## Executive Summary

This is an Operational Concept Document (OCD) for a Lexical analyzer, which is part of a code analyzer that is essentially a parser used to ease the process of developing and maintaining large software. During the maintenance phase when the client wants to add additional features or when a software project is distributed among various developers, a code analyzer can help in the task of logically studying and breaking down the software without the actual manual hard work.

The lexical analyzer is building block for the code analyzer, which helps in its primitive stage in the processing and breaking up of code. It accepts lines of code and outputs logically partitioned code based on a set of rules, which will be defined as we progress further in the document.

The project can be extended to accommodate a number of languages depending upon their syntax. Currently the Lexical analyzer has rules to handle code of software from the c# domain. The project in its current state can accomplish the following objectives: -

1. Break down the software code in to tokens where a token can be defined as a group of logically defined characters.
2. Form a semi expression from a set of tokens depending upon the terminating conditions. A semi expression is set of tokens that occur in a sequence and logically form a line of dependent code.
3. Is capable of dividing input file characters to a set of 7 token states depending upon the type of character.
4. Is capable of dividing the tokens to a set of 3 semi expression states depending upon the terminating condition.

***Users***

Given the objectives achieved by the project the intended users include: -

1. The next modules in the code analyser which will use semi expression as an input
2. Software developers that will use the codeAnalyser software will indirectly use the lexical analyser software
3. Since the software is simple to implement the client may also be able to use it.
4. Compilers use lexical analysers to convert code to machine code after syntactical analysis
5. A change in the rules of the lexical analyser can help introduce more applications.

***Uses***

The uses of the Lexical analyser include:

* Analyzing code while maintaining software
* Analyzing code while adding new features to the an existing software
* The lexical analyser can help search for specific instances in the code for error solving
* The lexical analyser can be extended to the Natural language domain and similar classes can be built to identify parts of speech.
* The Lexical analysers can be used to demonstrate how compilers function.

## Introduction

Lexical Analyzer is a code analysis tool, which we will use to analyze and test considerably large software.

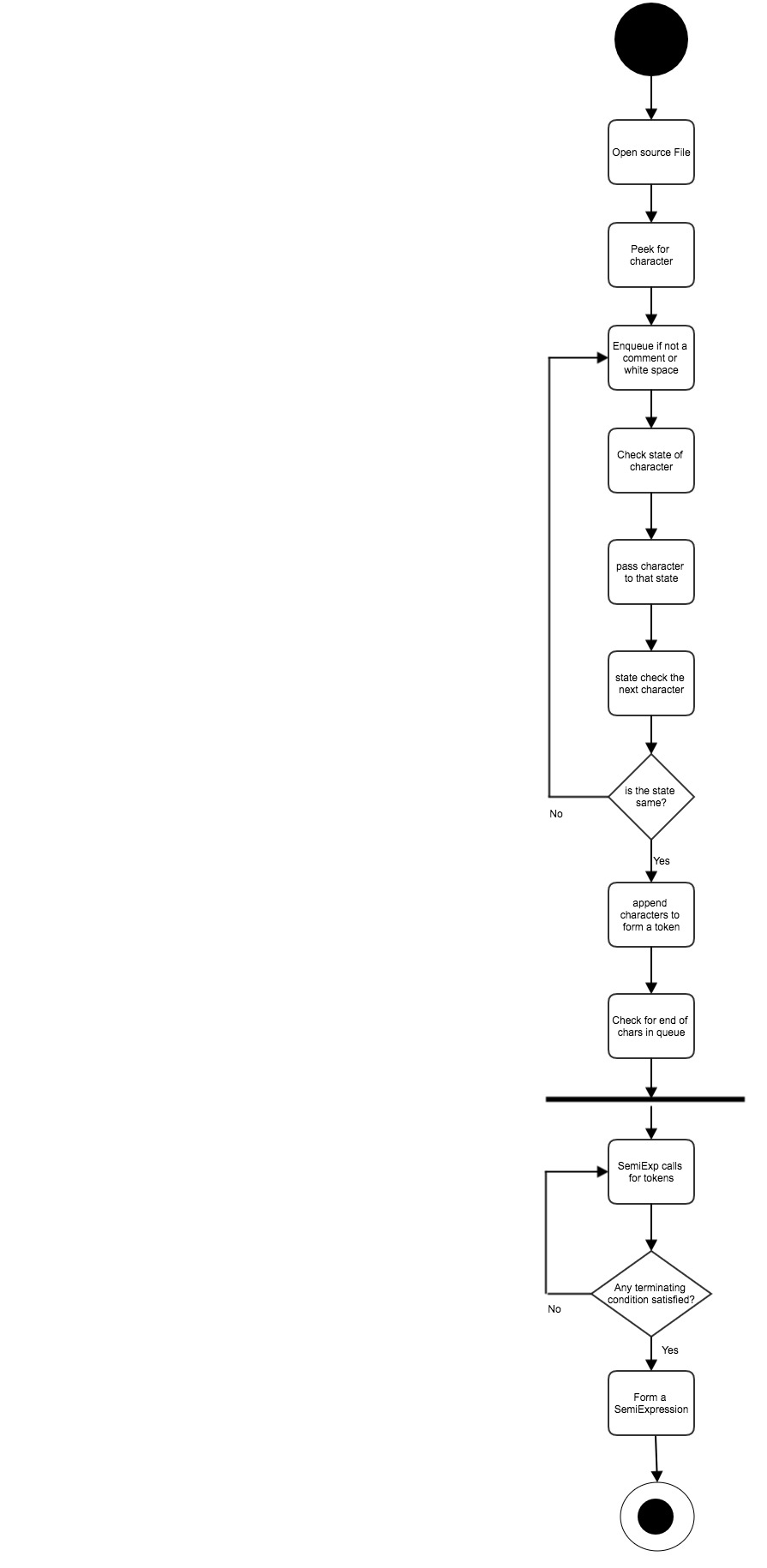
***Concept and key Architectural idea***

