

OCD for Code Analyzer

CSE681 – Software Modeling and Analysis

Instructor: Jim Fawcett

Rundong Zhou

SUID: 234944570

Sep 14, 2014

Contents

1. Executive Summary	1
2. Introduction	1
3. Uses.....	3
4. Application Activities	4
Parse command line arguments	4
Retrieve input file references.....	4
Identify defined types	6
Find type relationships.....	6
Display final results	6
5. Partitions	8
Executive	8
CMDParser	9
FileMgr	9
Parser	10
Semi	11
Tokenizer	12
Repository	13
Display	13
BasicDisplay	14
RelationDisplay	15
XMLGenerator	16
6. Critical Issues	17
7. APPENDIX	20

1. Executive Summary

Code Analyzer is a single executable consists of six main parts, Executive, Cmd Parse, File Managing, Repository, Parse and Display. Each part is composed of one or several packages. Code Analyzer is used to 1) analyze the size and complexity of member functions in C# projects. 2) construct an overview of C# projects and help developers get quick understanding of C# code. Critical issues like if/for statements without braces, the validity of C# file and the existence of files are discussed.

2. Introduction

When a function has too many lines of code or becomes too complex, it may be necessary for the developer to pay extra attention and make it simpler. For a large C# project, it's very important to ensure the quality of each function to make the whole project robust and easy to maintain. To this end, Code Analyzer will construct two metrics for each member function: size in lines of code and complexity defined as the number of elements in its scope tree.

Besides, C# does not separate interface from implementation syntactically like C++. The implementation code is forced to be placed together with the class declaration. Therefore, it's not quite easy to get an overview the code of a C# project. In order to help developers to get a quick understanding of the structure of a C# project, the Code Analyzer will find and display all the types, their members, and relationships with other types, given the files of a C# project.

Figure 1 Shows the package overview for Code Analyzer. Each package take some responsibility to separate tasks clearly and make the architecture as simple as possible.

The Executive package takes control of the whole application and interacts with four other packages in order. CMDParser parses the command line arguments and output the processed results. FileMgr take the path and file patterns and output the list of files to be analyzed. Parser will parse the C# file and store the results into repository. Semi takes tokens are combines tokens into semi expressions. Tokenizer takes file reference and output tokens. Repository can store analyzed results for later display.

Display will interact with BasicDisplay or RelationDisplay according to user's requirement to display the results. BasicDisplay displays function sizes and complexities for each function in each file, and a summary for all the files in the specified set. RelationDisplay displays 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. XMLGenerator can output the result in xml format.

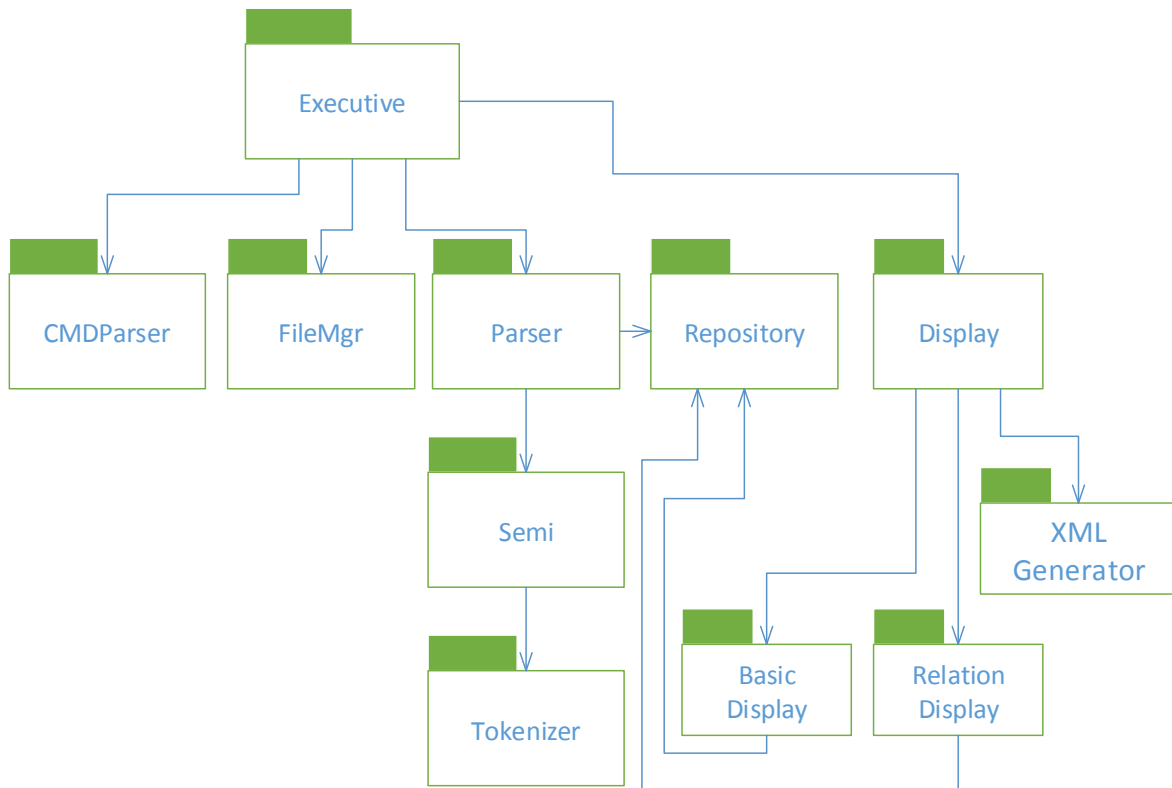


Figure 1 Package Overview for Code Analyzer

3. Uses

Developer's/QA's Use of Checking Function Size and Complexity

- Give the project path, file name patterns for analysis. The subdirectories may need to be traversed.
- Check the function size and complexity, and summary for all specified files on the standard output.
- Check the function size and complexity, and summary for all specified files in the xml file.
- Take the xml file for further analysis.

