GENERATING UML CLASS DIAGRAM FROM SOURCE CODES USING MULTITHREADING TECHNIQUE

SAIF KHALID ABDULLAH

A dissertation submitted in partial fulfilment of the requirement for the award of the Degree of Master of Computer Science (Software Engineering)

Faculty of Computer Science and Information Technology Universiti Tun Hussein Onn Malaysia

APRIL 2015
ABSTRACT

The traditional Software Development Life Cycle (SDLC) often includes four phases: analysis, design, implementation, and testing. Reverse engineering is the process of moving back those phases by analyzing the software system and then representing it at the higher levels of abstractions. The reverse engineering software process generates high level information from the implementation phase. This information includes generating several diagrams and specification documents that describe the implemented software. The UML class diagram represent a valuable source of information even after the delivery of the software. Class diagram extraction can be done either from software’s source code, or from the executable file. In the case of source code, a review of the current tools shows that many researchers have been extracting the UML class diagram from an object-oriented source code based on the sequential processing approach. In this research, a proposed approach for extracting a class diagram from the source code is presented. 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 research, 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.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>TITLE</th>
<th>i</th>
</tr>
</thead>
<tbody>
<tr>
<td>DECLARATION</td>
<td>ii</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>iii</td>
</tr>
<tr>
<td>ACKNOWLEDGMENT</td>
<td>iv</td>
</tr>
<tr>
<td>ABSTRACT</td>
<td>v</td>
</tr>
<tr>
<td>ABSTRAK</td>
<td>vi</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>vii</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>x</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>xii</td>
</tr>
<tr>
<td>LIST OF APPENDICES</td>
<td>xiii</td>
</tr>
</tbody>
</table>

## CHAPTER 1 INTRODUCTION

1.1 Background of research  1
1.2 Research motivations   3
1.3 Objectives             4
1.4 Scope of research      4
1.5 Expected outcomes      5
1.6 Chapter summary        6
1.7 Dissertation outline   6

## CHAPTER 2 LITERATURE REVIEW

2.1 Introduction            8
2.2 Reverse engineering derivatives  8
   2.2.1 Reverse engineering  9
   2.2.2 Restructuring       9
   2.2.3 Reengineering       10
2.3 Reverse engineering activities 10
   2.3.1 Data gathering      10
   2.3.2 Knowledge management 12
CHAPTER 3 RESEARCH METHODOLOGY

3.1 Introduction 31
3.2 Proposed methodology 31
3.3 Proposed methodology stages 32
  3.3.1 Stage 1: Map code files into tokens 33
  3.3.2 Stage 2: Extract classes and interfaces information 34
  3.3.3 Stage 3a: Extract relationships without using multi-threading technique 35
  3.3.4 Stage 3b: Extract relationships using multi-threading technique 39
  3.3.5 Stage 4: Compare execution time of both techniques 41
  3.3.6 Stage 5: Visualize class diagram 42

3.8 Related work 25
  3.8.1 ForUML: class diagram extraction from fortran source code 25
  3.8.2 Extracting class diagram from c++ code 27
  3.8.3 ReSeT: Reverse Engineering System Requirements Tool 28
  3.8.4 Related work summary 30

3.9 Chapter summary 30
### CHAPTER 4 DESIGN AND IMPLEMENTATION

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Introduction</td>
<td>44</td>
</tr>
<tr>
<td>4.2</td>
<td>The general flowchart of generating class diagram from source code</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>The proposed flowchart of generating class diagram from source code</td>
<td>45</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Without using Multi-threading technique</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2</td>
<td>Using Multi-threading technique</td>
<td>48</td>
</tr>
<tr>
<td>4.4</td>
<td>Tool implementation</td>
<td>50</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Build data structure</td>
<td>50</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Classes and interfaces names extraction</td>
<td>51</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Relationships extraction without using multi-threading technique</td>
<td>52</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Relationships extraction using multi-threading technique</td>
<td>53</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Relationships extraction function</td>
<td>54</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Classes attributes and operations extraction</td>
<td>56</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Processing time calculation</td>
<td>57</td>
</tr>
<tr>
<td>4.4.7</td>
<td>Sub-class diagrams merging</td>
<td>57</td>
</tr>
<tr>
<td>4.4.8</td>
<td>Interface implementation</td>
<td>59</td>
</tr>
<tr>
<td>4.5</td>
<td>Testing results</td>
<td>59</td>
</tr>
<tr>
<td>4.6</td>
<td>Chapter summary</td>
<td>66</td>
</tr>
</tbody>
</table>

