21

A Reverse Engineering System Requirements Tool

Rosziati Ibrahim and Tan Siek Leng
Faculty of Information Technology and Multimedia,
Universiti Tun Hussein Onn Malaysia (UTHM)
Parit Raja, Johor, Malaysia.

1. Introduction

Reverse engineering is the process of discovering the technological principles of a device or object or system through analysis of its structure, function and operation (Sommerville, 2007). Most of the time, it involves taking something apart, for example the device or the system program, and analyzing its working in detail, and trying to make a new device or program that does the same thing without copying anything from the original (Musker, 1998).

In reverse engineering, the process is often tedious but necessary in order to study the specific technology or device. In system programming, reverse engineering is often done because the documentation of that particular system has never been written or the person who developed the system is no longer working in the company. We use this concept to introduce an automatic tool for retrieval of requirements of a system from the program source codes.

The purpose of producing the tool is to be able to recover the system requirements of any system due to the cause that the system does not have the necessary documents. Documenting the process involved in developing the system is important. In many organizations, 20 percent of system development costs go to documenting the system (Heumann, 2001). In software development life cycle (SDLC), documenting process in requirements analysis ends with a system requirements document (SRD) (Sommerville, 2007). SRD is important in order to develop a system. It shows the system specification before a developer would be able to develop the system. Once the system demonstrates fault after implementation phase, the SRD can be used as a reference for finding errors of the system requirements. However, if documenting is not proper, the source codes of the system will be used to find errors. This is a difficult process considering the lines of the source codes would be thousand. Therefore, by having a tool that would be able to retrieve the system requirements back from the source codes would be an added advantage to the software developers of any system application.

This chapter discusses on retrieval of system requirements from its source codes. The rest of the chapter is organized as follows. Section 2 presents the related work, while Section 3 reviews the UML (Unified Modeling Language) and its specification and Section 4 discusses

the system requirements. We then present our idea on how to read the source codes, parse it to parser and then convert it to system requirements in Section 4. Section 5 discusses our tool in details, in particular on how to retrieve data from the source codes using the engine of the tool. Finally, we conclude our chapter in Section 6 and give some suggestions for future work of the tool.

2. Related Work

Reverse engineering has become a viable method to measure an existing system and reconstruct the necessary model from its original. In the older days, disassembler is used to recreate the assembly codes from the binary machine codes, where the assembler is used to convert the codes written in assembly language into binary machine codes. Decomplier, on the other hand, is used to recreate the source codes in some high level language for a program only available in machine codes or bytecodes.

Based on the decompiler and disassembler, there has been a significant amount of study focusing on disassembly of the machine code instructions. Schwarz et al. (2002), for example, study the disassembly algorithms and propose an hybrid approach in disassembly of the machine code. Tilley et al. (1994), on the other hand, propose the programmable approach using the scripting language to enable the users to write their own routines for common reverse engineering activities such as graph layout, metrics and subsystem decompositions.

There has been a significant amount of study for looking at the best technique in reverse engineering focusing on studying the source codes in order to get the design or requirements documents. Knodel et al. (2006) suggest using the graphical elements in order to understand the software architecture better. Graph-based technique is proposed to be one of the good techniques in reverse engineering in order to get the requirements documents. Cremer et al. (2002), for example, use the graph-based technique for COBOL applications and provide code analysis. Based on the graph-based technique as well, UML (Unified Modeling Language) reverse engineering (Altova, 2008) imports Java source codes and generates UML class diagram to facilitate requirements analysis.

For our approach, we use graph-based technique as well to get the necessary information from the C++ source codes, convert the information into necessary tokens and then use these detected tokens to generate the UML class diagram. The class diagram can be used for requirements analysis. Altova tool (Altova, 2008) is quite similar to our tool. However, Altova concentrates on Java program source codes for its input to generate the UML class diagram. Our tool, on the other hand, concentrate on C++ program source codes for its input to generate the UML class diagram.

3. Review of UML and its Specification

In producing a software product, there are four fundamental process activities. They are software specification, software development, software validation and software evolution (Sommerville, 2007). Modeling a software application in software specification before coding is an essential part of software projects. This modeling activitity involves the process of transformation from the users' requirements into the software application.

Software development life cycle (SDLC) is used to process the activities of software development. Four main phases are used in SDLC. There are analysis, design,

implementation and testing (Hoffer et al., 2008). In SDLC, modeling tool is usually used to do the analysis of a system. The modeling tool used can be either a structured approach or an object-oriented approach or a hybrid approach. A structured approach uses diagrams such as entity relationship diagrams (ERD) and context diagrams to model and analyze the system requirements. Object-oriented approach, on the other hand, uses diagrams such as use-case diagrams and class diagrams to model and analyze the system requirements. A hybrid approach is a combination of a structured and object-oriented approach.

