

**CSE 681-Software Modeling and Analysis
Project #1**

**CODE ANALYZER
OPERATIONAL CONCEPT DOCUMENT (OCD)
VERSION: 1.0**

**Neethu Haneesha Bingi
SUID: 525763056
Date: 09-15-2014**

Instructor-Dr. Jim Fawcett

Table of Contents

1	Executive Summary.....	4
2	Introduction	5
2.1	Obligations	5
2.2	Organizing Principles.....	5
2.3	Key architectural designs	5
3	Use Cases	7
3.1	Software Developer	7
3.2	Quality Assurance/ Testing team.....	8
3.3	Software Architect	9
3.4	Customer.....	9
4	Application Activities	10
4.1	Receive the input from the user and parse	10
4.2	Retrieve files	10
4.3	Analyze each file for types	10
4.4	Analyze each file again for type relationships	10
4.5	Display Results	11
4.6	Activity Diagram.....	11
5	Packages or Modules	13
5.1	Executive	13
5.2	Command Line Processing	13
5.3	File Manager	13
5.4	Code Analyzer	13
5.5	Parser	13
5.6	Semi-Expression	14
5.7	Tokenizer.....	14
5.8	Actions and Rules.....	14
5.9	Repository	14
5.10	Display.....	15
5.11	XMLWriter.....	15
5.12	PACKAGE DIAGRAM:	15
6	Critical issues.....	16

6.1	Performance for huge set of files	16
6.2	Parser not implementing discard comments feature	16
6.3	Unfeasible Code Analysis for distributed files	16
6.4	Code Analyzer program output	16
6.5	Dependency on Compilation status.....	17
6.6	Representation of results.....	17
7	Extended Applications	18
8	References	19
9	Appendix	20
9.1	Parser	20
9.2	Type Relationships	21
9.3	Scope related information	22

Table of Figures

Figure 1: Code Analyzer - Activity Diagram	12
Figure 2: Package Diagram	15
Figure 3: Parser Class Diagram.....	20
Figure 4: Sample output of Parser Code.....	21

1 Executive Summary

In a large project, code understanding or analyzing can become extremely difficult to manage and takes huge amount of time when performed manually. A developer tries to understand the code to make further changes; a software architect tries to get an overview on the code to make technical decisions for further enhancements; a software tester will be interested to know the code regions which are error prone. The Code Analyzer to help the users in such situations by providing them with its code summarization/analysis.

The Code Analyzer tool analyzes the code token by token using Parser project and focuses on answering questions about the code. It depicts a high level picture of the entities and its relationships. The project consists of following major tasks.

- Parse code files
- Analyze parse results to identify functions sizes and complexities
- Analyze parse results to identify types and their relationships
- Display Code Analysis results.

The primary users of the tool will be software developers, software testers and software architects. Developers can use this tool to understand the code structure for working on code changes/improvements and to estimate the impact of code changes. Quality assurance team can identify the tightly-coupled coding areas or the impacted areas of a code change that might be error prone. The team can utilize this information to ensure quality and stability of the system, by planning test cases appropriately. Similarly, software architect can take better planning decisions using the tool instead of going through the entire code. If the results are taken into consideration, system can have increased stability and maintainability.

Some critical issues listed below associated with this entire system that need to be addressed are identified and solutions are discussed in detail in further sections.

- Performance for huge set of files
- Parser not implementing discard comments feature
- Unfeasible Code Analysis for distributed files
- Code Analyzer program output
- Representation of results

This document further shows the interaction of modules with each other in the tool; the flow of tasks/activities to achieve the results with supporting diagrams. For example, how parser can be used without making changes; to whom the tool results will be more useful; how the results make sense to them. It discusses about the critical issues that can arise and the ways to solve them, if possible.

Code Analyzer is an important tool for understanding the structure of a given set of source code files. Code Analyzer facilitates to find the types that help to identify the entities and their relationships with each other in a set of packages. It also helps to understand the modularity characteristics of a package by analyzing each function's size (number of lines) and its scope.

2 Introduction

Software projects are often too large for any one person to understand completely. Academic projects range in size from a few hundred lines of code to perhaps 50,000 lines for group projects. Professional projects, entailing perhaps millions of source code lines in thousands of packages, are much larger than academic projects. Many commercial and industrial software systems are much larger than this. No one can understand the system structure that consists of millions of source code lines by looking at its source code alone. So Abstraction is essential as understanding of code structure is extremely difficult without any tool. Code Analyzer plays an effective role in helping the users by providing them with abstraction/summarization of information obtained from the analysis of a set of code files.