### CHAPTER 5 CONCLUSION AND FUTURE WORK

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Introduction</td>
<td>67</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of the research</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>Future work</td>
<td>69</td>
</tr>
</tbody>
</table>

### REFERENCES

<table>
<thead>
<tr>
<th>Reference</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>69</td>
</tr>
</tbody>
</table>

### APPENDIX

<table>
<thead>
<tr>
<th>Appendix</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>73</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

2.1 Reverse engineering derivatives (Nelson, 1996) 9
2.2 Class Diagram of monitoring system of postgraduate student (Ibrahim & Tiu, 2008) 14
2.3 Class diagram example of association between two classes (Wiki, 2014) 15
2.4 Class diagram showing Aggregation between two classes (Wiki, 2014) 16
2.5 Class diagram showing Composition between two classes at the top and Aggregation between two classes at bottom (Wiki, 2014) 17
2.6 Class diagram showing generalization between one superclass and two subclasses (Wiki, 2014) 18
2.7 Class diagram showing realization relationship (Nishadha, 2012) 19
2.8 Dependency relationship example (Wiki, 2014) 20
2.9 standard steps of Parallel approach (Mivule, 2011) 22
2.10 standard steps of Sequential approach (Mivule, 2011) 22
2.11 sequential and parallel execution example (Xmipp, 2010) 22
2.12 A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow), (Wiki, 2007) 25
2.13 Fortran model 26
2.14 Transformation process 26
2.15 ForUML class diagram view 27
2.16 An example of C++ code 28
2.17 Extracted tokens 29
2.18 Class diagram extracted out of C++ code 29
Proposed Methodology

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

General flowchart of generating class diagram

flowchart to extract class diagram without using multi-threading technique

flowchart to extract class diagram using multi-threading technique

Data Structure

Classes and interfaces information extraction

Relationships extraction without using multi-threading

Relationships extraction using multi-threading

Generalization and realization relationships extraction

Association relationships extraction

Dependency relationships extraction

Visibility representation of attributes and operations

Processing time calculation

Adding classes and interfaces to the class diagram

Adding relationships to the class diagram

Tool Main interface

Case study 1 class diagram

Case study 2 class diagram

Case study 3 class diagram
# LIST OF TABLES

<table>
<thead>
<tr>
<th>2.1</th>
<th>UML 2.4.1 defines 14 diagrams, (OMG, 2008)</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.2</td>
<td>Multiplicity Indicators (Agile, 2014)</td>
<td>17</td>
</tr>
<tr>
<td>2.3</td>
<td>Related work summary</td>
<td>30</td>
</tr>
<tr>
<td>3.1</td>
<td>Visibility types (MSDN, 2014)</td>
<td>34</td>
</tr>
<tr>
<td>4.1</td>
<td>Testing Results (Time is in ms)</td>
<td>60</td>
</tr>
</tbody>
</table>
# LIST OF APPENDICES

<table>
<thead>
<tr>
<th>APPENDIX</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Case study 1</td>
<td>73</td>
</tr>
<tr>
<td>B</td>
<td>Case study 2</td>
<td>75</td>
</tr>
<tr>
<td>C</td>
<td>Case study 3</td>
<td>78</td>
</tr>
</tbody>
</table>
CHAPTER 1

INTRODUCTION

1.1 Background of research

Software maintenance is the last phase in the life cycle of a software development process which often includes the following phases: requirement specification, analysis, design, implementation, testing, deployment and maintenance (Dennis, et al., 2006). However, this phase plays an important role because software maintenance activities ensure that a software system still works well without errors in new environments after it is released. According to Doan (2008), common maintenance activities include fixing bugs, adapting the system to a new environment, adding new features to the system to satisfy new requirements from the client, and updating documentation for the system. In order to do these tasks, software maintainers must understand the structure or architecture of the system. However, it is a hard task for them in case some changes happened in the structure of the system, which makes the system different from its original version. In some cases, system documentation is not up-to-date so it cannot provide explicit knowledge about the system. Source code is the most important available source to understand the structure of the system (Doan, 2008).

In the case of source code reuse, 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 system and then understand clearly their features and functions.
Reverse engineering tools are very useful in the above cases. The term reverse engineering was defined by Chikofsky & Cross (1990) as the process of analyzing a subject system to (i) identify the system’s components and their interrelationships and (ii) generate representations of the system in another form or at a higher level of abstraction. It is an activity that takes place frequently, for example, when understanding a system before making a change; when migrating a software system from one platform to another; when transforming source code from one object model to another; and when refactoring a set of classes to satisfy new requirements (Canfora & Penta, 2007).