A model is an abstract representation of a system, constructed to understand the system prior to building or modifying it. Most of the modeling techniques involve graphical notations, its syntax and semantics. One example of modeling techniques is Unified Modeling Language (UML). UML assumes a process that is use-case driven, architecture-centered, iterative and incremental (Bahrami, 1999). It is a modeling language that provide system architects, software engineers and software developers with tools for analysis, design and implementation of software based systems as well as for modeling business and similar processes. UML includes a set of graphical notation techniques to create abstract models of specific systems. UML modeling can also help software developers to understand the system requirements better using the UML diagrams (Vidgen, 2003).

UML is a standard modeling language for visually describing the structure and behavior of a system. The main purpose of UML is to provide a common vocablary of object-oriented terms and diagrammatical techniques that enable software developers to model any system development project from analysis untill implementation phase. Therefore, during analysis, system requirements are transformed into UML specification using diagrams. These diagrams have special rules and notations. Each diagram used in UML specification has its own syntax and semantics. The syntax is the notations for each element of the diagrams, whereas the semantics is the meaning of the notations.

Currently, UML specifies 13 UML diagrams. These diagrams are divided into two categories: structure and behaviour. Structure diagrams are diagrams that describe the structure of the systems at a particular time (static) while behaviour diagrams are diagrams that describe the behaviour of the systems according to time (dynamic). Structure diagrams in UML include class diagram, composite structure diagram, component diagram, deployment diagram, object diagram and package diagram while behaviour diagrams in UML include activity diagram, sequence diagram, communication diagram, interaction overview diagram, timing diagram, use-case diagram and state machine diagram. These diagrams are then formed to become the UML specification which represented the system specification. The UML specification can then be used to develop the system application.

Currently, UML specification consists of two interrelated parts: UML syntax and UML semantics. UML syntax specifies the abstract syntax and graphical notations of UML object modeling concepts wheras UML semantics describes the mapping of the graphical notations to the underlying semantics as well as the meaning of the graphical notations. Much effort has been given to gather UML semantics for various UML diagrams. Currently, the information relating to UML semantics is scatterd throughout the standard document (Selic, 2004). However UML specification and its dependency relations between diagrams are discussed in Pons et al. (2003). They suggest that the relationships between

software models need to be formally defined since the lack of accuracy in their definition can lead to wrong interpretations and inconsistency between models. The consistency between models need to be checked in order to conform that the models adhere to system requirements.

Not all diagrams are used to specify the behaviour and structure of any system. Activity diagram is an example of behavioral diagram that illustrates business workflows independent of classes, the flow of activities in a use case, or detailed design of a method. Activity diagram is capable of successfully specifying an entire set of use-case scenarios in a single diagram. In addition, it is potentially a rich source of test related information in both business and software-based models. UML activity diagrams are developed using elements that are divided into two groups: nodes and edges. Three types of nodes defined are action nodes, object nodes and control nodes; while edges are defined as the transitions that represent control flow between nodes.

UML class diagram, on the other hand, shows the state structure of object-oriented model, the object class, their internal structure and relationships in which they participate. UML class diagrams are developed using three expects: the form for graphical representation of class diagram, notation for the diagram and the association or relationships amoung classes.

In UML class diagram, a class is a description of a set of objects that share the same attributes, operation, relationships and semantics (Vidgen, 2003). Class is usually represented using a rectangle with three compartments separated by horizontal lines. The first part is class name, followed by list of attributes and list of methods (Rosziati, 2008). Attributes describe the data of the class while methods describe the services the class provided. The relationships for each of the class will then be described by connecting links. Methods are basically categorised into three types: constructor, query and update. Relationships, on the other hand, can be categorised into three types: association, inheritance and aggregation. Association is a simple relationship between classes where one class is associated with another class. Inheritance is a relationships between classes where one class is a superclass from another class (subclass). Aggregation is a relationships between classes where one class comprises of other classes (more than two classes).

In UML diagrams, a use-case diagram is used to describe the requirements of the system representing the behaviour of the system. In a use-case diagram, two important factors are used to describe the requirements of a system. They are actors and use cases. Actors are external entities that interact with the system and use cases are the behavior (or the functionalities) of a system (Rational, 2003). The use cases are used to define the requirements of the system. These use cases represent the functionalities of the system. Most often, each use case is then converted into a function representing the task of the system. The descirption of a use case defines what happens in the system when the use case is performed (Bahrami, 1999). The arrow that is connected from an actor into a use case represented a communication between the outside (actors) and inside (use-case) of the system's behaviour.

4. System Requirements

The foundation of a good application begins with a good analysis. In order to do a good analysis, we should be able to rigorously analyze the system requirements. Requirements analysis is an important phase during the software development life cycle (SDLC). In

UML specification, requirements analysis is usually done using diagrams (Bahrami, 1999). A use-case diagram is used to specify requirements of the system.
In this section, we present an example of an application for monitoring system of a postgraduate student submitting his/her progress report to Centre of Graduate Studies. The requirements of the system include the capability to submit progress report using the provided form, view the submitted progress report and evaluate the submitted progress report. These three requirements are then transformed into a use-case diagram as shown in Figure 1.

