VOL. 11, NO. 18, SEPTEMBER 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2016 Asian Research Publishing Network (ARPN). All rights reserved.

www.arpnjournals.com

GENERATING UML CLASS DIAGRAM FROM SOURCE CODES USING MULTI-THREADING TECHNIQUE

Saif Khalid[1] and Rosziati Ibrahim[2]
[1]Faculty of Computer Science and Information Technology, Department of Software Engineering, Universiti Tun Hussein Onn Malaysia
[2]Centre for Research and Development, Office of Research Management, Consulting, Commercialization and Innovation, Universiti Tun Hussein Onn Malaysia
E-Mail: saif.uthm@yahoo.com

ABSTRACT

Reverse engineering software is the process of moving back of the Software Development Life Cycle (SDLC) phases by analyzing the software system and then representing it at the higher levels of abstractions. Those processes generate high level information from the implementation phase, which can support the software understanding activities by generating several diagrams and specification documents that describe the implemented software. The UML class diagram represents a valuable source of information even after the delivery of the software. The importance of class diagram comes from its closeness to implementation phase. Class diagram extraction can be done either from software's source code, or from the executable file. This paper proposed approach for extracting a class diagram from the source code. The proposed approach relies on multi-threading technique in the class diagram extraction which is representing the parallel processing. The motivation behind using multi-threading technique is that, it gives an advantage of faster processing to any software because the threads of the program naturally lend themselves to truly concurrent execution. In this paper, a class diagram extraction using multi-threading technique is designed and implemented using the C# programming language. The implemented approach is tested on three case studies that contain several types of entities and relationships between them. Testing results show that the time needed to extract class diagram using multi-threading technique for the tested three cases is less than the time needed in extracting the same class diagram without using multi-threading technique.

Keywords: reverse engineering, multi-threading, C# source code, UML class diagram.

INTRODUCTION

Reverse engineering is the process of discovering the technological principles of a device, object or system through the analysis of its structure, function and operation (Sommerville, 2007). In order to conduct this process, software developers must understand the structure or architecture of the software system. Unfortunately, the documentation of that particular system is often has never been written or the person who had developed the system is no longer employee in the company (Ibrahim and Tiu, 2008). Therefore, explicit knowledge about the particular system could not be provided.

On the other hand, source code, as mentioned in (Doan, 2008), is the most important available source to understand the structure of the system. Therefore, the ability to reuse source code can be economical for software engineers; this is because, if some parts of a new software system can be reused from existing systems, software developer will save a large amount of money and effort in developing it (Doan, 2008).

In order to reuse the source code, software developers must realize the structure and architecture of the software as well as clearly understanding the software's features and functions. UML class diagram describes the structure the software by showing the software's classes, their attributes, operations and the relationships among objects (Nagappan, 2008).

Numerous researchers have developed techniques and tools of reverse engineering from source code to class diagram such as (Aziz *et al*., 2013), where they developed ForUML tool that extracts UML class diagrams from Fortran code. ForUML is able to produce an XMI document that describes the UML Class Diagrams. Another approach is proposed by (Mrinal *et al*., 2013). Where they used reverse engineering to generate UML class diagram from an object oriented system and analysis of its static behavior by considered java programs. Their approach able to sketch a method which determines classes and their attribute, operation and relationship: generalization, aggregation, association and various kind of dependencies in form of a simple class diagram that can be understood by a programmer when inspecting the source code of a given java programs. While (Jain *et al*., 2010) developed a reverse engineering method to automate the extraction of DFDs, CFDs, and class diagrams from any legacy C++ code. The extracted information is classified as structural, behavioural and constraint rules through which such information can be produced. In addition, there are also many tools that were developed for this purpose, such as those mentioned in (Ibrahim and Tiu, 2008), (Sutton and Maletic, 2007), (Vinita *et al*., 2008), (Keschenau, 2006) and (Tonella, 2005). However, none of these researchers have used multi-threading technique to extract UML class diagram from the source code. Therefore, we use multi-threading technique to improve the efficiency of UML class diagram extracted from source code.

The objectives of this paper work are as follows (i) To design an approach that generates class diagram from object-oriented source code using multi-threading technique. (ii) To implement the proposed approach using C# programming language. (iii) To test the proposed approach on C# source code and compare it with

VOL. 11, NO. 18, SEPTEMBER 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2016 Asian Research Publishing Network (ARPN). All rights reserved.

generating a class diagram without using multi-threading technique for its efficiency in terms of time.