2.1 Obligations

The main intention of Code Analyzer tool is to get useful information from a set of code files to the user. The obligations of the tool is

- To analyze the files for identification of types and their relationships with each other.
- To analyze the files for finding the functions present in a type, its size, its complexity and its scope structure. To provide an overview on modularity characteristics of each package.

Types are the constructs that represent the basic entities defined in a system like class, interfaces, structs and enums. Type relationships include 'inheritance', 'composition', 'aggregation' and 'using'.

2.2 Organizing Principles

- The parser package provides support for substitutability of its rules and actions without making changes to any of the parser architecture/code.
- The parser design structure follows the dependency inversion principle. Parser package uses interfaces like `ITokenCollection`, `IRule`, `IAction`, and `IBuilder` to decouple all the important parts so that the design is very robust under change. Thus, it increases the chances of reusability of Parser package.
- The parser project has been used successfully in several research activities and many classroom projects. So it has been thoroughly tested and does a fairly complex job with simple parts.

2.3 Key architectural designs

Code Analyzer tool uses the Parser project developed by Dr. Fawcett. The Parser package collects token sequences for analysis. It is essentially a container of rule detectors that implement an `IRule` interface. 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. For each parsing application, the only things that need to change are the rules and actions, the `ConfigureParser` builder that assembles the analyzer's parts, the `Repository` and the `Display`. All the complex parts - tokenizer, Semi-Expression, and Parser don't need to change at all. The rules and actions

will probably be modeled after already existing rules and actions, so reuse in our application is fairly straight forward.

The Code Analyzer tool will parse the code for two times. First time, to identify the types and its members defined. Second time, to identify the relationships between identified types. It uses repository to share resources across all modules that is to store and retrieve information.

3 Use Cases

A variety of users interact with the Code Analyzer system. The following are the typical users of the Code Analyzer system:

- Software Developer
- Quality Assurance/ Testing team
- Software Architect
- Customer

The expectations from the system and the features that are required to satisfy each of the user needs are described below.

3.1 Software Developer

Scope Analysis:

Scope defines the lifetime of a variable or an object. It can be defined through different blocks of a language using statements such as function, if, for, open-curly braces etc. The Code Analyzer system displays the scope structure and size of all the functions in a given set of code. This summary helps the developer to quickly understand the gist of the project code's scope structure.

Maintainability through Modular Programming:

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each module contains all the code necessary to execute only one aspect of the desired functionality. Code Analyzer system displays the scope structure and size of all the functions in a set of code. By observing the size and scope structure of the functions, the developer can understand whether the code is deviating from modular programming. For example, if the function size is more than 100 lines, there is a possibility of more than one functionality code in the same function. In such a case, the developer can have a look at the functions with an abnormal size or scope structure. If it is necessary, he can take a decision to modularize it further. As modules represent a separation of concerns and improve maintainability by enforcing logical boundaries between components, Developer will have a chance to ensure maintainability through the Code Analyzer tool.

Complexity Analysis:

This tool helps the developer to get the picture of each function's complexity in the project. The developer can have a look at the scope structure output of Code Analyzer tool. Based on the scope structure, the developer can get a sense on how complex each function is. If the developer feels that a function might need some refactoring, he can get down to the function and analyze it. Based on further analysis, he can decide whether to refactor it or not. In complex projects, such as scientific projects, the tool helps the developer to identify the places that might need attention.

Code understanding through types and relationships:

Code Analyzer tool displays all the types (class, struct, enum, interface etc.) defined in the input set of files and also displays the relationships between these types. When a new developer joins the team, he/she may be entitled to work on a project and assigned to do some code changes or improvements. The tool facilitates in overall high level understanding of the project by listing out the types and their relationships. The developer will know which code areas are more dependent or impacted. This helps in further understanding of the project.

Risk and Impact analysis for doing code changes/improvements:

When there are code changes to be made in a package file, it is very essential to analyze its impact on other code files. The types and their relationships help a developer to easily identify the impacted files for a particular change. This helps in determining files where code changes can be made without much impact on other files. Understanding the impact of code changes is very essential for the system to remain stable. Based on the volume of changes needed, the decision can be made on whether to go ahead with the change or not. If the change has a high impact, it could be reported to the concerned manager and then to the customer. Finally a suitable action can be taken. Depending on the impact associated with the change, the developer can carefully plan, schedule and provide a proper estimate to the manager.

3.2 Quality Assurance/ Testing team

Test process:

The Code Analyzer tool results can be used during test planning phase to determine priorities for software under test. The Code Analyzer helps the tester to identify high risk coding areas by providing them with the type relationship results. Using those results, the tester can identify specific components within an application that need to be focused and tested comprehensively to ensure stability to the project. The test planning may revise some of the test cases or add more test cases based on the results of Code Analyzer tool. They can estimate the depth of testing needed for a particular file based on its dependency information and make sure that it is achieved by suitable and sufficient test cases.