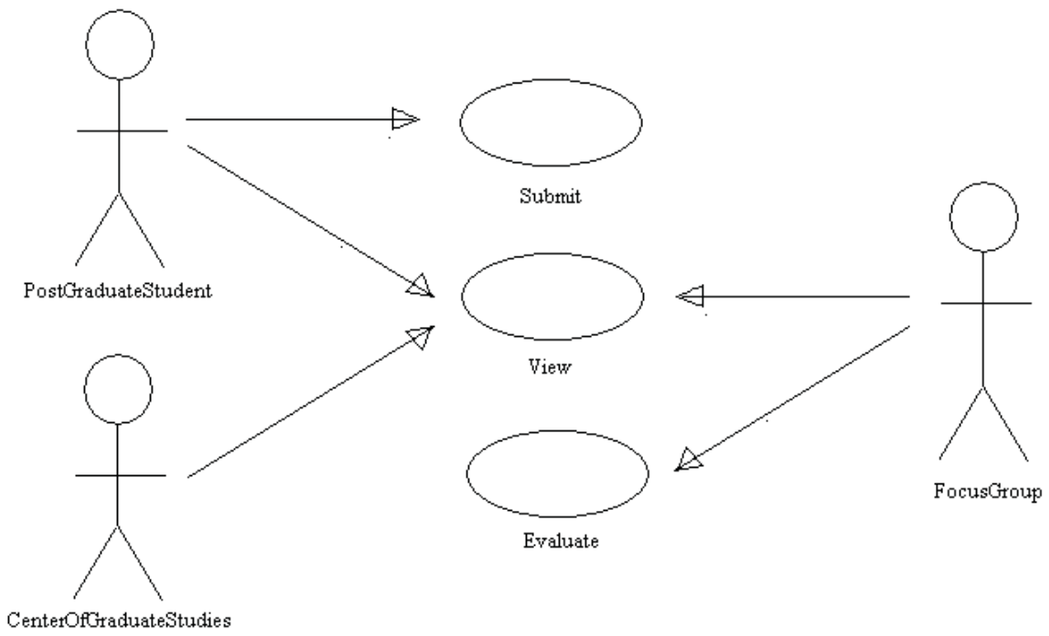

Fig. 1. A use-case Diagram for Monitoring System of Postgraduate Student

Figure 1 shows a simple use-case diagram for a monitoring system of postgraduate student where a postgraduate student (an actor) can submit his/her progress report to Centre of Graduate Studies. From Figure 1, a student is able to do two tasks: submit a progress report and view a progress report. A focus group is able to view and evaluate the progress report while the centre is able to view the progress report.
Once the use-case diagram is formed, the next diagram, an activity diagram can be developed. An activity diagram, on the other hand, describes the activities of the process. The purpose of an activity diagram is to provide a view of flows and what is going on inside a use case (Bahrami, 1999). Figure 2 shows an example of an activity diagram which exhibits the activities that can be performed by a postgraduate student. From a use-case diagram in Figure 1, a postgraduate student is able to submit and view the progress report. Hence, the activity diagram shows that these two activities can be performed by the postgraduate student.

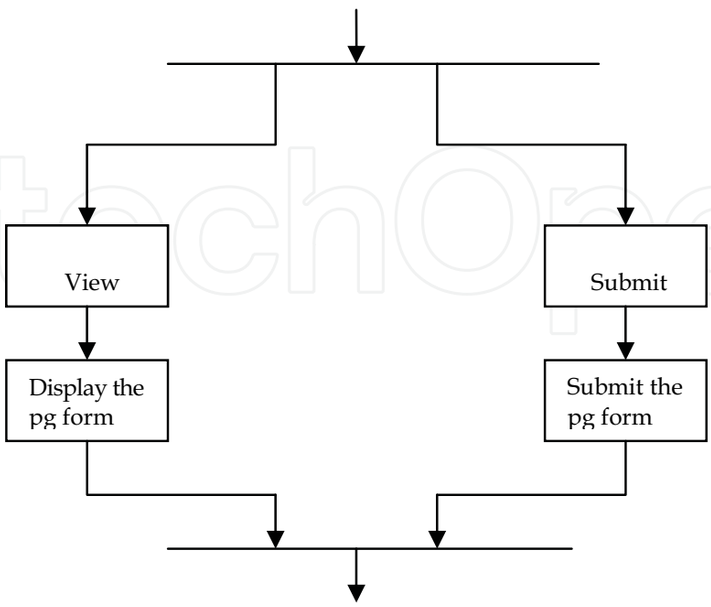

Fig. 2. An Activity Diagram for Postgraduate Student

Most often, use cases represent the functional requirements of a system. If the requirements are gathered correctly, then a good use-case diagram can be formed. From this use-case diagram, the use cases are usually used for the functions of the system. Table 1 shows the mapping of use cases to functions of a system. These functions can then be used in a class diagram of the system.

