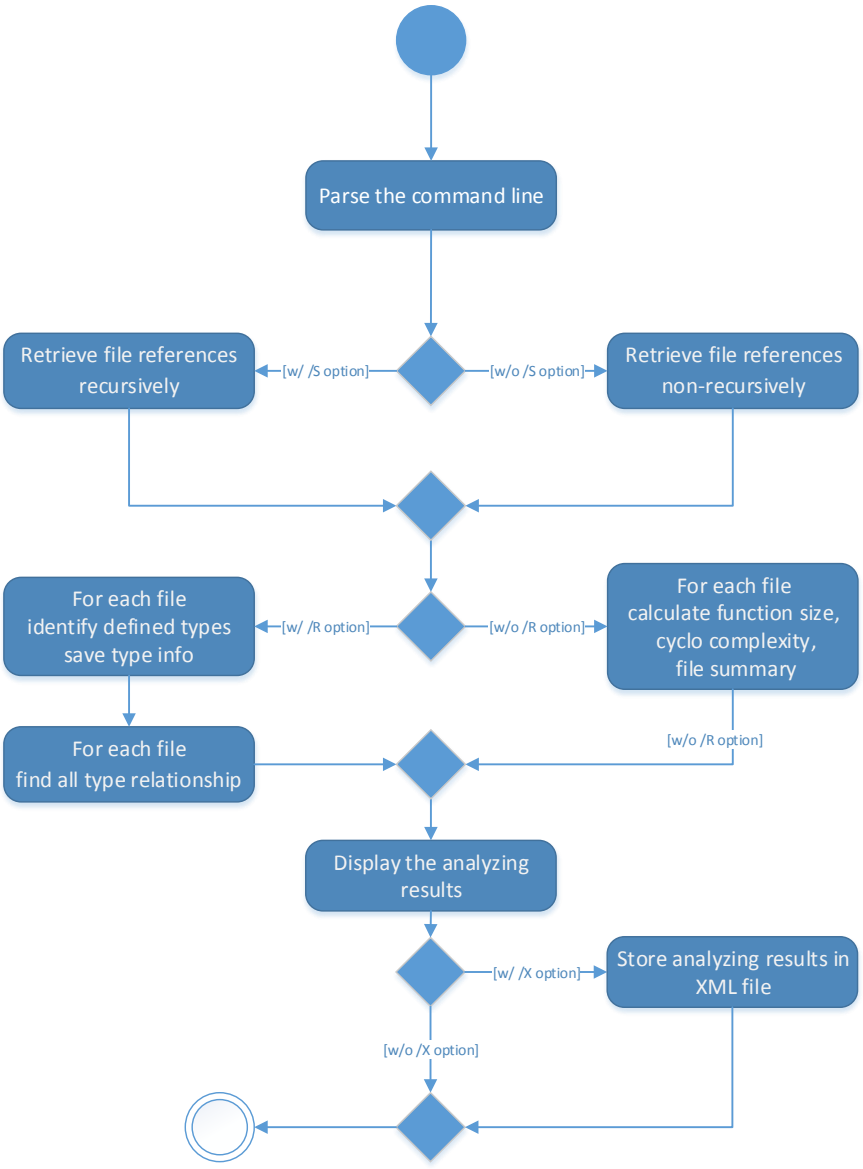


Figure 3 Activity Diagram for Executive

6.2. Activities of Parser Facility

Parser Facility is the core part of this code analyzer, comprehension of its structure and behavior can help to design the whole project.

6.2.1. Open one file as semi expressions

The parser facility will open a file and read its content as semi expressions.

6.2.2. Build the parser

The facility has to build a parser by adding rules and actions to the parser (Builder Pattern)

6.2.3. Parse each semi expression

User the parser to parse each semi expression in the file

6.2.4. Display or Pack info

Display the parsing result Or pack the results into specific data structure for further use.