Estimate quality:

The Code Analyzer tool will give some of the inputs needed for the quality assurance team to estimate the quality of the project. The function sizes of the project helps the team to determine whether each function is modularized appropriately. The scope structure of the code file helps the team to get an estimate of the performance in real-time scenarios. The types and the relationships determine how much each file is tightly coupled with other file. This information helps in finding the components which are more error prone by instabilizing code.

3.3 Software Architect

Technical design decisions:

A software architect is a person who makes high-level technical design decisions before the project kick-off. If any code enhancements or modifications are required to be performed on an existing system, the software architect checks the feasibility and provides a technical design document. The Code Analyzer tool provides the code structure which helps the architect to understand the code better. It also provides him the information about the tightly coupled code areas by which he can analyze the impacts of proposed changes to the system and he can check for feasibility. A proper decision at this level will help in implementing flaw-less extensions to the system.

3.4 Customer

Customizations impact:

Once the project is delivered to the customer, he may want to customize a bit to his needs. The Code Analyzer tool will help the customer to understand the code structure. It also helps him to determine the impacts of customization changes.

4 Application Activities

Code Analyzer deals with identifying the important parts of the code files that help the user to get some summary information about the code. For this process, the code needs to be parsed line by line and needs to be analyzed. Summarized information after analysis is displayed to the user appropriately. The output consist information about types, its members and their relationships. The relationships between types include inheritance, composition, aggregation and usage. The major tasks involved in the application are described below in a sequence.

4.1 Receive the input from the user and parse

The user inputs a path, which refers to a set of files that needs to be analyzed, optional file patterns and optional switches in the command line. The tool parses the command line and identifies the input parameters passed. This task helps to set the context of the application in the executive package.

4.2 Retrieve files

The files that need to be analyzed are retrieved from the given path and from the given file patterns. The files are retrieved recursively from all the sub directories if required with a specific switch in the command line. The retrieved file names are stored in a list.

4.3 Analyze each file for types

Parse each input file in the list line by line. Identify all the types defined in a file and store them in a list. Apart from finding type's information, parsing will also find its member's details such as its functions, their scope structure, line numbers etc. The information required to understand the structure of the code such as function names, its size, complexities and scope structure are also stored.

Types and its members in a file are identified by searching for a text pattern with the help of some predefined rules.

4.4 Analyze each file again for type relationships

The lists of input files are parsed again to identify the type usages. These are classified as type relationships only if the type used is defined in the current input files. The type information stored in the first parse is used here to know the types which are defined in the current input code files. So the type information stored will help to differentiate types defined in the given input files from the standard library types such as strings or any other types which does not exist in the given input files.

Types and type relationships cannot be identified in a single parse of a file, because of the file parse ordering. If a file which contains a type usage is parsed before the file which contains type definition, it is difficult to relate them. If a type relationship is encountered whose definitions information is not yet identified, it's impossible to store/identify them. So, second parse is mandatory for identifying type relationships.

Similar to the above parse, type relationships in a file are identified by searching for a text pattern with the help of some predefined rules. It also stores the type relationships information found.

4.5 Display Results

The final step after doing code analysis is to display the results. The results contain types, its members and their relationships information. The output can be displayed in two ways. The output can be either printed on the standard output terminal or written into a file.

The output is printed on standard output terminal in a tabular format. The results are written into a file in XML format.

4.6 Activity Diagram

The activity diagram below represents the sequential order of activities performed as part of the execution steps of this tool. The diagram also depicts the overall flow of control. The activity diagram gives reader an abstract idea of what activities the tool will perform for analyzing the code.