Use Case	**Function**
Submit	Submit
View	View
Evaluate	Evaluate

Table 1. Use Cases Mapping to System's Functionalities

The class diagram is the main static analysis diagram (Bahrami, 1999). It shows the static structure of the model for the classes and their relationships. They are connected to each other as a graph. Each class has its own internal structures and its relationships with other classes. Figure 3 shows an example of a class diagram for Monitoring System of Postgraduate Student. Note that the mapping from use-cases from Figure 1 into functions in the class diagram in Figure 3. This mapping is important for the consistency of the UML diagrams.

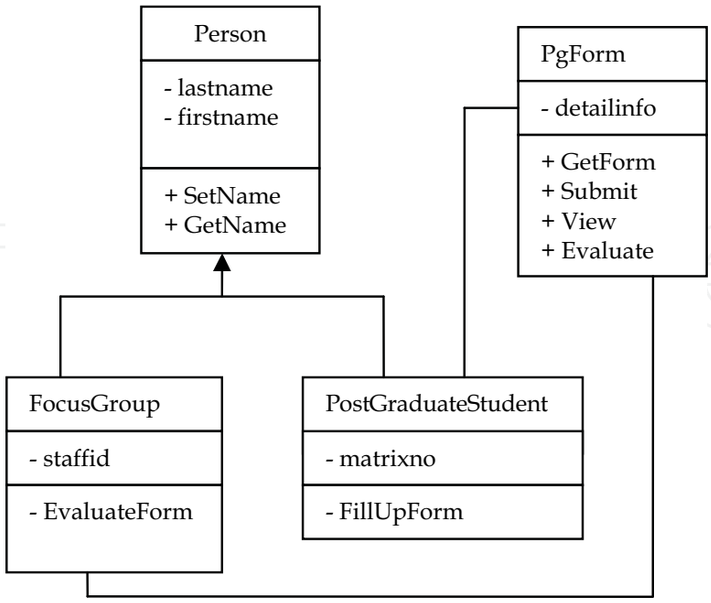

Fig. 3. A Class Diagram for Monitoring System of Postgraduate Student

From Figure 3, each class consists of a class name, its attributes and methods. For example, a class *Person* has attributes lastname and firstname with no method. Classes *FocusGroup* and *PostgraduateStudent* inherit class *Person*. Class *FocusGroup* declares its own attribute (staffid) and one method (EvaluateForm) and class *PostGraduateStudent* declares its own attribute (matrixno) and one method (FillUpForm). Note that, a subclass inherits all the attributes and methods of its superclass. Class *PgForm*, on the other hand, offers 4 methods namely GetForm, Submit, View and Evaluate. The three methods are translated from the three use cases declared in Figure 1.

5. The Tool - CDG

The tool, which we call CDG (Class Diagram Generator) is implemented using C++ programming language. The tool has two stages of activities. The first stage accepts the source codes of C++ programming language as the input and produces the output as detected tokens in term of the set of class name, its attributes and functions as well as its relationships with other classes. From this output, for the second stage, the tool will suggest the possibility of the class diagram. The targeted user of the tool is software developer who wants to get back the system requirements specification based on the program source codes. The main objectives of developing the tool are being able to detect the necessary tokens from the syntaxes of the program source codes and generate the class diagram automatically based on the detected tokens. Figure 4 shows the activity diagram to generate the class diagram from the tool.