6.2.5. Activity Diagram

The activity diagram can describe the work flow clearly as Figure 4 shows, parser facility will open a file and split it into semi expression sequence. Then it will build the parser by adding rules to the parser and actions to the rules. Then for each semi expression in the file, it parse the semi expression for analysis. Finally, the analyzing results will be display simply in an intuitive way or stored in the repository.

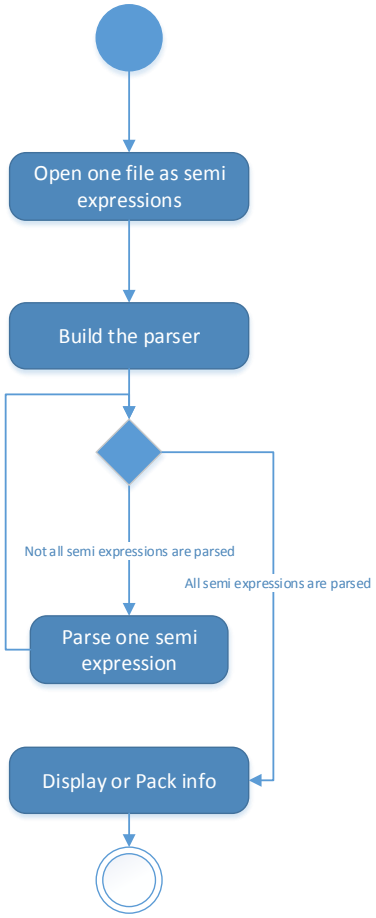


Figure 4 Activity Diagram for Parser Facility

6.3. Activities of File Manager

The program should have a file manager to retrieve all file references.

6.3.1. Get all file names

The file manager should first use the path and pattern given by users to get matched file names in the given path

6.3.2. Store filenames

The file name obtained in 6.3.1 should be stored in a file set.

6.3.3. Search all subdirectories

The file manager should search all the subdirectories of given path with specific pattern, which mean, a subdirectory gives a new path, file manager should repeat 6.3.1 and 6.3.2 recursively with respect of each new path.

6.3.4. Activity Diagram

The activity diagram can describe the work flow clearly as Figure 5 shows, file manager first get matched file names based on the user input path and pattern, the store the filenames in a file set. If user requires a recursive searching, it will get all the subdirectories in given path, and for each subdirectories as the new path and repeat searching, storing file names. If there is no subdirectory or user does not require recursive searching, the file manager will come to end.

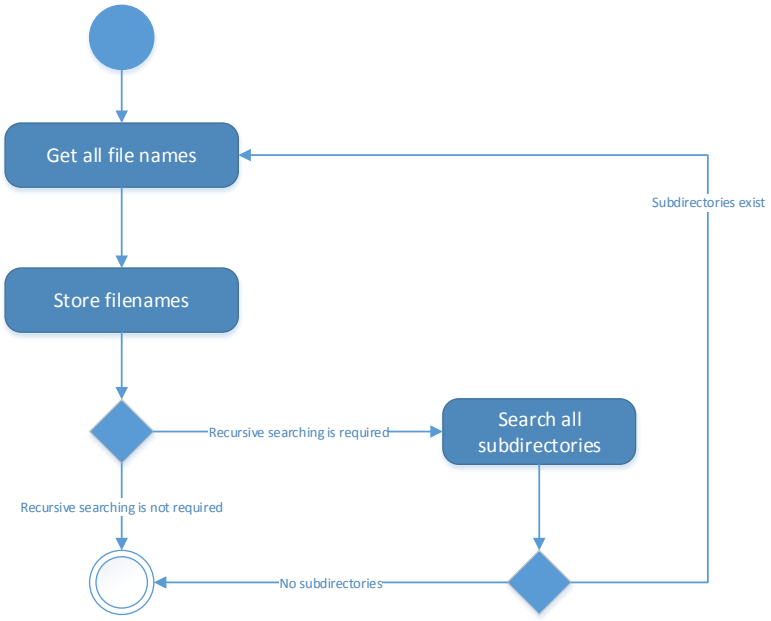


Figure 5 Activity Diagram for File Manager

7. Partitions

7.1. Module Partitioning

Based on the activities, each task becomes a package candidate. Because in this project, parser is playing the leading role, thus parser can break up into several packages. Thus the code analyzer will include following modules.