The main area of concentration is the part of reverse engineering that is pertaining to generate class diagrams from source code. The reverse engineering concept here is explained in terms of the transformation of object oriented source codes to UML diagrams using the suggested approach. The application scope of this approach will be the C# source code only. The main focus will be on using asynchronous threading. This application accepts codes that are free from any syntax errors, while the parser that will be built according to the suggested approach is limited to extract class diagram but not to compile the source code. The input will be source code of C# language and the output will be a UML class diagram. Four relationships between classes and interfaces will be extracted: generalizations, realizations, association, and dependency.

This proposed approach's aim is to compare the time needed in generating a class diagram with and without using the multi-threading technique; hence, this proposed approach is applied on three case studies to prove its validity. The three case studies are C# programs that contain several code files. Each code file contains a set of classes and interfaces. The time of execution was calculated for each case study and then listed in the testing results table.

MULTI-THREADING TECHNIQUE

A thread is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler (Butenhof, 1997). According to (Justia, 2011), multi-threading is the ability of a program or an operating system process to manage multiple requests by the same user without having to have multiple copies of the programming running on the computer. Multi-threading paradigm has become increasingly popular as efforts to further exploit instruction level parallelism have stalled since the late 1990s (Justia, 2011). This allowed the concept of throughput computing to re-emerge to prominence from the more specialized field of transaction processing.

According to (Barnes *et al*., 2012) the advantage of a multi-threaded program is that it allows the program to operate faster on computer systems. This is because the threads of the program naturally lend themselves to truly concurrent execution.

GENERATING CLASS DIAGRAM FROM SOURCE CODE

Through the literature reviews, it can be observed that there are several stages in generating a class diagram from the source code of any object oriented programming languages. These stages rely mainly on text analysis techniques, hence, generating a class diagram from the source code needs several processing iterations of the source code. The first iteration result will be the names of the code's main classes and interface. Then, in the second iteration, the relationship between all classes and interfaces will be extracted. The final iteration focused on extracting each class' attributes and operations. Finally, after gathering all of this information, a class diagram will be generated and can be presented as a graph.

Normally, every source code is divided into several files, with each file containing one or several classes and interfaces that are related to each other. This is done in object oriented programming to achieve the modularity objective. Based on this distribution, a class diagram extraction is done by individually handling the files that contain a project code. Files processing undergoes three main stages, which are (i) extracting classes and interfaces names, (ii) extracting the relationships between these classes, and (iii) extracting class operations and attributes.

PROPOSED METHODOLOGY

The proposed methodology for extracting class diagram with and without the use of multi-threading technique consists of several stages that are followed in order to achieve the objectives listed. The stages are illustrated in Figure- 2. The first stage is to map code files into tokens, in which code files are selected from the project folder, then, all unnecessary symbols are removed, and finally the results are passed, as tokens to the next stage. After extracting the tokens from the first stage, and prior to extract class diagram relationships, there is a need to extract classes and interface information in order to use them to extract relationships, these information include classes and interfaces information, including the names of classes and interfaces along with classes attributes and operations. The third stage of this proposed methodology consists of two parts which works synchronously; the first part is creating class diagram without the use of multi-threading technique. This part uses the information from the first two stages. While the second part is about generating a class diagram using multi-threading technique. This part also uses the information obtained from the first two stages. The result of this stage is a full class diagram and execution time for each part. The fourth stage of this proposed methodology is about displaying the execution time comparison results. The time of execution is taken from stage 3. Finally, the resulted class diagram from stage 3 will be visualized using graphics library.

VOL. 11, NO. 18, SEPTEMBER 2016 ISSN 1819-6608
ARPN Journal of Engineering and Applied Sciences
©2006-2016 Asian Research Publishing Network (ARPN). All rights reserved.

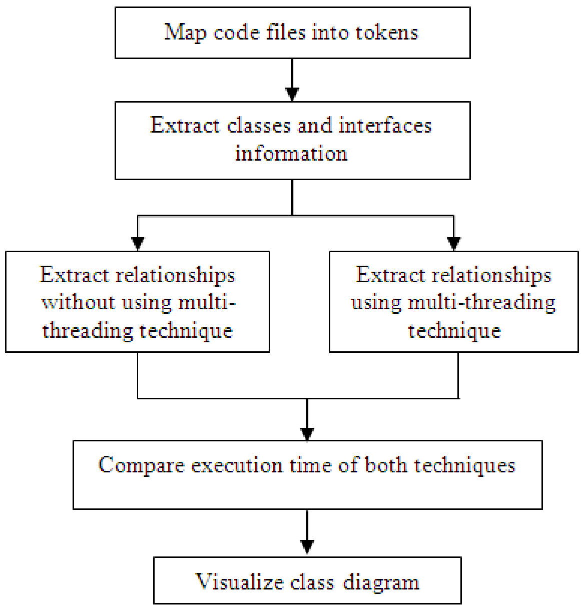
Map code files into tokens
Extract classes and interfaces information
Extract relationships without using multi-threading technique
Extract relationships using multi-threading technique
Compare execution time of both techniques
Visualize class diagram

Figure-1. The proposed methodology stages.