The basic concept of the software is to break up the source code into tokens and form semi expressions from the broken up tokens. It does this using a set of states for tokens and a set of terminating conditions for semi expressions. The aim is to form an Abstract Syntax tree from the above information to help in analyzing the code. For this purpose the Lexical analyzer is the first step. The current software is written in C# and will also work for the analysis of the same language. Since the analysis and grouping is based on these set conditions it can be extended to accommodate other languages like C, C++, Java. For this purpose the class partition becomes key in the software as only replacing the classes which check conditions can help in analysis of another language. We therefore divide the project in to two separate entities i.e. the Tokeniser and the SemiExpression.

***Application Activities***

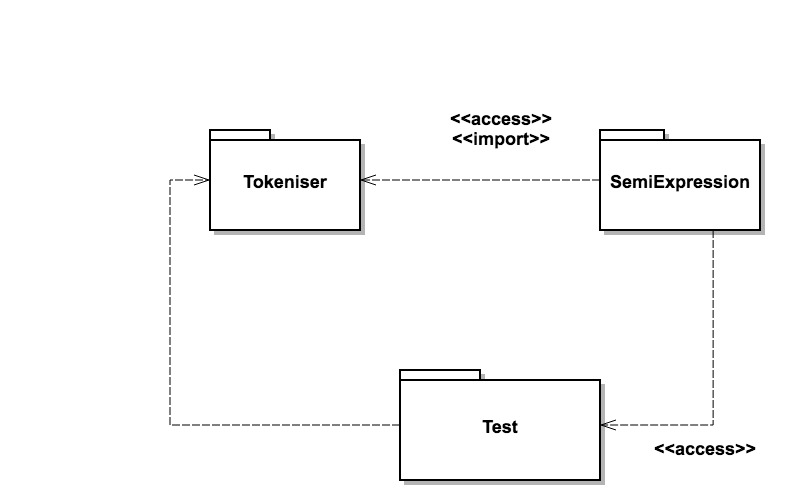
The figure below shows the activity diagram of the Lexical Analyzer.