Developer's Use of Understanding Code

- Give the project path, file name patterns for analysis. The subdirectories may need to be traversed.
- Read all the types, their members, and relationships with other types on the standard output.
- Read the all the types, their members, and relationships with other types in the xml file.
- Take the xml file for further understanding.

4. Application Activities

Parse command line arguments

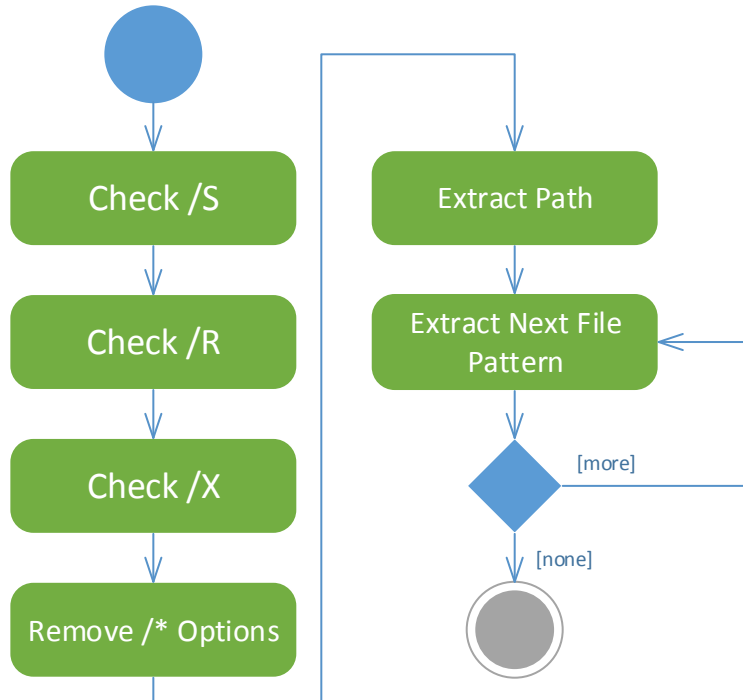


Figure 2 Activity Diagram for CMDParser

CMDParser first check the /* options. After the /* options are removed and the path is extracted, CMDParser will extract the file patterns one by one.

Retrieve input file references

FileMgr takes the root directory, retrieve all file reference matching the file patterns in the root directory and traverse subdirectories if /R option is present.

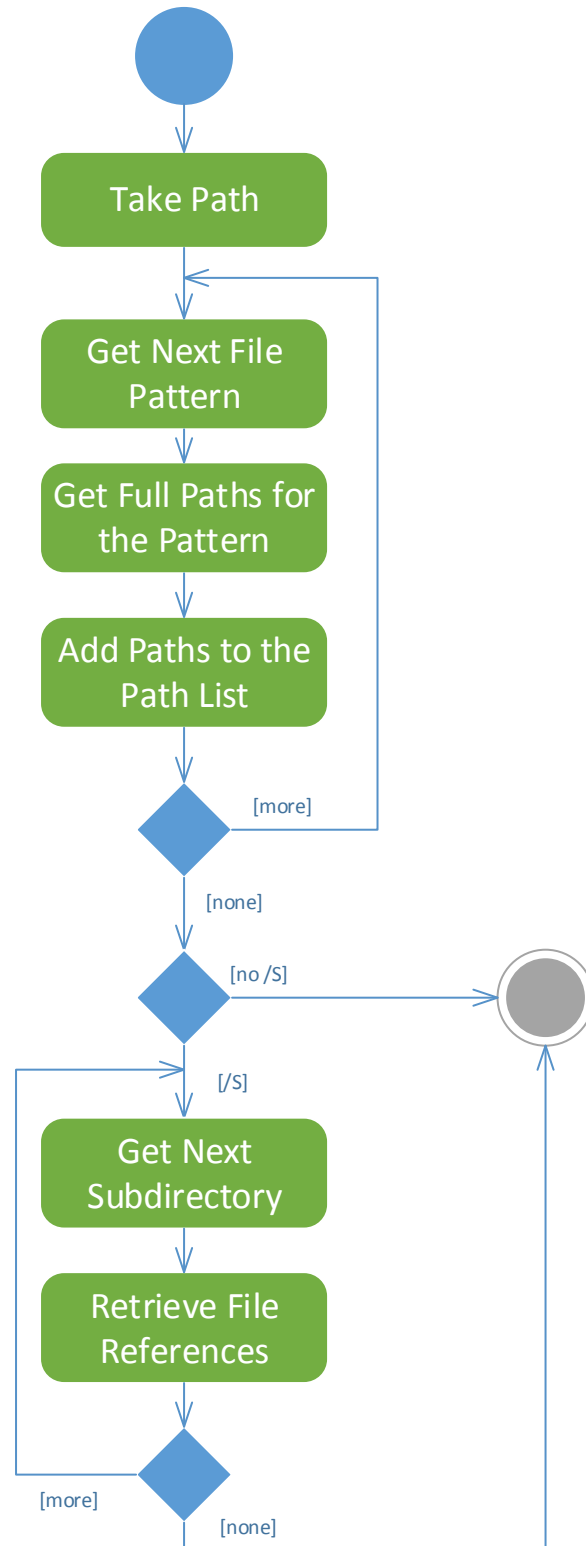


Figure 3 Activity Diagram for FileMgr

Identify defined types

For each file reference, Parser identifies the defined types and save the information into Repository.