1) Executive Module
2) File Manager Module
3) Command Line Process Module
4) Toker Module
5) SemiExpression Module
6) ActionAndRules Module
7) Repository Module
8) Parser Module
9) Analyzer Module
10) Display Module
11) XML storage Module

Among all modules, 4) to 8) completes the functionality of a parser. The responsibility of each module will be described in detail below.

7.1.1. Executive Module

Executive module is the main controller of the program in charge of calling other modules in sequence to finish the analysis. No code analyzing will be performed in this module. To be specific, it calls command line process to analyze the command line input and then based on the command line input to retrieve files call file manager, then perform analyzing calling analyzer, finally call display module to show the analysis results and XML storage module to save the XML file.

- Classes and interfaces Used:

Executive – to call each module it depends on and achieve functionality of the project.

7.1.2. File Manager Module

File Manager Module facilitates the C# built-in I/O API to access filesystem and get all files based on the path with specific pattern and the command determining whether to retrieve files

recursively with respect to subdirectories.

- Classes and interfaces Used:

FileMgr – pass in the path, pattern, the /S option, return file set after search.

7.1.3. Command Line Process Module

Command Line Process Module parses the command line option input following the executive name. Three options need to be parsed, whether to scan files recursively(/S), whether to save the results to XML(/X), whether to perform type relationship analysis or function evaluation analysis(/R).

- Classes and interfaces Used:

CommandLineParser – parse the command line input, and extract path, pattern and options.

7.1.4. Toker Module

Toker Module is a basic module which supports reading tokens, from a string or a file stream. The token can either be the letter and number or the punctuator characters. They are identified by the token boundaries. Quoted strings and comments are returned as single tokens.

- Classes and interfaces Used:

CToker – split the input string or file stream into tokens.

7.1.5. SemiExpression Module

SemiExpression Module converts the tokens into the semi-expressions. SemiExpression Module identify grammatical terminator such as curly brace, semicolon. The SemiExpression Module call the Tokenizer Module and return the semi-expression to the Parser Module.

- Classes and interfaces Used:

CSemiExp – extracts tokens sequences according to specific terminator and return semi expression for parsing.

7.1.6. ActionAndRules Module

ActionAndRules Module set the rules to the parser and define the actions according to the rules. Rules are used for detecting a specific syntax in a semi-expression, where actions affiliated to a specific rule are the analyzing actions when the rule is matched. Action will use repository module to store the analyzing results.

- Classes and interfaces Used:

1) IRule – An interface to be implemented by ARule.

2) ARule – An abstract class to define rules class structure and implement some common

operations.

3) IAction – An interface to be implemented by AAction.
4) AAction – an abstract class to define actions for rules and implement some common operations.
5) DetectAnonymousScope – rule to detect anonymous scope such as single left curly brace.
6) DetectLeavingScope - rule to detect leaving scope such as single right curly brace.
7) DetectClass – rule to detect class, struct, enum and interface name.
8) DetectFunction – rule to detect functions.
9) DetectNamespace – rule to detect namespace names.
10) DetectCycloComplexity – rule to detect keywords that contributes to cyclo complexity of function
11) DetectInheritance – rule to detect inheritance type relationship.
12) DetectCompositionAggregation - rule to detect Composition and Aggregation type relationship
13) DetectUsing - rule to detect use type relationship
14) PushStack – action to push the scope information into scopestack
15) PopStack – action to pop the scope information from scopestack
16) PrintFunction – action to print function without comment.
17) Print – action to print any semi expression passed in.
18) Increment – action to increment cyclocomplexity count when complexity keywords are detected.
19) PushInheritance – action to push inheritance relationship info to repository
20) PushCompositionAggregation – action to push composition and aggregation relationship info to repository
21) PushUsing – action to push use relationship info to repository

7.1.7. Repository Module

Repository serves as a temporary container of all the analyzed information. Repository is aggregated by different container, such as scopestack and ArrayList.