Code Analyzer - Activity Diagram

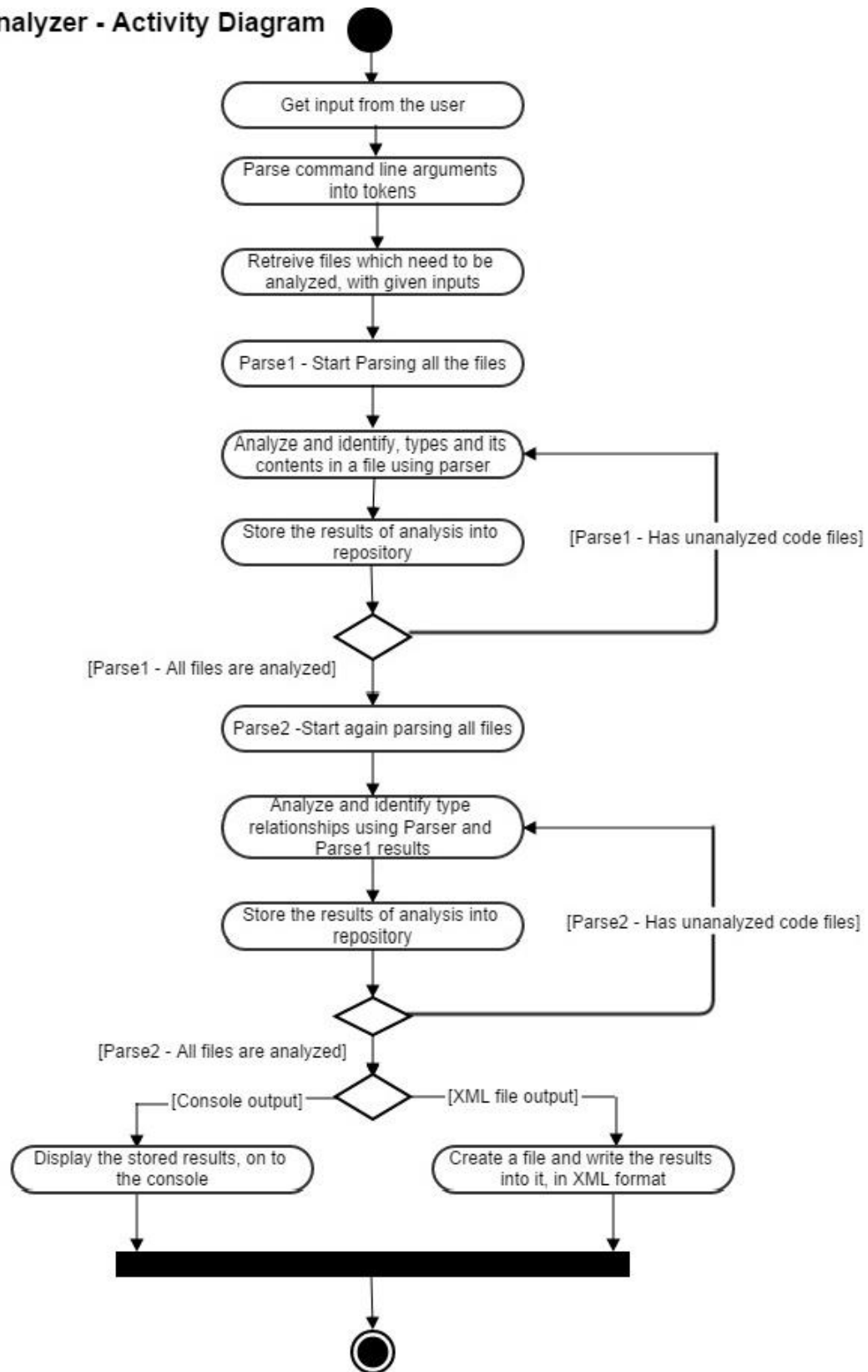


Figure 1: Code Analyzer - Activity Diagram

5 Packages or Modules

Based on the tasks or activities designed for the Code Analyzing process, the following packages or modules can be identified in the application.

5.1 Executive

This module acts as a global controller, which simply directs the control to various modules based on the actions to be performed. It does not look into the details of the implementation or the business logic of the tool. This package is the entry point of the application. All the calls to subsequent modules will be routed from this package. It takes the input from the command line, calls subsequent business logic modules and then finally calls display module for displaying results. It performs the executive functions alone.

5.2 Command Line Processing

This package provides support for parsing command line arguments. The arguments given in the command prompt are given as input to this package. It parses the arguments and classifies them as file path, file patterns and switches.

5.3 File Manager

File Manager Module is responsible for retrieving file names from the given path that needs to be analyzed. It provides support for retrieving files with a specific file name pattern in a directory and its sub directories, if required recursively. After retrieving, it stores them in a list data structure.

5.4 Code Analyzer

Code Analyzer module handles all the analysis or processing which needs to be done as part of Code Analyzer tool. It acts as a controller for the code analysis logic. This module is responsible for coordinating all the processing required.

It takes the list of file names as input. Processes each file by parsing them with the help of Parser and its dependent modules. As part of processing, all the files are parsed first time to identify types, its members, its member's sizes and complexities. The results are stored for future. After that, all the files are parsed for the second time to identify the type relationships, with the help of first parse results. The analysis results are stored for displaying the output.

This Code Analyzer uses the services of Parser, Semi Expression, Tokenizer, ActionsAndRules and Repository modules for its operation.