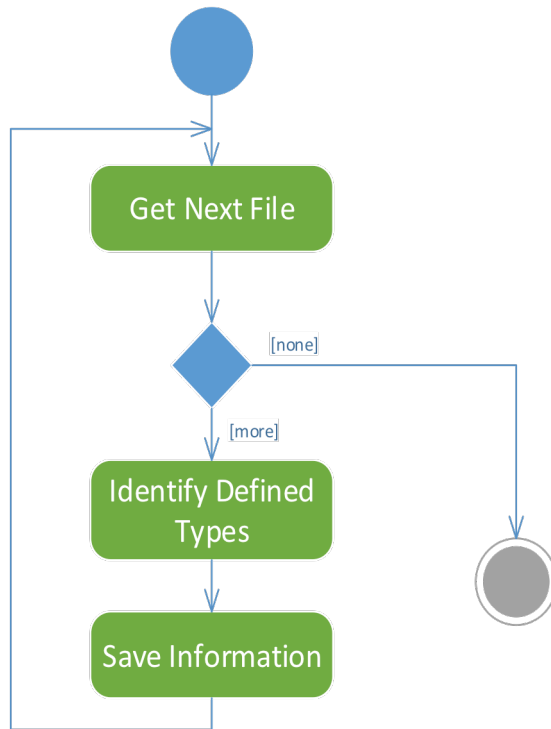


Figure 4 Identify Types

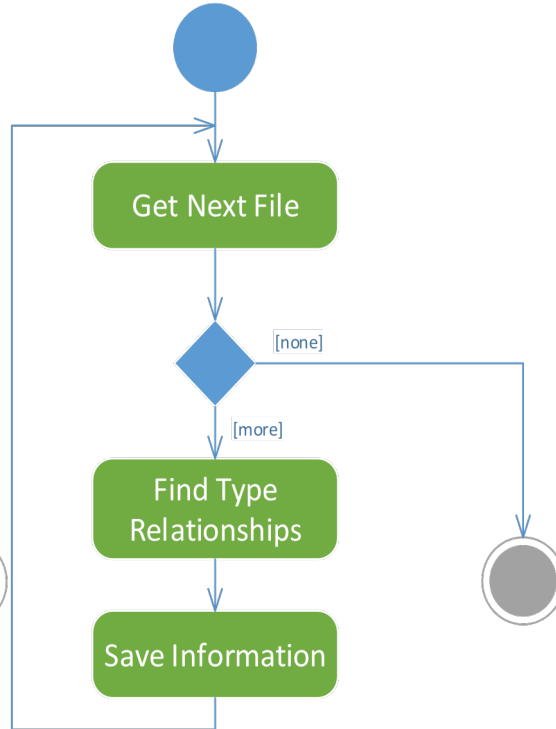


Figure 5 Find Relationships

Find type relationships

For each file reference, Parser finds its type relationships and save the information into Repository.

Display final results

Display selects the right display package to display the final result according to /R option. If /X is present, the xml file will be generated.

