**CSE 681 Software Modeling and Analysis**

**Project 1**

**Code Analyzer**

**Sahithi Desu**

**Graduate student, Computer engineering**

**Syracuse University.**

**Instructor: Jim Fawcett**

**Version 1.0**

**15th September 2014**

***Table of Contents***

1. Purpose………………………………………………………………………….3

2. Introduction …………………………………………………………….3

3. Use Case Analysis…………………………………………………….3

4. Users…………………………………………………………………….4

5. Use case Diagram for Code Analyzer………………………………5

6. Context Diagram for Code Analyzer……………………………... 6

7. Code Analyzer Modules……………………………………………….7

6. Module Diagram………………………………………………………7

7. Activities and Activity Diagram………………………………………9

8. Code Analyzer Architecture ………………………………………..11

9. Critical Issues…………………………………………………………14

10. Technological Advancements……………………………………..16

11. Summary…………………………………………………………….17

12. References………………………………………………………… 17

Code Analyzer

1. **Purpose:**

The Code Analyzer is an automated tool which will analyze different types and relationships between types which are present as a set of files in different directories. The goal of this project is to create an automated tool which will automatically traverse each directory and its subdirectories by extracting the file pattern given and displays all kinds of types and their relationships. The code analyzer will especially look in to different classes to display the kinds of relationships that a class has with another class, with in the file or across the files for subdirectories present in the directory. This document deals with the description of the architecture of the code analyzer. The critical issues of these system can be increase in the load of the system, crash recovery, security, non-terminating code, Change in File name and content, File pattern matching. These are discussed in detail in the later part of the document.

**2. Introduction:**

The term ‘Code Analysis’ is named for the analysis performed by an automated tool, with human analyses and understanding about the program. This tool is intended to resolve names to declarations, expressions to types from certain actions and rules and finally displaying the results, i.e. types and their relationships. A file in which a C# program resides may hold the following “types” like struct, enum, class, delegates, abstract class, non-abstract class, interfaces including all the built in types. These types in turn will hold different “relationships” with one another. This tool is intended to analyze the different relationships between classes.

* Inheritance
* Composition
* Aggregation
* Using

These Identified relationships will help the user to understand the structure of the program and enable him to make the decisions wisely which plays a vital role in writing a quality code considering size and complexity metrics in to consideration. In addition, this tool enables the user to deal with complex system easily. This tool also generates metrics like nesting depth, size of the code, Fan IN (tells how many modules are dependent on a single module), Fan Out (how many modules a single module is dependent on), Modularity index about the source code. These metrics generated will help to analyze code complexity and help the users to develop the code with less complexity.

**3. Use case Analysis:**

The behavior of the software that occurs when the user requests or interacts with the software is described as a “Use case”. This methodology helps to identify how this software can be used by a broad range of users and how do they interact with it, which is helpful for analyzing requirements.

**3.1 Principle Users**

* **Software Architect:** It is the prime responsibility of the software architect to make design choices and to recognize potential reuse in the organization or in the application.

If there is any system that can be build using the system which has already been analyzed, architect can use this tool for framing the architecture of the system by looking in to the structure of the system which has been already analyzed by the tool by looking at its class structure and its relationships.

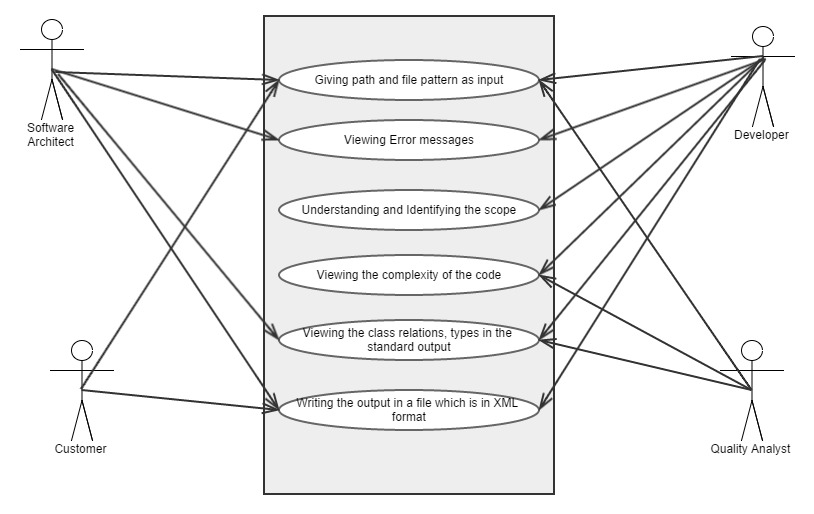
* **Quality Analyst:** The role of the QA is to determine the quality of the code and validate whether the code has met the requirements or not. He can use the tool to analyze the quality by calculating the values of the metrics produced and reduce the complexity of the code. The types and their relationships that are analyzed by this tool will help quality analyst to calculate the degree of coupling and cohesion. For a code to have high quality, low degree coupling is preferred as the degree in which each modules depend on each other should be less.
* **Developers:** Using this tool developers can identify the coding issues and bugs at the early stage of development cycle, passing better code into the later stage of development. He can estimate which area of code is prone to bugs by identifying the dependency relationships and looking in to the hierarchical structure of the classes. The more the depth of the hierarchical structure and dependencies, the more the degree of cohesion increases.
* **Customer:** Customer doesn’t care about the detailed working of the project. So, a comprehensive view would be sufficient for the customer to understand the working of the project. The relationships produced by the tool will help him to understand the necessary details.