Map code files into tokens

The first stage of the proposed methodology is illustrated in Figure-3. At the beginning of this stage, a C# project folder will be selected. A C# project folder contains several types of files, like ".cs" files which is the source code files, and ".csproj" files among other types of files. The processing is only conducted on ".cs" files because it is the only file type that contains the source code, while other files are generated by C# IDE to run the project. After selecting the code files, unnecessary symbols such as (',' , ';' , ')' , '(' , '<' , '>' , '.' , '+' , '=' , '-' , '/', '*'), are removed. These symbols are not necessary in generating a class diagram or extracting classes and interfaces information. The rest of the code is stored as a list of string, which is called tokens list. This list is passed to stage 2 for further processing.

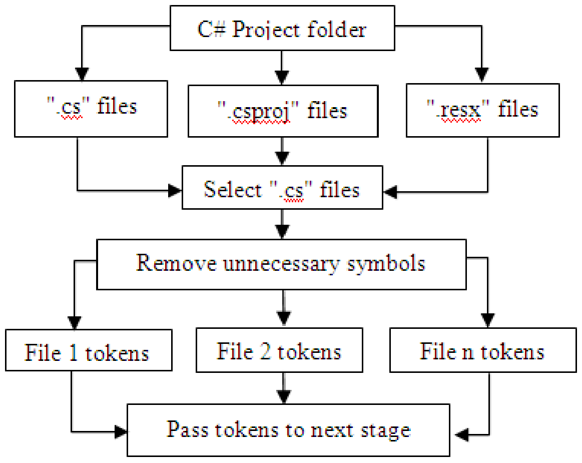

Figure-2. Map code files into tokens.

Extract classes and interfaces information

This stage works on the tokens resulted from stage 1. Figure-3 shows the details of stage 2. The first step is to detect the needed keywords out of the tokens extracted in stage 1. The important keywords are: "class", "interface", "public", "private", and "protected". When detecting the keyword "class", it means that the next word represents a class name, in which the detected class name is stored in class names list. The same case applies to keyword "interface". When detecting one of the following keywords: "public", "private", or "protected", it means that there is an attribute or operation ahead. The detected attribute or operation is stored in a special list with an indicator in its own class. The results of this stage along with the tokens from stage 1 are passed on to both parts of stage 3.

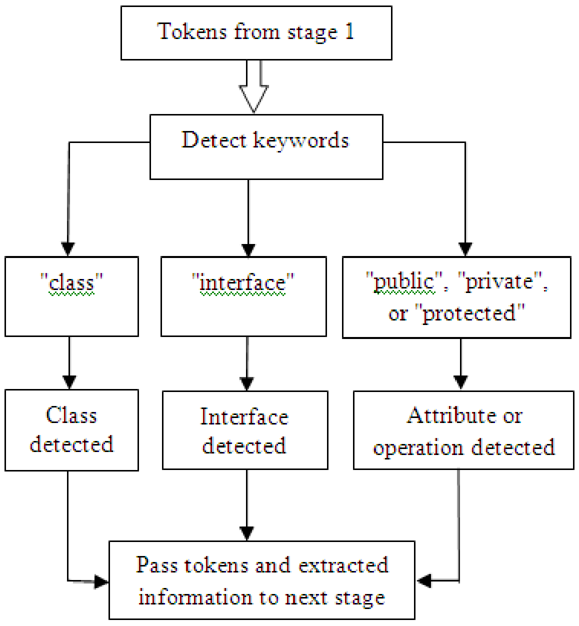
Tokens from stage 1
Detect keywords
"class"
"interface"
"public", "private", or "protected"
Class detected
Interface detected
Attribute or operation detected
Pass tokens and extracted information to next stage

Figure-3. Extract classes and interfaces information.