According to Tonella & Potrich (2001) reverse engineering tools provide useful high level information about the system being maintained. Their output diagrams can support the program in understanding activities, drive refactoring, and restructuring interventions, and also employed to assess the traceability of the design into the code. Therefore, it is important that the representations recovered from the code to be accurate, i.e., exploit all static information present in the code in order to reverse engineer entities and relations.

Enhancements can be easily done if the modeling diagrams are done during the initial diagram generation. Unfortunately, when the software is delivered, design diagrams are not packed with it. There are a large number of tools that are incorporated with reverse engineering modules (Nagappan, 2008). The most commonly implemented reverse engineering module is the reverse engineering of the codes (Nagappan, 2008).

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 more 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.
1.2 Research motivations

Software engineering has undergone a paradigm shift as the size of the software systems deployed increased dramatically and businesses began to rely increasingly on computers and information systems (Ramasubbu, 2001). A substantial portion of the software development effort is spent on maintaining existing systems rather than developing new ones (Rugaber, 1992). An estimated 50% to 80% of the time and material involved in software development is devoted to maintenance of existing code (Boehm, 1991). Crucial to the maintenance of existing systems is the task of program comprehension, an emerging area in software engineering. About 47% of the time is spent on enhancements to existing programs and 62% of that spent on program corrections involve program comprehension tasks like reading the documentation, scanning the source code, and understanding the changes to be made (Fjeldstad & Hamlen, 2001).

Software development as mentioned above is a growing field. However, developing software from scratch is no longer a situation faced by the developer. The challenge faced currently is how to use the minimum information about existing software and further enhance it to become a powerful tool (Nagappan, 2008).

Since the paradigm shift, developers who have embarked on the idea of enhancing any software are often faced with the problem of how to gather the initial requirements on which the existing software was built upon (Nagappan, 2008). Documentation that is often used to aid this process would be the user manual. However, user manuals only show how to use the system for the system user and not from the developer’s perspective. Design documentations are often not enclosed together with the software due to security reasons. UML models are used to document user requirements and design documentation. One of the most important models used is the class diagram. Class diagram describes the structure of a system by showing the system’s classes, their attributes, operations (or methods), and the relationships among objects (Nagappan, 2008).

According to Barnes, et al., (2012) the advantage of a multithreaded 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.
Many researchers have developed techniques and tools of reverse engineering from source code to class diagram such as (Ibrahim, R. & Yong, T.K., 2008), where they developed ReSet tool which is implemented using C++ programming language. The main objectives of developing this tool is being able to detect the necessary tokens from the syntax of the program source codes and generate the class diagram automatically based on the detected tokens. Another tool named ForUML is proposed by Aziz, et al., (2013). ForUML is a tool that extracts UML class diagrams from Fortran code. ForUML can produce an XMI document that describes the UML Class Diagrams. 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, behavioral and constraint rules through which such information can be produced. And also many tools are developed such as in (Matzko, et al., 2002, Sutton & Maletic, 2007, Keschenau, 2006, Tonella, 2005). However, none of these researches have used multi-threading technique to extract UML class diagram from the source code. Therefore, this research uses multi-threading technique to improve the efficiency of UML class diagram extracted from source code.

1.3 Objectives

The objectives of this research work are as follows:

- To design an approach that generates class diagram from object-oriented source code using multi-threading technique.
- To implement the proposed approach using C# programming language.
- To test the proposed approach on C# source code and compare it with generating a class diagram without using multi-threading technique for its efficiency in terms of time.

1.4 Scope of research

The main area of concentration in this research is the part on reverse engineering that is pertaining to generating class diagram from source code. The reverse engineering
concept here is explained in terms of 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. Asynchronous concept in C# programming language explained the running of two or more operations in different contexts (thread) so they can run concurrently and do not block each other (Mazhou, 2010). The application accepts codes that are free from any syntax errors. The parser that will be built according to the suggested approach is limited only to extracting class diagram, not compiling 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. The proposed approach’s aim is to compare the time needed in generating a class diagram with and without using the multi-threading technique. The proposed approach is applied on three case studies in order 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 is calculated for each case study and then listed in the testing results table.