5.5 Parser

The Code Analyzer tool uses the Parser module which is developed by Dr. Fawcett. It is a container of rules each of which is a detector for some grammatical construction like class, struct, interface, enum, functions etc. It collects semi-expressions (token sequences) of a code file using Semi-Expression module for analysis. It feeds each semi-expression to all the registered rules. It repeats this process until all the semi-expressions are analyzed in a file. Each rule has a set of actions which are invoked when a particular rule satisfies. For example when a class or interface definition rule is found out, appropriate action is taken. Each action is given a reference to a data repository to store and retrieve information to

carry out its task. The parser uses the services of Tokenizer module, Semi-Expression module and Repository module for its operation.

5.6 Semi-Expression

The Semi-Expression module written by Dr. Fawcett will be used in our Code Analyzer tool. It builds a sequence of tokens obtained from the Tokenizer module for analysis of code which are called as Semi-Expressions. Semi-Expressions are lists of tokens that terminate on one of the tokens: semicolons, curly braces, or newlines if the line starts with "#". This structure is designed to collect just the right amount of information from a code file for analysis. That is, it collects the smallest set of tokens that lead to detection of grammatical constructs, without having to save part of the sequence for later use or get more to complete the detection process. This simplifies the design of code analysis tools.

5.7 Tokenizer

The Tokenizer module written by Dr. Fawcett will be used in our Code Analyzer tool. It reads words from the given input file stream or a string and returns a token. The tokenizer promises to return all the input stream's characters, in the order provided by the source. It segregates them into words called tokens. Each token is a text word or group of punctuators. The transition between characters and white space are taken as token boundaries. You can ask the tokenizer to return or to throw away comment tokens. Thus it removes all textual structure of the source while preserving its compilable information.

5.8 Actions and Rules

The Actions and Rules module written by Dr. Fawcett will be used in our Code Analyzer tool. Rules can be defined as a detector for some grammatical construction such as class, struct, enum, functions etc. Actions are nothing but a code sequence which gets executed when a particular rule satisfies. Actions are configured to Rules and Rules are configured to Parser. IAction and IRule are two interfaces which are designed to decouple all the important parts of the parser, so that the design is very robust under change. IRule is an interface which is a contract for all the rules defined. IAction is an interface which is a contract for all the actions defined. Both of the interfaces facilitate in making the Parser project reusable, by providing a facility to configure a different set of rules and actions for different applications. ConfigParser is a class which builds parser and configures it with application specific rules and actions.

5.9 Repository

The Repository module written by Dr. Fawcett will be used in our Code Analyzer tool. It is a package which is used to share resources across other modules like actions in the process of code analysis. The intermediate results which are used for analyzing will also be stored as part of repository. For example, to calculate the number of lines in a function, when an open curly brace of a function is encountered its line number is stored. Once the closing curly brace is encountered, it retrieves the starting line number stored and calculates the size of a function. This is also the place where the final results are stored. Later this is used by the display package.

5.10 Display

The Display module provides support for displaying the Code Analyzer results onto the standard output. The results are taken from the shared Repository package.

5.11 XMLWriter

The XMLWriter module provides support to build XML. The Code Analyzer results can be modeled into a valid XML using this module. Apart from creating an XML, it also writes the xml into the output file, if required.

5.12 PACKAGE DIAGRAM:

The following package diagram depicts a high level view of the Code Analyzer modules and their interactions. From the package diagram it is observed the project control is handled by the Executive package. It redirects the functionalities to different modules.