Attributes in C# starts with one of the following scopes: Public, Private, or Protected. The scope may be followed by the word Static. After that, the type of attribute takes a place. It is then followed by attribute name. Attributes will be represented as:

[Visibility] [static] data_ type attribute_name [=initial_value];

Operations have similar parts to attribute, with the difference being that an operation takes several arguments. Operations will be represented as:

[Visibility] return_ data_ type function_name ([parameter_list])

Visibility of attribute and operations is identified using Table-1 (MSDN, 2014).

Table-1. Visibility types.

Keywords	Representation	Desecration
Public	+	Visible globally
Private	-	Not visible outside
Protecte d	#	Visible to types derived

Identifying relationship

The rules for identifying the relationship types are as follows:

Generalization relationship: Generalization is a type of relationship where all the derived classes are specialization of the base class, and all the base classes are generalizations of derived classes. Generalization can be detected when the parser finds the keyword ":", this mean that the name of the parent class is coming after it and the name of the child class will be before it. Generalization is represented in UML by a solid line from the child class to the parent class, drawn using an unfilled arrowhead.

Dependency relationship: Dependency is a kind of relationship which states that a change in the specification in one class may affect another class that uses it. In the context of class, the dependency relationship can be identified when one class uses another class as an argument in the signature of an operation and if the used class changes, the operation of other class may be affected as well. Dependency is represented in UML by a dotted line with an unfilled solid arrowhead between the classes.

Realization relationship: In a realization relationship, one entity (normally an interface) defines a set of functionalities as a contract and the other entity (normally a class) "realizes" the contract by implementing the functionality defined in the contract. This relation is between a class and an interface. When the parser match keyword ":", this mean that the name of an interface is coming after it. This relationship is the same as a generalization, with the difference being in the parent type. In generalization, the parent is a class, while in realization the parent is an interface. Realization is represented in UML by a dotted line with a filled solid arrowhead.

Association relationship: Association is a type of HAS_A relationship for whole/part relations. Association specifies that the object of one class is connected to the objects of another class. Association is represented in UML by a solid line between the two classes.

The association relationship also supports several adornments like: (i) name, (ii) role name, and (iii) multiplicity. Our tool focuses on multiplicity which is the number of instances of a class. If the declaration is a single object, then the multiplicity would be "0..1". While if a collection of objects is defined in another object, the multiplicity would be "0... *".

Extract relationships without using multi-threading technique

This stage is about extracting class diagram relationship out of the C# source code without using multi-threading technique with details shown in Figure-5. The starting point of this stage is the tokens and information from stage 1 and stage 2. In this stage, code files are processed sequentially. For each file, a sub class diagram is generated. The sub class diagram contains the relationship between the processed file classes and interfaces with the rest of the selected project classes and interfaces. Four types of relationships are considered for each file. Execution time is calculated using a timer which starts at the beginning of this stage and it is stopped at the end of the class diagram generation process. Processing time is taken from this timer after it is stopped. The generated class diagram and calculated time execution are passed on to next stages.

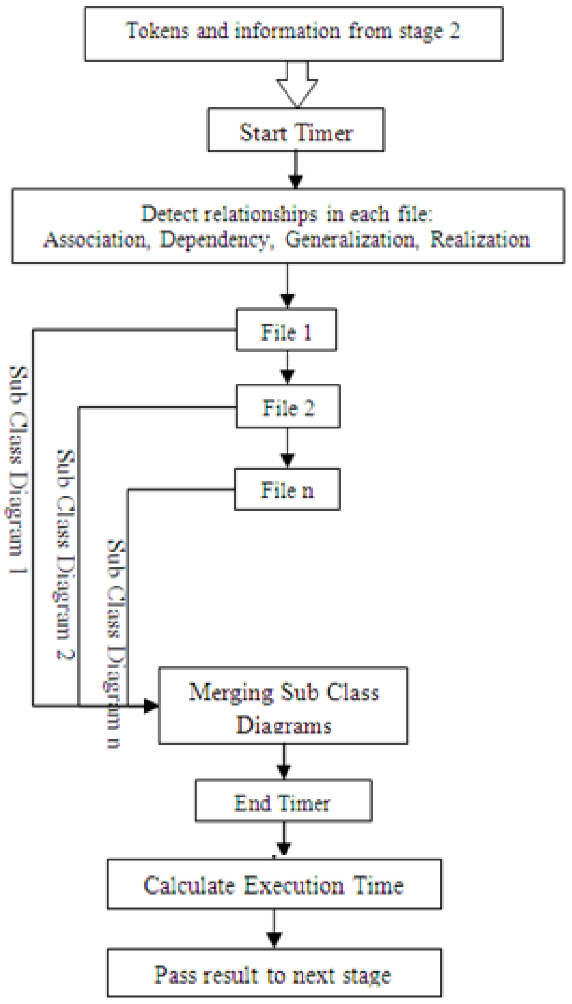
Tokens and information from stage 2
Start Timer
Detect relationships in each file:
Association, Dependency, Generalization, Realization
File 1
File 2
File n
Sub Class Diagram 1
Sub Class Diagram 2
Sub Class Diagram n
Merging Sub Class Diagrams
End Timer
Calculate Execution Time
Pass result to next stage

Figure-4. Extract relationships without using multi-threading technique.