1.5 Expected outcomes

The main aim of this research is to prove that using multi-threading technique in generating class diagram from source code is more efficient than generating it without using the multi-threading technique. The basic criterion of comparison is time. A tool will be developed to compare the time needed in generating class diagram with and without using multi-threading technique. This research outcome will be a tool that generates a class diagram from source code in two ways: i) Using multi-threading technique, ii) Without using multi-threading technique. The developed tool generates a class diagram that contains the following items: classes, interfaces, relationships, class attributes, and class operations. The developed tool still needs some enhancements in order to be able to extract class diagram using all possible rules of code writing.
1.6 Chapter summary

In this chapter the overall aim of the research which is generating UML class diagram from an implementation phase using reverse engineering is explained. Further illustrate on how the aims can be achieved, objectives are identified. The scope and the limitation in this chapter clearly narrow the broad area of research. This chapter also gives an introduction and a brief overview of the reverse engineering concept and its advantages. Followed by the expected outcomes and the organization of the dissertation. The next chapter will cover the literatures related to this research.

1.7 Dissertation outline

The rest of this dissertation is organized into the following chapters. Chapter 2 presents a review of several aspects that are related to this research, starting from general ideas of reverse engineering to UML and class diagram. After that, a description of C# programming language is introduced. This is followed by a description of multi-threading technique and how to apply it practically. Finally, some of the related works of this research are presented. Chapter 3 contains a description of the proposed methodology that will be followed and used in order to achieve this research objective. This chapter starts with an overall view of the proposed methodology. After that, a description of the methodology in brief details are presented. The methodology is composed of a set of vital steps. Each step takes the result of the previous step and provides a new input for the next step. While in Chapter 4, design and implementation phases of this research are introduced. The design part will be about how to generate a class diagram in general. Then the proposed work is presented. The proposed work consists of mainly two parts: generating a class diagram from source code using multi-threading technique, and generating a class diagram without using multi-threading technique. After design, the last two parts of this research are presented, namely tool implementation and testing. In the first section which is implementation, a detailed description of the proposed approach is to be presented. Implementation explanation contains also some important parts of the written code. The second section will be about tool testing.
Testing is done on three case studies that will be explained in the testing section. The main concentration in testing will be the time of execution. This means processing time that is needed to generate a class diagram with and without using multi-threading technique. Chapter 5 provides a summary of this research, which will be presented along with possible future development.
CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

This chapter covers the literatures related to this research. First of all, an overview of reverse engineering and its derivatives with its main activities are presented. After that, the description of UML as a prominent modelling language is given. Then, the class diagram and its relationship types are discussed. Next, an overview of C# language is given. This is followed by a description of multi-threading technique and how to apply it practically. After that, some of the related works of this research are presented. Finally, summarizes the topics discussed in this chapter.

2.2 Reverse engineering derivatives

In order to understand clearly about reverse engineering, one must distinguish it from other terms such as restructuring and reengineering. These processes are described in Figure 2.1. In the following sections, a description of each term is presented.
2.2.1 Reverse engineering

Reverse engineering in software engineering is the opposite of forward engineering, which is offered to indicate a traditional software development process (Nelson, 1996). The traditional software development often includes four phases: analysis, design, implementation, and testing. Through those phases, software is developed from the high level of abstraction (architecture) to the lowest level of abstraction (source code). Therefore, reverse engineering is the process which analyzes software system and then represents it at the higher levels of abstractions. The following definitions which is given by Chikofsky & Cross (1990) is widely used: "Reverse engineering is the process of analyzing a subject system to identify the system’s components and their interrelationships."

2.2.2 Restructuring

Restructuring is the transformation from one representation form to another form within the same abstraction level (Chikofsky & Cross 1990). An example would be to modify the source code in order to make the structure of the source code more
clear. This process only takes place in one abstraction level and its result is the representation of the system in another form depending on the purpose intended by the software engineers, but still at the same abstraction level, while reverse engineering deals with many abstraction levels and its result is the representation of the system at a higher level of abstraction. In addition, restructuring generates changes in the structure of the system, while reverse engineering only examines the structure of the system and does not make any changes in the system.

2.2.3 Reengineering

Reengineering is the examination, alteration, and modification of the system in order to regenerate a new system with new functions in another representation form (Chikofsky & Cross 1990). This term is wider than the reverse engineering term because it often includes both reverse engineering and forward engineering. The first phase in the reengineering process is using reverse engineering to understand the structure of the old system and represent it at a higher level of abstraction. At that time, some changes were generated at any level of abstraction. The second phase is developing the new system based on the new requirements or functions which have just been recently generated. This phase follows the steps in forward engineering. Hence, reengineering generates a new system with different features and functionalities from an old system, while reverse engineering does not make any changes in the features and functionalities of the system. Reverse engineering is a process of examination, not a process of replication.