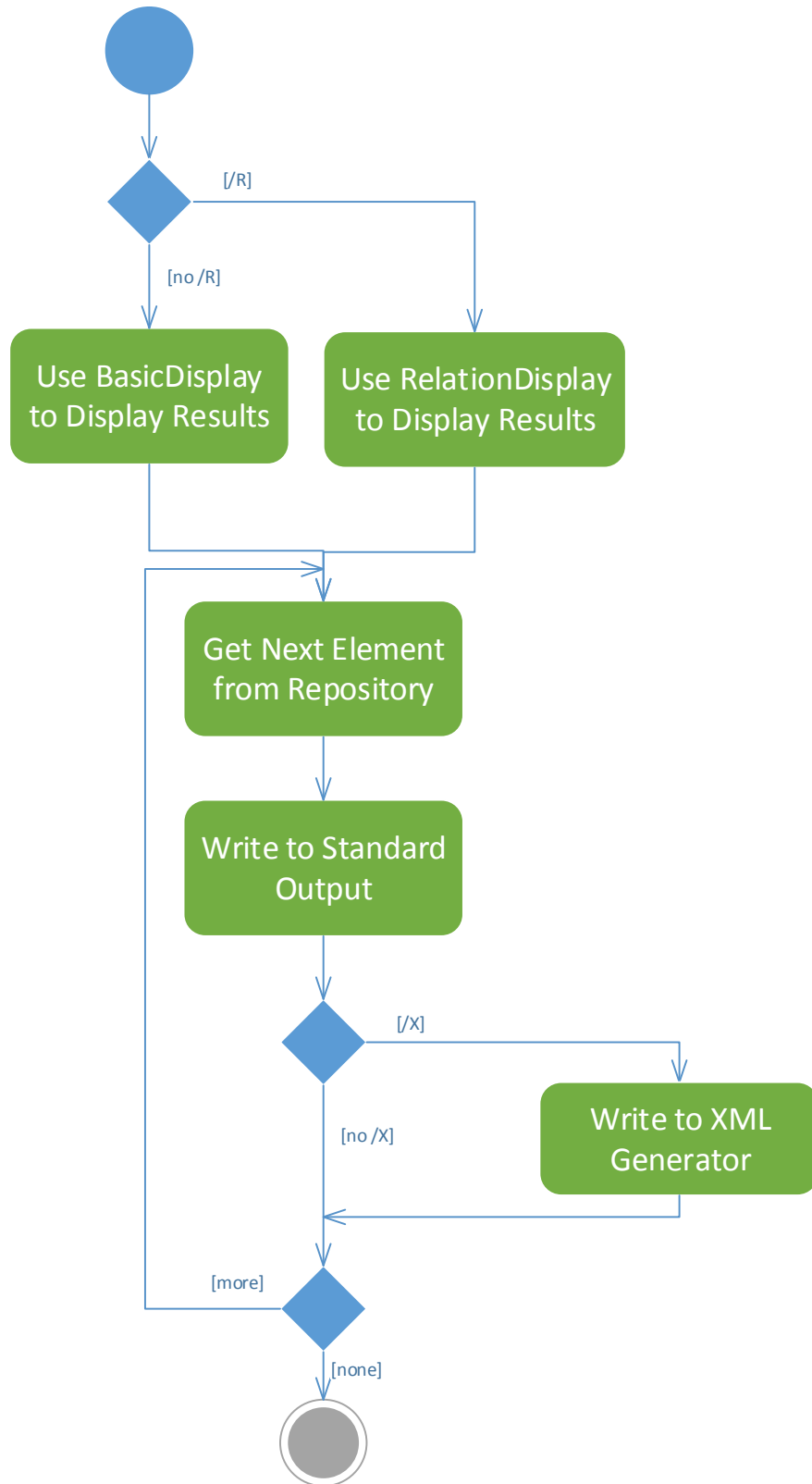


Figure 6 Activity of Display

5. Partitions

Executive

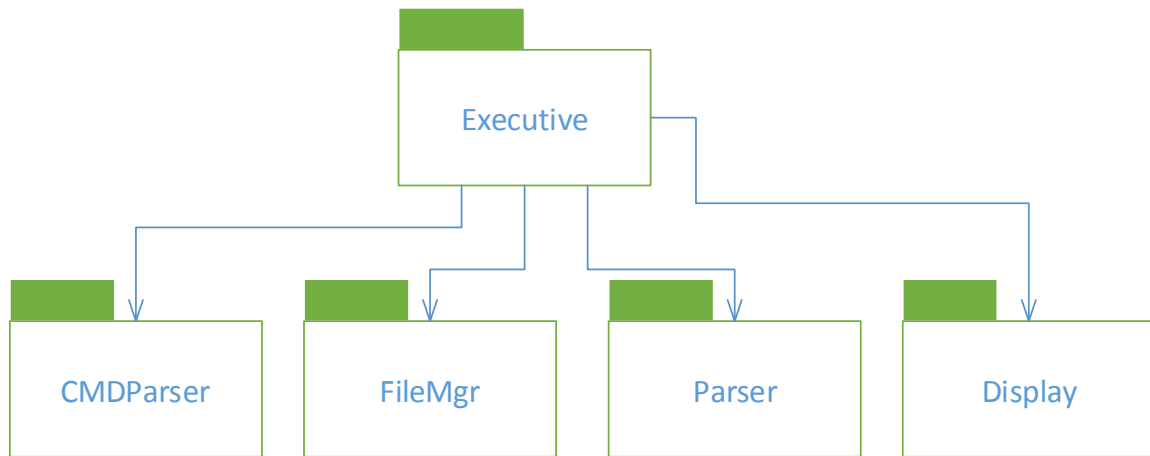


Figure 7 Executive Package

The responsibility of Executive package is to take control of the whole application. Executive package will conduct the command line parsing, file reference retrieving, file parsing and results displaying step by step.

Executive package sends the command line arguments to CMDParser package and receives the /S, /R, /X options, the root directory and the file patterns.

Executive package sends the root directory, the file patterns and the /S option to FileMgr package and receives the file full path list.

Executive package sends the file full path to Parser package for analysis. The analyzed results are stored by Parser package in Repository package so Executive package receive nothing from Parser package.

Executive package sends the /R and /X options to Display packages and receives nothing from it.

CMDParser

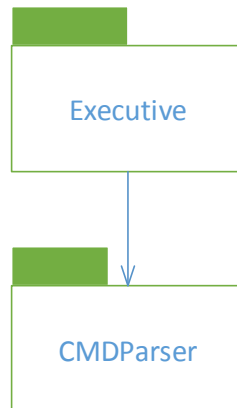


Figure 8 CMDParser Package

The responsibility of CMDParser package is to analyze the command line arguments and turn them into /S, /R and /X options, the root directory and the file patterns list in a more structured way. After the processing of CMDParser package, /S, /R and /X options are turned into boolean variables. Besides, the root directory and the file patterns are also stored in corresponding string variable and string list.

CMDParser package only interacts with Executive package. CMDParser package receives the command line arguments from Executive package and sends back the /S, /R, /X options, the root directory and the file pattern list.