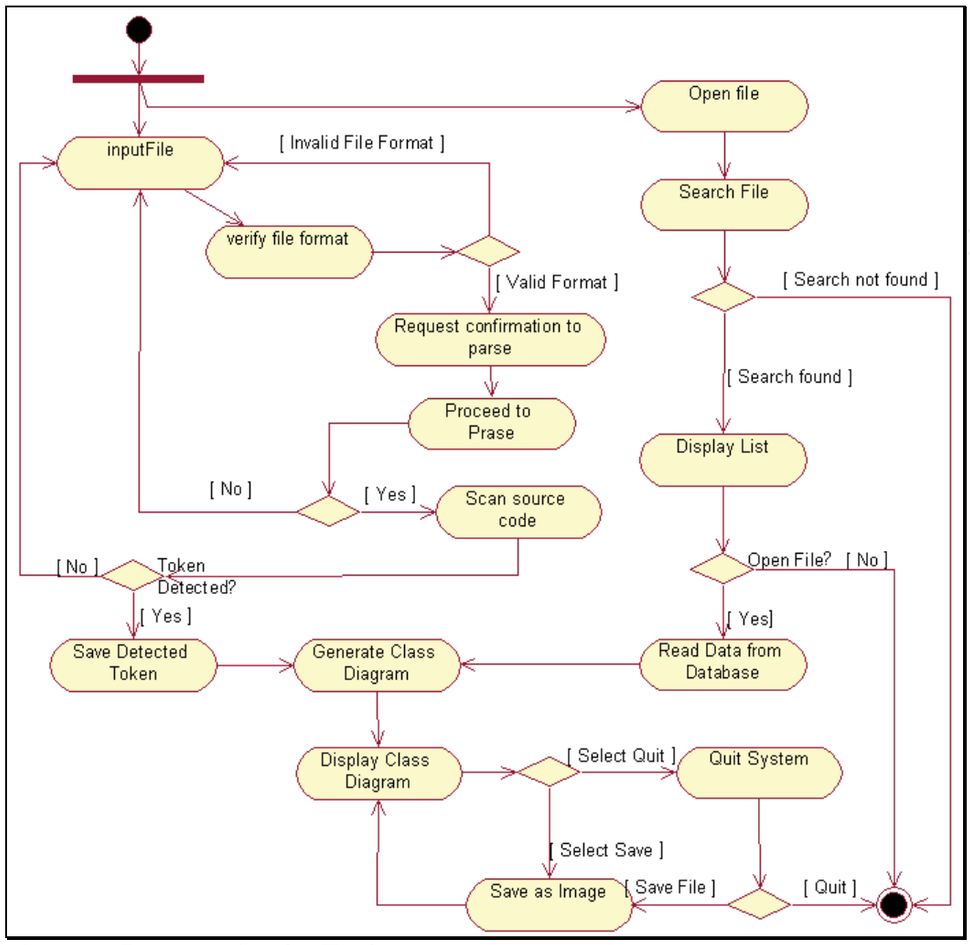

Fig. 4. Activity Diagram for the Tool

From Figure 4, in order to generate the UML class diagram, a user is required to input a C++ program source codes into the tool. After that, the tool will verify the file format as well as the filename. If an invalid file format has been entered or the file does not exist, the tool will prompt an error message to warn the user. Indeed, the user needs to reinsert the filename. However, if both the filename and file format is valid, the tool will reconfirm whether it is the file that the user needs. All the commands in the tool are case-insensitive where the tool will recognize both lowercase and uppercase command typed by the user. The tool will also provide files searching function in order to list out all the files' name in a folder. The tool will only accept a C++ source codes with the ".cpp" and ".h" extensions. When a user tries to insert an invalid file, the tool will display a warning message and ignore the file.

Once the correct source codes file has been verified, the source codes are parsed to the parser. The parser will read the file, line by line, detect the tokens and store the necessary tokens to form the class diagram. Note that, the tool will bypass all the comments found in the file. There are two types of comments which are single line comments (//) and multiple line comments (enclosed between /* and */).

Before the class diagram is displayed, the tool will display the scanning results to the user. The result will contain the set of class name, its attributes and methods as well as its relationships with other classes. The tool will then provide two log files to store the error occurred and parsing results. The detected tokens will be stored into another file for generating the class diagram. If the program source codes do not have any syntax errors and have been successfully passed the parser process, the class diagram will be generated by the tool and saved as an image file for later used. Figure 5 shows the class diagram for the tool.