**3.2 Use Cases:**

The use case depicts how the user interacts with the system. The following use cases are:

* **Giving the path as the input:** The user specifies the path and the file pattern in the command prompt. This file pattern is analyzed and all the files matching to the given pattern is displayed. The user can view the list of subdirectories rooted from the current path.
* **Viewing error messages:** Error handling mechanism has been incorporated in this tool. User can view the error messages if there is any exception or error.
* **Viewing the output in an XML file:** Code analyzer will read these files and identify all the different types. It will read again to derive the type relations and display them in a tree structure in an xml file. User can view the report for understanding different types and as well as their relationships among them.
* **Identifying and viewing the scope:** Code analyzer also displays the scope of a class, function or names cape using scope stack. By looking at this the user can understand the scope of the each type and identify in which scope he is currently viewing.
* **Viewing the complexity of the code:** The size of the code written and the metrics calculated by the analyzer can be viewed by the user to estimate the complexity of the code.

**Use Case Diagram for Code Analyzer:**

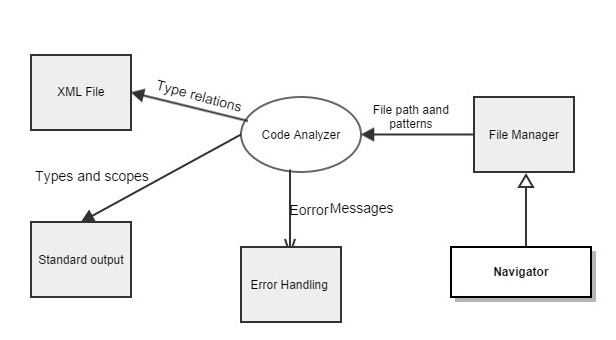
****

**3.3 User Interface:**

* The **command prompt** allows user to provide input, **file path and the pattern** to the code analyzer.
* The **File Manager** takes the input and calls “**process command line function”** to get all the absolute paths for the relative paths given. This extracts all the files in a list and gives as an input to the code analyzer for analyzing.
* **Navigator**, which is the part of file manager recursively iterates through the current path and displays all the directory and subdirectory structures rooted from the current path when the analyzer gives **/S** as an option in the command prompt.
* The output of the analyzer, identified **types** and **relationships**, are produced as a standard console output in a form of a tree on the **standard output** when user gives /R as an option in the command prompt.
* When the system is provided with an option of **/X**, the output will be written on a file with **XML** format.