FileMgr

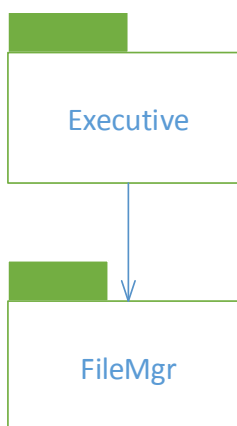


Figure 9 FileMgr Package

The responsibility of FileMgr package is to traverse the root directory, get all the file reference matching the file patterns, extend the file reference into full file paths and store the full paths into a path list. If /S option is present, FileMgr package will also traverse the subdirectories and find all the full file paths in the subdirectories of the root directory. After the processing of FileMgr package, all files needed to be analysed are places as plain full file path in a file path list. The full file path can be directly opened without any further conversion.

FileMgr package only interacts with Executive package. FileMgr package receives the root directory, the file patterns and the /S option from Executive package and sends back the file full path list.

Parser

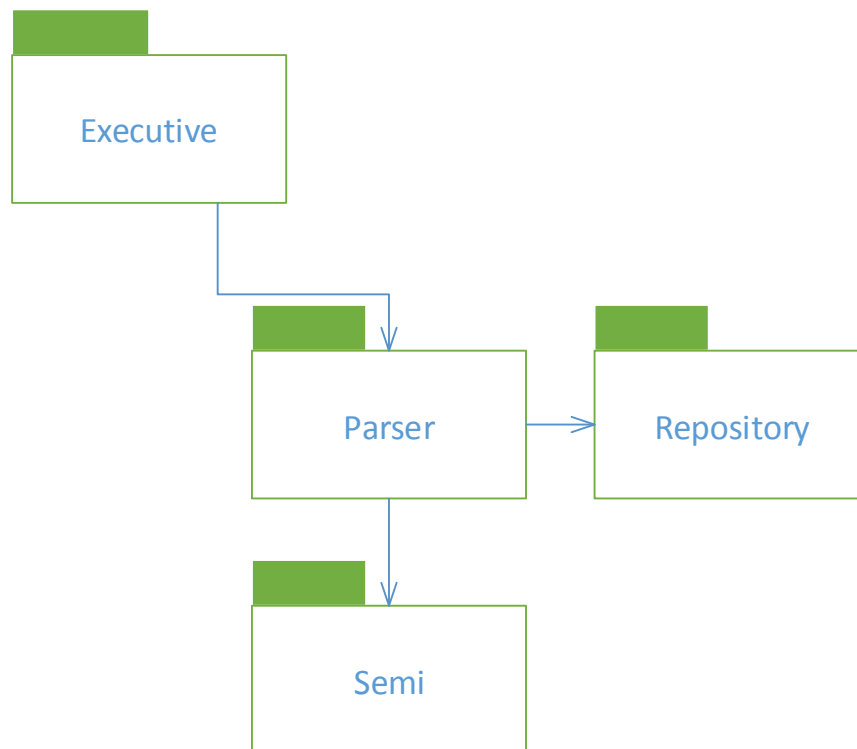


Figure 10 Parser Package

Given a C# file, the responsibility of Parser package is to parse it to get the corresponding information for the two main purpose and store it in the Repository package so that it's easy to display them. To find the functions, which need extra attention, Parser package constructs two metrics for each member function: size in lines of code and complexity defined as the number of elements in its scope

tree. To help developer get a quick understanding of C# code, Parser package finds all the types, their members, and relationships with other types. Parser is the most important and the most sophisticated package of Code Analyzer.

Parser package interacts with three packages, Executive package, Repository package and Semi Package. Parser package receives the full path for a C# code file and sends it to Semi package. Parser package then receives semi expressions from Semi packages and parses them to get the code information. Finally, Parser package sends the information to Repository package for later display.

Semi

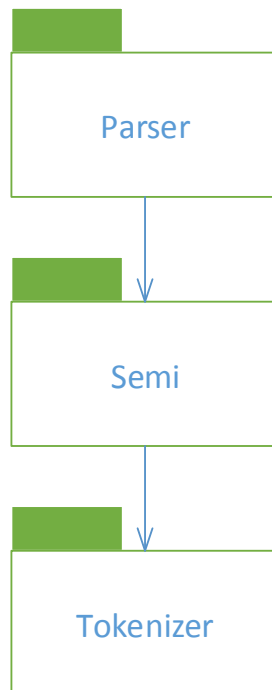


Figure 11 Semi Package

Given a C# file, the responsibility of Semi package is to receive tokens in the file and combine tokens into semi expression. A semi expression is a statement which ends with ';', a statement starts with '{' and ends with '}' or a new line start with '#'. Note that Semi package receives file reference but doesn't deal with file directly. Semi package deals with tokens directly.

Semi package interacts with two packages, Parser package and Tokenizer package. Semi package receives the full path for a C# code file from Parser

package and sends it to Tokenizer package. Semi package then receives tokens from Tokenizer packages and combines them into semi expressions. Finally, Semi package sends back the semi expressions to Parser package for further parse.

Tokenizer

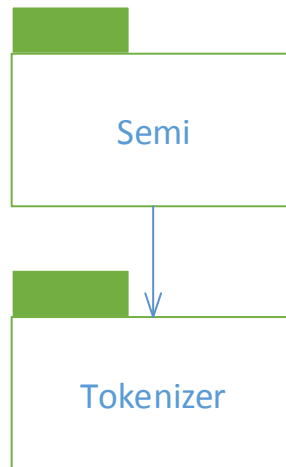


Figure 12 Tokenizer Package

Given a C# file, the responsibility of Tokenizer package is to parse the byte stream and split the characters into tokens. Though both Parser package and Semi package take file path, they don't deal with file directly. However, they take file path for final parsing by Tokenizer package. After the processing of Tokenizer package, the C# code is converted into the more clear unit, token. Since Tokenizer package deals with file directly, it also needs to deal with the issues related with file, such as file not exist and no permission to open.