1. The Lexical analyser takes input the source code of the software in consideration.
2. The tokeniser peeks for characters in the source file, which will be explained in detail further.
3. It is capable of ignoring white space and comments, which are not necessary for code analysis. All other characters are stored in a queue for further analysis.
4. The state of the character is checked depending upon the 7 predefined states.
5. The address location of the character is passed to the corresponding state.
6. The class checks if the sequence of characters belongs to the same state.
7. If not, it only creates a token of the current character and returns.
8. If yes it creates a token of all characters belonging to that state.
9. It then checks if the queue is out of characters or proceeds to create further tokens.
10. The semiexpression part does not start until the entire queue has been popped this is shown using the synchronization bar. It guarantees that each character is appropriately distinguished as a token.
11. Once this process is done the semiexpression gets triggered and starts asking for tokens.
12. It checks for terminating conditions and creates the semiexpression. This is the desired output from the lexical analyser.



***Partition***

***System Packages.***



***Tokeniser***

The tokeniser package has the responsibility of taking in the input source file and providing tokens to the SemiExpression package. Its activities include grouping characters in token as well as it holds the conditions for grouping them. It has a queue to peek for characters that matter in analyzing. It additionally contains a test for the package itself that checks for correct tokenization.

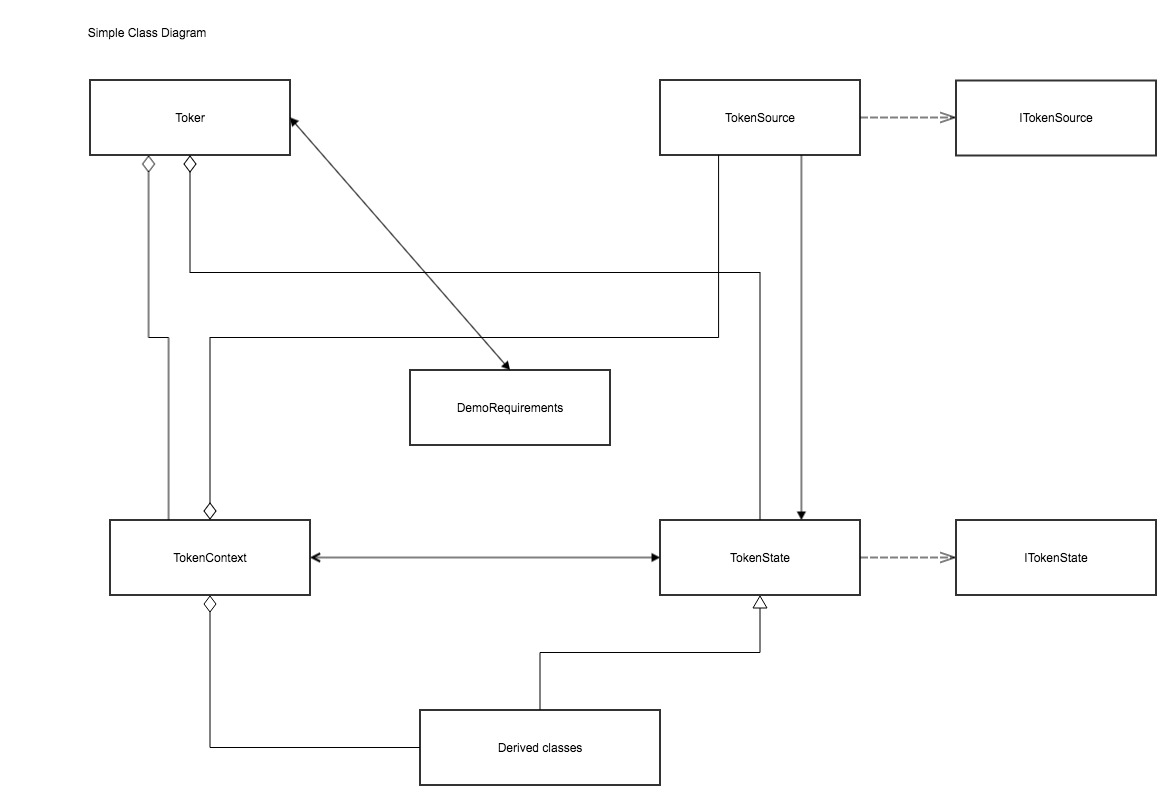
***SemiExpression***

The SemiExpression package is triggered when the source file is entirely converted to tokens. The package contains classes that call for tokens repeatedly until the termination condition to form a SemiExpression is satisfied. The output is a semi expression. It also has the rules for the terminating conditions. It additionally contains tests for the package, which tests the correct creation of semi expressions.