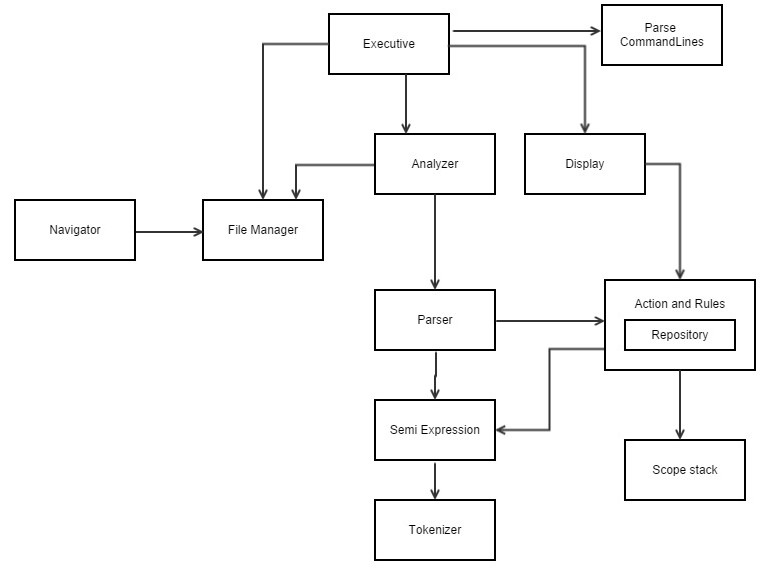
**3.4 Context Diagram for Code Analyzer:**

The context diagram describes about how the system being interacted with the surrounding environment. The user provides input through the command prompt. After processing the input, Code Analyzer lists all the types, their scopes along with the size of the input. Type relations are also produced on user’s input in the command prompt and results will be written in an XML File. Code Analyzer provides error handling mechanism. So, the system passes error messages when any error is occurred.



**4. Modules Involved:**

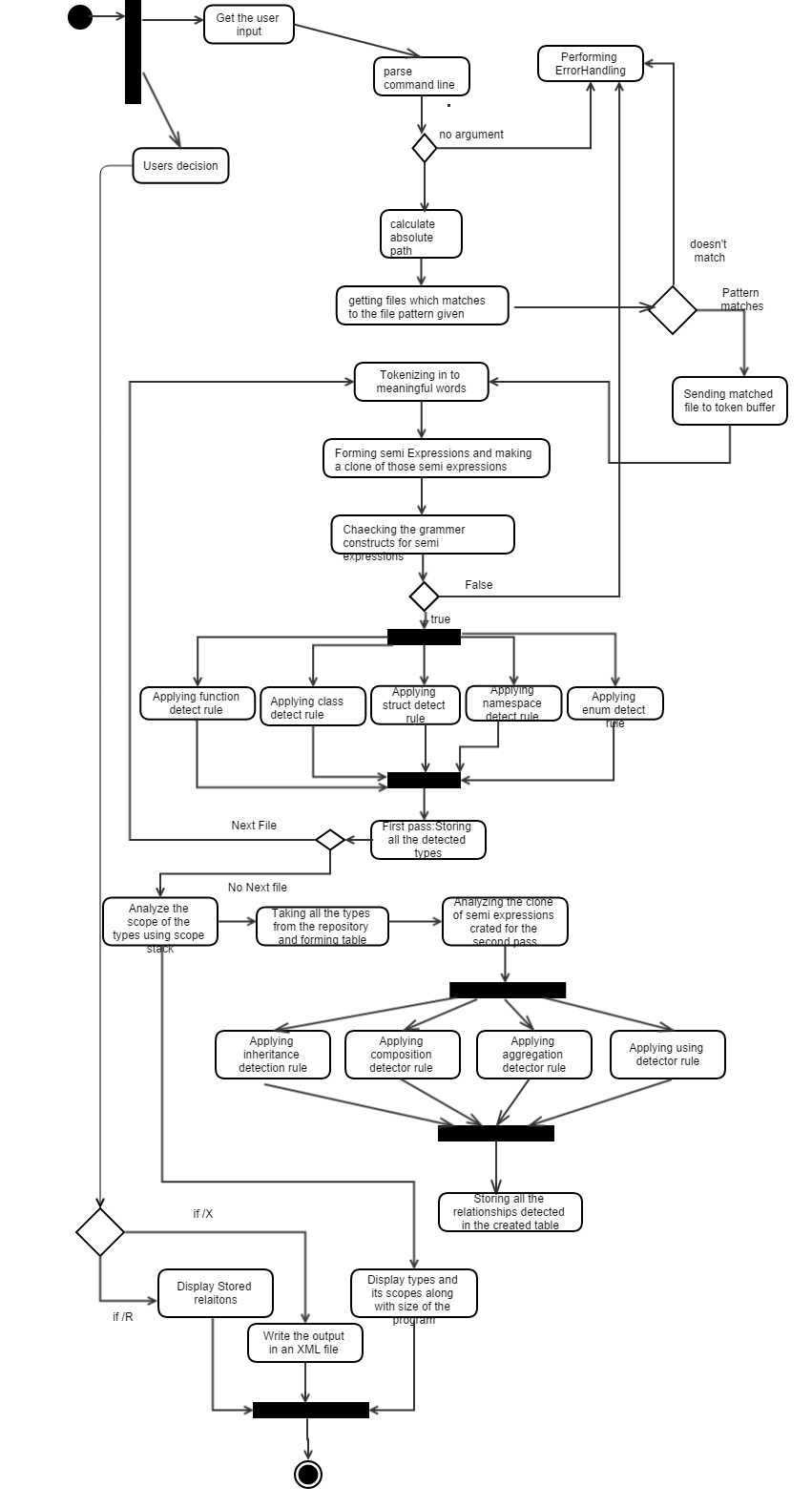
* **Executive**
* **File Manager**
* **Tokenizer**
* **Semi Expression Analyzer**
* **Parser**
* **Rules and Actions**
* **Relationship Analyzer**
* **Display**