Tokenizer package interacts with two packages, Parser package and Tokenizer package. Tokenizer package receives the full path for a C# code file from Parser package and sends it to Tokenizer package. Tokenizer package then receives tokens from Tokenizer packages and combines them into semi expressions. Finally, Tokenizer package sends back the semi expressions to Parser package for further parse.

Repository

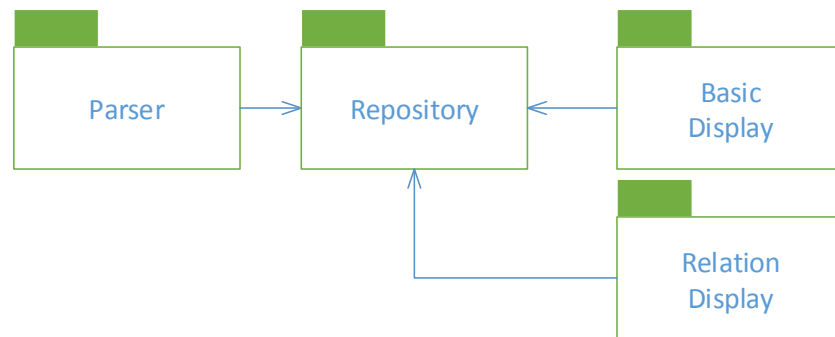


Figure 13 Repository Package

The responsibility of Repository package is to store the extracted code information and to provide the retrieve service when the information is to be displayed.

Repository package is relatively simple compared with other packages. Repository package interacts with three packages, Parser package BasicDisplay package and RelationDisplay package. Repository package receives the code information from Parser package and sends it to BasicDisplay/RelationDisplay package when necessary.

Display

The responsibility of Display package is to take the display command and /R and /X option, set the write the output stream for BasicDisplay and Relation Display packages and use the right package to display the code information. Display package will wrap the XML Generator into the output stream so that BasicDisplay and RelationDisplay packages only takes a uniformed output stream and don't need to worry about the xml file generation.

Display package interacts with four packages, Executive package, BasicDisplay package, RelationDisplay package and XMLGenerator package. Display package receives the display command and /R and /X option from Executive package and sends nothing back. Display package wraps the XML Generator into an output stream and sends the output stream to the BasicDisplay and RelationDisplay packages.

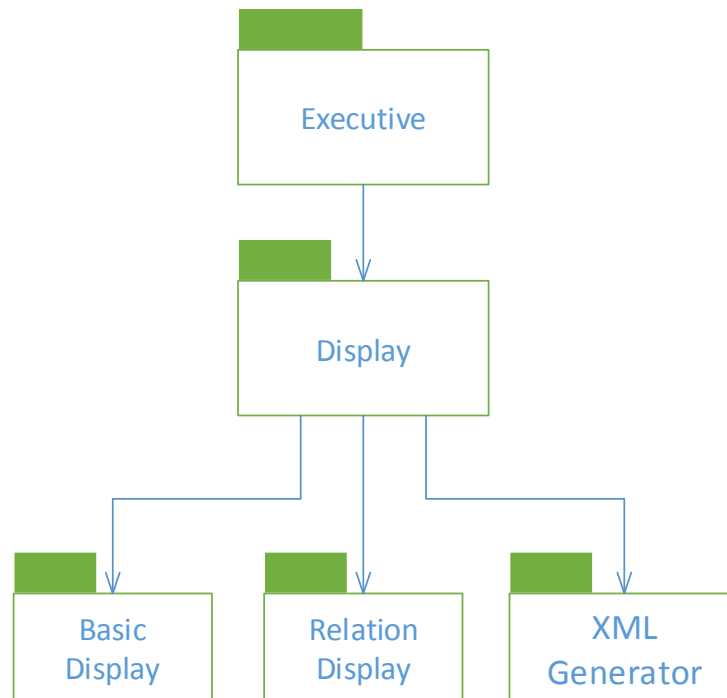


Figure 14 Display Package

BasicDisplay

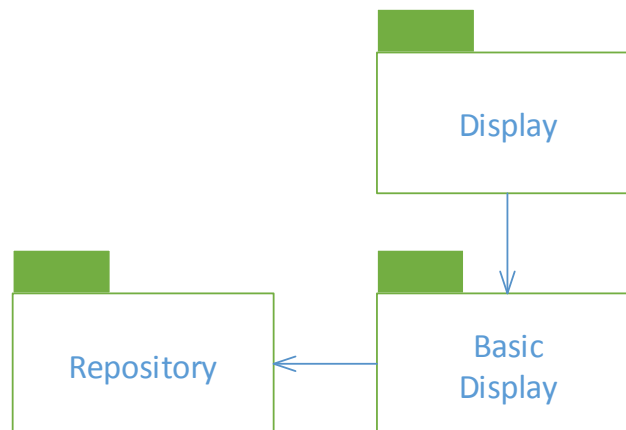


Figure 15 BasicDisplay Package

The responsibility of BasicDisplay package is to display function size in lines of code, and complexity defined as the number of elements in its scope tree to help developers/QA find the functions that need extra attention. Since Display package

wraps XMLGenerator package into a output stream. BasicDisplay package don't need to worry about generating xml or not.