***Test Package***

The test package contains an automated test that verifies the output of the system using test cases generated by the developer. The test cases should contain tests, which can trick the system to give incorrect results. It also contains the correct solution to verify the test.

***Simple Class diagram for tokeniser***

******

The above diagram is simple class diagram for the tokeniser package showing dependencies between the classes.

***TokenSource class***

The TokenSource class is the first step after that the input has to go through. Since the input is in character form and not all ‘terminators’ are synthesized correctly we first convert them to their equivalent integer values. This ensures that each character is read correctly. It contains the peek functions, which is used to get rid of comments and white space in the first step itself. The peek function checks for characters from the stream without extracting them and is a vital step. It uses methods from the TokenState class for this purpose and therefore a link between them is shown in the diagram.

***ITokenSource Interface***

It contains the methods used by the TokenSource class to check end of the stream as well as opening the stream.

***TokenContext class***

It holds all the tokeniser states that contains the instance context used by the different classes to work with the characters. It contains the source of the tokens and hence is qualified as internal to avoid any data manipulation from any other class. The states can be set depending upon what state condition it satisfies. All the derived classes uses its object to manipulate characters hence they have been linked with and aggregation as they implement this class.

***Toker class***

The toker class is used to collect tokens. It contains functions to check, open, close the source string. The main method contained in the toker class is getTok() which is used by the SemiExpression class in the SemiExpression package. For this purpose and security reasons this class contains methods only relevant to that function.

***TokenState class***

The main operation after peeking and collecting relevant characters begins in this class. It checks the state of the given character and sets the state in the context\_ instance of the TokenContext class and forwards the character to the appropriate state class. When the character reaches the next state it is already know that it belongs to that state so the next class only has to check the state of the forthcoming characters and form the token.

***ITokenState***

It contains the relevant methods for each state to form tokens after they have been passed to the derived states. They override them according to the needs of that particular derived class.

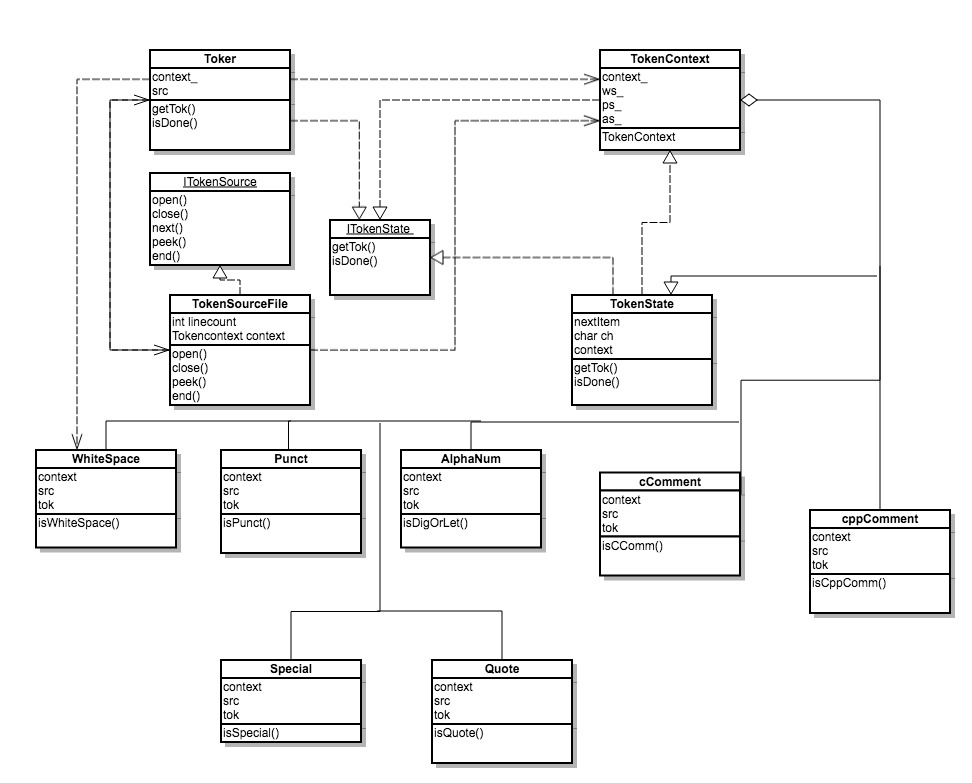
***Derived classes***

These classes are divided upon the test conducted in the TokenState class and hence that class invokes them only. They contain relevant methods to create tokens. Since in the beginning we converted the characters to integers theses classes have the methods to get the characters back.