2.3 Reverse engineering activities

According to Tilley (1998), the reverse engineering process includes three main activities: data gathering, knowledge management, and information exploration.

2.3.1 Data gathering

One cannot understand about the structure of a software system at higher levels of abstraction without having the necessary information pertaining to the subject.
Therefore, data gathering is often the first step, where several types of data about the system are gathered such as the source code, comments in source code, documentation about the system, and experts’ comments. Three techniques of data gathering which are widely used are: system examination, document scanning, and experience capture (Tilley, 1998).

2.3.1.1 System examination

System examination is often classified into two constricting ways: static examination and dynamic examination. Static examination concentrates on analyzing the source code. A source code parser is often used to analyze the source code and then transfers it to abstract syntax trees (Bellay & Gall, 1998). In contrast, the dynamic examination focuses on the executing system. It is useful for understanding component-based systems in which the static examination cannot apply because components do not come with the source code. Analyzing systems when they are running helps us to have the knowledge about the interactions between components in the system, types of messages and protocols used, and the external recourses used by the system (Tilley, 1998).

2.3.1.2 Documents scanning

Document scanning is the process of gathering documents, another type of information about the system. For example, comments in the source code are useful sources for understanding the system. However, automatic analysis of the comments is more difficult as they may be isolated in the source code or they do not provide explicit information about the source code when they are not updated. Therefore, comments are often analyzed manually by the experts (Tilley, 1998).

2.3.1.3 Experience capturing

Experience capturing is the approach to obtain knowledge about the system by interviewing the people who developed the system. The knowledge is very useful in
understanding the system. However, it is difficult to find out those that developed the system (Tilley, 1998).

2.3.2 Knowledge management

Knowledge management in reverse engineering is used to structure gathered data into a conceptual model of the application domain called a domain model. It includes three main steps namely knowledge organization, knowledge discovery, and knowledge evolution (Tilley, 1998).

2.3.3 Information exploration

According to Tilley (1998), the majority of program understanding takes place during information exploration, and it is arguably the most important of the three canonical reverse engineering activities. Data gathering is required to begin the reverse-engineering process. Knowledge management is needed to structure the data into a conceptual model of the application domain. But the key to increased comprehension is exploration because it facilitates the iterative refinement of the hypotheses. The process of information exploration includes three activities: navigation, analysis, and presentation (Tilley, 1998).

2.4 Unified Modelling Language (UML)

UML is defined by (OMG, 2008) as: "a graphical language for visualizing, specifying, constructing and documenting the artifacts of software systems". UML was originally derived from object modeling languages of three leading object-oriented methods: Booch, Object Modeling Technique (OMT), and Object-Oriented Software Engineering (OOSE). It is more compatible to be used to model object-oriented software systems.

Object Management Group (OMG) has approved UML in November 1997 as the standard notation for object-oriented analysis and design, and it became the industry standard for modeling objects and components. At the end of 2000, the OMG has issued a Request For Information (RFI) with regard to UML 2.0.
2.4.1 UML model

The main purpose of UML is to provide a common vocabulary of object-oriented terms and diagramming techniques that enable developers to model any system. It is clear that UML has gained widespread acceptance as a notation for the analysis and design of software systems. According to Vidgen (2003), UML modeling helps in having a better understanding about a system. Currently, UML 2.4.1 specifies fourteen (14) UML diagrams (Ibrahim, N., 2013). They can be used to describe different views of a system. Structural view is specified by structure diagrams such as profile diagram, class diagram, composite structure diagram, component diagram, deployment diagram, object diagram, and package diagram, while the behavioral view is specified by behavior diagrams such as activity diagram, sequence diagram, communication diagram, interaction overview diagram, timing diagram, use case diagram, and state machine diagram (Ibrahim, N., 2013). Among the diagrams, UML class diagram, use case diagram, sequence diagram, and activity diagrams are the most frequently used diagrams by the UML practitioners (Dobing & Parsons, 2008, Grossman, et al., 2005). Each of the UML diagram is used to describe various aspects of a system. For example, use case diagram is used to highlight the main functions of a system and the roles that interact to it, while activity diagram is used to model the scenario of use cases in terms of dynamic aspect of a society of objects. On the other hand, sequence diagram is modeled to show the communication between objects in terms of sequence of messages, while the class diagram is to show the classes of the objects in term of their attributes, methods, and relationship with other classes. Table 2.1 presents the types of UML diagrams.