- Classes and interfaces Used:

1) Scopestack – a defined data structure to store scope information

2) Elem – a container to store all information used during parsing analysis

3) Repository – a container storing scope stack and the scope list

7.1.8. Parser Module

SemiExpression module first partition file stream to semi expressions and pass each on to Parser Module. Parser module uses semi expression returned by SemiExpression module perform all rule tests one by one.

- Classes and interfaces Used:

Parser – a major controller of parsing facility and the window to interact with other modules

7.1.9. Analyzer Module

Analyzer Module is a major controller for file analysis who direct parser to analyze all files in the file set, including preprocessing the fileset by building a different parser. Additionally, analyzer has to build parser to meet different kinds of parsing needs.

- Classes and interfaces Used:

1) CodeAnalyzer – a control panel to wrap up calling of parser and for different analyzing needs it has different wrapper function.

2) BuildCodeAnalyzer – add actions to specific rules and add rules to the parser to build it, finally return the customized parser.

3) BuildCodeAnalyzerR – build a parser for type relationship analysis.

4) BuildCodeAnalyzerF – build a parser for function evaluation analysis.

7.1.10. Display Module

Display Module retrieves data stored in repository and display them in standard format on console.

- Classes and interfaces Used:

Display – based on the type of analysis, display the repository in different ways but all in standard format.

7.1.11. XML storage Module

Based on command line input, XML storage Module construct the analyzing results to structural format and stores analyzing results as XML format file.

- Classes and interfaces Used:

SaveToXML – based on LINQ to XML package built in C# (or XML DOM package for C#), save the analyzing result in xml file in a readable format.

7.2. Package Diagram

The package diagram is an abstraction of high level architecture of Code Analyzer and gives a dependency and communication understanding among packages. It is shown in Figure 66 as follows.