***DemoRequirements***

This class contains the test for the tokeniser package according to the developer requirements.

***Complex class Diagram for Tokeniser***

******

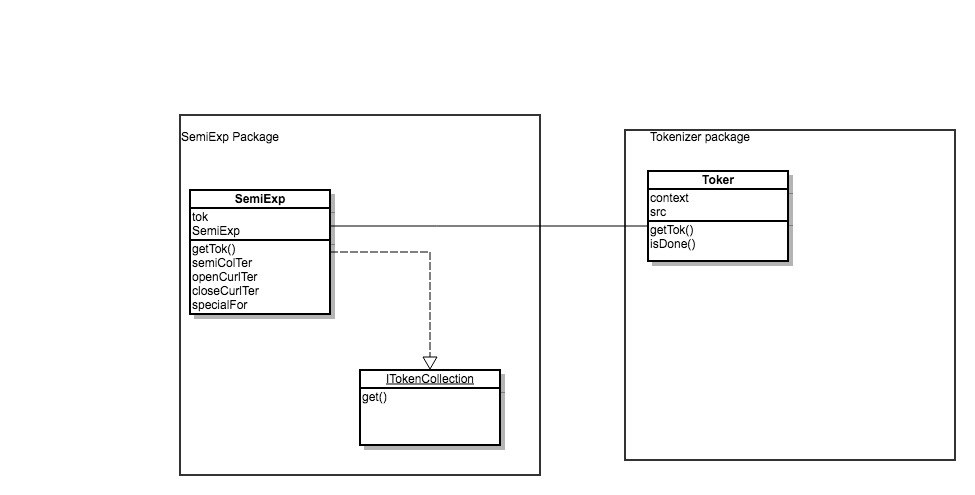
In this diagram the relevant methods of each class are specified. Since we have already explained the functions of each class we will only discuss the functions of the derived classes briefly.

1. WhiteSpace class is used to detect white space initially.
2. Punct class is used to detect punctuation marks. The logic is that if the character is neither white space nor Alphabet or number then it is punctuation.
3. Alphanum class detects a series of letters and numbers and makes them in a token.
4. cComment and cppComment are used to detect C and C++ comments using appropriate indentifiers for each language.
5. Special class includes methods to detect special one and two character tokens. Special one character tokens: <, >, [, ], (, ), {, }, :, =, +, -, \*, \

Special two character tokens: <<, >>, ::, ++, --, ==, +=, -=, \*=, /=

1. Quote class includes methods to handle strings that are quoted and retrieve the unquoted material.

***SemiExpression Package***

******

In the SemiExp class we have a method for each terminator. The terminators for the C# language are

* ;
* {
* }

The SemiExp class methods call getTok() repeatedly until a terminator is identified. Once the terminator is identified it forms a semi expression. The Semi Expression class then forwards the semi expressions for further analysis to the next software. The interface ItokenCollection() contains methods to get the semi expressions so created by the terminators and pass them to the next module of the code analyser. This is the desired output for the lexical analyser.

***Critical Issues***

Performance

If the semiexpression class cannot detect a terminator until the end of the source file the system may run into deadlock. For this purpose in the design we may include a method that tries to catch this error before it occurs.

Token peeking initially may contain files that have a lot of white space or comments this may overload the buffer. The solution to this is to specify the number of memory locations allotted for peeking until we find a character.

Reliability

If the tokeniser may come across characters, which are unknown, they may generate integer values, which will not be identified by the system. For this we may set a predefined set of integer values that form a boundary of accepted characters.

Since the procedure of the SemiExpression class only begins after the entire file has been tokenized we may have to put a limit on the size of the file that the tokeniser can accept.