The flowchart in Figure-5 describes class diagram extraction without using the multi-threading technique. The first step is to extract classes and interfaces names from all code files. After that, for each files, the relationship between the classes and interfaces that this file contains and the rest of project classes are extracted. The last step is to extract classes, attributes, and operations. Finally, a local sub-class diagram is generated. After finishing this stage, there will be a set of sub-class diagrams equal to the number of project code files. The next step will be merging all sub-class diagrams in order to generate the whole class diagram. By finishing this step, the class diagram can be visualized.

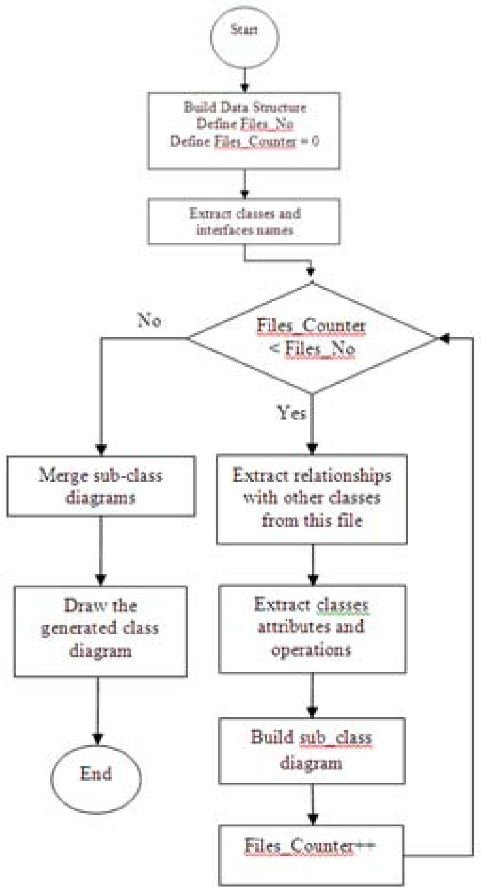
Start
Build Data Structure
Define Files_No
Define Files_Counter = 0
Extract classes and interfaces names
No
Files_Counter < Files_No
Yes
Merge sub-class diagrams
Extract relationships with other classes from this file
Draw the generated class diagram
Extract classes attributes and operations
Build sub_class diagram
End
Files_Counter++

Figure-5. Flowchart to extract class diagram without using multi-threading technique.

Extract relationships using multi-threading technique

This stage is similar to stage 3a; the difference is that the file processing in this stage is done in parallel and not sequentially as in stage 3a. The parallel processing of files is carried out using multi-threading technique, which is detailed in Figure-6. For each code file, a thread is generated and assigned to process this file. The processing result is a sub class diagram. When all threads are finished processing, the resulted sub class diagrams are merged to generate a full class diagram which represents the whole project. The processing procedure for each file is the same as in stage 3a, and the resulting class diagram is similar. The only difference is the total execution time for this stage is calculated using a timer which starts when the first thread starts, and stops when the last thread finishes processing. This total execution time and the class diagram are then passed on to the next stages.