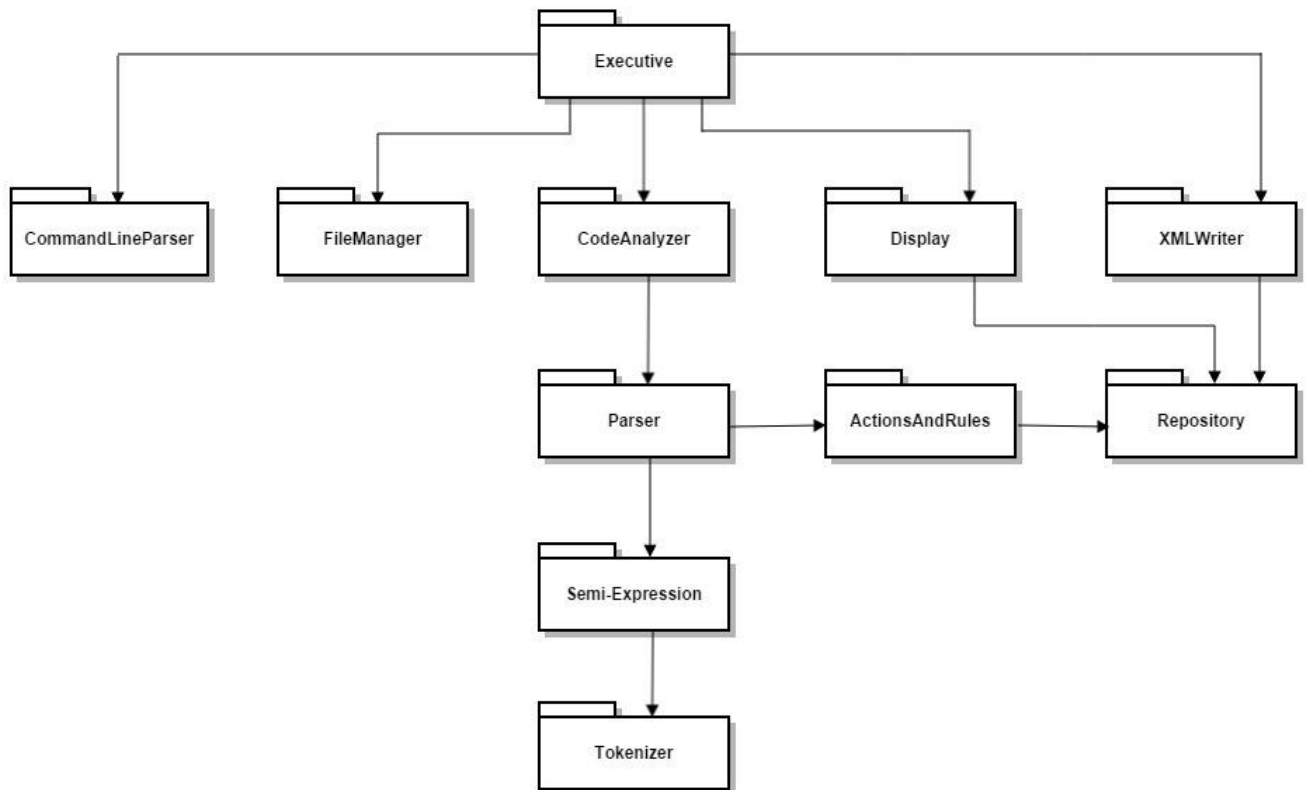


Figure 2: Package Diagram

6 Critical issues

6.1 Performance for huge set of files

If the input to the code analyzer tool is a huge set of files, the time taken by this tool to do code analysis increases. How the performance of the system is ensured with reasonable waiting times while accepting huge set of inputs files.

Solution: To handle the scenario of taking huge set of files as input, with unreasonable waiting times for processing, the scheme of parallel execution of dependency and type relationship analysis need to be followed. The tool will be changed to do parallel processing. As the parsing of one file is independent of parsing on other file, parallel processing of files processing would be the best way to increase performance. Multiple parallel threads will be launched where each thread will parse files one by one from the input list. But this would come at an extra cost of incorporating thread safe data structures. This will avoid the concurrent access of the data structures.

6.2 Parser not implementing discard comments feature

The stand-alone Code Analyzer displays functions defined in a package, its size and complexity. The parser helps in identifying size of the function by retrieving line numbers from the source file. The parser has a configuration property that decides whether to return new lines or not. Similarly, it does have a configuration property for discarding comments in the source file, but it is not yet implemented that is inactive. So the estimates of function sizes and its complexities may not be accurate in some cases. How can the user get the proper estimates of function sizes?

Solution: The Parser project should be enhanced to incorporate the discard comments property. Incorporating discard comments property helps in getting accurate estimations about the function sizes. Till the implementation of this property in Parser project, the user can have in mind that the results will also include comments and user can use that information further.

6.3 Unfeasible Code Analysis for distributed files

When the source code files are distributed across client network, it is not possible to analyze dependency and type relationship between them. The Code Analyzer takes input file set from a single path. Then, how can such distributed projects are analyzed with this tool?

Solution: The files which are distributed have to be gathered at a single location and given as input to the Code Analyzer tool. This might not be a feasible solution in scenarios when the files size is huge. As this introduces additional burden for the users, another possible solution is to take multiple file paths as inputs to Code Analyzer tool. Files need to be retrieved from this multiple input file paths and given to Parser.

6.4 Code Analyzer program output

The computer languages evolve frequently. The languages kept adding new features, new libraries, and new ways of achieving results etc. to improve efficiency and stability of the package. How will the Code Analyzer tool be able to incorporate the new features into its results?

Solution: The only thing that is constant is change. So as the language keeps changing the Code Analyzer needs to be maintained to mimic those changes and to produce accurate results. To incorporate new language features, rules and actions defined in the parses project need to be modified. This is a very safe and stable operation, as the rules and actions are substitutable and also highly independent on other packages.