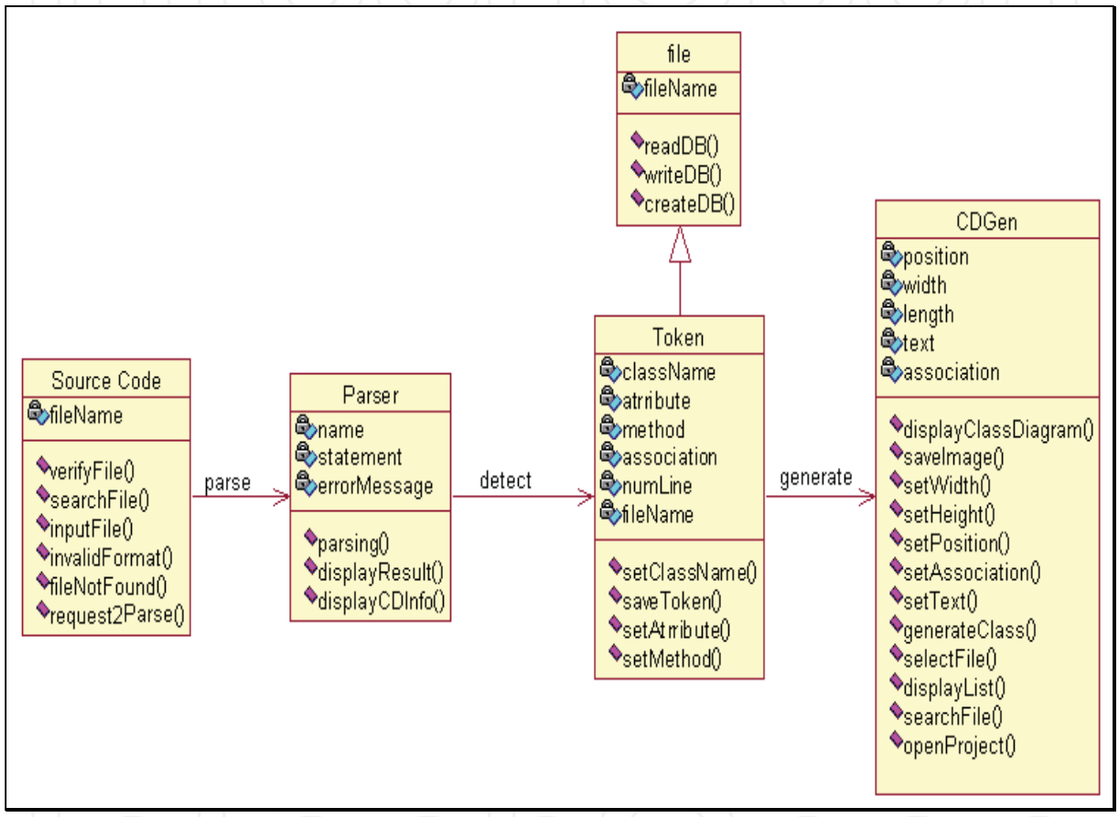

Fig. 5. Class Diagram for the Tool

From Figure 5, a class diagram of the tool represents the static view of the system. It shows classes in a system and relationships amoung themselves. Five classes are identifed for the tool. The internal structure of the system is illustrated by behaviour and attributes of each class, as well as relationships between classes. In Figure 5, the class Source Code manages the file verification and the class Parser focuses on parsing the source codes file input by the user. Class Token is inherited from class File where it focuses on detecting the tokens in the source codes when the system is parsing the program source codes. Lastly, class CDGen generates the drawing information in order to draw the UML class diagram as well as saving them into the database.

The tool is implemented using C++ programming language. Figure 6 shows the interface design of the tool.

Fig. 6. The Interface Design of CDG

From Figure 6, CDG offers a pop up menu for user friendly interface. If a New Project menu is clicked, another pop up menu will be displayed to ask for the program source codes files to be parsed as shown in Figure 7. The user can insert the files for the program source codes. Then, the user can click the Generate button to generate the UML class diagram for that particular project. Figure 8 shows the example of the generated UML class diagram.