Table 2.1: UML 2.4.1 defines 14 diagrams (OMG, 2008)

<table>
<thead>
<tr>
<th>No</th>
<th>Types of diagrams</th>
<th>No</th>
<th>Types of diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Package diagram</td>
<td>8</td>
<td>State Machine diagram</td>
</tr>
<tr>
<td>2</td>
<td>Class or Structural diagram</td>
<td>9</td>
<td>Communication diagram</td>
</tr>
<tr>
<td>3</td>
<td>Object diagram</td>
<td>10</td>
<td>Sequence diagram</td>
</tr>
<tr>
<td>4</td>
<td>Component diagram</td>
<td>11</td>
<td>Timing diagram</td>
</tr>
<tr>
<td>5</td>
<td>Deployment diagram</td>
<td>12</td>
<td>Interaction Overview</td>
</tr>
<tr>
<td>6</td>
<td>Use Case Diagram</td>
<td>13</td>
<td>Profile diagram</td>
</tr>
<tr>
<td>7</td>
<td>Activity diagram</td>
<td>14</td>
<td>Composite structure diagram</td>
</tr>
</tbody>
</table>
2.4.2 Overview of class diagram

According to Matzko, et al. (2002), a class is represented as a box with three vertical sections. The top section shows the name of the class. The middle section displays the variables belonging to the class, with symbols representing the visibility (public, protected, or private) and properties (constant or static). The bottom section contains the member functions of the class. Each method has a name, signature, and properties. Relationship for each of the class will then be described by connection links (Miller, 2003). This description is illustrated in Figure 2.2.

![Class Diagram of monitoring system of postgraduate student (Ibrahim & Tiu, 2008)](image)

Figure 2.2: Class Diagram of monitoring system of postgraduate student (Ibrahim & Tiu, 2008)

According to Vidgen (2003), a method also known as operation, is a function or a service that is provided by all of the instances of a class. There are basically three types of methods, namely constructor, query, and update. Attribute is a named property of a class that describes a range of values that instances of the property may hold (Vidgen, 2003).
Relationship for each of the class will then be described by connection links (Miller, 2003). There are several relationships to be discussed which are association, multiplicity, generalization, dependency, realization, and aggregation. Each relationship is represented in the diagram by a different type of arrow. The following sections illustrate all types of relations with its visualization.

### 2.4.2.1 Association relationship

According to Matzko, *et al.* (2002), an association is a structural relationship that describes a set of links, where a link is a connection among objects. Class A has an association with class B if class A has a data member of type B. Associations are often adorned with numbers or symbols representing the multiplicity of the relationship, or how many objects of one class from the other is used. Figure 2.3 shows an example of association relationship.

![Class diagram example of association between two classes (Wiki, 2014)](image)

Figure 2.3: Class diagram example of association between two classes (Wiki, 2014)

The association relationship is used as a measure of closeness among the system files. Also, it can be used for finding data sharing. So, it is very important to include reverse engineering of association (Vinita, *et al.*, 2008).

### 2.4.2.2 Aggregation relationship

According to Sparks (2001), aggregation is a variant of the "has a" association relationship; aggregation is more specific than association. It is an association that represents a part-whole or part-of relationship. As a type of association, an aggregation can be named and have the same adornments that an association can. However, an aggregation may not involve more than two classes; it must be a binary
association. Aggregation can occur when a class is a collection or container of other classes, but the contained classes do not have a strong lifecycle dependency on the container. In UML, it is graphically represented as a hollow diamond shape in the containing class with a single line that connects it to the contained class. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects. Figure 2.4 shows an example of aggregation relationship.

Figure 2.4: Class diagram showing Aggregation between two classes (Wiki, 2014)

2.4.2.3 Composition relationship

According to Scott (2009), the composition is a stronger variant of the "has a" association relationship; composition is more specific than aggregation. Composition usually has a strong lifecycle dependency between instances of the container class and instances of the contained class(es): if the container is destroyed, normally every instance that it contains is destroyed as well. The UML graphical representation of a composition relationship is a filled diamond shape on the containing class end of the tree of lines that connect contained class (Es) to the containing class. Figure 2.5 shows an example of composition relationship.
2.4.2.4 Multiplicity relationship

This association relationship indicates that (at least) one of the two related classes make reference to the other. This relationship is usually described as "A has a B" (a mother cat has kittens, kittens have a mother cat) (Scott, 2009). The UML representation of an association is a line with an optional arrowhead indicating the role of the object(s) in the relationship, and an optional notation at each end indicating the multiplicity of instances of that entity (the number of objects that participate in the association). Table 2.2 shows the multiplicity indicators.