6.5 Dependency on Compilation status

The Code Analyzer tool works as expected only on a set of input files which can be compiled. If the files are syntactically not compiled, it will not find the errors and also gives unexpected results.

Solution: The Code Analyzer tool assumes that all the input files are syntactically correct. This is necessary because the code is analyzed based on the syntax constructs. For example, a semi expression is defined as that it ends with semicolon. Code Analysis makes sense when the code is in use. If by any chance the code is not syntactically correct, there is even a chance of giving improper results to the user.

6.6 Representation of results

The Code Analyzer results are either in text or xml format. They both seem to be inefficient way of representing the code summarizations to the user.

Solution: Even though there are better ways of displaying the output to the user, currently the code analyzer targets to display results in only text and xml format. Also a final summary of results in Pie Charts or Bar charts gives quick overview for high level management. Representing results in a better visual format can be planned as a future enhancement for this tool.

7 Extended Applications

The Code Analyzer tool can be extended or customized to serve various other purposes. The following are some of the examples.

- Professors can use this as plagiarism detection software that categorically searches each line of code and displays the code duplication across student assignments.
- During the large development projects that involve many developers, it is common that code duplication of a specific fragment or large chunk, and incorrect copy of code in to other source code files can happen. The code analyzer tool can be customized to all such instances.
- As the tool analyzes the code line by line, it can be extended to detect security vulnerabilities and provide advice on how to remediate those problems.

8 References

- [1] Code Analyzer Requirements document, provided by Dr. Fawcett
<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/Projects/Pr2F14.pdf>
- [2] Parser code, provided by Dr. Fawcett
<http://ecs.syr.edu/faculty/fawcett/handouts/webpages/BlogParser.htm>
- [3] Parser design details, provided by Dr. Fawcett
<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Parser/>
- [4] Demonstration of type relationships, provided by Dr. Fawcett
<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project2HelpF14/TypeRelationships/ps/>

9 Appendix

9.1 Parser

In computers, Parsing code is the process of discovering and summarizing some form of syntactical information from a set of code files, which can be used for lot of reasons. For example, Parsing can be used for building code analysis tools, evaluating code metrics etc.

A rule-based parser is already developed by Dr. Fawcett. The Code Analyzer tool will use the Parser for its functionality. The Parser project is configurable with a different set of rules and actions for different applications. ConfigParser is a class which builds parser and configures it with application specific rules and actions. So, it helps us to use the Parser for our application specific rules.

Parser Class Diagram:

The following class diagram will help to understand the architecture design of Parser class in much detail.