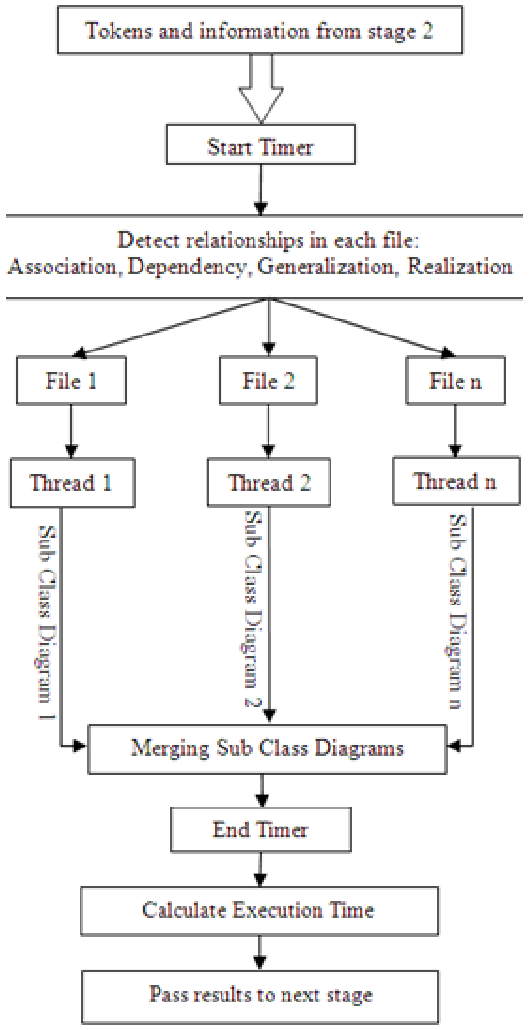
Tokens and information from stage 2
Start Timer
Detect relationships in each file:
Association, Dependency, Generalization, Realization
File 1
File 2
File n
Thread 1
Thread 2
Thread n
Sub Class Diagram 1
Sub Class Diagram 2
Sub Class Diagram n
Merging Sub Class Diagrams
End Timer
Calculate Execution Time
Pass results to next stage

Figure-6. Extract relationships using multi-threading technique.

It can be noticed from this stage that parallel processing can take place in only one part. The first round of code file processing should be done without the multi-threading technique to avoid classes and interfaces name duplication. So the multi-threading technique can be used only in classes and interfaces relationship extraction, which is the second round of code file processing.

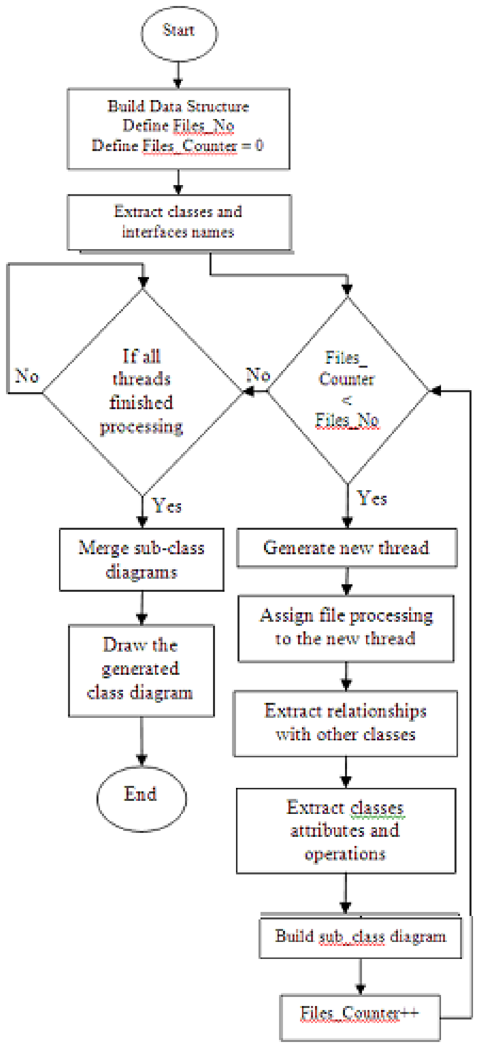
Start
Build Data Structure
Define Files_No
Define Files_Counter = 0
Extract classes and interfaces names
If all threads finished processing
No
No
Files_ Counter < Files_No
Yes
Yes
Merge sub-class diagrams
Generate new thread
Assign file processing to the new thread
Draw the generated class diagram
Extract relationships with other classes
End
Extract classes attributes and operations
Build sub_class diagram
Files_Counter++

Figure-7. Flowchart to extract class diagram using multi-threading technique.

In the flowchart shown in Figure-7, the first step is extraction of classes and interfaces names of code files. After that, a new thread is generated with file processing assigned to this new thread for each code file. Thus is repeated until all code files are assigned to the threads. Then, the program should wait until all threads have completed processing the code file and all sub-class diagrams are ready. The final step will be merging all sub-class diagrams to generate the class diagram.

Compare the execution time of both techniques

This stage uses the time calculated in stage 3a and stage 3b. The main aim of this stage is to compare the execution time of class diagram generation with and without using multi-threading technique. In order to measure the time needed in code files processing, a component called StartWatch is used to start implementing detect relationship function as start timer. After that, the component StopWatch is called to stop the timer and calculate the time taken. These components are provided by the C# programming language. The details of this stage are shown in Figure-8.