**Module Diagram**

* **Executive Module:** This is where all the activities in the system are integrated. It is the start of the project and initiates File manager. This parses the command line argument given by the user.
* **File Manager:** It processes the arguments in the command line and calculates absolute path from the given relative path and search for the files that match to the pattern given in the path specified with the help of navigator, which recursively traverses in each subdirectory.
* **Tokenizer:** It reads symbols, punctuations and characters from the file stream attach to it and convert them in to meaningful tokens removing white spaces and comments.
* **Semi Expression Analyzer:** It builds semi expressions from the bunch of tokens which has to go through parser. These sequences end with ; or { or }.
* **Parser:** It analyzes the source code and performs type and function analysis. It detects code constructs defined by the interface contract rules. Each of these rules have a grammar construct detector and also a collection of interface contract of actions.
* **Rules and Actions:** This has all the code specific to the application mostly required for analysis. It defines all the rules which incorporates grammar constructs. These rules are used by the parser to analyze the expressions given by semi expressions.
* **Relationship Analyzer:** It detects the relationships between classes by using four detector functions which detects inheritance, composition, aggregation and using.
* **Display:** This module deals with the displaying of namespace and type analysis along with the type relations detected. It also handles the job of printing the complexities and sizes along with the scope of the class(detected by the scope analyzer)

**5. Code Analyzer Activities:**

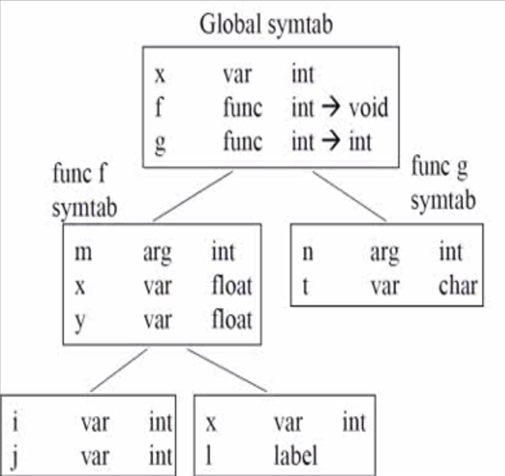
Here are the list of activities that code analyzer performs when the application is started.

* Get user input
* Parse the arguments given in the command line
* Get the absolute path from the relative path given
* Get all the files that match with the given pattern
* In First pass detect all the types present
* Store all the types in the repository
* Calculate no of namespaces, classes, functions, struts and enums along with the number of lines of code.
* Analyze the scope of the types using Scope stack
* Display all the types along with their scope and their line number
* In Second pass detect all the relationships present between each class
* Store all the relationships in tables
* ****On users decision display all the identified relations on standard output or write them in an XML file.

The above diagram shows how the control flows from one activity to another. It basically tells the operation of the system. Here are some of the important activities performed.

1. **Getting user input and fetching the matching files**: It will parse each path and analyze the file pattern and for each path and pattern, it will recursively traverse through all subdirectories and for each subdirectories, it will find all the files matching to the pattern and for each file, it will check the pattern and will add it to the file list. Where each of the files in that list will go inside Code Analyzer for processing.
2. **First Pass (Analyzing types)**: In the first pass the Analyzer detects all the types using type detectors and store them in the repository and this process happens till there are no files left in the file list. For each type it identifies, it will increment the count of that particular type and displays the count of the each type.
3. **Analyzing Scope**: The Scope Analyzer takes the input from the semi expressions and for each type stored in the repository it will analyze the scope using scope stack. In the repository a “map” data structure is created where all the types along with their values are stored in a **Symbol Table**. The occurrence of newscopes will be identified using scope stack by pushing it in the stack. While popping it from the stack, it is stored in the form of hierarchical tree and for each node in the hierarchical tree a symbol table is used.