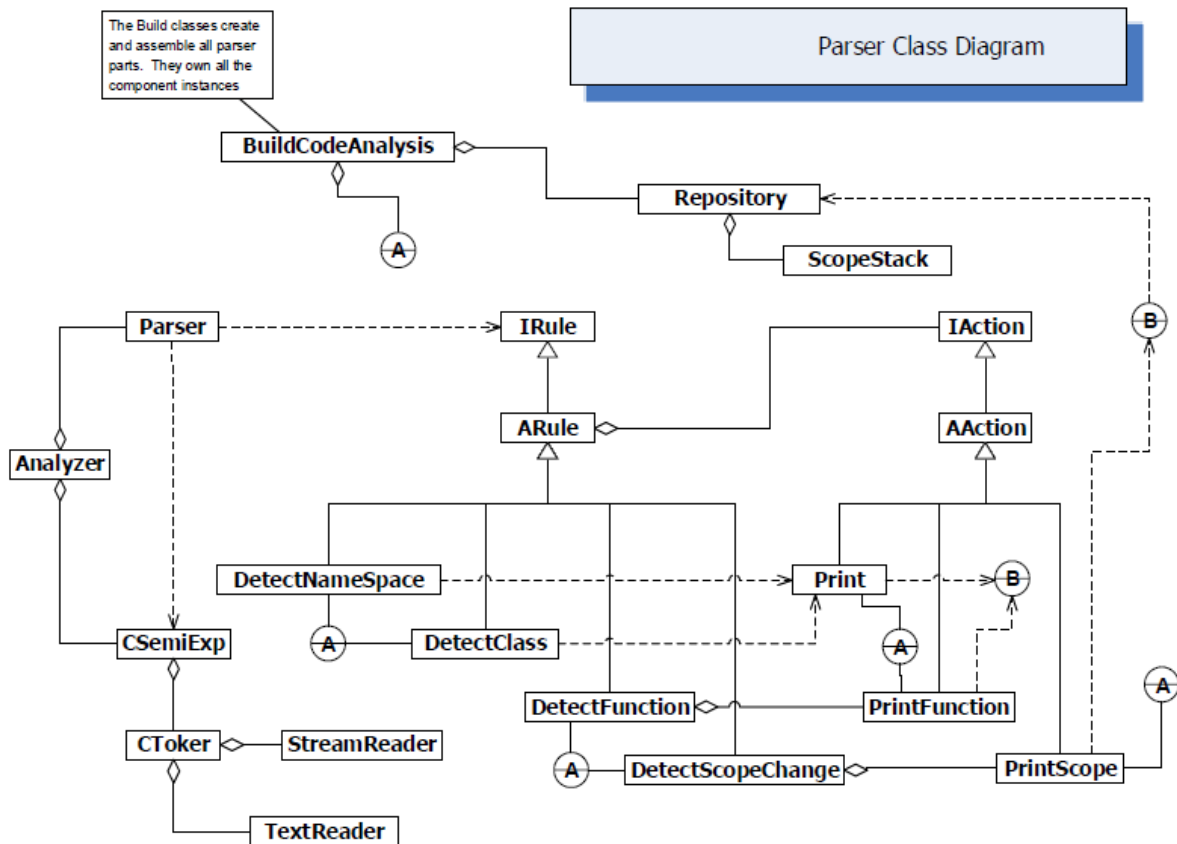


Figure 3: Parser Class Diagram

Sample output of Parser code:

```

C:\Windows\system32\cmd.exe

Demonstrating Parser
=====
Commandline args are:
..\..\ parser.cs RulesAndActions.cs token.cs TestCase-eligibleCheck.cs
current directory: C:\Users\neethu\Documents\Visual Studio 2013\Projects\SMA\SMSU\Parser\Parser\bin\Debug
Processing file C:\Users\neethu\Documents\Visual Studio 2013\Projects\SMA\SMSU\Parser\Parser\Parser.cs
Type and Function Analysis
-----
line# 44 entering namespace CodeAnalysis
line# 49 entering class Parser
line# 53 entering function Parser
line# 56 leaving function Parser
line# 57 entering function add
line# 60 leaving function add
line# 61 entering function parse
line# 72 leaving function parse
line# 73 leaving class Parser
line# 75 entering class TestParser
line# 79 entering function ProcessCommandLine
line# 95 leaving function ProcessCommandLine
line# 97 entering function ShowCommandLine
line# 106 leaving function ShowCommandLine
line# 112 entering function Main
line# 158 leaving function Main
line# 160 leaving class TestParser
line# 161 leaving namespace CodeAnalysis

locations table contains:
namespace, CodeAnalysis, 44, 161
class, Parser, 49, 73
function, Parser, 53, 56
function, add, 57, 60
function, parse, 61, 72
class, TestParser, 75, 160
function, ProcessCommandLine, 79, 95
function, ShowCommandLine, 97, 106
function, Main, 112, 158

That's all folks!

Processing file C:\Users\neethu\Documents\Visual Studio 2013\Projects\SMA\SMSU\Parser\Parser\RulesAndActions.cs

```

Figure 4: Sample output of Parser Code

From the above output it is understood that the current configuration of parser extracted namespaces, classes and functions information along with line numbers.

For more details on Parser design and its code, following links can be referred:

<http://www.ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Parser/>
<http://ecs.syr.edu/faculty/fawcett/handouts/webpages/BlogParser.htm>

9.2 Type Relationships

The Code Analyzer identifies the following four type relationships in a set of code files:

1. Inheritance
Derived class inherits most members of the base class. Exceptions are constructors and destructors.
2. Composition
Instances of value types can be composed by another class or struct. They occupy space within the memory footprint of the composing class.

3. Aggregation

Instances of reference types may be aggregated by another class or struct. They are constructed with a call to operator new and reside in the managed heap.

4. Using

Any type may be used by any class or struct. An instance of the used type is passed as an argument of a method of the using class or struct.

Refer to the following link for a demonstration of type relationships, provided by Dr. Fawcett:

<http://ecs.syr.edu/faculty/fawcett/handouts/CSE681/code/Project2HelpF14/TypeRelationships/>

9.3 Scope related information

Scope: Scope refers to the code block that starts with bracket and ends with closing bracket. The scopes defined will determine the "lifetime" of a name that does not denote an object of static extent. The blocks such as if, else, else if, for, switch, while, do-while, function, empty open-curly braces etc. will be considered as scopes.

Scope structure: The order of nested scopes defined in a particular code is called as Scope structure, which can also be referred as scope tree of the code.

Scope size: The number of lines of code in the defined scope.

Scope complexity: The number of scopes defined in a particular code. It can be also defined as the number of elements in the scope tree of a particular code.