The time needed to extract relationships of class diagram without using multi-threading is calculated at the beginning of stage 3a, where a timer is started and stopped at the end of the class diagram generation process. Processing time is taken from this timer after it is stopped. The generated class diagram and calculated time execution are then passed on to the next stages.

On the other hand, the time needed to extract relationships of class diagram using multi-threading is calculated at the beginning of stage 3b, where the execution time is calculated using a timer which starts when the first thread begins, and stops when the last thread finishes processing. Total execution time and the class diagram are passed on to the next stages. The final step is to compare the times that are calculated in the two steps above and have the results displayed.

Visualize class diagram

This stage is the last stage in this proposed methodology. The main aim here is to visualize the class diagram generated in stage 3a and stage 3b. Since both class diagrams which were generated in stage 3 are identical, this stage only utilizes one of them. Class diagram is visualized by using C# graphics library.

COMPARE BOTH TECHNIQUES

Both techniques, which were presented in Figure-5 and Figure-7 pertaining to class diagram extraction from OOP source code. These two techniques have the same core idea, which is analyzing a project code file by file. This analysis result will become a data structure which contains: class names, class attributes, class operations, interface names, relationship between class and interface. The last step in each technique is to visualize the result obtained in the analysis phase.

The main difference between both techniques is in the relationship extraction step; where in one of them extraction is done by processing one file after another, which follows the sequential approach. While in the second, file processing is done using the multi-threading technique in which files are processed using separate threads of the process, following the parallel approach.

TOOL IMPLEMENTATION

There are two interfaces for this tool. The main interface is shown in Figure-8. This interface contains several buttons and results view labels. A user needs to select a project folder at the beginning. After that, the Generate Class Diagram button will be activated. Pressing the Generate Class Diagram button will invoke the generating class diagram with and without using the multi-threading technique. After finishing this step, the View Class Diagram button will be activated. Pressing this button will open a new window which contains the class diagram as a visualized class diagram.

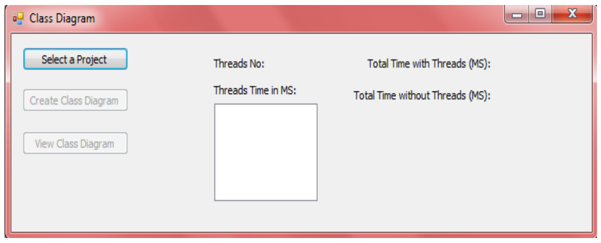

Figure-8. Tool main interface.

TESTING RESULTS

The developed approach and program testing will be done on 3 case studies generated just for the purpose of testing. The testing cases consist of several C# source code files. In each file there are some class, interface, relationship between them, class attributes, and class operations. The implemented program contains two interfaces: one shows the execution time with and without multi-threading, the other one is for the class diagram drawing. The testing procedure that is followed in this research is based on the generating class diagram for the three case studies. Testing results are shown in Table-2.

Table-2. Testing results (Time is in ms).

Case study	Time using multi-threading	Time without using multi-threading
1	1.2987	5.8305
2	4.9049	9.4205
3	2.2108	7.6495

As seen from the presented results, the time needed to extract the class diagram from the source code of the first case study using the multi-threading technique is much less than the class diagram extraction from the same source code without using the multi-threading technique. The same conclusion applies for the other two case studies.

From the presented results, it can be said that using multi-threading in class diagram extraction is more efficient than extracting class diagram without using this technique. And simply the bigger the project code the larger the difference in needed time to extract the class diagram.

The case study shown in Figure-9 represents a simple banking system. This case study consists of 6 classes with 4 relations. The class Bank is inherited from Company, so the relation between them is generalization relation. There is another generalization relation which is between class customer and person. Class Bank is associated with class Bank Account, so the relation between them is association as shown. Another association relation appears between class Bank Account and class Customer.

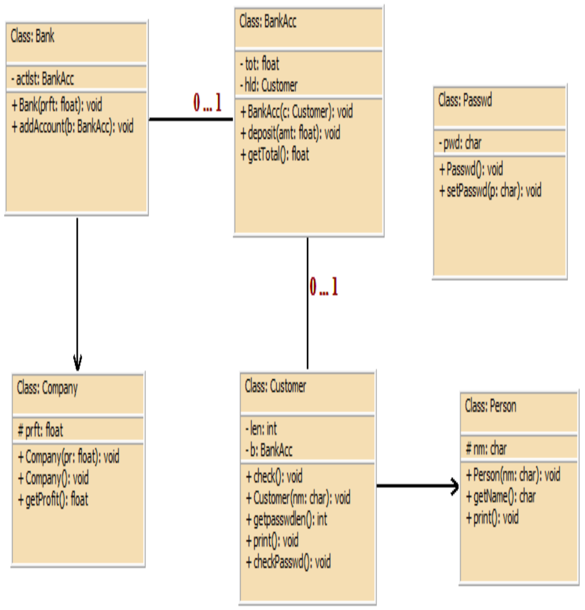

Figure-9. Case study implementation.

CONCLUSIONS AND FUTURE WORK

This paper contained the proposed approaches of extracting UML class diagram from source codes using multi-threading technique and without multi-threading technique along with related testing for them. The testing results have shown that using multi-threading technique in class diagram extraction is more efficient in the aspect of time than without using multi-threading technique, which is common with current tools.

There are a few research recommendations can be considered for future works. Those recommendations can be considered in order to enhance and improve the functionalities of the current approach. One is develop an approach in extracting class diagram using the multi-threading technique. Unlike the current approach that is developed in this research, which aims to compare class diagram extraction with and without using the multi-threading technique, the future developed approach can aim to extract class diagram from source codes using the multi-threading technique. In addition, comparison with other approaches built within other programming languages can also be conducted.

Furthermore, this current approach can be applied to other object oriented programming languages (OOP). For this research scope, this approach is to be implemented in the C# language. On the other hand, the proposed approach can also be extended to work on other programming languages other than C#, like, Java.

The proposed approach presented the limited relationships of a class diagram. As mentioned in this paper, the aim of this research was to compare the class diagram extraction efficiency with and without multi-threading. So, it can be observed that this approach can enhance class diagram extraction by including all of the relationships of the class diagram with all possible rules of code writing.

REFERENCES

Aziz, N., Karla Morris and Salvatore Filippone. 2013. Extracting UML class diagrams from object-oriented Fortran: ForUML. Proceedings of the 1st International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE '13). ACM, pp 9-16.

Barnes, A., Ryan Fernando, Kasuni Mettananda and Roshan Ragel. 2012. Improving the Throughput of the AES Algorithm with Multi core. Processors. Proceeding of the 7th International Conference on Industrial and Information Systems (ICIIS). IEEE. pp. 1-6.

Butenhof, D.R. 1997. Programming with POSIX Threads. 1st edition. Boston, USA: Addison-Wesley.

Doan, T. 2008. An evaluation of four reverse engineering tools for C++ applications. University of Tampere: Master's Dissertation.

Ibrahim, R. and Yong, T.K. 2008. ReSeT: Reverse Engineering System Requirements Tool. World Academy of Science. 14(4), pp. 238-241.

Jain, A., Sooner, S, and Holkar, A. 2010. Reverse engineering: Extracting information from C++ code. Proceedings of the 2nd International Conference on Software Technology and Engineering (ICSTE). San Juan: IEEE. pp. 154 -158.

Justia. 2011. Obfuscated hardware multi-threading. http://patents.justia.com/patent/8621186.

Keschenau, M. 2006. Reverse Engineering of UML Specifications from Java Programs. Proceedings of the 19th companion annual conference on Object-oriented programming systems (OOPSLA '04). ACM, pp. 326-327.

Nagappan, S.D. 2008. A reverse engineering uml modeling tool. University of Malaya: Master's Dissertation.

Mrinal Kanti Sarkar, Trijit Chatterjee, Dipta Mukherjee. 2013. Reverse Engineering: An Analysis of Static Behaviors of Object Oriented Programs by Extracting UML Class Diagram. International Journal of Advanced Computer Research, 12(3). pp. 135-141.

MSDN. 2014. Microsoft Corporation. http://msdn.microsoft.com/en-us/default.aspx.

Sommerville I. 2007. Software Engineering, 8th Edition, Addison Wesley, England.

Sutton, A. and Maletic, J.I. 2007. Recovering UML class models from C++: A detailed explanation. Information and Software Technology. 49(3), pp 212-229.

Tonella, P. 2005. Reverse Engineering of Object Oriented Code. Proceeding of the. 27th International Conference on Software Engineering ICSE 2005.IEEE .pp. 724-725.

Vinita, Amita Jain, Devendra K. Tayal. 2008. On reverse engineering an object-oriented code into UML class diagrams incorporating extensible mechanisms. ACM Sigsoft Software Engineering Notes, 33(5). pp. 1-9.