BasicDisplay package interacts with two packages, Display package and Repository package. BasicDisplay package receives the display command and output stream from Display package. BasicDisplay package sends GetElement command to Repository package and receives code element. Finally, BasicDisplay package write the information to the output.

RelationDisplay

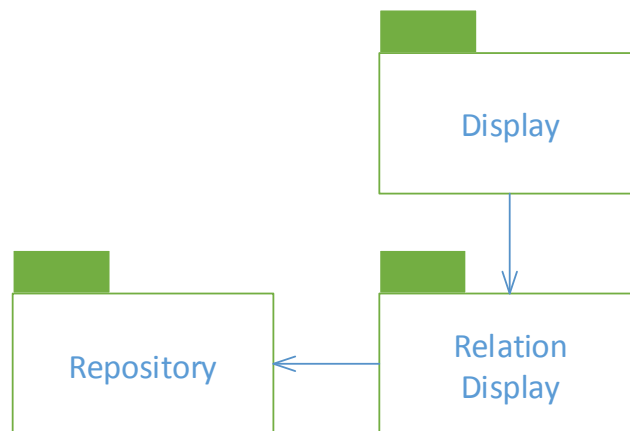


Figure 16 RelationDisplay Package

The responsibility of RelationDisplay package is to display all the types, their members, and relationships with other types to help developers get a quick understanding of C# code. Since Display package wraps XMLGenerator package into a output stream. BasicDisplay package don't need to worry about generating xml or not.

RelationDisplay package interacts with two packages, Display package and Repository package. RelationDisplay package receives the display command and output stream from Display package. RelationDisplay package sends GetElement command to Repository package and receives code element. Finally, RelationDisplay package write the information to the output.

XMLGenerator

The responsibility of XMLGenerator package is to write the information of C# code into xml format. XMLGenerator package will provide the interfaces to be wrapped into a output stream so that Display package can wrap it. Therefore BasicDisplay and RelationDisplay package can take a uniformed output stream.

XMLGenerator package interacts with only one package. XMLGenerator package receives the get instance command and send back its instance to Display package.

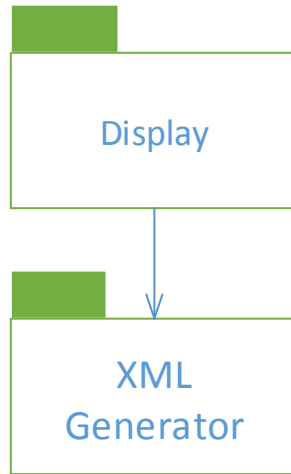


Figure 17 XMLGenerator Package

6. Critical Issues

- (1) Is it enough to simply count the number of braces to compute the complexity of a member function?

No. When if statements and for statements are followed by only one statement, braces can be omitted. Though the number of braces remains the same, the number of scopes has increased. Therefore, the special cases need extra consideration.

There are two possible solutions:

- a) Insert extra braces after if/for statements when the braces are omitted before the parse is conducted. In this way, some preprocessing needs to be done before the parse. The good thing is that we can simply count the number of braces to compute the complexity of a member function without paying attention to special cases. Note that extra braces should be inserted with extra new line character; otherwise the function size in lines of code will be modified.
- b) Consider the special cases of if/for statements without braces during the process of parse. The logic of parse will become a little more complex. However, the good point is that no extra preprocessing is needed. Since we don't insert extra tokens, there is no risk of modifying the function size.

In general, the above two solutions are both feasible and both have pros and cons. Either one can be adopted.

- (2) Is it possible that the file to be parsed is not a valid C# file?

Yes. There are two kinds of invalid C# files.

- a) The file is not a C# file. The user may provide some other files by mistake to Code Analyzer.
- b) The file is a C# file. However, there may be some syntax errors. For example, the braces don't not match.

The above two cases are very normal and reasonable. If we don't deal with them carefully, the Code Analyzer may crash or behave incorrectly. There are three possible solutions:

- a) Don't consider the above two cases and let Code Analyzer continue parsing. As long as we don't make the assumption that the code is correct, Code Analyzer won't crash. The good point of this solution is that Code Analyzer will function well for valid functions inside one file. For example, a developer may be developing one package so some functions are under development and incomplete. The result may be meaningless for incomplete functions but still valuable for complete functions. However, the output for invalid functions or invalid file may not be predictable and may mislead users.
- b) Check the validity of the file before the parse is conducted. If the file is not a valid C# file, Code Analyzer just print a warning and continue parsing next file. The good thing is that we can make the assumption with confidence that there is no special case. The parse logic will be much simpler. We also don't need to worry about cleaning the half-parsed result if the file is not valid.
- c) Don't conduct any validity check. Directly parse the code, if we find something is wrong during the parsing process, clean the half-parsed result, print a warning message and continue parsing next file. The good thing is that no preprocessing is need. However, the parsing logic will become more complex and error-prone.

To sum up, the above three solutions are all feasible. The results of the last two solution make more sense. Furthermore, the logic of b) is more clean and simple. Therefore, b) is the preferred solution.

- (3) Can we ensure that the file given by the user will always be successfully open?

No. When open a C# file for read, there are three possible cases when the file cannot be successfully opened.

- a) The file doesn't exist at all.
- b) The file exists when FileMgr traverse the directory. However, before the Tokenizer opens the file, the file is removed due to some reason.