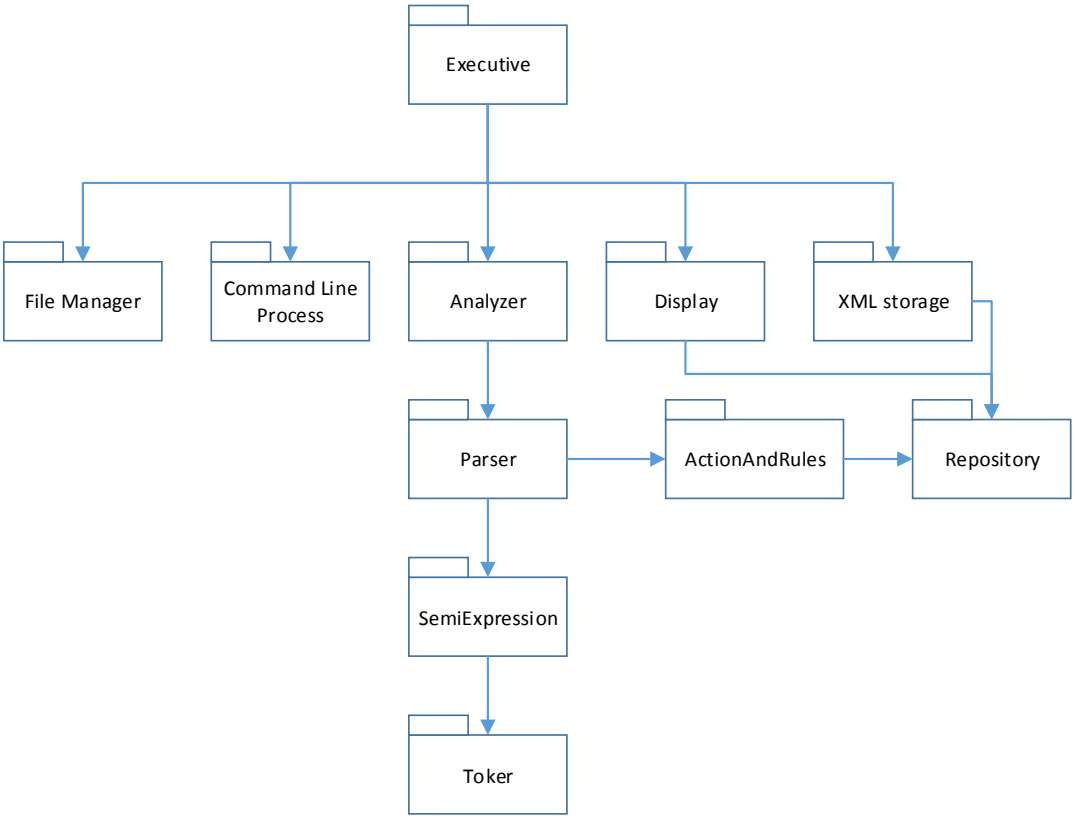


Figure 6 Package Digram for Code Analyzer

This diagram shows that the dependency and communication between packages. Executive act as the human-machine interface which will controls the operation of the whole software. First, executive will call command line process to parse the command line input into path, pattern and options. Then it uses the path and patterns returned to call file manager to retrieve file set. Then the file set is passed to analyzer who facilitate parsers to parse the files in file set.

For each file, Toker will split it into sequence of tokens and semiexpression will construct semi expressions. The parser will use these semi expression to take rule tests, if the rule tests passes, the affiliated action will be executed to push the analyzing information into repository. After analyzing. Executive will call display to show the results. And based on the option parsed by command line process module, executive will call XML storage package which use repository's analyzing data to construct XML file.

8. Critical Issues

8.1. Invalid Input

In reality, users will always enter command using command line in a different way from what developers expects. This kind of situation causes invalid input which terminates the program at the very beginning of run time.

- **Problem**: if the user enter the command arguments without order of path, pattern and options, the program will throw an exception because of no correct files are retrieved.

  Solution: the command line process module should be able to detect and handle invalid argument input and interrupt the execution of the program throwing a warning to the user. But more user friendly way is to parse path, pattern and options even the argument order is scrambled, and let the program run as usual. Identifying options is intuitive by judge if the string match option identifiers. And identifying path can be based on identifying the last back slash to tell if it is a path. The left one is the pattern. Another way is education. A readme file can embedded to show at the beginning of the program telling the format of argument input.

- **Problem**: if the user enter unexpected character in any command argument (it is OK in options but serious bug in path and pattern), the program will retrieve no correct files.

  Solution: This problem won't cause any exception before the file manager uses command line arguments to search files. This can be handled by sending a warning information to users when no expected files are found. Education is the best way for this kind of problem, and our responsibility is to maintain program flexibility so that it won't crash with invalid input.

8.2. Excessive file set

Code Analyzer need to handle thousands of files in the path directory or moreover including files in subdirectories in the path. This may cause performance problem and display problem

- **Problem**: A huge project can have thousands of files and millions of lines of code, if the code analyzer analyze files one by one, it will take a lot of time.

  Solution: To improve the performance in time domain, we may exploit the high performance computing ability of processors and memory, which means we may use multithread to handle the file parsing process. Multiple files can put into a blocking queue, and the code analyzer will start several new threads to get file from the queue and run the parsing process. Then they save the analyzing info in a thread-safe repository. In this way, it saves lot of time compared to analyzing files one by one.

- **Problem**: Due to the large amount of analyzed file, the console output can occupy thousands of pages in console that exceed the page limit, thus users will get the output at the front lost.