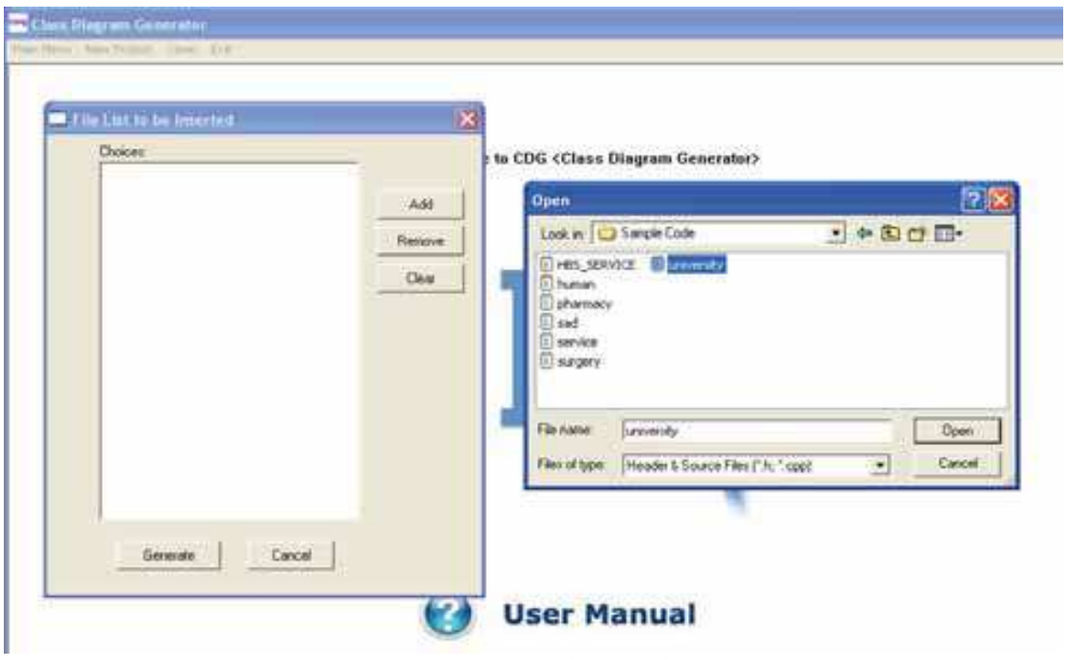

Fig. 7. A Pop Up Menu for Adding The Program Source Codes into the Tool

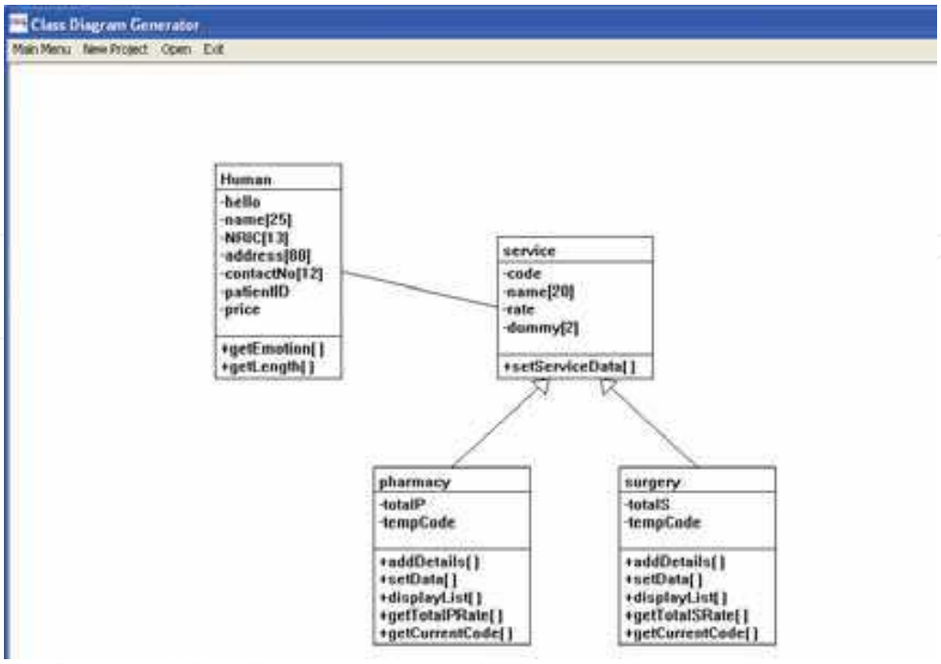

Fig. 8. The Generated Class Diagram from the Tool

For the purpose of ease in understanding in this chapter, we present an example of an application for monitoring system of a postgraduate student submitting his/her progress report to Centre of Graduate Studies as our case study using the tool. From Figure 3, a class *Person* is a superclass of classes *FocusGroup* and *PostGraduateStudent*. Therefore, classes *FocusGroup* and *PostGraduateStudent* inherit all attributes and methods of class *Person*. Figure 9 shows some of the extracted source codes from the program of this system.

```
class Person {
      private:
         char lastname [30];
         char firstname [30];
      public:
         void SetName();
         char *GetName();
};
…
```

```
class FocusGroup : public Person {
      private:
         char staffid [10];
      public:
         Void EvaluateForm();
};
…
```

```
class PostGraduateStudent : public Person {
      private:
```

```
            char matrixno [10];
      public :
            void FillUpForm();
};
…
class pgForm{
      private :
            struct Data {
                  char lastname [30];
                  char firstname [30];
                  char matrixno [10];
            } detailinfo;
      public:
           void GetForm();
            void Submit();
            void View();
            void Evaluate();
};
```

Fig. 9. Extracted Source Codes

From Figure 9, adopting the hybrid algorithm (Schwarz et al., 2002) by using the linear sweep and recursive traversal algorithms, the tool is able to read the source codes line by line and detect the necessary tokens. Then the tokens are stored. Figure 10 shows the extracted tokens from reading of the program source codes.

Class name: Person Association: Inheritance: Attributes: lastname, firstname Methods: SetName, GetName
Class Name: FocusGroup Association: Inheritance: Person Attributes: staffid Methods: EvaluateForm
Class Name: PostGraduateStudent Association: Inheritance: Person Attributes: matrixno Methods: FillUpForm
Class Name: PgForm Association: Person Inheritance: Attributes: detailinfo Methods: GetForm, Submit, View, Evaluate

Fig. 10. Extracted Tokens from the Source Codes

From Figure 10, the hybrid algorithm (combination of linear sweep and recursive traversal algorithms) is used in order to identify and extract the necessary tokens. Once the tokens have been identified and extracted from the source codes, the graph-based approach is used in our engine of the tool in order to develop and generate the class diagram from the extracted tokens. Figure 11 shows the generated class diagram. The generated class diagram can be saved for later use. The extension of the saved file is ".cdg".