Table 2.2: Multiplicity Indicators (Agile, 2014)

<table>
<thead>
<tr>
<th>Multiplicity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..1</td>
<td>No instances, or one instance</td>
</tr>
<tr>
<td>1</td>
<td>Exactly one instance</td>
</tr>
<tr>
<td>0..*</td>
<td>Zero or more instances</td>
</tr>
<tr>
<td>1..*</td>
<td>One or more instances</td>
</tr>
</tbody>
</table>

2.4.2.5 Generalization relationship

According to Rohitha (2011), the generalization relationship ("is a") indicates that one of the two related classes (the subclass) is considered to be a specialized form of the other (the super type) and the superclass is considered a 'Generalization' of the subclass. In practice, this means that any instance of the subtype is also an instance
of the superclass. The UML graphical representation of a Generalization is a hollow triangle shape on the superclass end of the line (or tree of lines) that connects it to one or more subtypes. The generalization relationship is also known as the inheritance or "is a" relationship. The superclass (base class) in the generalization relationship is also known as the "parent", superclass, base class, or base type. The subtype in the specialization relationship is also known as the "child", subclass, derived class, derived type, inheriting class, or inheriting type. Figure 2.6 shows an example of generalization relationship.

![Class diagram showing generalization between one superclass and two subclasses (Wiki, 2014)](image)

One of the major activities in software development is the process of generalizing components. We like generalized (inherited) code because it can be reused in various applications. When it has been thoroughly tested and its use properly documented, generalized code saves the user of that code considerable time and effort (Vinita, et al., 2008).

2.4.2.6 Realization relationship

In UML modeling, a realization relationship is a relationship between two model elements, in which one model element (the client) realizes (implements or executes) the behavior that the other model element (the supplier) specifies (Sparks, 2001). The UML graphical representation of a realization is a hollow triangle shape on the interface end of the dashed line (or tree of lines) that connects it to one or more implementers. A plain arrow head is used on the interface end of the dashed line that
connects it to its users. Realizations can only be shown in class or component diagrams. A realization is a relationship between classes, interfaces, components, and packages that connects a client element with a supplier element. A realization relationship between classes and interfaces and between components and interfaces shows that the class realizes the operations offered by the interface. In the example below, the printing preferences that are set using the printer setup interface are being implemented by the printer. Figure 2.7 shows an example of realization relationship.

![Figure 2.7: Class diagram showing realization relationship (Nishadha, 2012)](image)

The main use of realization or interface is reusability. It allows us to reuse a lot of code and make things simpler at the same time. Interfaces promote abstraction. In case if we want to use some class for its functions only, we have to define the interface in which we just use the implementation part only. So, realization rule specifies the corresponding alternate UML class diagram representations (Vinita, et al., 2008).

### 2.4.2.6 Dependency relationship

According to UML (2014), dependency is a directed relationship which is used to show what some UML element or a set of elements requires, needs or depends on other model elements for specification or implementation. Because of this, dependency is called a supplier - client relationship, where supplier provides something to the client, and thus the client is in some sense incomplete while semantically or structurally dependent on the supplier element(s). Modification of the supplier may impact the client elements. Figure 2.8 shows an example of dependency relationship between "Car" class and "Wheel" class (An even clearer example would
be "Car depends on Wheel", because Car already aggregates (and not just uses) Wheel).

![Dependency relationship example](image)

Figure 2.8: Dependency relationship example (Wiki, 2014)

Dependency plays an important role in the understanding of code as the relationship correctly depicts how the effects of a change in one class are being propagated into another class. During the process of reverse engineering it is necessary to incorporate dependency not to lose any important information regarding lower level abstraction and/or any other functional or structural representation (Vinita, et al., 2008).

### 2.5 C# programming language

C# (pronounced C sharp) is a programming language designed for building a wide range of enterprise applications which runs on the .NET Framework. It evolved from Microsoft C and Microsoft C++ and is a simple, modern, type safe, and object oriented programming language. C# code is compiled as managed code, which means that it benefits from of the services of the common language runtime. These services include language interoperability, garbage collection, enhanced security, and improved versioning support (MSDN, 2014).

C# sources are stored in the files with .cs extension. There are also compiled components that are stored in the DLL (binary source code) files. The scope of this dissertation is parsing C# source files stored with .cs extension. The C# programming language provides a unified type system. The main type is object and every other
Value types differ from reference types, and this is because value types directly contain their data, whereas variables of the reference types store references to the object. With reference types, it is possible for two variables to reference to the same object, and it is possible for operations of one variable to act as the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations of one variable to act as the operation of the other (MSDN, 2014).

2.6 Sequential and Parallel approaches

The parallel approach involves the concurrent computation or simultaneous execution of processes or threads at the same time (Mivule, 2011). While sequential approach involves a consecutive and ordered execution of processes one after another (Britannica, 2010). In other words with sequential, processes are executed one after another in succession while in parallel, multiple processes are executed at the same time. In sequential approach, computation is modeled after problems with a chronological sequence of events (Harvey & Wright, 1999). The program in such cases will execute a process that will in turn wait for user input, then another process is executed that processes a return according to user input generating a series of cascading events. In contrast to sequential computation, parallel approach, while processes might execute concurrently, its sub-processes or threads might communicate and exchange signals during execution and therefore programmers have to place measures in place to allow for such transactions (Tokhi, et al., 2003). Figure 2.9 shows the standard steps of sequential while Figure 2.10 shows the standard steps of parallel approach.
Xmipp (2010), mentioned that a program that perform tasks A, B, and C sequentially, and tasks A and C task can be threaded. So, task A can be split in several concurrent tasks and implemented in parallel as A1, A2, A3... An, and the same for C. Figure 2.11 shows the sequential and parallel version of the program execution.
2.7 Multi-Threading technique

According to Butenhof (1997), a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. The implementation of threads and processes differ from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources. In particular, the threads of a process share the latter's instructions (its code) and its context (the values that its variable reference at any given moment) (Butenhof, 1997).

2.7.1 Multi-Threading description

According to Justia (2013), multi-threading paradigm has become more popular as efforts to further exploit instruction level parallelism have stalled since the late 1990s. This allowed the concept of throughput computing to re-emerge to prominence from the more specialized field of transaction processing:

Even though it is very difficult to further speed up a single thread or single program, most computer systems are actually multi-tasking among multiple threads or programs. Techniques that would allow speedup of the overall system throughput of all tasks would be a meaningful performance gain (Justia 2013).

Justia (2013) highlighted two major techniques for throughput computing as being: multiprocessing and multi-threading. These two techniques are explained in the following paragraphs:

Multi-threading is the ability of a program or an operating system process to manage its use by more than one user at a time and to even manage multiple requests by the same user without having to have multiple copies of the programming running on the computer. Central processing units have the hardware support to efficiently execute multiple threads. These are distinguished from multiprocessing systems (such as multi-core systems) in that the threads have to share the resources of a single core: the computing units, the CPU caches and the translation look aside buffer (TLB) (Justia, 2013).
Multiprocessing systems include multiple complete processing units, while multi-threading aims to increase the utilization of a single core by using thread-level as well as instruction-level parallelism. As the two techniques are complementary, they are sometimes combined in systems with multiple multi-threading CPUs and in CPUs with multiple multi-threading cores (Justia, 2013).

According to Barnes, et al., (2012) the advantage of a multithreaded program allows it to operate faster on computer systems that have multiple CPUs, CPUs with multiple cores, or across a cluster of machines, because the threads of the program naturally lend themselves to truly concurrent execution.

2.7.2 Thread code implementation

The term thread code refers to a compiler implementation technique where the generated code has a form that essentially consists entirely of calls to subroutines (Ertl, 2003). The code may be processed by an interpreter, or may simply be a sequence of machine code call instructions. However, a program small enough to fit fully in a computer processor's cache may run faster than a larger program that suffers many cache misses (Bell & James, 2003). Threaded code is best known as the implementation technique commonly used by programming languages.

2.7.3 Thread pool concept

In computer programming, the thread pool pattern is where a number of threads are generated to perform a number of tasks, which are usually organized in a queue (Thaara, 2002). The results of the tasks being executed may also be placed in a queue, or the tasks may return no result. Typically, there are many tasks more than threads. As soon as a thread completes its task, it will request the next task from the queue until all tasks have been completed. The thread can then terminate, or sleep until there are new tasks available (Thaara, 2002). Figure 2.12 shows the thread pool sample.
REFERENCES