And since so many files' results listed in the screen, the scroll bar will be very small and it is hard for user to locate specific file's result.

Solution: A GUI in further work will be developed to handle this problem, a file name list with the order of first letter will be provided for users to locate the specific file's result by clicking. Moreover, user can search the file name to locate its results. In this way, the results will be displayed clearly to users.

8.3. Unexpected changes during the operation

A file whose name is stored in the fileset after searching is modified or deleted before being parsed. A mechanism to handle this situation should be considered.

- **Problem:** suppose a file exists when file manager searches the files and stores its name into file set, but later before the parser want to parse this file, the file is modified or deleted. Code analyzer will perhaps fail to parse this file.
- **Solution:** the file set can not only record the file name but also its metadata including last modified time. When parsing, before parser open the file, it need to check the metadata to ensure it is the same as what file manager searched. If it is modified, due to last modified time, parser will find that analyzed file has been changed and throw a warning to users letting them check the file. If it is deleted, surely parser will know something wrong. A warning will prompt to users to check the deleted file.

8.4. Non-writable path

- **Problem:** Some paths in users' OS is write-permission denied such as a user has a MAC OSX and he use bootcamp to install a Windows 8.1, when launching Windows, he will see the MAC OSX files is in D:, but path D:/ is not writable (but it is readable)because it is reserved for MAC OSX. At run time, if we set XML file path as non-writable path, in this example, if the file storage path is D:/, the storing process will fail, and the analyzing result will be lost after program execution.
- **Solution:** An exception handle mechanism should be explore when the file writing attempt fails. The program should give the user a warning first to tell users, the path is not valid. And give user second chance to enter a valid path to save the file. Repeat this process until the file is successfully saved.

9. Conclusion

This operational concept document first analyzed the requirement of the code analyzer project, and then specified potential users who will facilitates this tool. The use case diagram showed the interaction between users and the program. And detailed analysis about different actors was stated later including

the role they are playing for using this program, based on their roles why they may use this program and what they can benefit from this program. Moreover, the extension was addressed to make this tool more robust and effective as well as enhance this tool's user experience.

Then we exploit context diagram to see the interaction between the major part of the tool and external environment. This will help us to consider potential critical issues for the code analyzer which may rise during the interaction with external environment.

The detailed analysis for the program requirement was delivered through the application activities in the following section. The development of the tool was separated into tasks. The tasks described by activity diagram were discussed detailed later to take a top-down, high level design. Next partitioning the tasks to modules (packages) was described in which each module's responsibility is clearly discussed in response to the activities. A package diagram then showed the interaction and relationship among different modules to give a high level overview of the code analyzer. This kind of top-down design methods help developers understand the architecture of the whole project and take an increment design.

Finally, we discussed the critical issues which may cause problem when using this tool, and possible solutions are given to deal with the issues.

Generally, based on above analysis, a clear reasonable structure of the code analyzer is developed which meets the requirement. To achieve the program coding process module by module helps developers finish designing this tool incrementally. And the code analyzer can be an effective and efficient tool for different kind of users analyzing their code.

Appendix – Execution Results

By the submission of this OCD project, I have finished basic functionality of the code analyzer, thus execution results will be shown. The input of the code analyzer will be a path and the pattern followed by options.

- If the input is "executive.exe ./ CodeAnalyzer.cs /S ", the code analyzer will search files recursively into subdirectories of current path (./) named CodeAnalyzer.cs and analyze its function information. The result is shown as follows,

```
 Testing Function Size And CC
 ========================


 Processing file D:\Dropbox\Syracuse University\SEM 5\CSE 681\Project2\Project2
\CodeAnalyzer.cs

 Function Analysis
 --------------------------

 Function Name              Size   CC
 PreProcess                 33     7
 AnalyzeRelationship        74     2
 AnalyzeFunctionSizeAndCC   50     6
 Main                       21     1

 Module Summary
 --------------------------
 lines of function code: 178
 lines of code text: 198
 cyclomatic complexity: 16
```

Figure 7 Testing Function size and CC analysis

- If the input is “executive.exe ./ *.cs /X /R”, the code analyzer will first preprocess all the .cs files retrieved and the analyze type relationship file by file. At last, the relationship information will be stored in XML file. The result is shown as follows,

```
 Testing Relationship
 ========================


 Processing file D:\Dropbox\Syracuse University\SEM 5\CSE 681\Project2\Project2
\Parser.cs

 Type and Function Analysis
 --------------------------


 locations table contains:
 relationship              source name            target name
 Aggregation,                    IRule,                  Parser
      Using,                     IRule,                  Parser
      Using,                  CSemiExp,                  Parser
 Aggregation,         BuildCodeAnalyzer,              TestParser
 Aggregation,                    Parser,              TestParser
 Aggregation,                Repository,              TestParser
 Aggregation,                      Elem,              TestParser
```

Figure 8 Testing type relationship analysis

And the XML file is stored in following format,

```
▼<Source name="BuildCodeAnalyzer">
   <Relationship name="Inherit From"/>
  ▼<Relationship name="Aggregates">
     <Target>TestParser</Target>
   </Relationship>
   <Relationship name="Composes"/>
   <Relationship name="Uses"/>
 </Source>
```

Figure 9 XML file format

Code snippet is pasted here, but they are probably modified during final release of the code analyzer

```
public class FileMgr
{
    private List<string> _files = new List<string>();
    //-----------< Default Constructor >-----------------------------
    2 references
    public FileMgr() { }
    //-----------< get the file name list >-----------------------------
    2 references
    public List<string> getFiles()
    {
        return _files;
    }
    //-----------< get the filename with specific pattern recursively >-----
    3 references
    public void retrieveFiles(string path, string pattern,bool isRecursive)
    {
        path = Path.GetFullPath(path);

        // get all files in this directory and display them

        string[] files = Directory.GetFiles(path, pattern);
        foreach (string file in files)
        {
            string name = Path.GetFileName(file);
            _files.Add(path+name);
        }
        // for each subdirectory in this directory display its files
        // recursively
        if (isRecursive)
        {
            string[] dirs = Directory.GetDirectories(path);
            foreach (string dir in dirs)
                retrieveFiles(dir, pattern,isRecursive);
        }
    }
```

```
public class SaveToXML
{
    private Dictionary<string, Dictionary<string, HashSet<string>>> relationmap_;
    1 reference
    public SaveToXML(Dictionary<string, Dictionary<string, HashSet<string>>> relation_map)
    {
        relationmap_ = relation_map;
    }
    1 reference
    public void writeXML()
    {
        XmlDocument xmldoc = new XmlDocument();
        XmlDeclaration xmldecl = xmldoc.CreateXmlDeclaration("1.0", "UTF-8",null);
        xmldoc.AppendChild(xmldecl);
        XmlElement xmlelem = xmldoc.CreateElement("RelationshipMap");
        foreach(var c in relationmap_)
        {
            XmlElement xe1 = xmldoc.CreateElement("Source");
            xe1.SetAttribute("name", c.Key);
            foreach(var r in c.Value)
            {
                XmlElement xe2 = xmldoc.CreateElement("Relationship");
                xe2.SetAttribute("name", r.Key);
                foreach(var t in r.Value)
                {
                    XmlElement xe3 = xmldoc.CreateElement("Target");
                    xe3.InnerText = t;
                    xe2.AppendChild(xe3);
                }
                xe1.AppendChild(xe2);
            }
            xmlelem.AppendChild(xe1);
        }
        xmldoc.AppendChild(xmlelem);
        xmldoc.Save("./data.xml");
    }
}
```

References

1) http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Lectures/StudyGuideOCD.htm
2) http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/
3) http://edn.embarcadero.com/article/31863
4) http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/projects/Pr2F14.pdf