**This diagram illustrates how the values are stored in the Symbol table**.

 int x;

void f(int m){

float x,y;

…….

….

{int i,j;}

{int x,l;}

}

Int g(int n)

{

char t;

…….

}

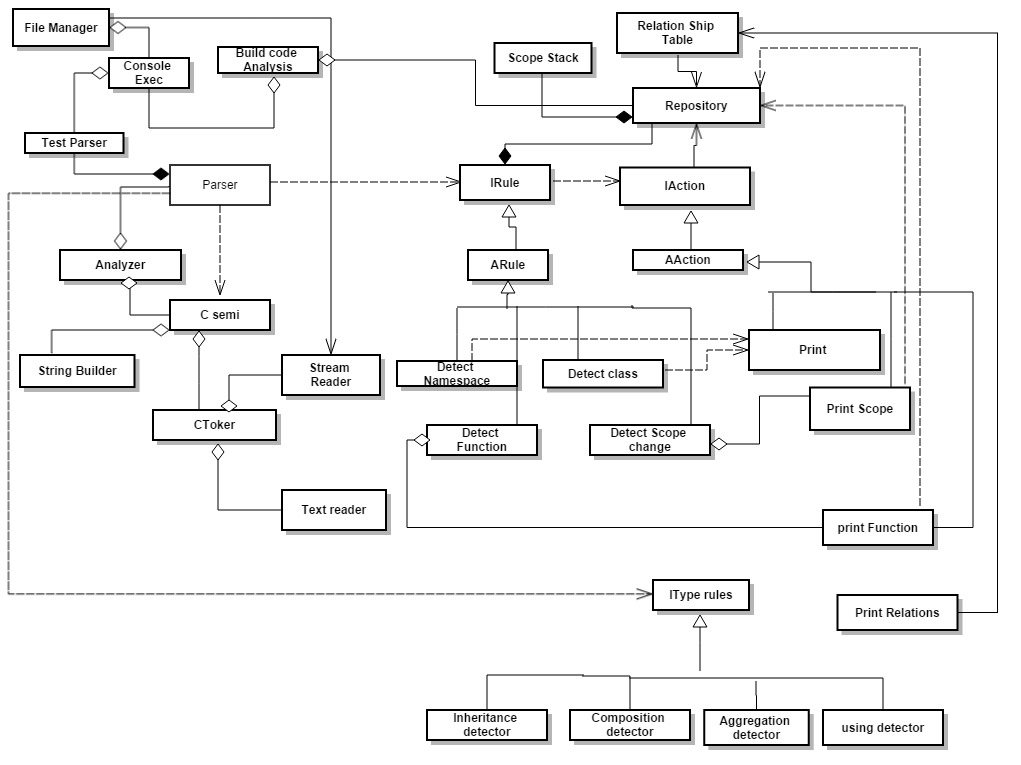
1. **Operations performed for identify relations before second pass**: Before the analyzer gets started with the second pass all the types are stored in the hash table in a sorted order and all the types are numbered, which forms a key for the hash table.
2. **Second Pass (Identifying Relationships):** In the second pass the analyzer will detect all the relations between the types and for each relation detected, it will update the hash table by finding the key (type name no) and insert them in the form of linked list corresponding to the key. This operation takes just O(1). And for displaying detected relations for a particular type, the analyzer has to search the key, (type name) and traverse the linked list attached to that class till the end of the link list, which takes O(K) complexity, where K is the length of the linked list . This process is repeated till there are no files to analyze.
3. **Display**: Based on the users decision the output will be shown on the standard output or will be written in an XML file.

**5. Code Analyzer Classes:**

* The executive class is the main class and it is the start of the process. The **BuidCodeAnalysis** class is the one which configures the parser. It builds and assembles all parser parts. It has all the component instances. **Console Exe**

Class has an instance of **Test parser** class, which parses the command line argument and calculates the absolute path. This creates the list of files and gives

them to the **File manager**. File Manager uses **Navigator** class which

**Code Analyzer Architecture**

Navigates through the directories. File Manager returns the file to be analyzed to the stream reader after fetching all the files which matches to the pattern.

* Token buffer, which is the temporary storage of all the characters opens the file in the stream reader and fetches the characters to tokenize using Text reader, which takes strings from the stream reader. Toker provides a class **CToker** which facilitates Tokenizing. Here Tokenizer doesn’t handle strings that are going to start with ‘@’. Tokenizer throws an exception when it is encountered. When it encounters a leading comment or trailing comment, stores the leading and trailing comment in the token buffer and will pass back the entire string. It will consider the comment as a single token. If it encounters carriage return (/r) it removes it from the string and then tokenize the rest. The main responsibility of this Tokenizer is to remove all tha white spaces and comments.
* Semi Expression has **CSemiExp** class which extracts all the semi expressions from the tokens formed. As discussed before, Semi Expressions are token sequences that end with “}”,”{ “ and “;”. Semi Expression needs tokenizer for building semi expressions. It creates an object of **C-Toker** class which helps semi to get the tokens form tokenizer to from semi Expressions from the attached file. It creates expression, which parser needs to analyze. Semi Expression uses string builder to append the tokens coming from Tokenizer instead of string. Since it is not advisable to create instances for all.
* Rules and Actions has classes which are used more for analyzing. This defines all the detection rules by inheriting Abstract rules and interface rules. Parser uses this detection rules to detect the types. Then it will request the **IAction** and **Aaction** to perform actions. Without request from Parser, Rules and Actions cannot execute by itself. The Rules and Actions package defines **Repository** class which uses **Scope stack** to calculate the scope of the variables. Scope Stack recursively calculates the entry and exit of the scope. This Repository is used for passing data between the actions. Semi takes the line count of the types from the tokenizer and repository uses semi to get the line count of the type. Scope information of the particular type is calculated by fetching expressions from semi and push them in to the stack when it encounters { and pop when it encounters }. This information is taken by the repository and a “location list table “ is created inside the repository in which line number, type, type name along with the scope is stored and displayed.
* The “I**typeDetector**” has a contract of all type relation ship rules and is inherited by Relationship detected rules (Inheritance detected rule, Composition detector rule, aggregation detector rule, using detector rule). The relationships are identified and they are stored in relationship table. The detailed explanation of this process is explained in the activity diagram (refer to point 4 and 5)

**6.Critical Issues:**

Identifying critical issues will help a lot to identify the pitfalls that might occur during developing, designing and maintaining. This issues need a special focus to improve the performance, reliability, accuracy of the system.

* **Load on the System:** When user has given a path where in which there are large number of files to process the load of the system will increase and the performance of the system degrades. Even in this situation the system should be able to withstand this without getting crashed. It should calculate the estimated time and display error message to the user telling that the program takes large amount of time notify him that it will abort.
* **Namespace Resolution:** Each reference of a class in C# has a namespace resolution. There might be two classes of same name with different namespaces. Code analyzer can identify the type relations in two pass or single pass. To resolve this issue code analyzer will identify the relations in two pass rather than identifying in a single pass, though single pass has high performance when the tool is dealing with large number of files. In the Single pass we will have to restrict the user to give the class reference with fully qualified names otherwise we will have to store every reference to the class name with all the possible name spaces that this class might exist in which is very complicated. So here we decided that the program runs in two passes.
* **Concurrent Requests:** At a particular point of time if there are N number of concurrent requests coming from N number of clients for analyzing their code the system has to log the no of client requests to estimate how much memory all the processes can take. If the memory allocated is not sufficient in the server, then either a notification message has to be sent to the client that memory is not sufficient or increase the memory allocation in the server.
* **Change in File name and Content inside the file:** When file name has been changed during the process of analyzation, then system has to log the changes happening. After the process is done system has to check with the log and notify user that the file content has been changed and ask him whether he wants to see the result for the previous information or he wants to start the process. Other solution is we can lock the files that has gone in to code analyzer so that the user cannot make any changes.
* **Occurrence of nonterminating process:** When the userencounters infinite loop or deadlock, the user can’t sit for a long time in front of the computer for retrieving the results. In this situation a user should be notified that analysis is taking longer time than expected and will abort. Information about the program’s state has to be displayed so that the user exactly knows how many files has been processed and how many relations has been identified and exactly know in which file the process has been aborted. By giving this information, he can rectify his mistake and can give for processing again from the place he got aborted instead of starting of starting the process again.
* **Security**: When the client sends the file to the server in a network, the file should be secured till it reaches server. This can be done by sending the file in https protocol or openSSC.
* **Crash Recovery**: If the program is taking longer time, it might crash because of any reason. The system should create log of error handling messages so that it will be helpful for the developer to fix the bugs. If crash occurs the in the initial stage itself (when listing the files) recovery is very easy. Only file path (input) can be saved so that the user need not give the input again. If the crash occurs while analyzing the types (in 1st pass) then the system has to make the index for the file name and as well as for the list of types. The index numbers where the crash occurred will be saved along with the information (types) analyzed till now. If the error occurs when analyzing the type relations (2nd pass) it also has to make an index for semi expressions and as well as for the relationships analyzed. This index is saved along with the list of type relationships analyzed. Now when the error has been recovered the user can start the process from the index where he has stopped as he has saved the index. But before doing that he has to check with all the file list and their modified date before proceeding otherwise start the process from the beginning for the reliable output to be produced.
* **File Pattern matching:** This system is been restricted only to analyze .cs files. If any other file pattern is given file won’t be taken for analyzing. Message will be sent to the user that this system doesn’t accept any other file pattern other than .cs.
* **Occurrence of Semantic errors:** In the process of identifying different types and their relations we are checking our semi expressions formed are following the rules which has grammar constructs in it. We are not checking whether the semi expression formed satisfies the semantic rules or not. These semantics rules can be type checking rules or checking variables whether they are defined before without declaring it. For example, we are not checking for the array bounds when we are analyzing, we are just checking whether the syntax of the type, array, is declared properly or not.

1. **Technology Transitions**:

Initially the system handles particular number of files. In future depending on how successful this project is the number of files that can be processed will be increased by incorporating some performance, reliability and reusability measures to the system.

**Redundancy:** If there is a situation where a client gives a path where files in that path has already been analyzed, the system should not start analyzing that code, which will be a redundant task. This issue can be resolved by having a temporary storage of all the XML files generated and all well as the list of all the files and processed and its last modified date. If the user gives a path where the file name and last modified date are same, then the system directly gives the corresponding XML file to the user instead of going through the whole process again. By doing this my system load decreases and will analyze for more number of files.

**Implementing Factory:** The location of the concrete class where all the objects are constructed is factory. This pattern is used to insulate the of reation of objects from their usage and to create related objects without they having to depend on their concrete class. This allows new derived types to be created with no change in the code that uses the base class. Here we can change the concrete code with changing the code that uses them. By doing this changes made wont effect much of the code and reliability increases. For example in my system I can create a factory method which can configure my parser and have all component instances.

**Reusing objects:** For increasing the performance the code analyzer can develop a ware house where initially according to the initial client requests some number of objects are created and instantiated and stored in the warehouse. When the new client requests comes it will check in the warehouse whether they are any instances available, otherwise it will create a new object. If the client completes his task and leaves his object instance will be stored again in the warehouse. The main purpose of this is to avoid release of resources by recycling the objects that are no longer in use. Factory takes care of instantiation of objects but it doesn’t care about the objects after instantiation. The object Pool takes care of it.

**Making system to analyze the files written in different languages:** To increase the scalability, the code will be designed in such a way that one can add new plugins which can support new protocols. For example, the present system will execute only .cs files now the system will analyze .cpp by just adding that plugin.

**Adding behavior of an object dynamically**: If a behavior has to added to an object dynamically doing with inheritance is not a feasible thing since it applies to the entire class. So a decorator class is being created

1. **Summary:**

Code Analyzer is a tool which analyzes the types and the relationships between them. It has console interface with the user to provide input path and file pattern. The output is shown to the user according to his decision whether he want to see the types, complexities, relationships between types or write the output in an XML file.

This System comprises of 4 main modules: Executive, File manager, parser, display. Each module is discussed in this document and a module diagram is given to show. The flow of the process from the user’s input till the output generation was clearly explained in the activity diagram. Critical issues of this project has been discussed here and also some thoughts of technological transitions were discussed briefly.

1. **References**:

* Sample OCD: “Test Harness “ Sample (Given in Jim Fawcett’s Website).
* For definitions, sample codes, Diagrams: Jim Fawcett Lecture #1, #2, #3 and handouts
* Design Patterns: <http://www.dofactory.com/Patterns>

<http://sourcemaking.com/design_patterns>