- c) The file exists and is not removed. However, the file need extra permission to be opened and Code Analyzer doesn't have the permission.

To solve this issue, Tokenizer must check if the file can be successfully opened. What's more, Parser and Semi also need check when try to open the file to get semi expressions or tokens. Note that even if the file is successfully opened, the stream may be broken during the subsequent reading. Therefore, Tokenizer needs to check each time it reads a file.

- (4) Tokenizer can read C# file on the fly and get extra tokens from the file on demand. Tokenizer can also read the whole file into memory at once, convert the file into tokens and store in the memory for subsequent analysis. Is there any problem for this two ways? Which way is better?

The first way needs little memory space because it only store one token at one moment, but this will not be a problem for a small file. In most cases, a C# file should not be too big. Besides, it takes more time to read in all the tokens. Therefore, more unexpected problems may occur during the process (e.g. the file is removed). However, as long as Tokenizer checks the status each time it reads, it will not be a problem.

The second way may consume more memory space but this can be omitted for normal C# file. The good thing is that after reading in all the tokens and storing them in a list. Tokenizer just needs to pop one from the list when requested. Therefore, the logic is simpler and clearer. Besides, it takes less time to read in the whole file so there is less possibility of unexpected problems during the reading process.

In short, there is no problem for this two ways. And the second way is better.

7. APPENDIX

Prototype for Executive package

```
////////////////////////////////////
// Executive.cs - Executive manages the control flow of Code Analyzer//
// ver 1.0                                                    //
// Language:    C#, 2013, .Net Framework 4.5                //
// Platform:    Mac Air 2013, Win8.1                        //
// Application:  Project #1 for CSE681, Fall 2014            //
// Author:      Rundong Zhou, 234944570, Syracuse University //
//              (315) 696-1898, rzhou02@syr.edu              //
////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeAnalyzer
{
    class Argument
    {
        public bool bS, bR, bX;
        public string rootdir;
        public string[] filePatterns;
    }

    class Executive
    {
        static void Main(string[] args)
        {
            // Process comand line
            Argument argu = CMDParser.ParseCommand(args);

            // get all file reference
            List<string> files = FileMgr.GetFiles(argu);

            // Parse the file one by one
            foreach(string file in files)
            {
```

```

        Parser parser = new Parser(file);
        if (!parser.Parse())
        {
            Console.WriteLine("The file is invalid. {0}", file);
        }
        else
        {
            // display the final results
            Display.Write(argu.bX, argu.bR);
        }
    }
}
}
}
}

```

Prototype for CMDParser package

```

////////////////////////////////////////////////////////////////
// CMDParser.cs - CMDParser parses command line          //
// ver 1.0                                                //
// Language:      C#, 2013, .Net Framework 4.5          //
// Platform:      Mac Air 2013, Win8.1                  //
// Application:    Project #1 for CSE681, Fall 2014      //
// Author:         Rundong Zhou, 234944570, Syracuse University //
//                (315) 696-1898, rzhou02@syr.edu        //
////////////////////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeAnalyzer
{
    class CMDParser
    {
        public static Argument ParseCommand(string[] args)
        {
            Argument ret = new Argument();
            // TODO: parse the command line and fill ret
            return ret;
        }
    }
}

```

```

    }
}
}

```

Prototype for FileMgr package

```

////////////////////////////////////
// FileMgr.cs - FileMgr convert path and patterns to path list      //
// ver 1.0                                                            //
// Language:    C#, 2013, .Net Framework 4.5                        //
// Platform:    Mac Air 2013, Win8.1                                //
// Application: Project #1 for CSE681, Fall 2014                     //
// Author:      Rundong Zhou, 234944570, Syracuse University        //
//              (315) 696-1898, rzhou02@syr.edu                      //
////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeAnalyzer
{
    class FileMgr
    {
        public static List<string> GetFiles(Argument argu)
        {
            List<string> ret = new List<string>();
            // TODO: get all files
            return ret;
        }
    }
}

```

Prototype for Parser package

```

////////////////////////////////////
// Parser.cs - Parser detects code constructs                        //
// ver 1.0                                                            //
// Language:    C#, 2013, .Net Framework 4.5                        //
// Platform:    Mac Air 2013, Win8.1                                //
// Application: Project #1 for CSE681, Fall 2014                     //
// Author:      Rundong Zhou, 234944570, Syracuse University        //

```

```

//          (315) 696-1898, rzhou02@syr.edu          //
////////////////////////////////////////////////////////////////
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace CodeAnalyzer
{
    class Parser
    {
        public Parser(string file)
        {

        }

        public bool Parse()
        {
            // TODO: parse the file
            return false;
        }
    }
}

```

Prototype for Display package

```

////////////////////////////////////////////////////////////////
// Display.cs - Display takes display command          //
// ver 1.0                                              //
// Language:    C#, 2013, .Net Framework 4.5          //
// Platform:    Mac Air 2013, Win8.1                  //
// Application: Project #1 for CSE681, Fall 2014       //
// Author:      Rundong Zhou, 234944570, Syracuse University //
//          (315) 696-1898, rzhou02@syr.edu          //
////////////////////////////////////////////////////////////////

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

```

```
namespace CodeAnalyzer
{
    class Display
    {
        public static void Write(bool bX, bool bR)
        {
            // TODO: Write the final results to standard output
            // and XML file
        }
    }
}
```