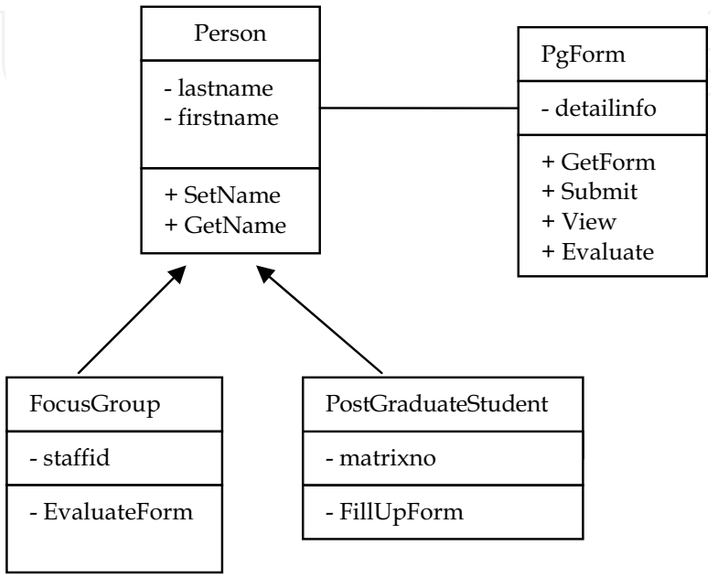

Fig. 11. Generated Class Diagram

Based on Figure 11, the tool is able to generate the possible class diagram for the system. However, comparing from Figure 11 and Figure 3, we still have problems to overcome the associations of the classes and relationships among the classes. The ambiguities of the system requirements are still existed. We are currently looking at the possible solutions to reduce these ambiguities.

The tool offers the system requirements by means of extracted tokens of class name, its attributes and functions as well as its relationships with other classes. Then the tool suggests the possible class diagram based on the extracted tokens.

6. Conclusion and Future Work

Modeling is an important part of any projects. A model plays a major role in system development life cycle. A model is also served as a blueprint for the system. During requirements analaysis, a model is usually developed using a modeling language. If however, due to poor documentation, a model does not exist, the tool can be used to get the system requriements from the program source codes. In this chapter, we discuss the tool that provides the ease in coming up with the system requirements when the system does not support the proper documents for requirements analysis.

We have also described in this chapter our idea on how to read the program source codes, parse it to parser and then convert it to system requirements. We are currently improving our algorithm of extracting the tokens in order to reduce the ambiguities of the system requirements. For future work, the tool can also be designed to parse other types of programming languages such as Java and C#.

7. Acknowledgements

The authors would like to thanks Universiti Tun Hussein Onn Malaysia (UTHM) for supporting this research under the short term research grant.

8. References

Altova (2008). UML Reverse Engineering, http://www.altova.com/features_reverse_engineer.html

Bahrami A. (1999). Object-Oriented Systems Development, Mc-Graw Hill, Singapore.

Cremer K., Marburger A. and Westfechtel (2002). Graph-based Tools for Re-engineering, Journal of Software Maintenance and Evolution: Research and Practice, Voulme 14, Issue 4, pp 257-292.

Heumann J. (2001). Generating Test Cases from Use Cases, Rational Software, IBM.

Hoffer J., George J. and Valacich J. (2008). Modern Systems Analysis and Design, 5th Edition, Pearson International Edition, New Jersey.

Knodel J., Muthig D. and Naab M. (2006). Understanding Software Architectures by Visualization – An Experiment with Graphical Elements, Proceeding of the 13th Working Conference on Reverse Engineering (WCRE 2006).

Musker D. (1998). Reverse Engineering, IBC Conference on Protecting & Exploiting Intellectual Property in Electronics.

Pons C., Giandini R., Baum G., Garbi J.L., Mercado P. (2003). Specification and Checking Dependency Relations between UML Models, in UML and the Unified Process. Hershey: IGI Publishing, 2003, pp 237-253.

Rational. (2003). Mastering Requirements Management with Use Cases, Rational Software, IBM.

Rosziati I. (2008). An Introduction to Object-Oriented Programming with UML using Borland C++, UTHM Publication. ISBN 9789832963776.

Schwarz B., Debray S. and Andrews G. (2002). Disassembly of Executable Code Revisited, Proceedings IEEE Working Conference on Reverse Engineering, October 2002, pp 45-54.

Selic B.V. (2004). On the Semantic Foundations of Standard UML 2.0, in Formal Methods for the Design of Real-Time Systems, Vol. 3185/2004, Springer Berlin, pp 181-199.

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

Tilley S., Wong K., Storey M. and Muller H. (1994). Programmable Reverse Engineering, Journal of Software Engineering and Knowledge Engineering.

Vidgen R. (2003). Requirements Analysis and UML: Use Cases and Class Diagrams, Computing and Control Engineering, April 2003, pp. 12-17.

Recent Advances in Technologies

Edited by Maurizio A Strangio

ISBN 978-953-307-017-9

Hard cover, 636 pages

Publisher InTech

Published online 01, November, 2009

Published in print edition November, 2009

The techniques of computer modelling and simulation are increasingly important in many fields of science since they allow quantitative examination and evaluation of the most complex hypothesis. Furthermore, by taking advantage of the enormous amount of computational resources available on modern computers scientists are able to suggest scenarios and results that are more significant than ever. This book brings together recent work describing novel and advanced modelling and analysis techniques applied to many different research areas.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Rosziati Ibrahim and Tan Siek Leng (2009). A Reverse Engineering System Requirements Tool, Recent Advances in Technologies, Maurizio A Strangio (Ed.), ISBN: 978-953-307-017-9, InTech, Available from: http://www.intechopen.com/books/recent-advances-in-technologies/a-reverse-engineering-system-requirements-tool

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen