A MODEL OF SOFTWARE COMPONENT INTERACTIONS USING THE CALL GRAPH TECHNIQUE

NOORAINI ISMAIL

MASTER OF SCIENCE (INFORMATION TECHNOLOGY)

UNIVERSITI UTARA MALAYSIA

2013

A MODEL OF SOFTWARE COMPONENT INTERACTIONS USING THE CALL GRAPH TECHNIQUE

A thesis submitted to the UUM College of Arts and Sciences in
fulfilment of the requirements for the degree of Master of Science
Universiti Utara Malaysia

by
Nooraini Ismail

Permission to Use

Abstrak

Maklumat interaksi yang berkaitan dengan operasi antara komponen adalah penting, terutama apabila program perlu diubahsuai dan diselenggara. Oleh itu, komponen yang terlibat perlu dikenal pasti dan dipadankan berdasarkan keperluan sistem. Maklumat berkenaan boleh diperolehi menggunakan teknik kod ulasan. Walau bagaimanapun, proses ini mengambil masa yang panjang. Penyelidikan ini mencadangkan suatu model untuk mewakili maklumat tersebut yang mana ia diperolehi secara automatik daripada kod sumber untuk menyediakan paparan yang berkesan untuk perwakilan interaksi antara komponen perisian. Untuk mencapai objektif kajian, metodologi reka bentuk kajian yang mengandungi lima fasa telah diadaptasikan iaitu kesedaran kepada masalah, cadangan, pembangunan, penilaian dan kesimpulan. Fasa pembangunan mempunyai proses yang lebih terperinci yang mana maklumat interaksi antara komponen perlu diperolehi secara automatik menggunakan peralatan kejuruteraan balikan dan program tambahan. Program ini digunakan untuk mendapatkan maklumat perisian, maklumat interaksi komponen dalam program perisian, dan untuk mewakili model dalam bentuk *call graph*. Graf yang dihasilkan ini dinilai melalui dua cara, iaitu menggunakan alatan penggambaran yang bersesuaian dan juga melalui kajian oleh pakar. Alatan penggambaran digunakan untuk memaparkan graf yang dihasilkan daripada format teks ke paparan grafik. Proses penilaian model pula dijalankan melalui teknik kajian pakar. Hasil kajian ini menunjukkan bahawa model yang terhasil boleh digunakan dan dimanipulasikan untuk tujuan menggambarkan maklumat interaksi antara komponen. Model ini boleh digunakan untuk menyediakan paparan penggambaran bagi penganalisis untuk melihat interaksi maklumat yang relevan dalam komponen perisian. Ia juga dalam meningkatkan pemahaman mengenai integrasi komponen itu sendiri, supaya penganalisis boleh memanipulasi dan mengekalkan perisian untuk tujuan tertentu.

Kata kunci: Model interaksi komponen, Perwakilan *call graph*, Kefahaman program, Alatan penggambaran

Abstract

Interaction information that is related to operations between components is important, especially when the program needs to be modified and maintained. Therefore, the affected components must be identified and matched based on the requirement of the system. This information can be obtained through performing the code review technique, which requires an analyst to search for specific information from the source code, which is a very time consuming process. This research proposed a model for representing software component interactions where this information was automatically extracted from the source code in order to provide an effective display for the software components interaction representation. The objective was achieved through applying a research design methodology, which consists of five phases: awareness of the problem, suggestion, development, evaluation, and conclusion. The development phase was conducted by automatically extracting the components' interaction information using appropriate reverse engineering tools and supporting programs that were developed in this research. These tools were used to extract software information, extract the information of component interactions in software programs, and transform this information into the proposed model, which was in the form of a call graph. The produced model was evaluated using a visualization tool and by expert review. The visualization tool was used to display the call graph from a text format into a graphical view. The processed model evaluation was conducted through an expert review technique. The findings from the model evaluation show that the produced model can be used and manipulated to visualize the component interactions. It provides a process that allows a visualization display for analysts to view the interaction of software components in order to comprehend the components integrations that are involved. This information can be manipulated and improved the program comprehension, especially for other software maintenance purposes.

Keywords: Component interaction model, Call graph representation, Program comprehension, Visualization tool

Acknowledgement

Alhamdulillah. A thousand gratitude to Allah S.W.T for giving me strengths to finish my studies.

First of all, I would like to express my deepest gratitude to my supervisor, Dr Nor Laily Hashim for her excellent guidance, moral support and advice throughout my studies.

For my thesis examiners, who give me their valuable comments and feedbacks.

I would like to acknowledge my sponsor, Ministry of High Education for granting sponsorship of my research study.

My dearest thanks I own to my husband Mohd Hafiz Abdullah, for his encouragement, patient, prayers and support throughout my studies.

I also thanks to my family, my masters colleagues and friends for their help and encouraged me through this thesis. May Allah bless us, InsyaAllah.

Table of Contents

List of Tables

List of Figures

CHAPTER ONE
INTRODUCTION

1.1 Overview

This chapter includes an overall research plan of this study by introducing the research foundation and motivation to be undertaken in this research. It also includes a detailed description of the issues to be studied, the research objectives, scope of the research, the research framework, and the contribution expected to be gained in this research.

1.2 Introduction

A software component can be a single element of software that can be integrated with other components (Szyperski, 1998). Two components are integrated if they can potentially react to the same events (Fiege, 2005), which is bypassing messages through their interfaces when the components are provided or required for specific events (Inverardi et al., 2003). The communication between components typically is realized by procedure calls or any kind of messaging (Bure et al., 2009).

When new components are integrated, a newly added component has an effect on another component and it can be used by other components. Because of this situation, the program may crash or immediately stop the execution of the system. For this reason, a programmer must scan through the program and investigate which components are causing the errors.

To identify the component interactions of the system, the programmers need to refer to the software program to examine lines of program code. This task will become more complex and time consuming when it requires scanning through the lines of code and this requires the knowledge of developers. For the users who are not programmers or developers, it is difficult to trace through the program code.

Most of the users faced many issues when trying to learn a program. According to Winslow (1996), many novice programmers are unable to transform the problem solution into source code. They must take a great deal of time to comprehend the syntax and semantics of a system. In addition, they need a technique and tools to help them in the learning process. This process is related to program comprehension.

Program comprehension is concerned with the individual programmer's understanding of "what a program does and how it does it in order to make functional modifications and extensions to program without introducing errors" (Corritore and Wiedenbeck ,2001). A suitable visual representation must be considered carefully to ensure the effectiveness of visualization used. Based on Cross et al. (1998), representing objects, processes, and ideas in pictures rather than words is intuitively more appealing. This is because a large amount of software can be displayed clearly as a visualization.

Software visualization is one of the alternative ways to represent the information of software components. This is because humans are better at deducing information from graphical images than from numerical information (Caserta & Zendra, 2001).

By using visualization, the interaction of software components is a more natural, effective, and perhaps essential way to help programmers to avoid tracing through the codes. To fulfil the objective of this study and gain greater comprehension of the component interaction in the software structures, a model that can represent this information and the process to extract this information needs to be explored.

1.3 Problem Statement

At the time this research was conducted, several reverse engineering tools were able to extract details of software component information. The focus on component information and their traces mostly depend on the programming languages used.

Therefore, the programmers need to be expert in the language itself. The problem in extracting and tracing will be more difficult when the programming language applied is in a *custom user library* or third party component (Acharya, 2013). Among the available reverse engineering tools, SOMOX and JCE are the tools that are similar to the study. However, these two tools cannot be used because they focus less on extracting detailed information concerning component interactions and their traces.

The Software Model Extractor (SOMOX) is also a reverse engineering tool that can be used to predict whether a program complies with a component-based development approach. Aside from this, the Java Component Extractor (JCE) tool analyses the main Java elements that are classes (abstract or not), interfaces, inner classes, subtyping and composition relationships, and method signatures (Royer, 2010). These tools have the strength of being able to improve the overall component structure and make it more explicit. They do not focus on revealing detailed information of the services involved in the interactions in terms of which classes implement the services and the traces of a program involved when the services are invoked. This information is important for future maintenance purposes, especially in creating test cases and to identify any changes that happen to the services when newer versions of components are produced. The above studies are similar in nature to this research as they focused on extracting component information; however, their focus differs slightly from this research, as they have different rules used to extract provided and required interface information.

Even though JCE and SOMOX are dealing with extraction of software components and classes, the details of the processes involved were not discussed. Based on these studies, neither tool covered the processes involved in extracting component information of its interaction. They are more focused on the extraction that was done on all components and class levels in the software. This is difficult for users who need to know the processes involved in order to manipulate the component interfaces to keep track of the flow of the component integrations. To overcome the problem, this proposed model was built to provide the stage by stage process of how to extract and trace the component integration of the program.

1.4 Research Questions

To conduct the main goal of this research, this research will answer the following questions:

1. What is the type of visualization technique to model software component interactions?
2. What are the suitable processes used to extract the software component interactions?
3. How should the proposed model be evaluated?

1.5 Research Objectives

This researcher proposed a model for representing component interactions in software using a visualization technique. The following specific objectives are formulated:

1. to define the process to extract component interaction using a suitable visualization technique;
2. to propose a model for representing software component interactions; and
3. to evaluate the proposed model in term of its ease of use.

1.6 Research Scope

For this study, a software component interaction model representation is used for software written only in Java. The sample of programs used in this study is focused on open source, which have no documentation and use an object-oriented component-based approach as their development method. The criterion that was selected is based on open source applications developed in Java Swing. The samples for the experiment are chosen from software that was published and used by other researchers in their work, which were developed using an object-oriented component approach. The samples are selected partially because they executed on all platforms supported by Java (e.g., Windows, Linux, and Mac), and they are 100% open source. Aside from this, the applications also can be downloaded from http://sourceforge.net.

This study is focused on how to extract and model component interaction traces. In extracting the component interaction information, this study filters out any methods that refer to Java API component libraries. Therefore, any methods that are not listed as interface operations, such as execution of Java API methods or class methods, are ignored. The component information is focused only at their interface interactions, and not on control or data dependencies, and class interactions.

The technique to extract traces of the software component interaction used in this research is static analysis, which examines code without performing the program execution. The results do not consider loops, which lead to a reduction in the path length of the model. This operation will produce a model that shows component interactions involved starting from the main entry point of the program (the “main” function), the following call, and integration dependencies. This model only shows the integrations, positions in the code, and names of the components involved in the program.

An objective of this research is to model component interaction information in the form of a call graph; the amount of textual information contained in nodes will be limited to reduce the number of overlapping edges, which is focused only at the component level of interaction. Furthermore, this research also only focuses on visualizing software as it is coded, and dealing with information that is valid for all possible executions of the software, as an effort to show that the source code of the method can, thus, be seen without losing the call graph context, which is very convenient to understanding communications between their methods.

1.7 Significant of Study

This researcher has proposed a model that supports visualization component interaction information in the form of a call graph to help enhance program comprehensibility. The proposed model provided processes to translate a code program to a call graph in order to show the component integration in the form of a graphical view. These processes are significant for software engineering researchers and practitioners, as they improve the program comprehension, so that it is easier for them to conduct the process of extracting component interaction information. The proposed model contributes to the domain of software comprehension, especially in extracting and modelling in component interaction. If users do not understand what the system can do for them, they cannot move even one step with it.

In presenting a large amount of evolutionary information in a scalable way, this thesis is significant by giving some benefit for people who are in the area of manipulating data, which required extraction of the information in many purpose of the study. Using a graph-based approach can help in better understanding software evolution (Bhattacharya et al., 2012). By viewing in the form of a graph, users are not required to trace the program through the source code to retrieve the program information; they can study just on the flow of the graph itself. It is important, especially for large systems, that evolutionary program maintenance will become costly, and difficult to manage and understand (April & Abran, 2012). This is because a programmer should have the ability to explain the program, understand the program's structure, its behaviour, and its relationships to its application domain (Ruiz et al., 2012).

1.8 Overview of Thesis

This thesis consists of five chapters organized as follows. Chapter 1 is a description of the formulation of the problem statement, research objectives, and scope of study to achieve the desired goals. Chapter 2 is a discussion of the background information and related work, including an overview of the area of software components and representation. This chapter will be an explanation of the issues concerning how to represent software components in the form of a call graph. Next, the discussion is toward the model of software representations. The chapter ends with an investigation concerning the techniques that are used in detail. Chapter 3 is an introduction to the process of the methodology used in this research. The evaluation techniques are revised and the reverse and re-engineering technique is selected. In addition, Vaishnavi and Kuechler (2008) was used as a research model. The detailed process to construct a call graph is also discussed in this chapter. Chapter 4 is an overview of the proposed model discussed to show the flow of the process to capture component extraction in software. Chapter 5 is a presentation of the details of the procedure to evaluate the proposed technique and results of the research. The evaluation includes a questionnaire by Baecker (1988), data analysis, and its finding. Chapter 6 is the conclusion of the evaluation of the component representation. The evaluation is done by using a method defined in Chapter 3. The research findings, suggestions for future work, and conclusions are outlined in this chapter.

1.9 Summary

In this chapter the background of this study and the layout of this research were covered. This chapter is a discussion of the introduction of this research, followed by the problem statement, research objectives, scope, framework, research contribution, and research questions. The details of this study are discussed briefly in the following chapter.

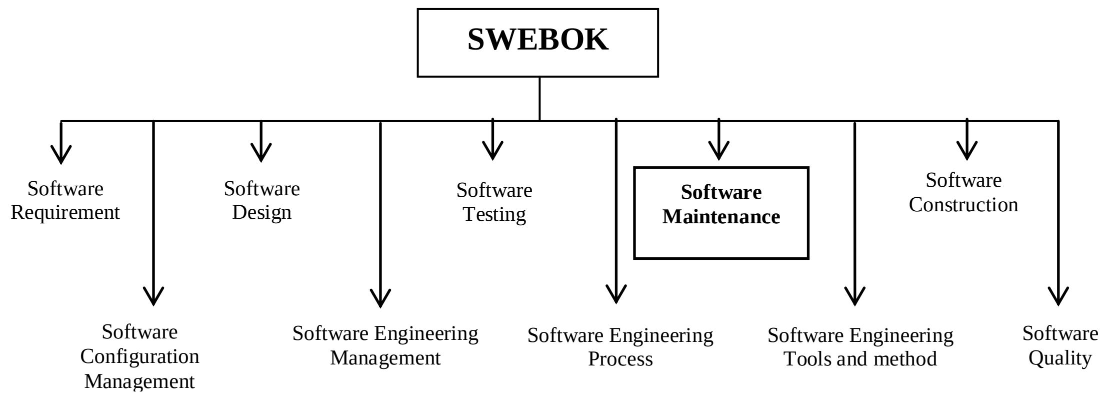
CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

This chapter is a presentation of an overview of existing studies in the area of software visualization, components, and component interactions. Section 2.2 is a description of software maintenance. Section 2.3 is a discussion of software comprehension. Software components are described in section 2.4. An existing study concerning related component extraction tools is explained in section 2.5. Section 2.6 covers software representation. Section 2.7 is a discussion of the dependence graph. Last, section 2.8 is a summary of this chapter as a whole.

2.2 Software Maintenance



SWEBOK
Software Requirement
Software Design
Software Testing
Software Maintenance
Software Construction
Software Configuration Management
Software Engineering Management
Software Engineering Process
Software Engineering Tools and method
Software Quality

Figure 2.1. Ten knowledge areas adapted from Abran et al. (2004)

This area of study falls under the area of software maintenance. According to SWEBOK, software maintenance is referred to as modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment (Abran et al., 2004).

Software maintenance is a difficult and expensive process. Continually changing requirements must be incorporated into the software system and bugs must be fixed (Ackermann & Lindvall, 2007). As explained by Grubb and Takang (2003 p. 370), software maintenance is defined as "modifying a process of a software system or component after its release to correct faults, improve performance or other attributes, or adapt to change of an environment". This is a predictable process, as a software system must go through repetitive changes to maintain its usefulness during its lifecycle (Ackermann & Linvall, 2006).

Sherburne and Fitzgerald (2004) have defined software maintenance slightly differently. They defined software maintenance with several activities related to processes of changing software, including fixing bugs, enhancing function and performance, providing backward compatibility, updating algorithms, covering errors in hardware, and creating user-interface access methods.

Grubb and Takang (2003) suggested that software maintenance should be first understood by separating the meaning of the software and maintenance. Thus, software can be defined as documentation and operating procedures by which computers can be made useful to users, while maintenance is the act of improving the state of software in an existing state of repair, efficiency, or validity, to guard against failure or decline. The combination of these two can be declared as a discipline, which is concerned with changes related to a software system after delivery, which is traditionally known as software maintenance. This definition is supported by the roles of software maintenance, which are:

- The need-to-adapt view—maintenance is making changes to software when its operation environment or original requirement changes.
- The bug-fixing view—maintenance is detecting and correcting errors.
- The user-support view—maintenance is the provision of support to users.

This research is focused on the latter role of software maintenance, which is the user-support view. This is because this research is focused on extracting and representing components for use in identifying component interactions

Software maintenance can be categorized into four areas of study, which are fundamentals, key issues, process, and techniques for maintenance. The details of software maintenance are shown in Figure 2.2.

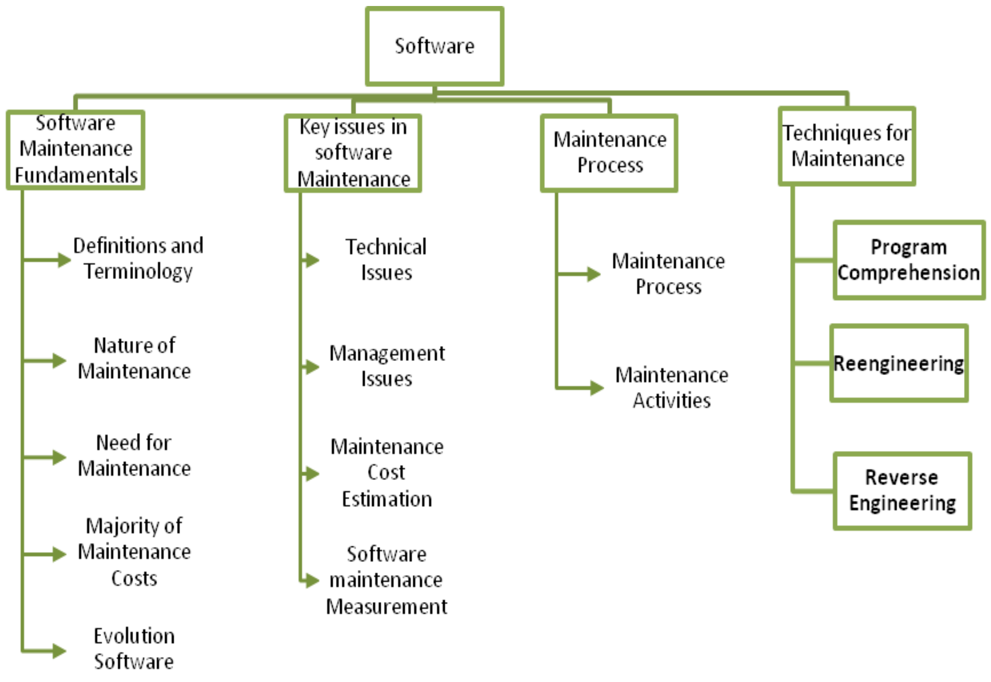


Figure 2.2. Area of software maintenance adapted from Abran et al. (2004)

Based on Figure 2.2, this study is focused on the aspects of techniques for maintenance only. There are three types of techniques for software maintenance: program comprehension, re-engineering, and reverse engineering. For this study, only the area of program comprehension and reverse engineering technique are applied.

Various studies have been conducted in program comprehension over the last few decades (Storey, 2005). The suggested idea, techniques, and tools can help the users to comprehend a program. Software comprehension is a major activity during software maintenance. Program understanding is important because it can explain how programmers understand a program or software (Maletic & Kagdi, 2008). Further discussion of software comprehension is provided in section 2.3.

As discussed by Chikofsky and Cross II (1990), reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationship and creating representations of the system in another form at a higher level of abstraction. As a result, the maintainers can perform reverse engineering starting from any level of abstraction or any stage of the lifecycle. However, what is important to notice is, reverse engineering does not involve changing to create a new system; it is a process of examination, not a replication. For this study, reverse engineering is used in extraction of component information. The program will extract the component in the software that already exists. The further process of reverse engineering is provided in Chapter 4.

2.3 Software Comprehension

According to Corritore and Wiedenbeck (2001), program comprehension is concerned with the individual programmer's understanding of "what a program does and how it does it in order to make functional modifications and extensions to program without introducing errors". Von Mayrhauser (1995) stated that program comprehension is "an activity in which the program reader extracts meaning by understanding how a particular program or code fragment performs its task, or what task a particular item performs".

There are two models in software comprehension, which are the mental model and the cognitive model. A mental model describes the constructed combination of information contained in the source code and documentation with the assistance of experts and domain knowledge that the programmer brings into the task (Grubb & Takang, 2003). In other words, a mental model shows the maintainer’s mental representation of the program that needs to be understood. A cognitive model describes the processes and information structures used to form the mental model (O’Brien, 2003). De´tienne (2001) also reviews cognitive models and conducts the experiment in this area.

Although much research has been done, Corritore and Wiedenbeck (2001) indicated that studying program comprehension remains incomplete and it should be continued in order to produce the best strategies to improve program comprehension.

2.3.1 Cognitive models of program comprehension strategies

Many studies have been conducted to observe the process of how programmers understand the code. There are five cognitive models of program comprehension strategies: bottom-up (Shneiderman & Mayer, 1979), top-down (Brooks, 1983), the integrated approach (Von Mayrhauser & Vans, 1985), knowledge-based (Letovsky, 1986) as well as systematic and as-needed.

Shneiderman and Mayer (1979) suggested that some programs are understood from the bottom-up comprehension strategy where programmers read the source code by constructing a multilevel internal semantic structure to present the program. Low-level software artefacts are mentally chunked or the lines of code are grouped into meaningful high-level abstraction. Chunking is the process of recognizing the function of program components and fragments. These pieces are then grouped until understanding is formed. This strategy can help to improve program comprehension, especially to novices because users can focus on smaller programs.

Pennington (1987) observed how programmers understand a program by using the bottom-up strategy, which focuses on gathering statements and controlling flow information. In the bottom-up strategy, understanding the overall control flow is more important than understanding the function of programs. This strategy produces at least two mental models, which are the program model and domain model. The microstructure will be chunked and cross-referenced by macro structure to form a program model. The domain model relates objects and functions in the problem domain to language entity sources.

According to O’Brien (2003), the bottom-up model of program comprehension primarily addresses situations where the programmer is unfamiliar with the domain. Comprehending a program by using bottom-up strategy needs a mental model and cognitive model of a program. The process of chunking the source code will be based on the program domain. However, this strategy is not applicable to novices because they do not have the capability to determine the program domain.

The top-down strategy is the understanding by comprehending the top-level detail program, such as what it does and when it executes. It also includes the understanding of low-level details such as data type, control and data flow, and arithmetic patterns. Brooks (1983) proposed the top-down strategies in which the programmer develops a hierarchy of hypotheses concerning what the program does and how the program works. The verified hypotheses depend heavily on the presence and absence of guidance, where indicators present a particular structure or operation of the internal and external program.

According to Soloway and Ehrlich (1984), a top-down strategy is used when the syntax of the program is familiar to the programmer. They also observe how expert programmers recognize program plans and exploit programming conventions during comprehension. In this strategy, they determine the hypotheses to know the program domain. Users must select the beacons based on their knowledge foundation, mental model, and external representation. Thus, novices must have the ability to select the guidance and hypotheses.

Letovsky (1986) studied programmers who used either a bottom-up or top-down strategy to comprehend a program that is called a knowledge-based strategy. The author mentions that program understanding depends on the programmer's knowledge foundation and the assimilation process involving both top-down and bottom-up strategies. Von Mayrhauser and Vans (1985) suggested the integrated approached strategy to improve program comprehension.

Most users face difficulty in determining the flow of a program (Pennington, 1987), which causes them to fail in understanding what happens inside a program (Brusilovsky *et al.*, 2006). By using the combination of bottom-up, top-down, and external representation, this study is an attempt to reduce problems in comprehending a control flow of a program. The bottom-up strategy can be used to determine the flow of a program and the top-down strategy can be used to recognize the function and process of the program.

This study utilizes the combination of these strategies to visualize a control flow of control structures from the source code. Bottom-up strategy is used to determine the flow of a program and top-down strategy is used to show the flow of a program when users trace the program through the source code and to recognize the function or process of the program. This technique transforms the program source code into a graphical view. The user can trace the code through the graphical view to determine the component integration in the program. The user can follow the flow of a program through graphical visualization and further read the program's textual context that is provided so that it can improve viewing and readability, thus, making it easily understood.

2.4 Software Component

There are many different definitions for the term *component*. Among those commonly used is the adopted definition provided by Szyperski (1998, as cited in Briand et al., 2006; Hopkins, 2000; Inverardi et al., 2003; Wu, 2003) as, "A unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subjected to composition by third parties".

Two components are related if they can potentially react to the same events (Fiege, 2005), which is by passing the message through its interface when a component provides or requires specify events (Inverardi et al., 2003). The communication between components typically is realized by procedure calls or any kind of messaging (Bure et al., 2009).
When designing integrated systems, components are required to refer to other components using simple object-oriented techniques to create an interaction between the components. To detect the most wanted behaviours, components will need to call each other. When the interactions succeed in the dependence between components, it results in coupling, which prevents a separate compilation of integrated components (Rajan & Sullivan, 2005).

A component can be a single part of software that can be integrated to each other part. Szyperski (1998, as cited Reekie, 2002) defined software components as "binary units of independent production, acquisition, and deployment that interact to form a functioning system". Reekie (2002) clarified "binary" means any format that can be executed by a target machine. This may be serial coded for a specific processor, or virtual machine code, or in some cases even source code (as in some scripting languages).

Suman et al. (2009) clarified that the software components are independently developed *plug and play* parts of software. Components are increasingly being used in some situations to build critical applications such as safety critical real-time systems, embedded systems, and distributed control applications.

Components usually use more than one interface to manage interactions between the components. The interfaces provided by components are similar to service interfaces. As a result, the components can be candidates for implementing services where a service can be provided for the components to interact (Arafa et al., 2012).

According to Jin and Han (2005), the component services are able to use the services of a provider component based on its interaction definition without knowing its implementation details. To determine the definition of the component interaction, Broy and Kruger (1998) introduced a model to define the component interactions. For them, the interaction between an environment and its system can be described as a unique case of interaction between two components. The model used is shown below.

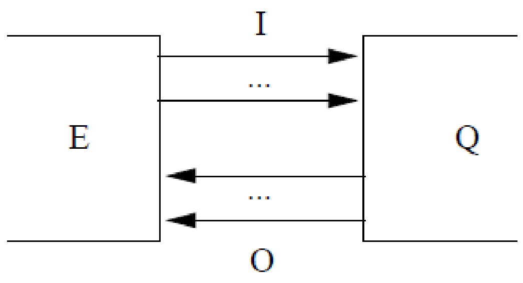


Figure 2.3. Graphical description of the interaction interface between two components (adopted from Broy & Kruger, 1998)

Based on Figure 2.3, they assumed that the components communicate to exchange messages via directed channels. As a result, they represent a component by a box labelled with the component's name. They represented channels by arrows directed from the source to the destination component. The set of channels can be communicated from left to right and vice versa.

In a software component, interaction is one of the foci in the development of a software system. It can be approached from different views including infrastructures and characteristics of individual components, which might support integrations (Rader, 1997).

To uncover errors in the interactions between their environment and components, the components need to be tested before use. The integration of a system must be assessed on the final platform, either when the system is modified or when the system is starting (Piel & Gonzalez-Sanchez, 2009). Therefore, in this study, the focus is to show the interaction between components. This is to ensure that the proposed module that was created will provide the right information concerning the software component interactions.

2.4.1 Existing study concerning software component extraction techniques

The common technique used during implementation and review to detect software implementation errors is by performing static analysis of the source code. The technique can also locate bugs before the program is executed. Another technique used in the area of quality assurance is model checking (Chen et al., 2004). Aside from that, static analysis also can find errors such as memory leaks, buffer overflows, timing anomalies (e.g., deadlocks, race conditions, and lovelocks), security vulnerabilities, and other general faults in programming mistakes (Schilling & Alam, 2008; Schilling, 2007).

Static program visualization is another type of program visualization, which refers to the process of visualizing aspects of a program without the need to run it. Examples in this field are call graphs, class diagrams, file explorers, and architecture diagrams. Visualizations of this type are used in several software engineering tasks. Dynamic program visualization is the process of visualizing the dynamic aspects of a program with moveable elements, which represents events or communication between the system components such as in algorithm animation tools (Ali, 2008). Static analysis also is defined as an automatic method to extract runtime properties of program source code without actually executing the program (Emanuelsson & Nilsson, 2008). It other words, other than to discover errors, static analysis also is helpful in producing code that is more efficient.

There are two important characteristics of static analysis, which are soundness and completeness. A static analysis tool is considered sound if it never gives false warnings. Aside from that, the completeness of an analysis tool can be defined where every fault is detected when static analysis is performed on a given source code (Schilling, 2007).

Three steps in achieving static analysis of a binary executable were defined by Bergeron et al. (2001), creation of an intermediary representation, flow-based analysis that captures security-oriented program behaviour, and static verification of critical behaviours against security policies (model checking). Bergeron et al. examined the advantages of static analysis over dynamic analysis as follows.

Table 2.1

Advantages of static analysis and dynamic extraction analysis

Advantages of static analysis	Advantages of dynamic analysis
• Static analysis techniques allow complete analysis, because they are not bound to a specific execution of a program and can give guarantees that apply to *all* executions of the program.	• Dynamic analysis techniques only allow examination of behaviours that correspond to selected test cases.
• Judgment can be given before execution.	• Difficult to determine the proper action to take in the existence of weak.
• There is no run-time overhead.	• Perform on execution programs.

For a new static program analysis technique, Scholz et al. (2008) studied the area for locating user input dependencies in programs using graph reachability. It was decided that the static program analysis can consider control dependencies and the analysis can consider all paths in the program, whereas dynamic program analysis exercises a single execution path.

Murata et al. (2006) addressed the benefits of static analysis in their new approach for XML access control, which helps developers to ensure that their queries do not cause access-control errors. Since access-denied *XPath* expressions in queries can be rewritten as empty lists at compile time, static analysis facilitates query optimization. In addition, static analysis guarantees complete automation, scalability, and handles larger classes of properties and larger classes of systems (Saidi, 2008).

Static analysis is a method of computer program debugging that is conducted by examining the code without executing the program. The process provides an understanding of the code structure and can help to ensure that the code adheres to industry standards (Ayewah et al., 2008). For that reason, many static analysis tools were used in diverse areas including in critical source code development such as aircraft and rail transit. These static analysis tools are able to detect software faults within certain industrial segments (Schilling & Alam, 2008).

For this study, static analysis is chosen as a technique to process an input. This technique is chosen because it offers a technique for predicting properties of the behaviour of programs without running them. Different from dynamic analysis, the process of visualizing the dynamic aspects of a program with moveable elements, which represents events or communication between the system components such as in algorithm animation tools (Ali, 2008). As an objective of this study is only to visualize the component information, the static analysis is a suitable technique used in extracting the component integration.

2.5 Existing studies on related component extraction tools

To predict whether a program is compliant with component-based development, reverse engineering software named Software Model Extractor (SOMOX), which is used by Royer (2010), can be applied. SOMOX is reverse engineering software for component models. To perform the component extraction, SOMOX has a certain set of rules to be followed, which involves rules to define the basic components, composite structures, interfaces, ports, and connectors in extracting data (somox.org). SOMOX is more focused on identifying the static structure of the component model. It is suitable to enhance the understandability of existing software by extracted the models of software. Aside from this, SOMOX also provided a part of a complementary tool to analyse software reliability, performance, and maintenance. The extraction of the software in SOMOX is based on the evaluation of multiple source code metrics.

For individual projects, the extraction can be guided by metric settings, which allow the adaptation of SOMOX to the specific component definitions. Currently, SOMOX supports source code for C/C++, Delphi, and Java. However, its general source code model can be extended easily to other programming languages (somox.org).

The Java Component Extractor (JCE) is a re-engineering tool, which is able to perform component extraction to improve the component structure. It is also able to show the communications between components in terms of what the services involved in the component interactions are (Royer, 2010).

JCE is more focused on the component model, which is implemented by Java classes. There are sets of rules supported by this tool, such as checking inner classes, checking array, and generic usages. This set of rules is applied to avoid components escaping from their enclosing parents. JCE provides information and graphical views related to component quality of the code, component and data types, component structures, and communications, which are suitable to be used in bug finding. The extraction of the software in JCE is based on the type of component from a Java source code. For individual projects, the JCE plugin allows a user to enforce the choice of component and data types. The plugin processes choose packages and compute some data types. This is useful to gain some understanding of what could happen in a component-oriented restructuring process. The resulting information is shown in some textual views when running the extractor. Currently, JCE supports source code only for Java language (emn.fr/z-info/jroyer/JCE/index.html).

2.5.1 Appropriate tool to be used

For this study, Graphviz is an appropriate tool to be used because it is able to accept the dot format that was created from the component extractor tool. This shows that the dot file that is produced from this research is valid and capable of representing a call graph to archive the objective of this research. Graphviz is open source graph visualization software. It represents the structural information as diagrams of abstract graphs and networks.

An in-depth comparison with SOMOX and JCE was done by Arboleda and Royer

(2011) to verify that both tools do not extract the same information. SOMOX and JCE are likely the same, (i.e., able to produce component types, primitive, composite, and data types, and can analyse Java source code). However, SOMOX is a component extractor or recovery tool, which is able to suggest component types in a source code, even if it is a pure object-oriented programming application. However, JCE is more specific. It relies on a component model with a true first-class component, that is, a component has types, which are implemented by Java classes. It also supports a notion of subtyping, abstract, and concrete component types. SOMOX is a metrics-based tool and it clusters some parts of the application, which could become true component types in a new restructuring step (Arboleda & Royer, 2011). Usually, most of the component recovery tools and architecture compliance checkers consider that a combination of components is implemented by packages (http://www.emn.fr/z-info/jroyer/JCE/index.html.

2.6 Software representation

Software visualization is a broad research area involving a large number of visualization terms and techniques. The term *software visualization* has many meanings (Ali, 2008). There are different terms in software visualization and *program visualization*; the term *software* includes documentation with the source code while the term *program* refers only to the source code.

A definition of software visualization suggested by Diehl (2007) and another for program visualization defined by Price (1992) will be used. Diehl defined software visualization as “the art and science of generating visual representations of various aspects of software and its development process”. Price defined program visualization as “the visualization of the actual program code or data structures in either static or dynamic form”. Software visualization can be seen as a specialized subset of information visualization because software visualization is the process of creating graphical representations of abstracts; while program visualization focuses on improving program presentation and representation techniques. Considering these two definitions, it can be concluded that software visualization is a generic field that contains all visualization classes while program visualization is a specific part of this field.

Generally, *presentation* refers to viewing of the source code while *program representation* is the visualization of aspects of the code. Source code highlighting and a call graph are two typical examples of program presentation and representation. The ultimate objective of program visualization is to enhance the developer’s understanding of various aspects of the source code under investigation. It can enable the software engineer visually to perceive program features that are hidden or difficult to obtain without the visualization (Ali, 2008).

Software visualization supports software engineers coping with software complexity and reduces understanding time by supporting the cognition process during development and evolution of the software (Ali, 2008). Therefore, by using program visualization to present and request component interactions, it can help the software developer to understand better the components and interactions involved at the component level.

2.6.1 Techniques Representation

There are two techniques in representing software, which are visual modelling and program visualization. The visual modelling field involves the visualizations that help software engineers in developing new software by reducing time and complexity of engineering tasks. Hence, these visualizations are mostly used in the later phases of software development. The concept of visualization can be thought of as the process of visualizing ideas, plans, analyses, processes, characters, and aspects of a system to model the system. The details of these techniques are shown in Figure 2.5.

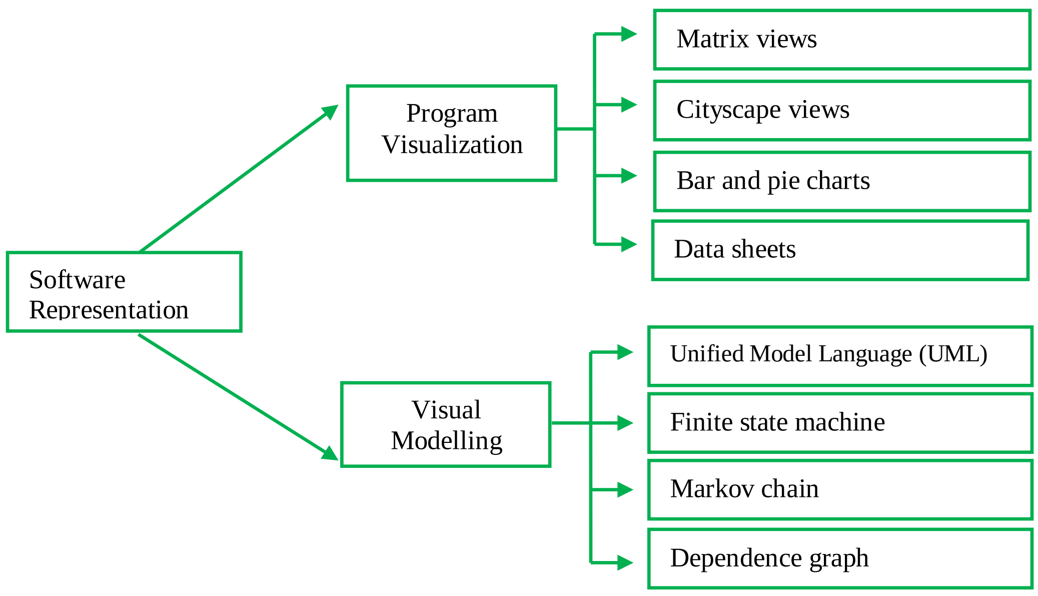


Figure 2.5. Different techniques in representing software

Based on Figure 2.5, to identify the software component representations, two different techniques have been studied. These are program visualization and visual modelling. The details of these two areas are discussed in the following sections.

2.6.2 Program Visualization

Program visualization or principal visualization metaphors are an effective visual representation to represent the software components. The five primary forms of visualization are matrix views, cityscape views, bar and pie charts, data sheets and network views that are related to components of software structure (Eick et al., 2002; Lanza, 2001). It collects the data about the software routinely and shows the components based on colours; different aspects of the data components will use a different visual metaphor for each. The details of each form of representation software are based on the analysis done by Eick and others (2002). Because of printing limitations, colour images are rendered in grey scale.

Matrix views

Matrix views are useful ways for displaying one or more responses as a function of two categorical indices. The view itself, as in Figure 2.6, is a two-dimensional grid with rows corresponding to one index and columns to the other. The strength of matrix display is that many cells are visible; also, there is no over-plotting.

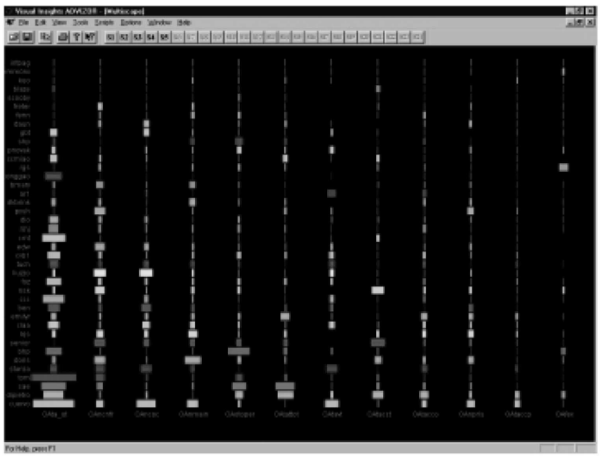


Figure 2.6. Example of a matrix view showing component patterns for the top developers (adopted from Eick et al., 2002).*Note*: Because of printing limitations, colour images are rendered in grey scale.

Figure 2.6 shows rows that represent developers and columns that represent software spaces at the level of aggregation of modules. The bar width represents module size and bar colour represents the developer.

Cityscape views

Cityscape views, also known as three-dimensional bar charts, are three-dimensional extensions of matrix views. There are two indices and one or more responses. The cityscape view has the same representation as a matrix view, which uses high vertical towers to represent the number of components made by the software engineer. Compared to matrix views, cityscape uses 3D towers to represent information, which provided more pixels per cell than a matrix view. An example of one of the views is shown in Figure 2.7.

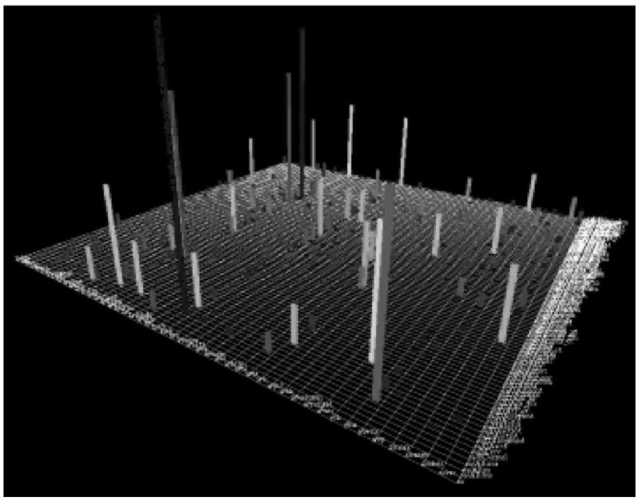


Figure 2.7. Example of a cityscape view (adopted from Eick et al., 2002). Note: Because of printing limitations, colour images are rendered in grey scale.

The cityscape view in Figure 2.7 uses the heights of vertical towers to represent the number of components, which has the same indices as the matrix view in Figure 2.6. This view shows patterns of software components made by each developer. The rows represent developers and the columns are software space at the level of aggregation of modules. Even the colour of the cityscape view is redundant, but it is effective to represent the number of components.

Bar charts and pie charts

Bar charts and pie charts are views that differ from matrices and cityscape views. Bar charts are most effective as selectors linked to other views in many perspectives. Aside from this, mostly bar charts and pie charts are based on colour code in representing the components of software.

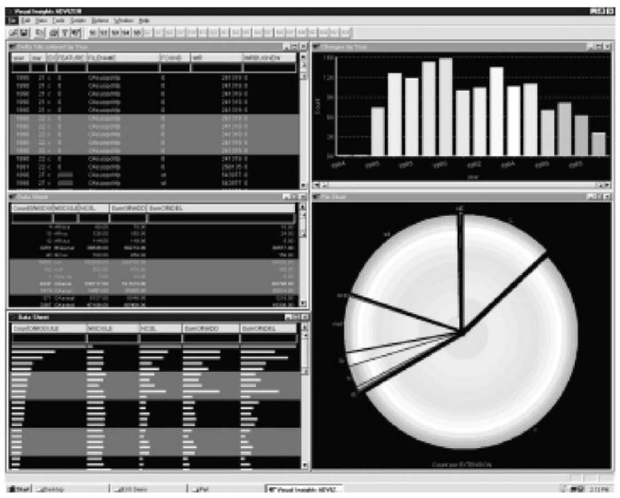


Figure 2.8. Example of a bar chart: numbers of software changes by year (top right). Pie chart: number of changes by file type (bottom right) (adopted from Eick et al., 2002). Note: Because of printing limitations, colour images are rendered in grey scale.

Figure 2.8 shows the simple illustrative example of bar chart and pie chart views, which show the yearly component changes of file type. The numbers of components are shown based on the colour representation.

Data sheets views

A data sheets view is a scrollable text visualization (Eick et al., 2002), a representation of data that may include structures, attributes, and relationships of the software components. Figure 2.8 is a simple illustrative example of the data sheets view (left); individual deltas (top); components aggregated by module (middle); and subsystem (bottom).

2.6.3 Technique for program visualization

By using different techniques, software component representation allows for critical code review, which can support relatively easy formalization and understanding.

Program visualization metaphors are not suitable for this research because each representation is generated to represent software components at the data level, especially for exploratory analysis, which is not at the component level. Additionally, the purpose of these visualizations is focusing on understanding the components of software, which are component time, component type, and software space (Eick et al., 2002). This research is only focused on how to represent component interactions in software components without considering any dimension or time.

2.6.3.1 Visual Model

A model is an overview of an idea. It uses exactly defined notation to describe and simplify a complex and important phenomenon, structure, or relationship of the system. Models of software systems help developers to visualize, communicate, and validate a system before spending greater amounts of money. They also help a developer structure and coordinate the efforts of a software development team (Arrington, 2001).

The model of software is necessary for the development of complex and large systems and it is very useful when dealing with code directly. Aside from this, software models are abstractions from code. They can serve as inputs for program generators and provide documentation to developers (Clauß, 2001). The main purpose of engineering models is to make it possible for a developer to understand the important aspects of a complex system before constructing them.

The quality of the model can help a developer concerning features of a system where there is uncertainty, either about requirements or about the capability of a proposed solution (Lavagno et al., 2003). For this research, there are four models to be studied to identify the best technique to choose for representing component interactions in software, which are Unified Model Language (UML), finite state machine, Markov chain, and dependence graph.

Unified Model Language (UML)

UML is a standardized notation for specifying object-oriented software systems and designs (Lilius & Paltor, 1999; Clauß, 2001; Arrington, 2001; Bruegge & Dutoit, 2009). The attributes and behaviours commonly show a group of objects that are documented in a class diagram, which documents structures, behaviours, and usage of the software system.

Although the model is easy to use and apply, some issues still need to be managed. If the model fails to support version management in multi-site or in multi-partner development environments, the modelling process may be ineffective and the modelling tool may not be used optimally. Ineffective tool deployment is expensive since there will be substantial costs without realizing the potential benefits (Koivulahti-Ojala & Käkölä, 2010).

Finite State Machine

Finite State Machine (FSM) is a model containing inputs and outputs (Suman & Mall, 2009). After receiving an input on state transition, FSM will produce a finite number of states and produce an output. FSM is represented by a state transition diagram. The diagram contains a direct graph, which is communicated to the states of the machine. Each edge matches the transition's state where it is recognized with the input and output related with the transition (Lee & Yannakakis, 1996).

Markov Chain

A Markov chain is defined as a visible model of the software that is involved in the application of inputs. It is a set of states, arcs, and usage (i.e., transition) probabilities associated with a software program. The probabilities in the Markov chain are presented in state transitions. This model contains two states, which are the invoke state and terminate state. The invoke state represents the software prior to the invoke and the terminate state represents the software after execution has terminated. The Markov chain transactions are established and suitable to estimate real usage of a model (Prowell, 2005).

Dependence Graph

A dependence graph represents dependencies between operations in a program (Jackson & Rollins, 1994). The model has nodes and edges to represent its dependencies. The nodes of the graph represent functions in the program, and edges connecting the nodes represent call paths in the program (Kaeli & Calder, 1997). Furthermore, the dependence graph also can be used to determine which functions are called by a particular function. In addition, the dependence graph also can be generated by using a dot file, which is a file that can be manipulated and reused by users for their own purposes.

2.6.4 Technique for model visualization

As a conclusion, to show the components in a program, programmers must understand its operations. Understanding the operations is one of the most time-consuming activities, especially when the programs are complex, and relevant information must be extracted from the system.

For this research, the technique to visualize a model is chosen to represent the component interactions in software. This is because it can help developers to understand the important aspects of a complex system. These models are Unified Model Language (UML), Finite State Machine, Markov Chain, and Dependence Graph. Based on these four models, this researcher decided to choose a dependence graph. This is because it is a directed graph representing dependencies of several components towards each other. In addition, it can also model the relationship of component interactions with their traces. Aside from that, the dependence graph focuses on the interactions and dependencies of the components in the software system while UML only focuses on the cooperation between objects.

Furthermore, the dependence graph can describe static representation, while FSM is more suitable to describe models for run time system communications (Iwanari et al., 2009). In representing the software components, especially in the component interactions, this research is not intended to consider probability interactions associated with the program, while this is the focus of a Markov chain.

The studies concerning properties of the dependence graph are referred to as graph theory, which has been widely used as a core subject computer science (Chatzigeorgiou et al., 2006) and in diverse areas such as in mathematics, engineering, physical, social, biological sciences, linguistics, and others (Deo, 2004). In addition, it can be helpful in many ways for understanding the characteristics of the software systems (Chatzigeorgiou et al., 2006), as well as in almost any physical situation to model the greatest pair of relations between objects from a certain collection (Deo, 2004).

In conclusion, a dependence graph will be adapted in this research to represent the component interactions in software, to provide a direct graph, which is easy to understand. In addition, a dependence call graph can also be represented in *dot* file format, which is easily manipulated and used.

2.7 Dependence Graph

The study of the dependence graph is referred to Bollobás (1998) study, which is based on the basic concept of graph theory. He defined a graph *G* as an ordered pair of disjoint sets *(V, E)* such that *E* is a subset of the V^2 of unordered pairs of *V*. Unless it is explicitly stated otherwise. He considered only a finite graph, that is, *V* and *E* are always finite. The set V is the set of vertices and E is the set of edges. If *G* is a graph, then *V* = *V (G)* is the vertex of *G*, and *E = E (G)* is the edge set. An edge *{x, y}* is said to join the vertices x and y and denoted by *xy*. Thus *xy* means exactly the same edge; the vertices *x* and *y* are end vertices of this edge. If $xy \in E(G)$, then x and y are neighbouring, vertices of *G*, and the vertices *x* and *y* are incident with the edge *xy*. Two edges are adjacent if they have exactly one common end vertex. The diagram of the basic concept of a dependence graph is shown in Figure 2.9.

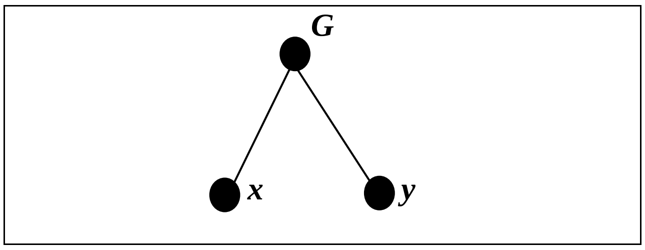


Figure 2.9. Basic concept of dependence graph

Figure 2.9 shows the basic concept of a dependence graph as defined by Bollobás (1998). Aside from this, there are many types of dependence graph representations. Table 2.2 is a summary of the different types of dependence graph representations, which are known as intermediate representations.

Table 2.2

Types of dependence graph representation

Types of graph	Features
Call graph (CG) (Ottenstein, 1978)	CG represents calling relationships between various modules of a program.
Control flow graph (CFG) (Ottenstein & Ottenstein, 1984)	CFG shows the flow of control between several program elements during execution time.
Data flow graph (DFG) (Ottenstein & Ottenstein, 1984)	DFG shows information of data flow between various program elements.
Program dependence graph (PDG) (Najumudheen et al., 2009)	PDG considers a single procedure where it captures both control and data dependences.
System dependence graph (SDG) (Najumudheen et al., 2009)	SDG represents dependences and procedure calls between multiple procedures.

Based on Table 2.2, there are five different types of dependence graphs summarized in this section: call graph, control flow graph, data flow graph, program dependence graph and system dependence graph. For this research, a call graph is used to represent the software components.

It is because a CG can represent calling relationships between various modules of a program, especially on a component level, without considered execution time such as CFG, while DFG, SDG and PDG are not appropriate because they consider the relationship on data flow of program elements.

Additionally, a call graph is suitable to be used to represent connectivity of interactions between the components in their relationships of a program. Aside from this, the call graph also is suitable in analyzing tracks of the flow's values between various modules of a program. By constructing a call graph, nodes of the graph represent functions in the program, and edges connecting the nodes represent call paths (Kaeli & Calder, 1997). The issue that needs to be considered when constructing this call graph is how the call graph can be viewed as a simple visual representation so that people will be able to read through the program easily.

2.7.1 Call graph

A call graph depicts the flow of control between various program elements. Aside from that, it also represents the program structure and defines possible execution paths. Each path through the graph is a possible path of execution in a process of a program. The nodes in the graph are linked to possible branch points of the program. The edges of the graph will work as a system call among the nodes (Xue et al., 2009). It has a simple visual representation of the calling relationships between functions or procedures. Furthermore, the call graph also can be used to determine which functions are called by a particular function.

It is important to note that the call graph is directed from caller to callee. Specifically, each node represents a procedure and each edge (*a*,*b*) indicates that procedure *a* calls procedure *b*. Thus, a cycle in the graph indicates recursive procedure calls. A call graph is a basic program analysis result that can be used for human understanding of programs, or as a basis for future analysis (Najumudheen et al., 2009).

The call graph is one form of useful analysis representation to extract executions. It has a simple visual representation of the calling relationships between functions or procedures. Furthermore, the call graph also can be used to determine which functions are called by a particular function. When trying to understand a system, the call graph is chosen in software engineering to ensure that the functions of the system are executed correctly.

The program call graph is a directed graph that represents the calling relationships between the program procedures. There are two types of program procedures, contact-insensitive and contact-sensitive (Grove & Chambers, 2001; Grove et al., 1997). Figure 2.10 (a) and Figure 2.10 (b) show the differences between these two procedures adopted from Grove and Chambers (2001).

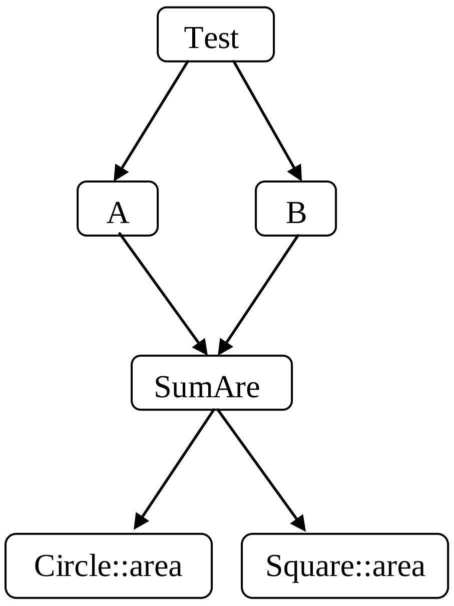


Figure 2.10(a). Contact-Insensitive

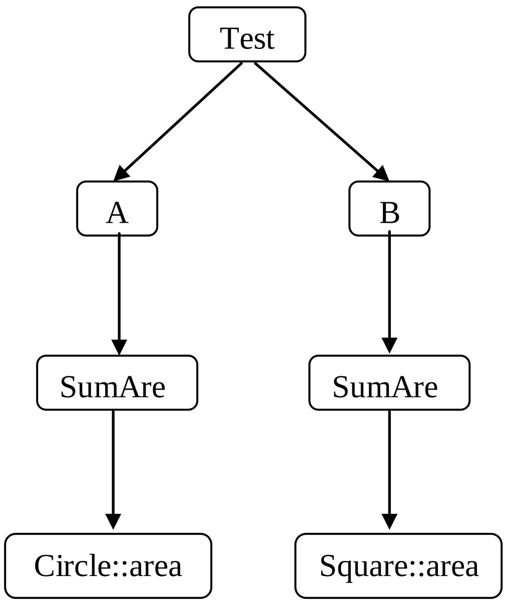


Figure 2.10(b). Contact-Sensitive

Figure 2.10(a) shows the context-insensitive call graph. A context-insensitive analysis can be modelled by restricting the call graph. It has only a single line for each source-level procedure. It is shown that the two nodes (*A* and *B*) are pointed to the same operation of the SumArea. Figure 2.10(b) describes a context-sensitive call graph. This figure shows two nodes (*A* and *B*) that are pointed to different operations of the SumArea. From there, it has created additional lines to separate the flow of Circle and Square objects (Grove & Chambers, 2001). For this research, both of the program procedures will be concerned.

Call graphs can represent static and dynamic information. A static call graph is a call graph intended to represent every possible run of the program. A dynamic call graph records an execution of the program such as output by a profiler. Thus, a dynamic call graph can be exacted, but only describes run time of the program (Najumudheen et al., 2009).

The static call graph can be constructed from the source text of the program. However, discovering the static call graph from the source text would involve two difficult steps: finding the source text for the program, (which may sometimes not be available), scanning, and parsing the code, which may be written in any one of several languages. However, in some circumstances where source code is available, the graph is still very difficult to obtain. This is because great understanding is needed from the programmer to recreate the call graph from an observed system call trace, which requires time and expertise to complete the task (Xue et al., 2009).

2.7.2 Related studies using call graph

The call graph representation in this study represents components in the software version at the component level. The call graph has been very popular in the reverse engineering literature and tools (Wen & Dromey, 2004).

Table 2.3

Existing tools for software representation using graph

Name	Description	Comment
GEVOL (Graph-Based Visualization of the Evolution of Software) Source: (Collberg et al., 2003)	• GEVOL extracts information from Java programs that are stored within a CVS version control system. • It extracts inheritance graphs, call graphs, and control-flow graphs of the program. • It displays the graphs from the beginning of the program.	GEVOL is a tool that is related to this study. However, the source code of this tool is not available to the public.
VIFOR (Visual Interactive Fortran) Source: (Knight and Munro, 2001)	• VIFOR works by using a database of code detail from FORTRAN code and then allows it to be viewed and queried. • Programs can be displayed and edited in two forms: as code and graph. • Hence, it is suitable for re-engineering and maintenance of existing code.	The language used by VIFOR is Fortran, which is not meeting the requirement of this research. This research will use Java language.
RIGI (Reverse Engineering tool suite) Source: (Martin et al., 2000)	• RIGI includes parsers to read the source code of the subject software. • Extracted artefacts such as procedures, variables, calls, and data are accessed into the RIGI file format. Uses simple file format to represent graphs. • Manage the complexity of the graph produce.	The language used by RIGI is C, C++, and COBOL.
PROBE (Program behaviour) Source: (Lhoták, 2007)	• Call graph comparison tool to aid in finding the root causes of call graph differences. • Compares two call graphs and reports their differences. • Language: Java	It is unidirectional tool: it reports only methods and edges present in the first graph and absent in the second graph.

SHriMP (Hierarchical Multi-Perspective) Source: (Storey and Müller,1995)	• Displays software architectural diagrams using nested graphs. • Designed to enhance how people browse, explore, and interact with complex information spaces. • Combines a hypertext following metaphor with animated panning and zooming motions over the nested graph to provide continuous orientation and contextual cues for the user. • Language: C and C++	The language used by SHriMP is C and C++.

Although there are many existing tools that can support the software representation, especially in the form of a call graph, many problems still arise, such as the unfamiliarity of the language used, flow of the tool's operations and the source code of the tool, especially when modifications must be done on the tool to meet the requirement of this study.

Moreover, the domain of this study is only at the component level, where tools that were studied are not focusing only on that component. The components are often related to each other, by data flow, control flow, or definition-reference relationship (Chen & Rajlich, 2001; Rajlich, 1997).

2.7.3 Call graph techniques

In representing software in the form of a call graph, Wen and Dromey (2004) proposed a formal method for mapping software in system functional requirements to changes in the behaviour of individual components, and their interfaces. They used a design behaviour tree (DBT) to show different parts of the components. To identify the changes, a new DBT and an old DBT will be compared to discover the following:

- new behaviours that are introduced into the new tree,
- behaviours in the old tree but not in the new tree, and

- behaviours unchanged in the two trees.

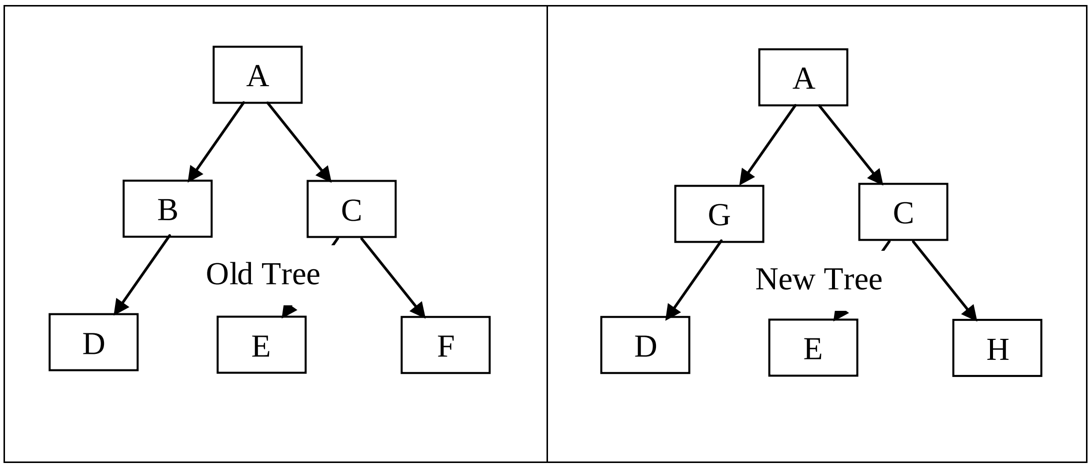


Figure 2.11. Design Behaviour Tree (DBT)

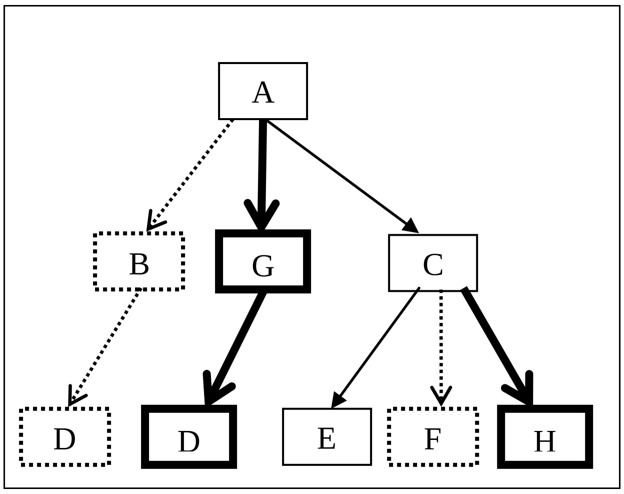


Figure 2.12. Edit behaviour tree (EBT)

After the newly designed behaviour tree and the original are compared and merged, an edit behaviour tree (EBT) is created to record any addition, deletion, or modification and to show the unchanged functional requirements. Figure 2.11 shows the old and new DBT. The EBT produced from Figure 2.11 is shown in Figure 2.12. The changes show:

- The new part of the tree is drawn with bold lines.

- The old part of the tree is drawn with dotted lines.
- The unchanged part is drawn in the normal style.

Lin et al. (2009) introduced a new source-code-level (static) call graph construction approach for AspectJ software. To update the call graph in an attempt to perform an amount of work relative to the source changes, the results from their previous analysis are reused. To begin the experiment, they assume that a call graph has been fully constructed for the initial software version, and after a session of source modifications, the incremental call graph construction algorithm is invoked in accordance with program changes without global reanalysis.

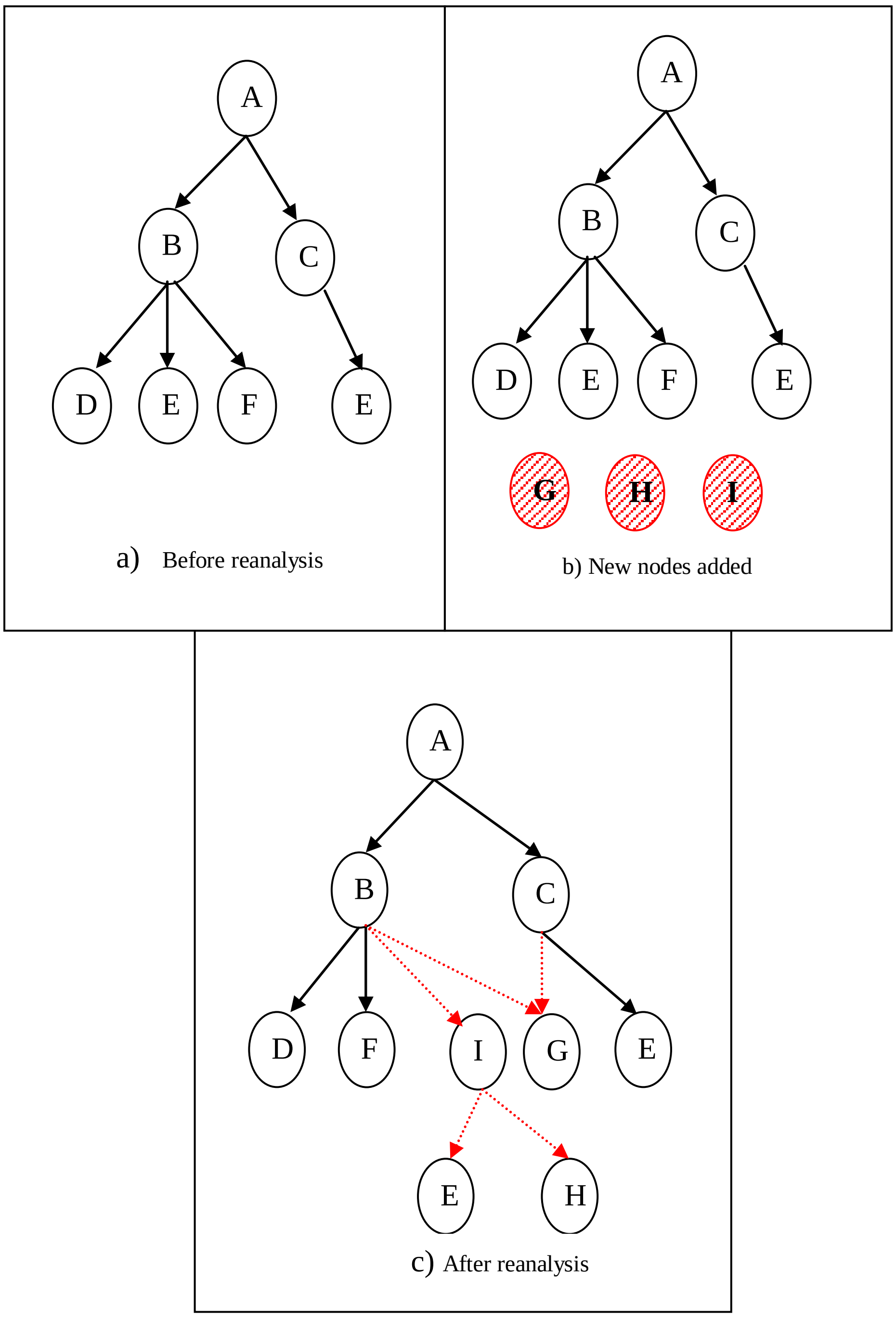


Figure 2.13. Representing change nodes adopted from Lin et al. (2009)

Figure 2.13(a) shows a call graph, which represents the original nodes before reanalysis. When new nodes are added, Figure 2.13(b) shows the initial constructed call graph before the original source is edited. The newly additional nodes are shown in red shadow. After reanalysis, the updated nodes in the red dotted lines are linked as shown in Figure 2.13(c).

2.7.4 Call Graph Interpretation

To support the understanding of the graphical representation, the graph can be interpreted in a set of statements that evaluate the expressions of a dependence graph (Xue et al., 2009), which allows description of relations of the nodes. This expression can recognize the order of operation, recognize its equivalent, and understand the sign of the symbol.

Kuck et al. (1981) transformed the call graph into an expression language. Their study is focused on the transformations of dependence call graph optimization to enhance the performance of the programs. A simple example of the transformation is shown in Figure 2.14.

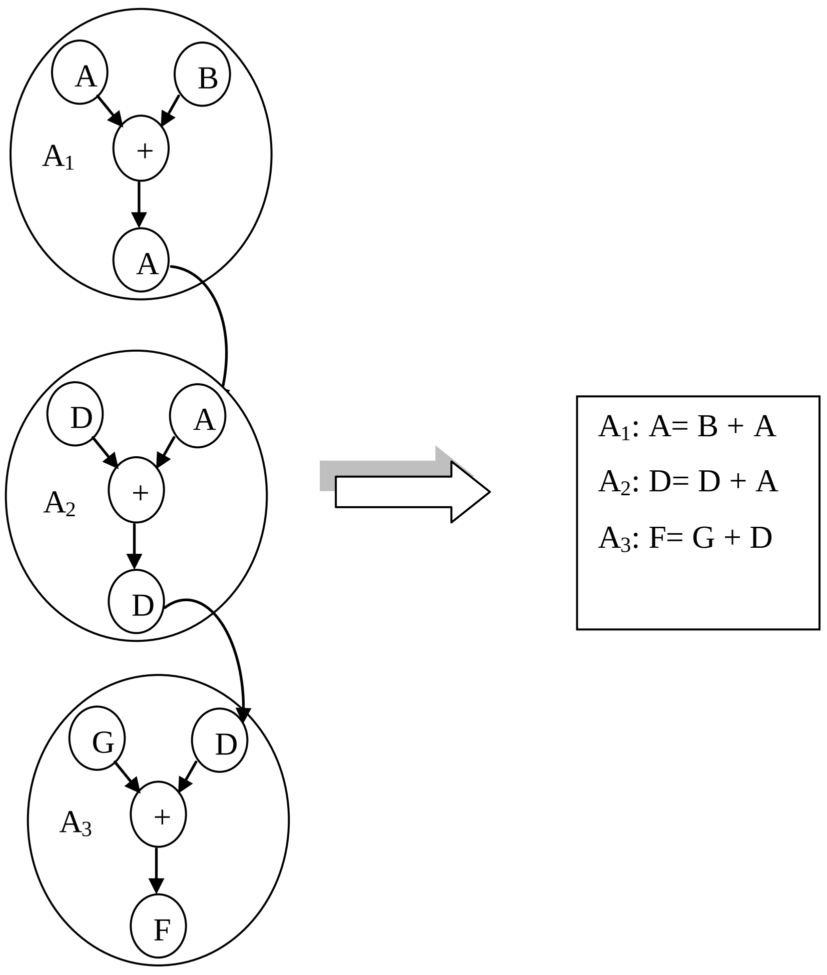


Figure 2.14. Transformation dependence call graph to expression language (adapted from Kuck et al., 1981)

The transformation shown in Figure 2.14 is explained as a process by which a set of nodes and their external nodes are connected to each other. The program is divided into three parts of programs, *A1*, *A2*, and *A3*. *A*, *B*, *D*, *F*, and *G* represent nodes, which have mathematical functions that have connections. The operation plus (+) represents the larger node. For this operation, program *A1* represents node *A* that is connected to node *B*. As a result, node *A* is larger than node *B*. For the next operation, node *A* is connected to program *A2*, where node *A* is connected to node *D*. As a result, node *D* is larger than node *A*. The same operation occurs in program *A3*. These notations are commonly displayed after the call graph is constructed.

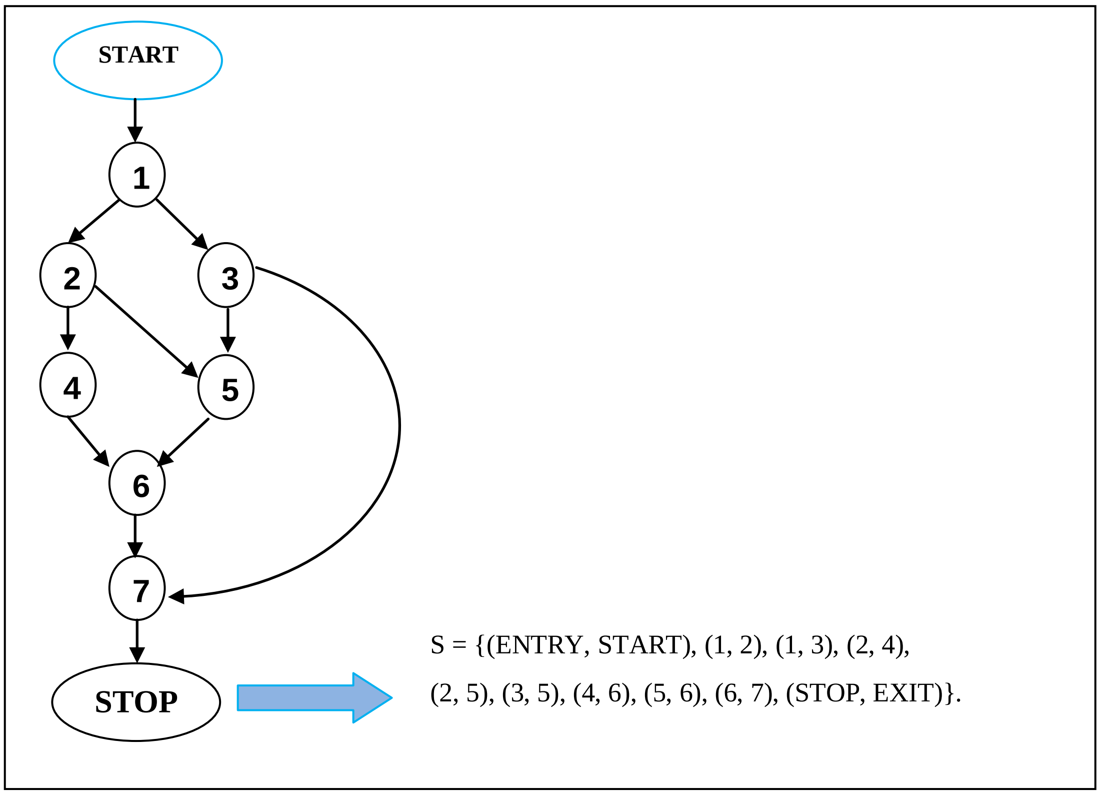

START
1
2
3
4
5
6
7
STOP
S = {(ENTRY, START), (1, 2), (1, 3), (2, 4),
(2, 5), (3, 5), (4, 6), (5, 6), (6, 7), (STOP, EXIT)}.

Figure 2.15. Transformation of dependence call graph (adapted from Ferrante et al., 1987)

Ferrante and others (1987) show their transformation of nodes slightly differently. They assigned the start node and stop node in the form of a call dependence graph, which construct the call notations of the process in a range of brackets. Figure 2.16 is an explanation of the process of the call graph and represents it in the form of call notation. Node start, *1, 2, 3, 4, 5, 6, 7,* and stop are nodes that represent functions in a program. The line in the graph represents the connection between nodes. The graph shows the called node begins with the start node to node *1*, node *1* called node *2* and *3* and continued until the path shows the stop node. The notation is constructed based on a call graph, which is displayed in Figure 2.15.

2.7.5 Application model

A call graph is much easier for developers to build, extend, or evaluate visualization than to work directly on source code (Koivulahti-Ojala & Käkölä, 2010). Moreover, software representation significantly decreases the time required to learn the system and code review (Arrington, 2001). Some particular pieces of a software representation should be reviewed for quality, understandability, and consistency with the rest of the representation.

Preparation time for review of a representation is dramatically lowered when compared to the code walkthrough. In general, teamwork and review of software representations lead to lower defect rates and fewer difficulties during integration. On the lowest level of the components are components without dependencies to another component (Crnkovic, 1991).

2.8 Graphviz as a visualization tool

Graphviz or Graph Visualization is an open source graph visualization tool developed at AT&T used for evaluation in this research (Gansner, 2012). The dot language is preferred because it can specify three types of objects: graphs, nodes, and edges.

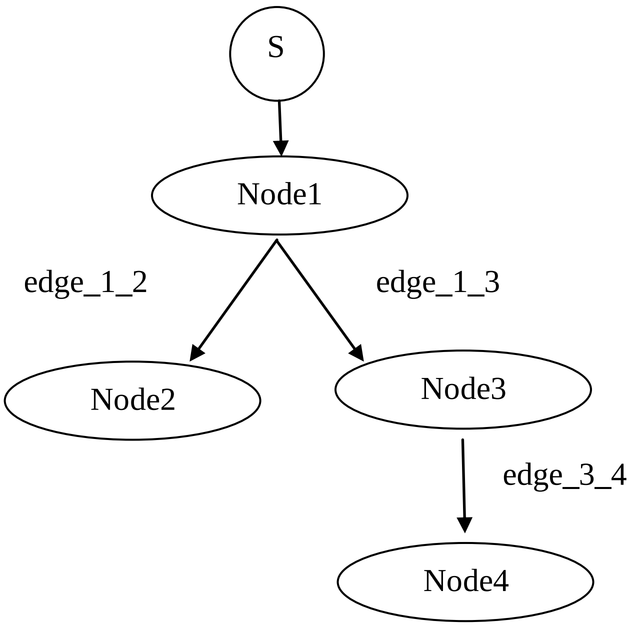
Listing 1 is an example that illustrates all three elements that are built to understand how these objects are represented. This representation presents a simple directed graph consisting of three nodes in dot notation. Line 1 declares a graph, called *G*, and its type (a digraph). The next three lines create the nodes of the graph, named *node1*, *node2*, and *node3*. Nodes are created when their names appear in the graph specification. Edges are created when two nodes are joined together by the edge operator (->), as shown in lines 6-8. An optional attribute is applied to the edge -- label -- that names the edge on the graph. Finally, the graph specification is completed at line 9.

Listing 1: Sample graph in Dot notation (test.dot)

```
1: digraph G {
2: node1;
3: node2;
4: node3;
5:
6: node1 -> node2 [label="edge_1_2"];
7: node1 -> node3 [label="edge_1_3"];
8: node3 -> node4 [label="edge_3_4"];
9: }
```

To translate this dot file into a graph image, the dot utility is used, which is provided in the Graphviz package. Listing 2 shows the image translation of this test.dot file.

Listing 2: Graph visualization



Listing 3: Example command prompt

```
C:\Program Files\Graphviz 2.28\bin>sfdp –Tpdf c:\test.dot > c:\test.pdf
```

2.8.1.1 Model elaboration

To elaborate the model, this section illustrates object representation based on Graphviz representation. The figure below shows the graphical representation for research of our model, which is adopted from Gansner (2012).

2.8.1.2 Representation for vertices and edges

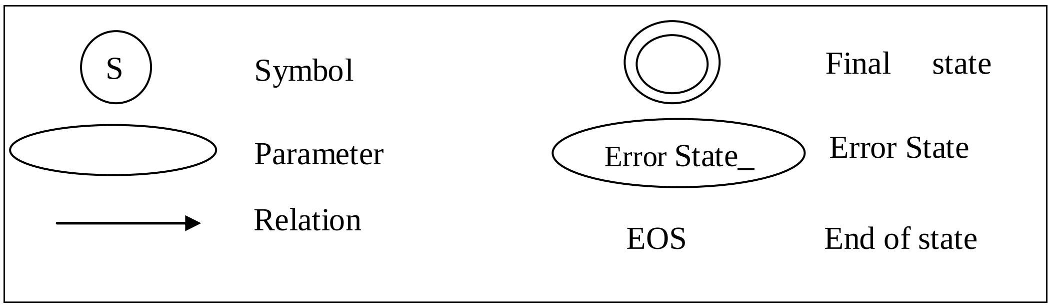


Figure 2.4. Graphical representation for call graph

Figure 2.4 shows the representation for vertices and edges for a call graph that was produced by Graphviz. ***S*** symbol shows the start state of the model. The parameter symbols show the parameter name of the state for the model. The relation symbol shows the parameter (i.e., the interface) invoked that is executed by a service. The last state symbol shows the final state of the model. The error state symbol shows the state that was no relation. The symbol ***EOS*** shows the last traces of interfaces invoked in a service.

2.9 Summary

This chapter is an extensive review of literature related to software comprehension and their cognitive models of program comprehension strategy, software components, component interactions, and software visualization. In this chapter, software and program visualization were examined to define the details about the techniques in representing software. Aside from this, the dependence graph concept was also examined to study in-depth a call graph and research related to a call graph. Because the domain of study is a component, study concerning a software component and its extraction techniques also are covered.

The first topic in this chapter is software maintenance. In conducting this research, the area of software engineering is considered where software maintenance is a sub-element of study, which is considered the area in the technique for maintenance. For this study, only the area of program comprehension and the reverse engineering technique are applied.

This chapter also is a review of several issues that can be addressed according to this research such as software representation, component integration, and existing software. It is conducted to gain more understanding especially to emphasize the issues concerning how to represent component interaction in software in the form of a call graph.

To represent software components, the dependence graph technique is chosen, as it is suitable to represent the connectivity of interactions between the components in their relationships with a program. Based on literature review, there are many types of dependence graph representations. A call graph is one of the dependence graphs that chosen to represent the software components because it depicts the flow of control between various program elements. Aside from that, it can also represent the program structure and define possible execution paths. To construct a call graph, this chapter covered software components as a domain of this research.

Call graphs that belong to the field of static program visualization will be used in the maintenance phase to support the software-understanding process, which is in the area of techniques for maintenance. This process requires tools to help in automatically running code to generate a result. For the time being, only one type of area in the technique for maintenance will be considered, which is reverse engineering.

Specifically, static analysis is chosen in this chapter as a technique to process an input. This technique is chosen because it offers a technique for predicting properties of the behaviour of programs without running them.

CHAPTER THREE

RESEARCH METHODOLOGY

3.1 Introduction

This chapter is a discussion of the research methodology used in this study. Research methodology is a disciplined way to solve the problem of research, which provides a collection of methods to conduct research in a specific sector (Kothari, 1985).

3.2 Research Design

The methodology applied in this study is based on a general methodology in research design as proposed by Vaishnavi and Kuechler (2008). This methodology is also an agreeable method, excellently chosen, described, and accepted among the experts in information system research and design (Vaishnavi & Kuechler, 2008). The major steps of this research methodology are shown in Figure 3.1 below.

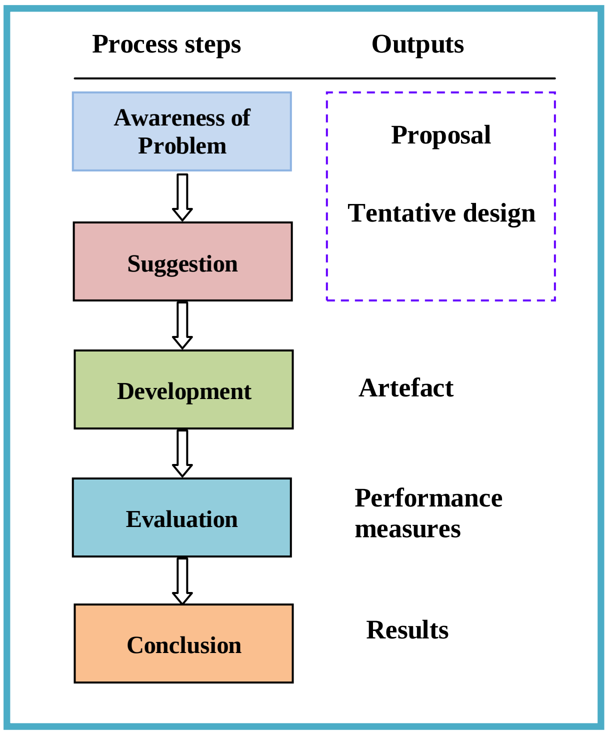


Figure 3.1. Research Design Methodologies (Vaishnavi & Kuechler, 2008)

Figure 3.1 shows the research design process, which starts from awareness of the problem, followed by suggestion of the phase: the development phase, the evaluation phase, and lastly the conclusion phase. The details of every phase are discussed briefly in the following section.

3.2.1 Phase 1: Awareness of Problem

The importance of this methodology is to understand the scope and objective of the research. For this research, to define the characteristics and understanding of these research problems, studies have been conducted on primary and secondary sources. In primary sources, the original materials were studied, including books, articles, and journals written by researchers interested in the area of software components, representation, and related tools. The secondary sources are based on the primary source, where the study is more focused on analysing, evaluating, interpreting, or criticizing the research problem from the primary sources.

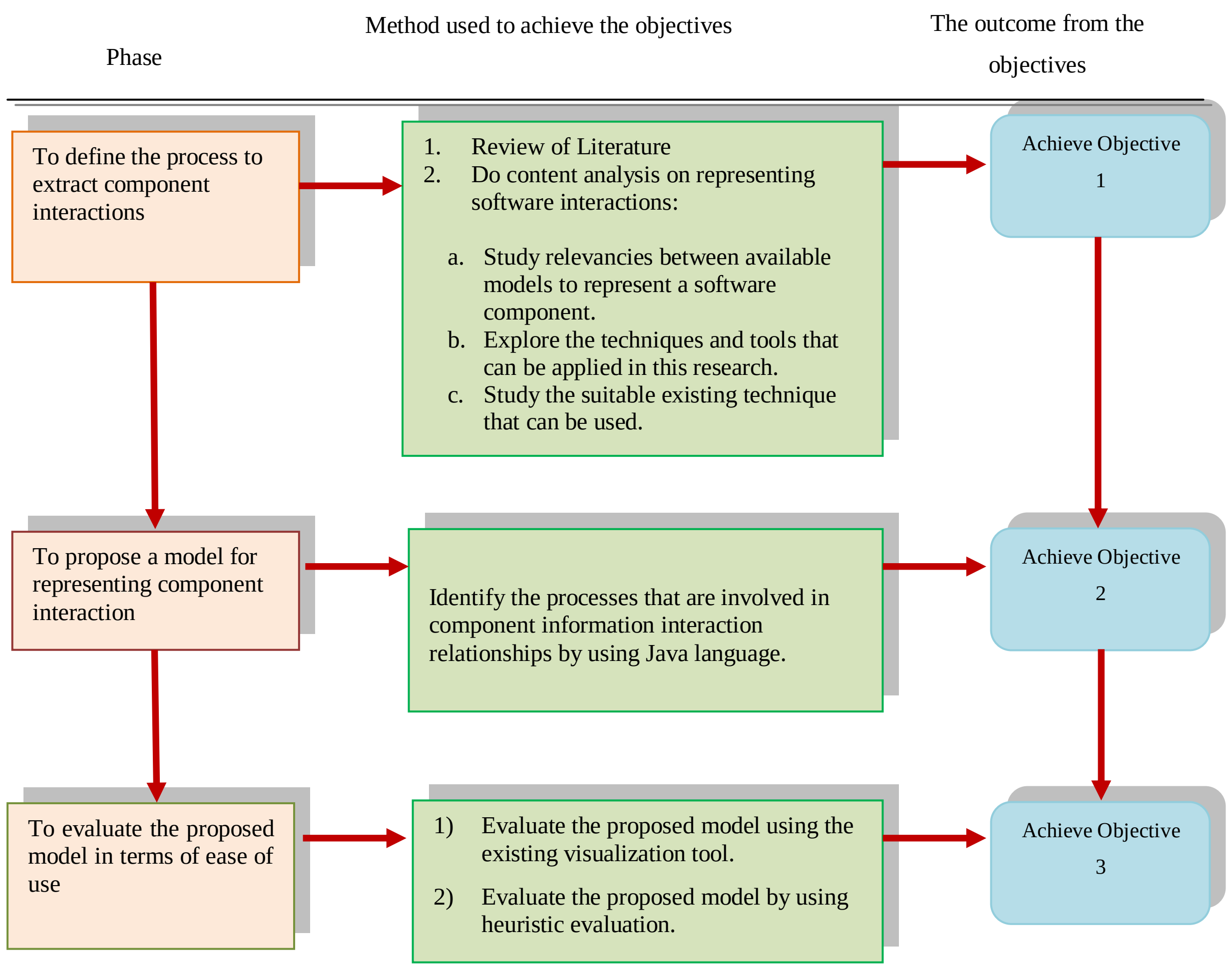
By assessing and reusing the information, secondary sources make the information more accessible. Secondary sources for this study are articles in newspapers, magazines, and scholarly journals that address someone else’s original research. The area that is covered in this phase is the problem statement as stated earlier in section 1.2, which is later used to create the research objectives as presented in section 1.3, and the overall study of literature review, which is presented in chapter 2. The output of this phase is a proposal for a new research effort (Ardakan & Mohajeri, 2009).

3.2.2 Phase 2: Suggestion

The suggestion phase follows immediately after the awareness of the problem phase (Ardakan & Mohajeri, 2009). It is directly connected with the proposal and tentative designs, which are the outputs. In this phase, this study suggests a dependence call graph to represent component interactions in software components to solve the problem as mentioned in chapter 1. For this study to be applicable, two existing tools will be used to extract data from software, which is in the format of a *JAR* file, and creating a dependence call graph. To extract the component interaction information, a tool will be developed because of unavailability of existing tools that can be used.

3.2.2.1 Research Framework

A research framework is one of the methods in the suggestion phase, which defines the outcomes and a set of different research activities. This framework shows the components of this research diagram that are related to one another and built into this framework. Figure 3.2 shows the framework of this research.



Phase
Method used to achieve the objectives
The outcome from the objectives
To define the process to extract component interactions
1. Review of Literature
2. Do content analysis on representing software interactions:
a. Study relevancies between available models to represent a software component.
b. Explore the techniques and tools that can be applied in this research.
c. Study the suitable existing technique that can be used.
Achieve Objective 1
To propose a model for representing component interaction
Identify the processes that are involved in component information interaction relationships by using Java language.
Achieve Objective 2
To evaluate the proposed model in terms of ease of use
1) Evaluate the proposed model using the existing visualization tool.
2) Evaluate the proposed model by using heuristic evaluation.
Achieve Objective 3

Figure 3.2. Research Framework

Based on Figure 3.2, this research framework shows phases and methods that are involved in achieving these research objectives. In this framework, the first phase is to define the process to extract component interactions. It involves literature review, where this phase is focused on finding a technique and solution on how to model software component interactions to represent components to accomplish the first objective. The second phase is to propose a model for representing software component interaction, which involves the development of a program that can automatically trace component relationships of the system. It consists of developing and verifying the program before it is released. Last is the third phase; this phase is to define the evaluation of the proposed model in terms of ease of use. This model will be evaluated by using the existing visualization tool and heuristic review. The details of all phases will be discussed in the next section.

3.2.3 Phase 3: Development

The tentative design is developed further and implemented in this phase. This phase presents a method of the overall development of a call graph. The goal is to give a reader a high-level view of the development process and highlight the role of the end-user. The processes are shown in Figure 3.3.

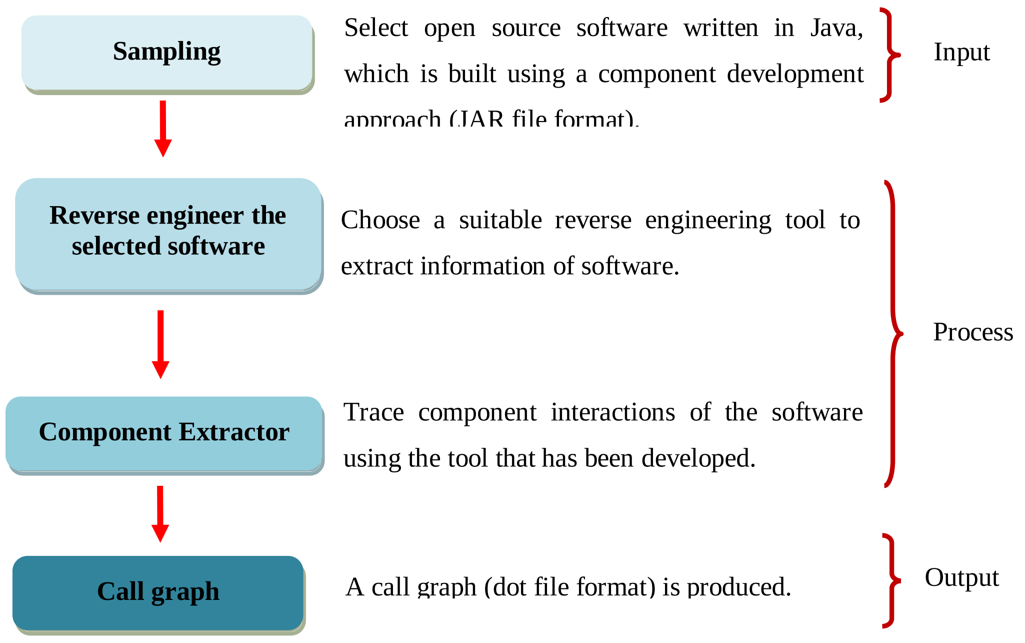

Sampling
Select open source software written in Java, which is built using a component development approach (JAR file format).
Input
Reverse engineer the selected software
Choose a suitable reverse engineering tool to extract information of software.
Process
Component Extractor
Trace component interactions of the software using the tool that has been developed.
Call graph
A call graph (dot file format) is produced.
Output

Figure 3.3. Flow chart of the development process

For this study, three tools will be used to manage the software that has been chosen to represent the component interactions of the proposed model. First, a tool named Ptidej will be used to extract all information from the software (Gueheneuc, 2005). Second, a component extractor tool will be developed to extract the information of component interactions in software programs. By using the information from this tool, the model representing software component interactions will be displayed using Graphviz (Gansner, 2012). The details of the representation process of constructing a call graph are mentioned in the next chapter.

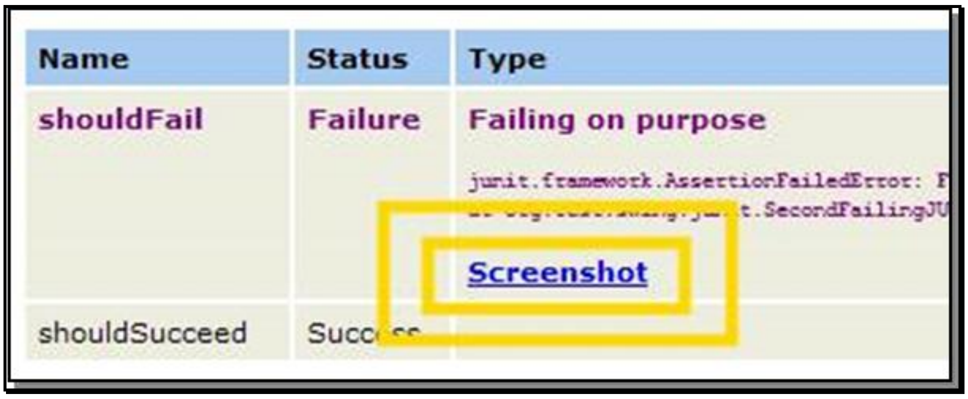
3.2.3.1 Process 1: Sampling

The first activity requires software applications, which have different versions of software as samples. Before beginning work, the team who has full access to their environment should be able to ensure that applications of the components work appropriately.

For this study, two open source applications developed in Java Swing were chosen as samples. These two applications were selected partially because they executed on all platforms supported by Java (e.g., Windows, Linux, and Mac), and they are 100% open source. These two applications also can be downloaded from http://sourceforge.net. The two applications are Junit 4.7 and UML2 0.9. This software also has been used by Zhang et al., (2011) and Koehler and Vanhatalo (2007) as samples in their studies.

JUnit 4.7

JUnit is a simple framework for writing and running automated tests, which does not have a user interface. It is usually used by programmers in testing their own software.



Name	Status	Type
shouldFail	Failure	Failing on purpose junit.framework.AssertionFailedError: F [illegible] t.SecondFailingJU Screenshot
shouldSucceed	Succ[illegible]	

Figure 3.4. Screenshot of a JUnit HTML report (Source: http://docs.codehaus.org)

Figure 3.4 shows the screenshot of a JUnit HTML report. It can generate HTML reports with or without frames. It has been tested with Ant 1.7 and requires Apache-Commons Codec 1.3.

UML2 0.9

UML2 is designed in Java and can be executed from the Command Line Interface. It can be used for converting UML activity diagrams to test cases.

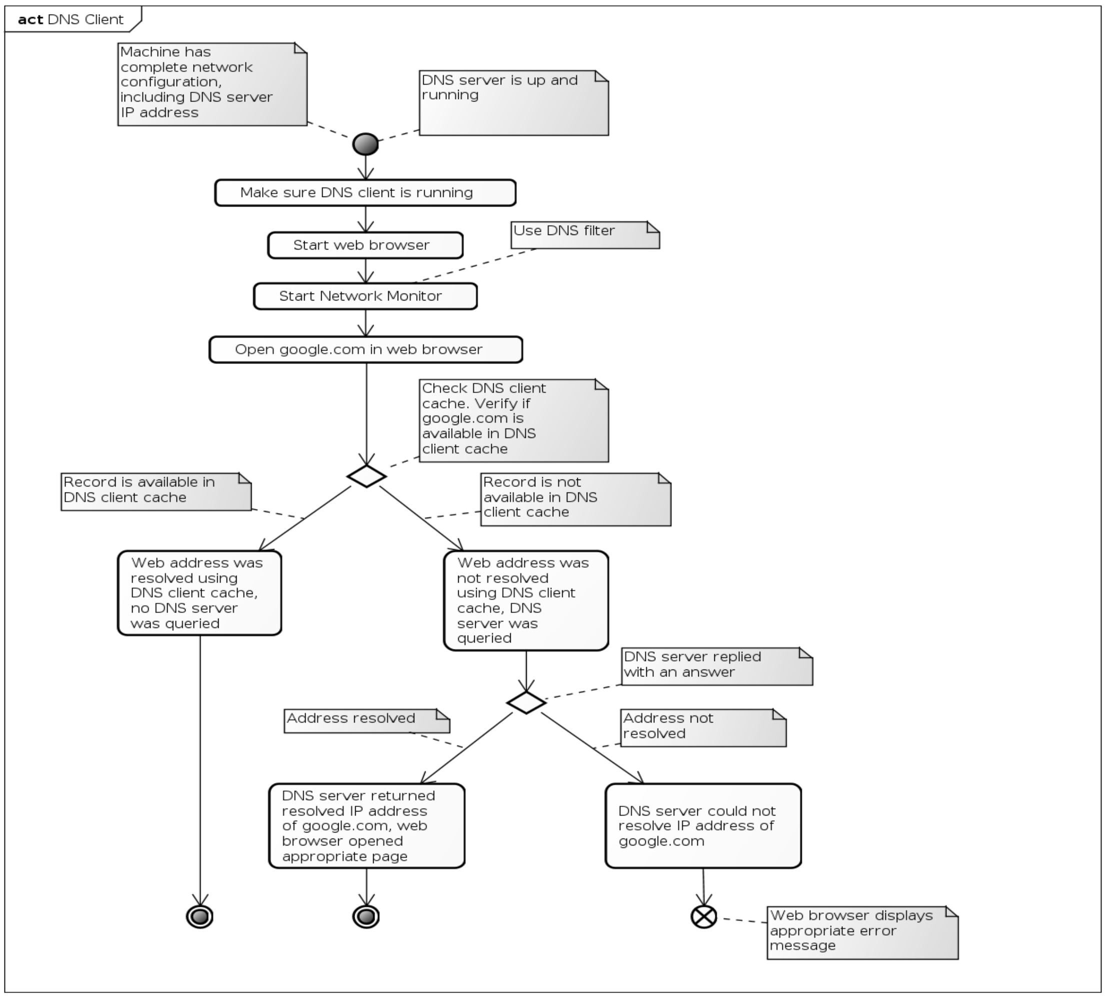
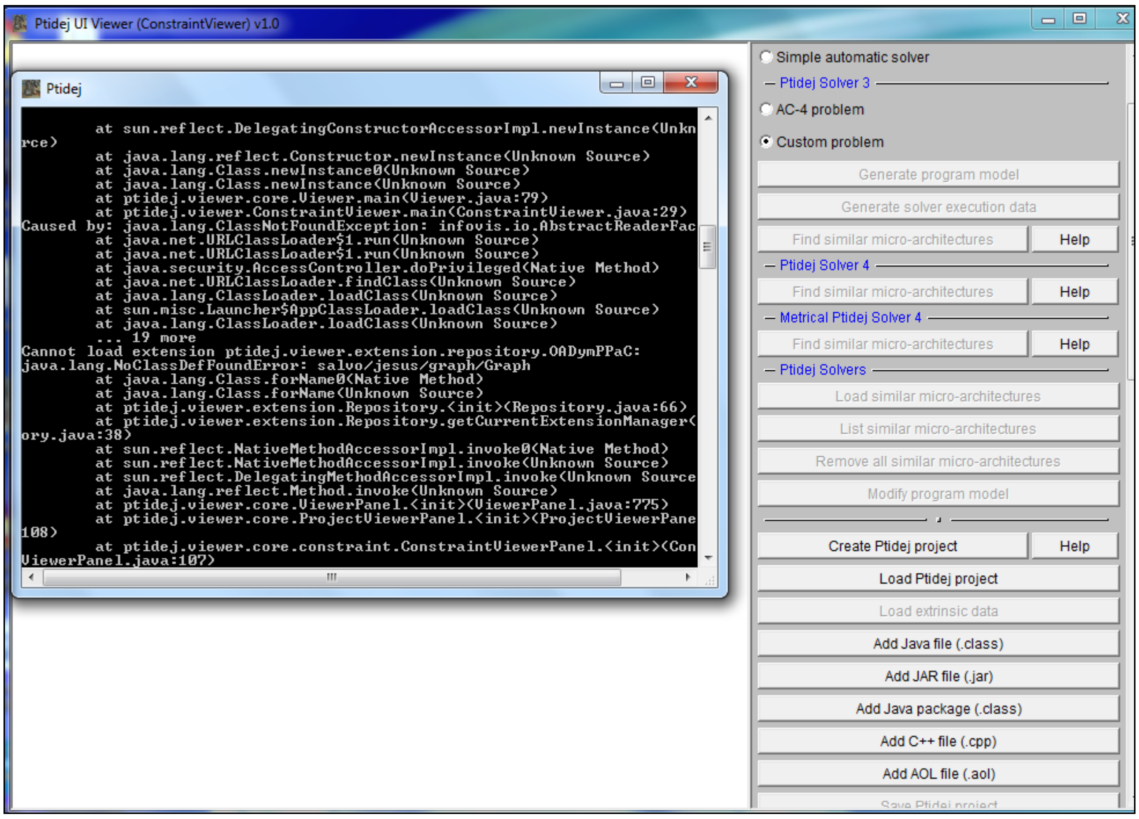

act DNS Client
Machine has complete network configuration, including DNS server IP address
DNS server is up and running
Make sure DNS client is running
Start web browser
Use DNS filter
Start Network Monitor
Open google.com in web browser
Check DNS client cache. Verify if google.com is available in DNS client cache
Record is available in DNS client cache
Record is not available in DNS client cache
Web address was resolved using DNS client cache, no DNS server was queried
Web address was not resolved using DNS client cache, DNS server was queried
DNS server replied with an answer
Address resolved
Address not resolved
DNS server returned resolved IP address of google.com, web browser opened appropriate page
DNS server could not resolve IP address of google.com
Web browser displays appropriate error message

Figure 3.5. Screenshot example of UML2 0.9 (Source: http://www.sourceforge.net)

Figure 3.5 shows an example of a UML2 0.9 screenshot. This simple UML activity diagram is a description of DNS name resolution and was designed according to their guidelines.

3.2.3.2 Process 2: Reverse engineering to extract software information

After deciding the samples that are to be used, this process will reverse engineer the selected system, as open source software is often not equipped with documentation. Therefore, information related to component interfaces can only be derived by extracting it from the program code. Aside from this, this process also requires collecting component information from the lines of code. Existing reverse engineering software named Ptidej, which is a text parser generator tool, will be used to handle this process, to extract the information of component interactions, which are derived from the program source code. The outputs from Ptidej are in text and UML files format. A screenshot of Ptidej is shown below.



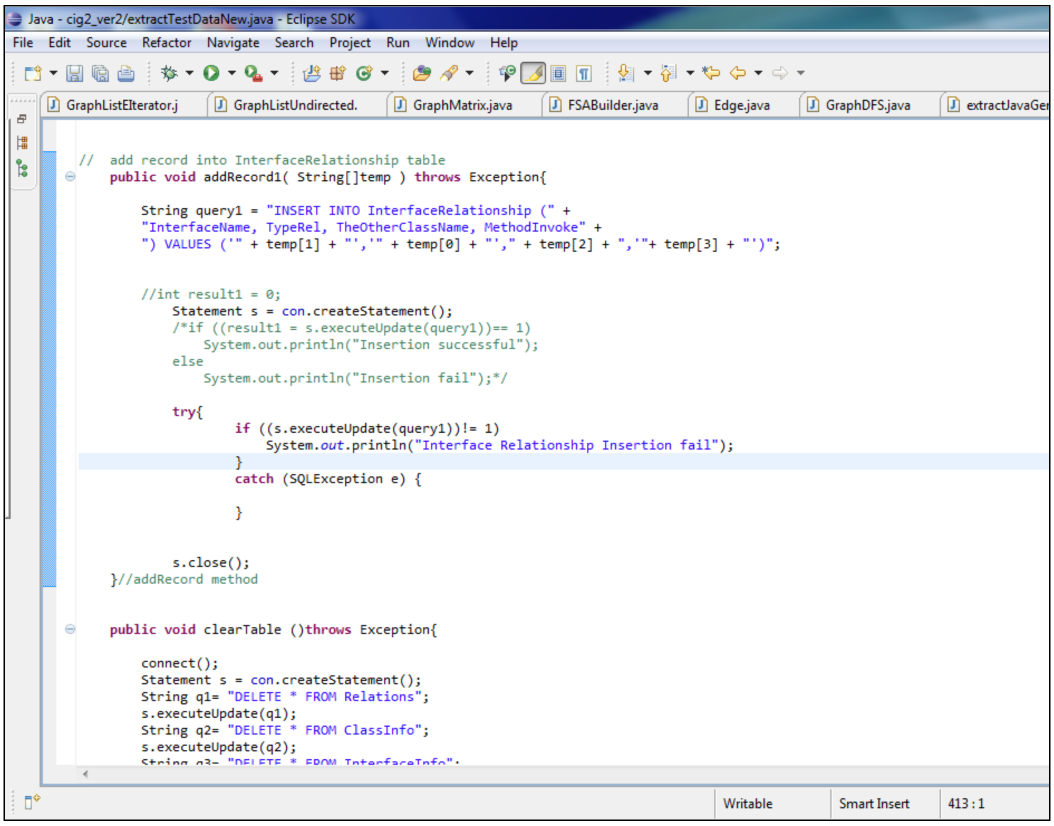
```
        at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(Unkn
rce)
        at java.lang.reflect.Constructor.newInstance(Unknown Source)
        at java.lang.Class.newInstance0(Unknown Source)
        at java.lang.Class.newInstance(Unknown Source)
        at ptidej.viewer.core.Viewer.main(Viewer.java:79)
        at ptidej.viewer.ConstraintViewer.main(ConstraintViewer.java:29)
Caused by: java.lang.ClassNotFoundException: infovis.io.AbstractReaderFac
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        ... 19 more
Cannot load extension ptidej.viewer.extension.repository.OADymPPaC:
java.lang.NoClassDefFoundError: salvo/jesus/graph/Graph
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Unknown Source)
        at ptidej.viewer.extension.Repository.<init>(Repository.java:66)
        at ptidej.viewer.extension.Repository.getCurrentExtensionManager(
ory.java:38)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source
        at java.lang.reflect.Method.invoke(Unknown Source)
        at ptidej.viewer.core.ViewerPanel.<init>(ViewerPanel.java:775)
        at ptidej.viewer.core.ProjectViewerPanel.<init>(ProjectViewerPane
108)
        at ptidej.viewer.core.constraint.ConstraintViewerPanel.<init>(Con
ViewerPanel.java:107)
```

Figure 3.6. Interface Ptidej screenshot (Source: http://www.ptidej.net)

3.2.3.3 Process 3: Component Extraction

A component interaction extractor tool will be developed in this process. A result from process 2 is required. Based on the information of component interactions, this process will be used to trace and extract component interactions from the file that is produced from Process 1. This tool will identify any invocation of component interfaces in the program and will trace the implementation of an invoked interface. From this information, we are able to know which program invoked the component interfaces and which classes in a component implement and execute the interfaces. After this program has extracted the component interactions, it will transform this information in a call graph format, which is in the form of a *dot* file.

A screenshot of component extraction is shown below.



```
//  add record into InterfaceRelationship table
    public void addRecord1( String[]temp ) throws Exception{

        String query1 = "INSERT INTO InterfaceRelationship (" +
        "InterfaceName, TypeRel, TheOtherClassName, MethodInvoke" +
        ") VALUES ('" + temp[1] + "','" + temp[0] + "','" + temp[2] + "','"+ temp[3] + "')";


        //int result1 = 0;
            Statement s = con.createStatement();
            /*if ((result1 = s.executeUpdate(query1))== 1)
                System.out.println("Insertion successful");
            else
                System.out.println("Insertion fail");*/

            try{
                    if ((s.executeUpdate(query1))!= 1)
                        System.out.println("Interface Relationship Insertion fail");
                    }
                    catch (SQLException e) {

                    }


            s.close();
    }//addRecord method


    public void clearTable ()throws Exception{

        connect();
        Statement s = con.createStatement();
        String q1= "DELETE * FROM Relations";
        s.executeUpdate(q1);
        String q2= "DELETE * FROM ClassInfo";
        s.executeUpdate(q2);
        String q3= "DELETE * FROM InterfaceInfo";
```

Figure 3.7. Screenshot of the extract component program

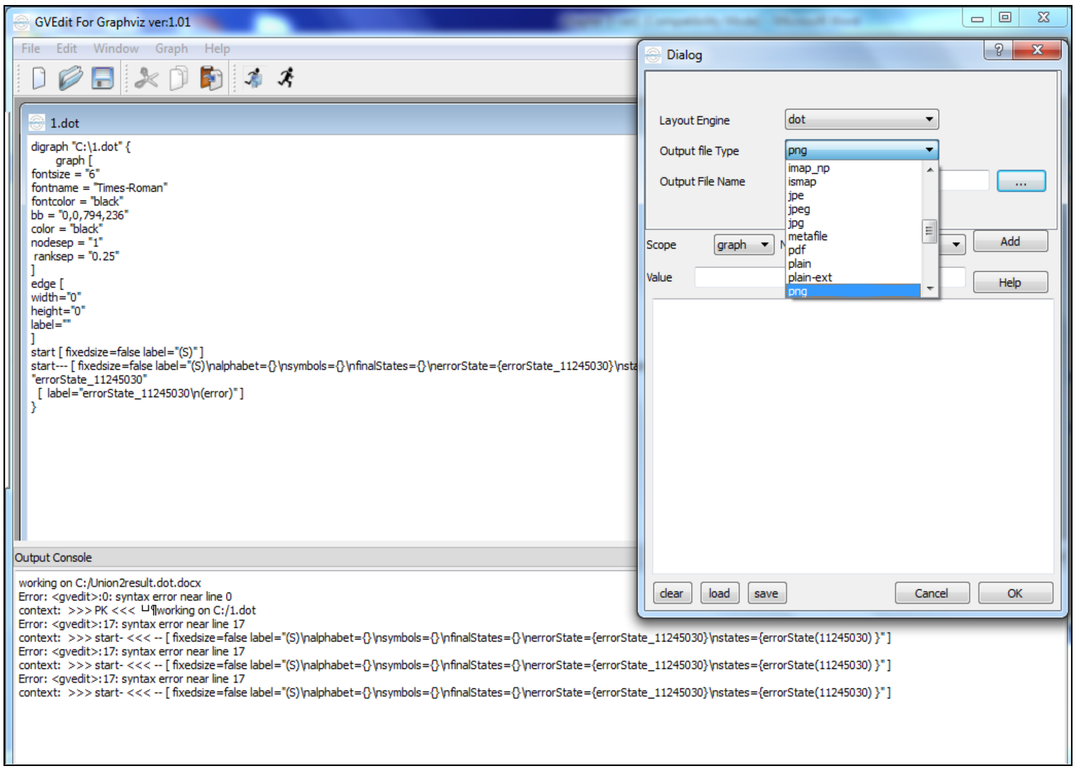
3.2.4 Phase 4: Evaluation

For this research, the evaluation will be conducted after the development phase, whereas the results from the development phase will be evaluated to ensure that this research has achieved its objectives.

3.2.4.1 Evaluation 1: Goal-based evaluation using existing tool

The assessment will be conducted using goal-based evaluation. The objective of goal-based evaluation is to investigate whether a project has achieved its goals (http://www.evalguide.ethz.ch). From the development phase, the component interfaces also will be extracted and call graphs in the form of a dot file will be produced. Based on these two results, component interfaces will be evaluated by comparing results from our program and from Ptidej.

For the evaluation of the produced dot file, the tool named Graphviz will be used to ensure that the dot format produced is valid to use and can display the call graph in the visualization model form. If the Graphviz is able to accept the dot format that was created from the component extractor tool, this shows that the dot file that is produced from this research is valid and capable of representing a call graph to archive the objective of this research. To conduct the evaluation, a result from process 3 in phase 3 is required. The figure below shows a screenshot from Graphviz.



```
digraph "C:\1.dot" {
      graph [
fontsize = "6"
fontname = "Times-Roman"
fontcolor = "black"
bb = "0,0,794,236"
color = "black"
nodesep = "1"
 ranksep = "0.25"
]
edge [
width="0"
height="0"
label=""
]
start [ fixedsize=false label="(S)" ]
start--- [ fixedsize=false label="(S)\nalphabet={}\nsymbols={}\nfinalStates={}\nerrorState={errorState_11245030}\nsta
"errorState_11245030"
  [ label="errorState_11245030\n(error)" ]
}
```

Layout Engine: dot
Output file Type: png
imap_np
ismap
jpe
jpeg
jpg
metafile
pdf
plain
plain-ext
png
Scope: graph

```
working on C:/Union2result.dot.docx
Error: <gvedit>:0: syntax error near line 0
context:  >>> PK <<< └┘¶working on C:/1.dot
Error: <gvedit>:17: syntax error near line 17
context:  >>> start- <<< -- [ fixedsize=false label="(S)\nalphabet={}\nsymbols={}\nfinalStates={}\nerrorState={errorState_11245030}\nstates={errorState(11245030) }" ]
Error: <gvedit>:17: syntax error near line 17
context:  >>> start- <<< -- [ fixedsize=false label="(S)\nalphabet={}\nsymbols={}\nfinalStates={}\nerrorState={errorState_11245030}\nstates={errorState(11245030) }" ]
Error: <gvedit>:17: syntax error near line 17
context:  >>> start- <<< -- [ fixedsize=false label="(S)\nalphabet={}\nsymbols={}\nfinalStates={}\nerrorState={errorState_11245030}\nstates={errorState(11245030) }" ]
```

Figure 3.8. Graphviz screenshot

Tools used in experiment

A programming tool is a program or application that is used by developers to create, maintain, debug, or otherwise support other programs and applications. The term typically refers to relatively simple programs that can be combined together to accomplish a task.

In conducting an experiment in this research, two types of existing tools are used: Ptidej and GraphViz. Ptidej was introduced by Gueheneuc in 2004. This reverse engineering tool is used because of its ability to extract the needed system structure information. This tool can be used to reverse engineer any software in bytecode written in Java or C++.

Ptidej allows information related to classes and relationships between them to be extracted. The relationship information is derived in terms of what methods or attributes are invoked, from what classes and to what classes. These relationships are labelled based on their type inheritance, use, association, aggregation, and composition. Ptidej's outputs are represented in the text format. This feature gives more flexibility to users in manipulating the types of output text files and storing it in a database. This tool is used in the early processes of an experiment.

Graphviz is open source graph visualization software. It represents the structural information as diagrams of abstract graphs and networks. It has important applications in networking, bioinformatics, software engineering, database and Web design, machine learning, and in visual interfaces for other technical domains.

The Graphviz layout programs take descriptions of graphs in a simple text language and make diagrams in useful formats, such as images and SVG for Web pages, PDF, or Postscript for inclusion in other documents, or display in an interactive graph browser. Graphviz also supports GXL, an XML dialect. Graphviz has many useful features for concrete diagrams, such as options for colours, fonts, tabular node layouts, line styles, hyperlinks, roll, and custom shapes. This tool is used in the last phase of the experiment.

3.2.4.2 Evaluation 2: Heuristics Review

An evaluation based on heuristics reviews (also called expert reviews) can identify most of the usability issues that are likely to occur. A heuristic review is a consultative process. While there are a number of different techniques that can be used, the core process is to ask experienced users, to walk through the software, and to adopt as far as possible the viewpoint of novice users (Travis, 2007).

Travis (2007) also defined that one of the approaches of heuristics review is the cognetics approach, which is to conduct a heuristic review with two to three experts. This is a large enough group to provide a diversity of opinion yet small enough to be efficient. Before coming together, each expert looks at the software alone. The first look at a program is very valuable. It is important for the experts to make notes of initial reactions before they become too familiar with the software. Having walked through the software for initial reactions, the expert revisits the software. He or she now looks at each screen looking for violations of the cognetics heuristic guidelines.

Comparative study on the call graph representation model

The comparative study is used in the proposed model as a base line, and the manual model is conducted to capture the component manually. To help in ensuring a realistic setting for this experiment, three computer science lecturers who teach programming courses were selected as respondents to study and make a comparison on the proposed model.

Ten criteria based on Baecker (1988) were used in this study. When conducting the experiment, the respondents discussed with each other to answer each question given. They must answer "Yes" or "No" to the questions. The respondents must compare these two techniques based on the guidelines from Beacker (1998), as stated in Table 3.1 below. A set of questionnaires is distributed to the respondents. The respondents must compare each technique based on requirement given.

The requirements of this experiment begin from extracting data information components from JAR archives from two samples that have been stated in Chapter 3. Existing tools are used to extract all the information from the source code. Then, the experts need to filter and extract only component information from the files using the proposed model. After they achieved the results, a call graph will be created. Once again, they need to create the call graph using the manual method. In the manual method, experts are required to open the Jar file manually and trace line by line the component information from the sample files. After they have completed the task, they manually need to trace again the line of every component that is integrated. Based on the results, they can use Graphviz to visualize their call graphs. Based on the two processes experienced by the experts in producing the call graphs, they evaluated the two processes using the evaluation criteria shown in Table 3.1.

Table 3.1

Criteria used in the comparative study (adopted from Beacker, 1998)

No	Criteria	Descriptions
1.	Maintain the information	Does the technique maintain a repository of architectural and/or behavioural information about a program?
2.	Presentation model	Does the technique provide a presentation model for visualizing information about programs in various ways (e.g., graph, code)?
3.	Transformation	Does the technique provide transformations for going from one kind of representation to another (e.g., text to graph)?
4.	Readability	Does the technique increase the speed by which the users are able to read code?
5.	Appeal	Do users prefer the technique representation?
6.	Memorability	Do users remember more of what they read when they use the technique?
7.	Learnability	Do users learn more easily or more quickly if they use the technique?
8.	Use basic symbols	Does the technique use basic symbols, which are simple, elegant, and easier to be understood in the flow of a program?
9.	Symbols indication	Does the technique use special graphical symbols to indicate the task of the program?
10.	Clear describing	Does the technique clearly describe the flow of the program?

3.2.5 Phase 5: Conclusion

The results of this phase will be the output of this research. First, the output of this research is to produce a call graph representing component interactions in the software and to evaluate the process that has been proposed. Second, to evaluate the process that has been proposed is by using a heuristic review. This review refers to the 10 criteria that were proposed by Baecker (1988).

3.3 Summary

This chapter is a description of the processes involved in this study to achieve all objectives. It starts with the research design of the research methodology. Each phase describes the understanding of a research problem and the area of studies to achieve objective 1, which involves phases 1 and 2. The processes that are proposed in phase 3 are used to create a call graph, which is conducted to achieve objective 2. Last, phase 4 is the process to verify that the output produced by this research is valid to represent the call graph to achieve objective 3. In addition, for this research to be appropriate, it is necessary to use existing tools to conduct an experiment, which are Ptidej and Graphviz (as mention in section 3.3). These two tools will help in software extraction and call graph construction. Finally, expert review is used to evaluate the process of this model.

CHAPTER FOUR

MODEL IMPLEMENTATION

4.1 Introduction

This chapter is a presentation of the overview of the proposed model to show the flow of the process to capture component extraction in software and later create the graph. The figure below shows the model that was generated in this research.

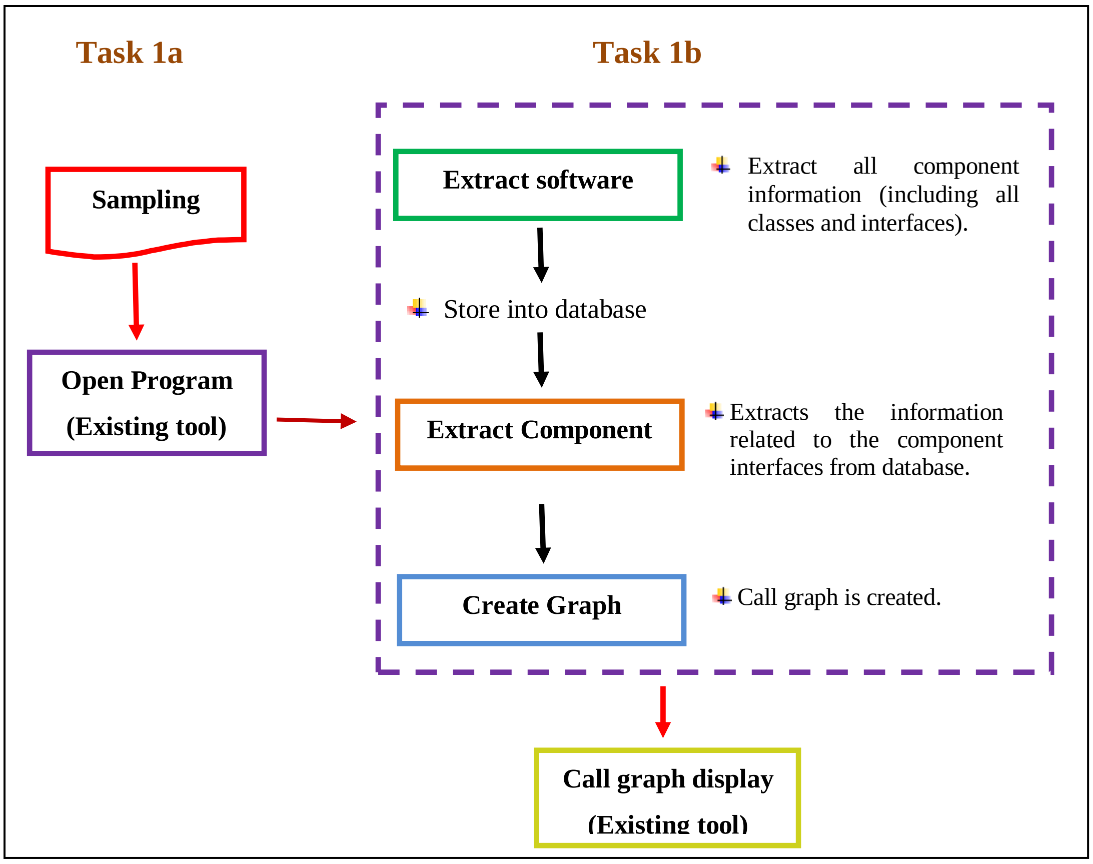
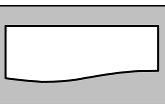
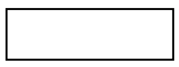


Figure 4.1. Model representing a call graph

The requirements of this model begin from extracting data information components from JAR archives, and filtering and extracting only component information from the files. These two steps should be successful to create formats of graph source files. The results are gathered from two samples that have been stated in chapter 3. Table 4.1 shows the action symbol for representing a call graph model in Figure 4.1.

Table 4.1

Symbol in model representation

Symbol	Description
	This symbol presents the symbol for input data.
	This symbol presents the symbol for processing.

The detailed description of the process of the model is mentioned in section 4.2 below.

4.2 Model Implementation

In conducting this experiment, there are five steps to gather the file's information, which are sampling, extracting software information using the reverse engineering tool, extract all component information, extract information related to the component, create a call graph, and call graph display using the existing tool. These tasks are clarified as follows:

- Task 1a: The reverse engineering tool is used to extract information related to all classes, components, and their interactions as the output.
- Task 1b: Use the static component extraction program to trace and extract component interactions to produce formats of graph source files.

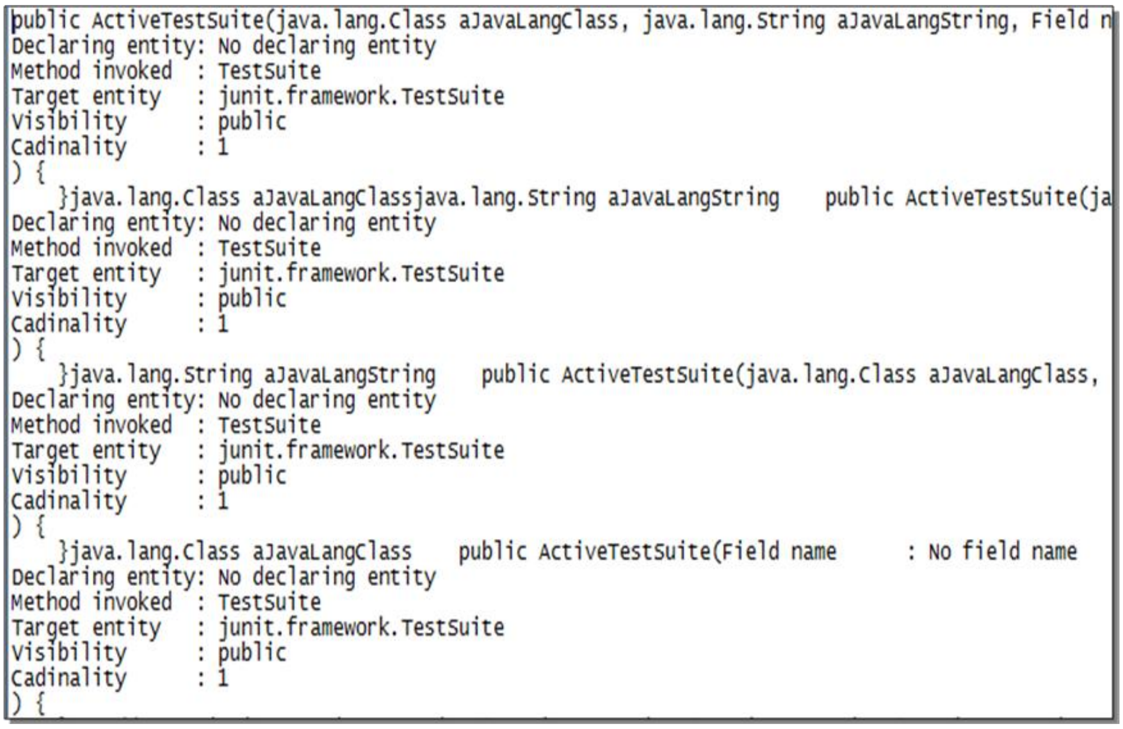
4.2.1 Sampling

The samples used in this experiment are identified based on the criteria that have been mentioned in section 3.2.3.1 which are:

1. JUnit 4.7 (Garousi & Koochakzadeh, 2010), this sample is referred to only as JUnit throughout this thesis.
2. UML2 0.9 (Becker, 2011), this sample is referred to only as UML2 throughout this thesis.

4.2.2 Extract the program by using the existing tool

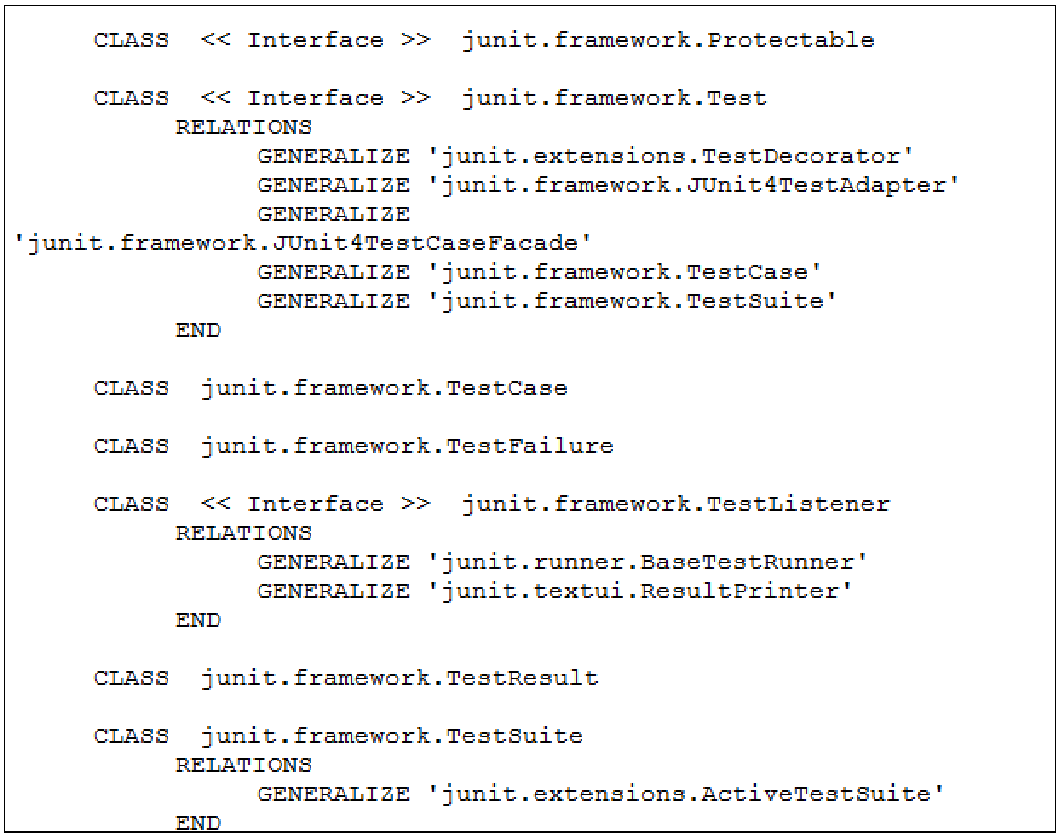
This process is implemented in task 1a in section 4.2. A suitable reverse engineering tool is selected based on the working environment. For this study, a tool named Ptidej is used to extract software information related to all classes, components and their interactions as the output. Two output files, which are in the text and document format, are required for this experiment.



```
public ActiveTestSuite(java.lang.Class aJavaLangClass, java.lang.String aJavaLangString, Field n
Declaring entity: No declaring entity
Method invoked  : TestSuite
Target entity   : junit.framework.TestSuite
Visibility      : public
Cadinality      : 1
) {
    }java.lang.Class aJavaLangClassjava.lang.String aJavaLangString    public ActiveTestSuite(ja
Declaring entity: No declaring entity
Method invoked  : TestSuite
Target entity   : junit.framework.TestSuite
Visibility      : public
Cadinality      : 1
) {
    }java.lang.String aJavaLangString    public ActiveTestSuite(java.lang.Class aJavaLangClass,
Declaring entity: No declaring entity
Method invoked  : TestSuite
Target entity   : junit.framework.TestSuite
Visibility      : public
Cadinality      : 1
) {
    }java.lang.Class aJavaLangClass    public ActiveTestSuite(Field name      : No field name
Declaring entity: No declaring entity
Method invoked  : TestSuite
Target entity   : junit.framework.TestSuite
Visibility      : public
Cadinality      : 1
) {
```

Figure 4.2. Screenshot of example output file in text format

Figure 4.2 shows the screenshot of an example of an output file that is produced by Ptidej in the form of a text document. This file shows the information of class interface names, which also provides the details of information about the declaring entity, which methods are invoked, their target entity, visibility, and cardinality of the selected class interface.



```
     CLASS  << Interface >>  junit.framework.Protectable

     CLASS  << Interface >>  junit.framework.Test
          RELATIONS
               GENERALIZE 'junit.extensions.TestDecorator'
               GENERALIZE 'junit.framework.JUnit4TestAdapter'
               GENERALIZE
'junit.framework.JUnit4TestCaseFacade'
               GENERALIZE 'junit.framework.TestCase'
               GENERALIZE 'junit.framework.TestSuite'
          END

     CLASS  junit.framework.TestCase

     CLASS  junit.framework.TestFailure

     CLASS  << Interface >>  junit.framework.TestListener
          RELATIONS
               GENERALIZE 'junit.runner.BaseTestRunner'
               GENERALIZE 'junit.textui.ResultPrinter'
          END

     CLASS  junit.framework.TestResult

     CLASS  junit.framework.TestSuite
          RELATIONS
               GENERALIZE 'junit.extensions.ActiveTestSuite'
          END
```

Figure 4.3. Screenshot of example output file in UML document

Figure 4.3 shows the screenshot of an example of the output file in the form of a UML document that is produced by Ptidej. This file captured the information of the classes and interfaces. As shown in the screenshot, this file is made of classes and interfaces based on the keyword CLASS for classes and CLASS <<Interfaces>> for interfaces. Additionally, this file also provides the information related to class relationship based on the “RELATIONS” keywords of class that are related to those classes.

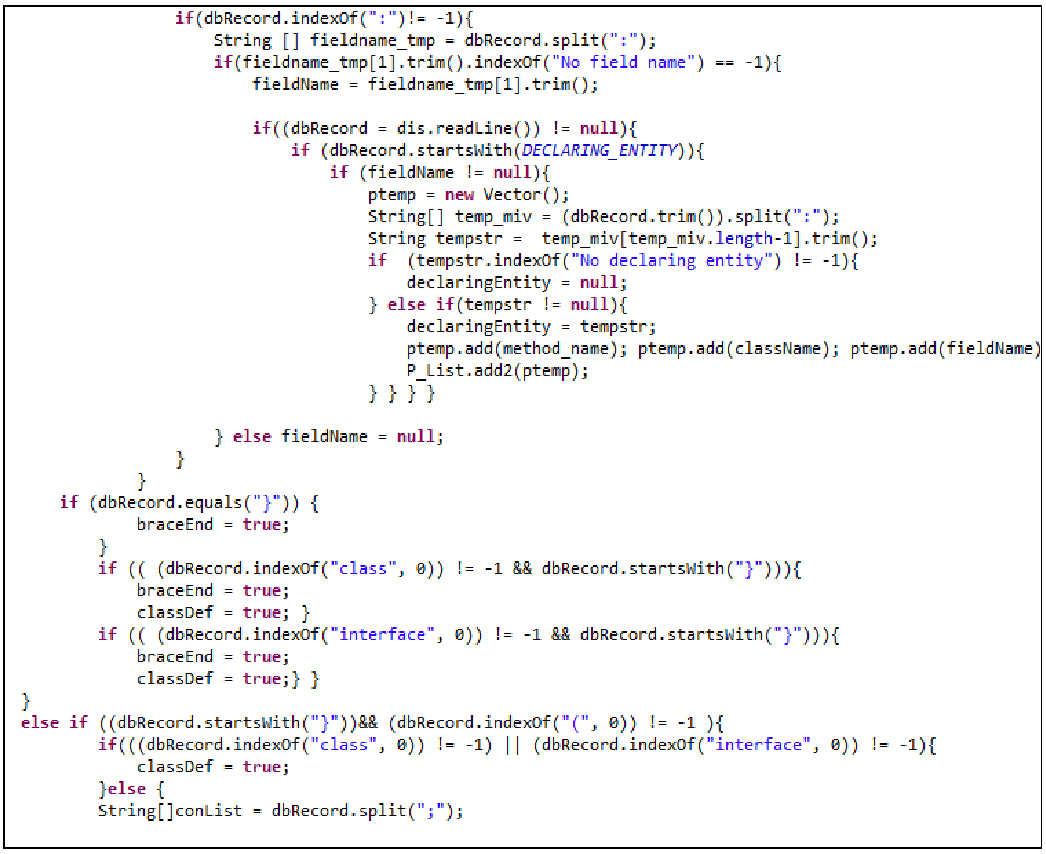
4.2.3 Component Extraction

This process is implemented in task 1b in section 4.2. The purpose of this process is to extract the component information in the program. Any program can be used as long as they can extract the component interaction in a program. This process needs the information input from Ptidej. A database is required because in this process the program will extract component interaction information and store it in the database. From the information gathered in the database, this program would generate a call graph in *dot* file format.

In this study, to extract the component interaction until a dot file is produced, three separated programs will be executed, two programs to extract the component and store it in a database. One program is to retrieve information from the database, and another program is to create a dot file. The purpose of storing data is for fast retrieval. The programs that are mentioned above are *ExtractJavaGenerator*, *ExtractTestDataNew* and *createGraph*. The details and screenshots of these programs are shows below.

4.2.3.1 Extract Software Information

The purpose of this section is to extract software information; the program name *ExtractJavaGenerator* is developed, the output file from the existing tool in text document form is required. This program extracts the information related to the component interfaces, the methods that invoked these interfaces, and the methods that implement and execute the invoked interfaces.



```
                if(dbRecord.indexOf(":")!= -1){
                    String [] fieldname_tmp = dbRecord.split(":");
                    if(fieldname_tmp[1].trim().indexOf("No field name") == -1){
                        fieldName = fieldname_tmp[1].trim();

                        if((dbRecord = dis.readLine()) != null){
                            if (dbRecord.startsWith(DECLARING_ENTITY)){
                                if (fieldName != null){
                                    ptemp = new Vector();
                                    String[] temp_miv = (dbRecord.trim()).split(":");
                                    String tempstr =  temp_miv[temp_miv.length-1].trim();
                                    if  (tempstr.indexOf("No declaring entity") != -1){
                                        declaringEntity = null;
                                    } else if(tempstr != null){
                                        declaringEntity = tempstr;
                                        ptemp.add(method_name); ptemp.add(className); ptemp.add(fieldName)
                                        P_List.add2(ptemp);
                                    } } } }

                    } else fieldName = null;
                }
            }
    if (dbRecord.equals("}")) {
            braceEnd = true;
        }
        if (( (dbRecord.indexOf("class", 0)) != -1 && dbRecord.startsWith("}"))){
            braceEnd = true;
            classDef = true; }
        if (( (dbRecord.indexOf("interface", 0)) != -1 && dbRecord.startsWith("}"))){
            braceEnd = true;
            classDef = true;} }
}
else if ((dbRecord.startsWith("}"))&& (dbRecord.indexOf("(", 0)) != -1 ){
        if(((dbRecord.indexOf("class", 0)) != -1) || (dbRecord.indexOf("interface", 0)) != -1){
            classDef = true;
        }else {
        String[]conList = dbRecord.split(";");
```

Figure 4.4. Screenshot of ExtractJavaGenerator program.

Figure 4.4 shows a screenshot of *ExtractJavaGenerator* program. This program will extract information from the text file that is produced by Ptidej by parsing parameters and types of relationship as shown below.

Table 4.2

Parsing for parameters in ExtractJavaGenerator program

Parsing Parameter
private static final String InterfaceWord = “Interface”
private static final String ClassWord = “Class”
private static final String fWHITESPACE = “\r\t”
private static final String ExtendsWord = “ Extends”
private static final String ImplementationWord = “implements”
private static final String COMMA = “,”

Table 4.3.

Parsing for type of relationships in ExtractJavaGenerator program

Type of relationship
private static final String FIELD_NAME = “Field”
private static final String FIELD_Extract = “Field name”
private static final String DECLARING_ENTITY = “Declaring entity”
private static final String METHOD_INVOKED = “Method invoke”
private static final String TARGET_ENTITY = “Target entity”

Based on Table 4.2 and Table 4.3, this program will extract the text file based on the terms that have been selected. All information that was extracted will be stored in the database. Three tables will be created to store this information. By parsing the content that has the term “Declaring entity”, the research is able to know what method implements the selected interface. By parsing the content for the term “Method invoke”, the research is able to know whether the interfaces are invoked or not.

Table 4.4

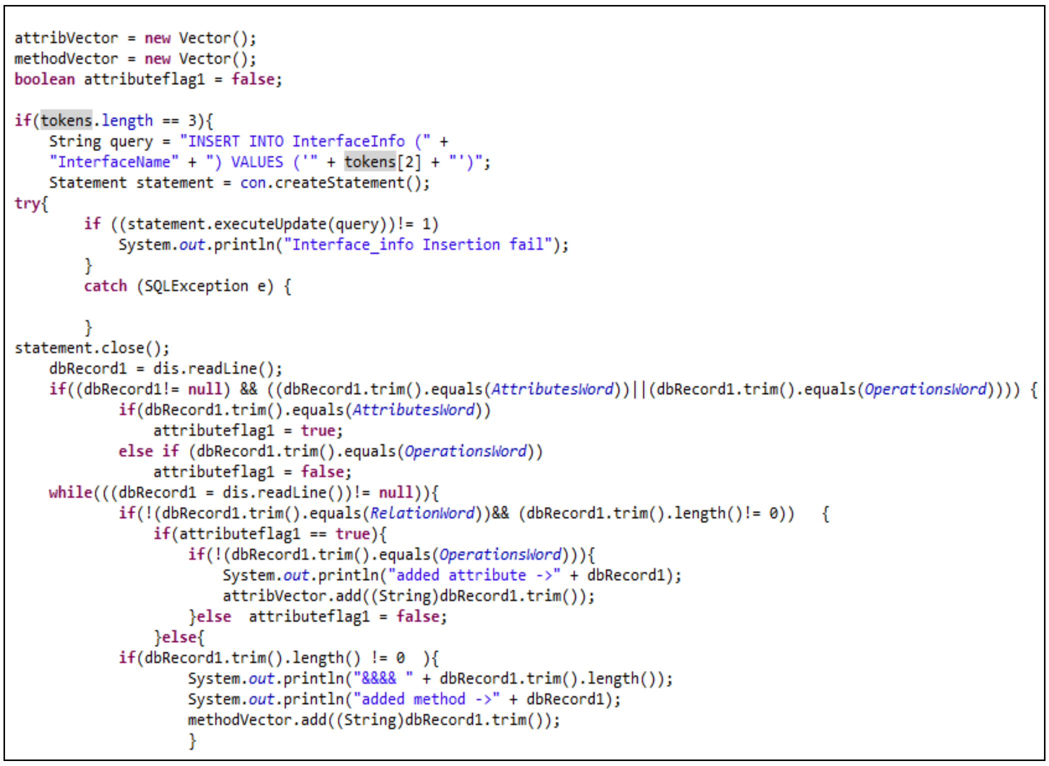
ExtractJavaGenerator tables descriptions

Table Name	Description
ClassInterface_extension	Capture component interface interactions which are class name, extend class, implement class, and type of class (class or interface).
ClassMethod_Detail	Capture the implementation information of all interactions among methods at class level, which are class method, class name, method, invoke, target entity and their parameter.
ClassMethod_Brief	Capture class method interaction, class name and their parameters.

Table 4.4 shows the detail of three tables that were created to store the information into the database, ClassInterface_extension, ClassMethod_Detail and ClassMethod_Brief.

4.2.3.2 Extract Component Program

The purpose of this section is to extract the component, the program name E*xtractTestDataNew* is developed, the output file from the existing tool in UML format is required. Four tables were created to store the data that are captured by this program. This program extracts information related to the class and interfaces together with their relationships.



```
attribVector = new Vector();
methodVector = new Vector();
boolean attributeflag1 = false;

if(tokens.length == 3){
    String query = "INSERT INTO InterfaceInfo (" +
    "InterfaceName" + ") VALUES ('" + tokens[2] + "')";
    Statement statement = con.createStatement();
try{
        if ((statement.executeUpdate(query))!= 1)
            System.out.println("Interface_info Insertion fail");
        }
        catch (SQLException e) {

        }
statement.close();
    dbRecord1 = dis.readLine();
    if((dbRecord1!= null) && ((dbRecord1.trim().equals(AttributesWord))||(dbRecord1.trim().equals(OperationsWord)))) {
            if(dbRecord1.trim().equals(AttributesWord))
                attributeflag1 = true;
            else if (dbRecord1.trim().equals(OperationsWord))
                attributeflag1 = false;
    while(((dbRecord1 = dis.readLine())!= null)){
            if(!(dbRecord1.trim().equals(RelationWord))&& (dbRecord1.trim().length()!= 0))   {
                if(attributeflag1 == true){
                    if(!(dbRecord1.trim().equals(OperationsWord))){
                        System.out.println("added attribute ->" + dbRecord1);
                        attribVector.add((String)dbRecord1.trim());
                    }else  attributeflag1 = false;
                }else{
            if(dbRecord1.trim().length() != 0  ){
                    System.out.println("&&&& " + dbRecord1.trim().length());
                    System.out.println("added method ->" + dbRecord1);
                    methodVector.add((String)dbRecord1.trim());
                    }
```

Figure 4.5. Screenshot of ExtractTestDataNew program.

Table 4.5

Parsing for parameters in ExtractTestDataNew program

Parsing Parameter
private static final String InterfaceWord = “Interface”
private static final String ClassWord = “Class”
private static final String fWHITESPACE = “\r\t”
private static final String ExtendsWord = “ Extends”
private static final String ImplementationWord = “implements”
private static final String COMMA = “,”

Table 4.6

Type of relationships in ExtractTestDataNew program

Type of Relationship
private static final String USE = “Knowledge>>DEPENDS”
private static final String INHERITANCE= “GENERALIZE”
private static final String MANY_ASSOCIATION = “ASSOC_TO_[*]_ROLE”
private static final String ASSOCIATION= “ASSOC_TO_ROLE”
private static final String MANY_AGGREGATION= “ASSOC_TO_[*]_aggregate_ROLE”
private static final String AGGREGATION= “ASSOC_TO_aggregate_ROLE”

Based on Table 4.5 and Table 4.6, this program will extract the UML file based on the terms that have been selected. All information that was extracted will be stored in the database. Four tables will be created to store the information.

Table 4.7

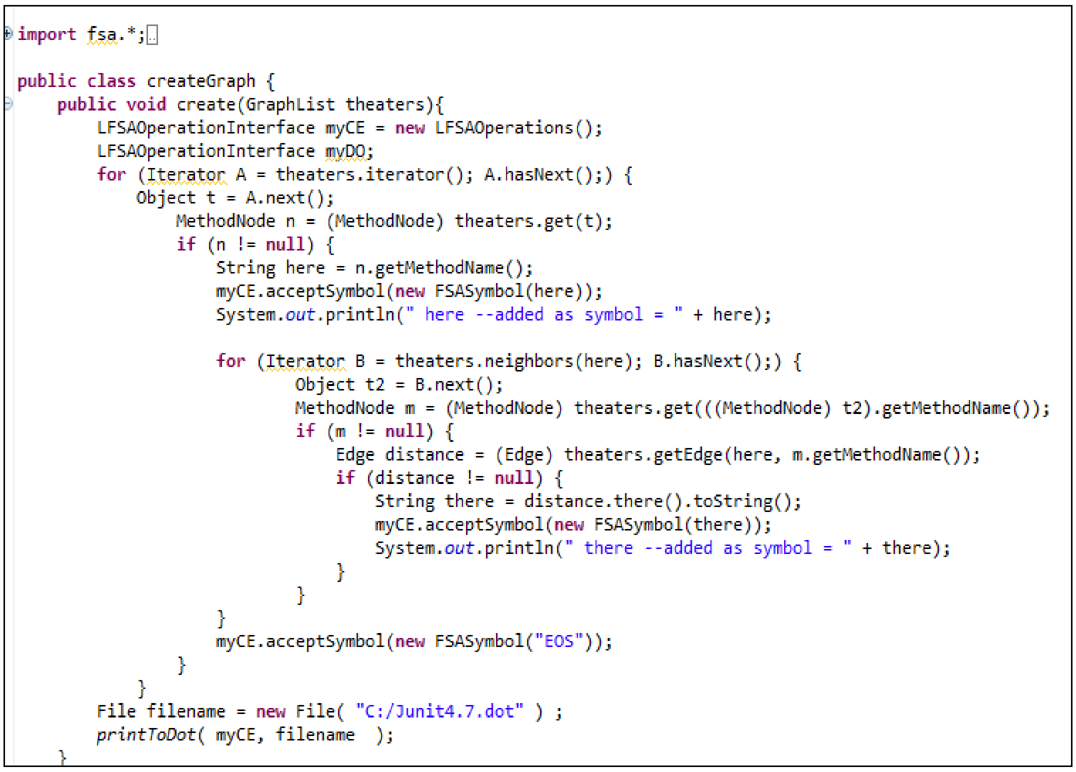
ExtractTestDataNew tables descriptions

Table Name	Description
ClassInfo	Capture all class names
ClassRelationship	Capture class names and their relationships
InterfaceInfo	Capture all interface names
InterfaceRelationship	Capture interface names and their relationships

Table 4.7 shows the details of four tables that were created to store the information into the database, which are ClassInfo, ClassRelationship, InterfaceInfo, and InterfaceRelationship. At this stage, class methods that implement the component interfaces will be extracted, leaving other methods not involved with the interfaces and there traces out of the database.

4.2.3.3 Create Call Graph Program

The purpose of this section was to create a call graph and the program name CreateGraph was created to retrieve information that was stored from the database. This program will create a call graph in a *dot* format for visualizing the component interaction information.



```
import fsa.*;

public class createGraph {
    public void create(GraphList theaters){
        LFSAOperationInterface myCE = new LFSAOperations();
        LFSAOperationInterface myDO;
        for (Iterator A = theaters.iterator(); A.hasNext();) {
            Object t = A.next();
                MethodNode n = (MethodNode) theaters.get(t);
                if (n != null) {
                    String here = n.getMethodName();
                    myCE.acceptSymbol(new FSASymbol(here));
                    System.out.println(" here --added as symbol = " + here);

                    for (Iterator B = theaters.neighbors(here); B.hasNext();) {
                            Object t2 = B.next();
                            MethodNode m = (MethodNode) theaters.get(((MethodNode) t2).getMethodName());
                            if (m != null) {
                                Edge distance = (Edge) theaters.getEdge(here, m.getMethodName());
                                if (distance != null) {
                                    String there = distance.there().toString();
                                    myCE.acceptSymbol(new FSASymbol(there));
                                    System.out.println(" there --added as symbol = " + there);
                                }
                            }
                    }
                    myCE.acceptSymbol(new FSASymbol("EOS"));
                }
            }
        File filename = new File( "C:/Junit4.7.dot" ) ;
        printToDot( myCE, filename  );
    }
```

Figure 4.6. Screenshot of createGraph program

Figure 4.6 shows a screenshot of the createGraph program. This dot file is a result of this research.

4.3 Call Graph Size

Based on the program running, the size of the call graph was captured.

4.3.1 JUnit 4.7

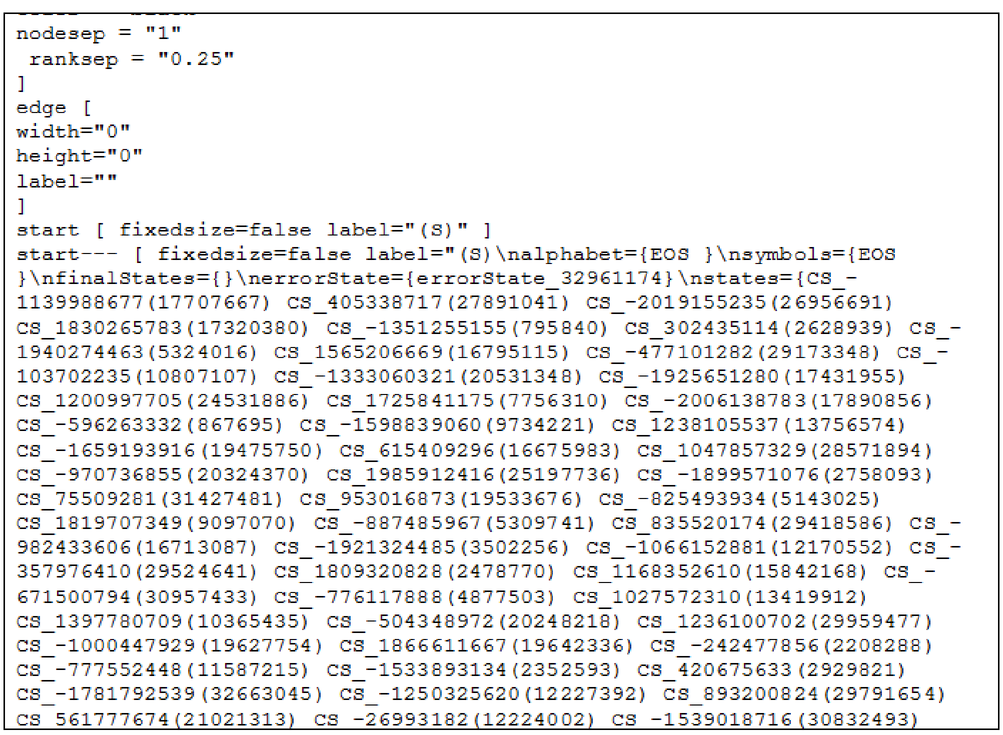
a. Extract from JAR archive

Table 4.8

The number of classes and interfaces that are extracted from the program

No. of classes	No. of Interfaces
134	29

Table 4.8 shows the size of JUnit 4.7 in terms of the number of classes and interfaces that exist after it has been statically reverse engineered. For the program, 134 numbers of classes and 29 numbers of interfaces were identified.



```
nodesep = "1"
 ranksep = "0.25"
]
edge [
width="0"
height="0"
label=""
]
start [ fixedsize=false label="(S)" ]
start--- [ fixedsize=false label="(S)\nalphabet={EOS }\nsymbols={EOS
}\nfinalStates={}\nerrorState={errorState_32961174}\nstates={CS_-
1139988677(17707667) CS_405338717(27891041) CS_-2019155235(26956691)
CS_1830265783(17320380) CS_-1351255155(795840) CS_302435114(2628939) CS_-
1940274463(5324016) CS_1565206669(16795115) CS_-477101282(29173348) CS_-
103702235(10807107) CS_-1333060321(20531348) CS_-1925651280(17431955)
CS_1200997705(24531886) CS_1725841175(7756310) CS_-2006138783(17890856)
CS_-596263332(867695) CS_-1598839060(9734221) CS_1238105537(13756574)
CS_-1659193916(19475750) CS_615409296(16675983) CS_1047857329(28571894)
CS_-970736855(20324370) CS_1985912416(25197736) CS_-1899571076(2758093)
CS_75509281(31427481) CS_953016873(19533676) CS_-825493934(5143025)
CS_1819707349(9097070) CS_-887485967(5309741) CS_835520174(29418586) CS_-
982433606(16713087) CS_-1921324485(3502256) CS_-1066152881(12170552) CS_-
357976410(29524641) CS_1809320828(2478770) CS_1168352610(15842168) CS_-
671500794(30957433) CS_-776117888(4877503) CS_1027572310(13419912)
CS_1397780709(10365435) CS_-504348972(20248218) CS_1236100702(29959477)
CS_-1000447929(19627754) CS_1866611667(19642336) CS_-242477856(2208288)
CS_-777552448(11587215) CS_-1533893134(2352593) CS_420675633(2929821)
CS_-1781792539(32663045) CS_-1250325620(12227392) CS_893200824(29791654)
CS_561777674(21021313) CS_-26993182(12224002) CS_-1539018716(30832493)
```

Figure 4.7. Screenshot dot file for JUnit 4.7

Figure 4.7 shows the screenshot of the output interface of the dot file. Based on the actual result that was produced, the size of the dot file is 767 Kb.

4.3.2 UML2 0.9

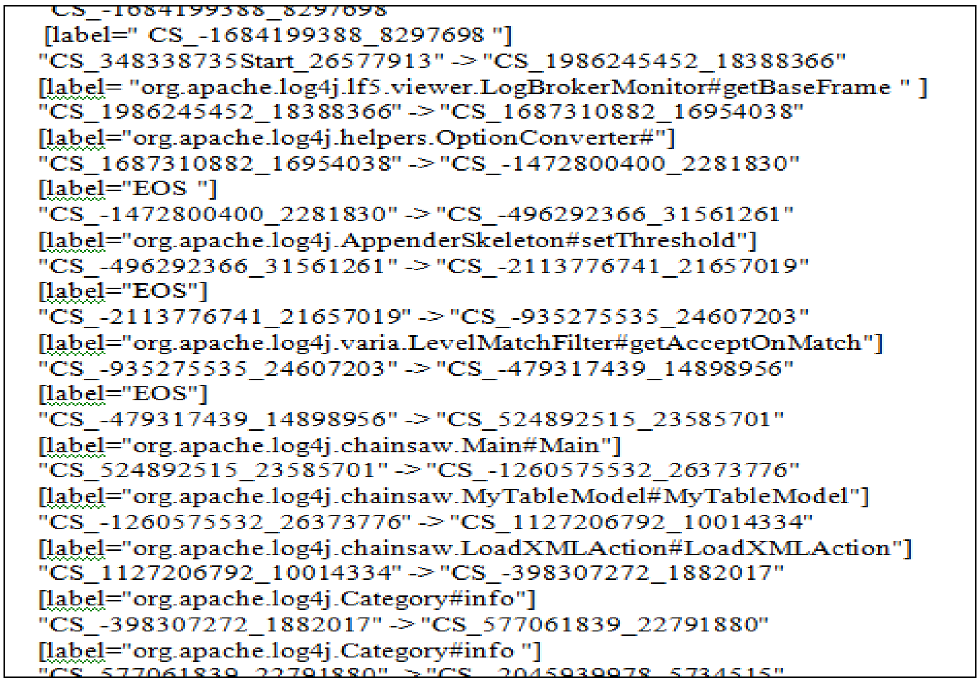
a. Extract from the JAR archive

Table 4.9

The number of classes and interfaces that are extracted from the program

No. of classes	No. of Interfaces
252	20

Table 4.9 shows the size of UML2 0.9 in terms of the number of classes and interfaces that exist after it has been statically reverse engineering. For the program, 252 numbers of classes and 20 numbers of interfaces were identified.



```
CS_-1684199388_8297698
 [label=" CS_-1684199388_8297698 "]
"CS_348338735Start_26577913" -> "CS_1986245452_18388366"
[label= "org.apache.log4j.lf5.viewer.LogBrokerMonitor#getBaseFrame " ]
"CS_1986245452_18388366" -> "CS_1687310882_16954038"
[label="org.apache.log4j.helpers.OptionConverter#"]
"CS_1687310882_16954038" -> "CS_-1472800400_2281830"
[label="EOS "]
"CS_-1472800400_2281830" -> "CS_-496292366_31561261"
[label="org.apache.log4j.AppenderSkeleton#setThreshold"]
"CS_-496292366_31561261" -> "CS_-2113776741_21657019"
[label="EOS"]
"CS_-2113776741_21657019" -> "CS_-935275535_24607203"
[label="org.apache.log4j.varia.LevelMatchFilter#getAcceptOnMatch"]
"CS_-935275535_24607203" -> "CS_-479317439_14898956"
[label="EOS"]
"CS_-479317439_14898956" -> "CS_524892515_23585701"
[label="org.apache.log4j.chainsaw.Main#Main"]
"CS_524892515_23585701" -> "CS_-1260575532_26373776"
[label="org.apache.log4j.chainsaw.MyTableModel#MyTableModel"]
"CS_-1260575532_26373776" -> "CS_1127206792_10014334"
[label="org.apache.log4j.chainsaw.LoadXMLAction#LoadXMLAction"]
"CS_1127206792_10014334" -> "CS_-398307272_1882017"
[label="org.apache.log4j.Category#info"]
"CS_-398307272_1882017" -> "CS_577061839_22791880"
[label="org.apache.log4j.Category#info "]
"CS_577061839_22791880" -> "CS_2045939978_5734515"
```

Figure 4.8. Screenshot dot file for UML2 0.9

Figure 4.8 shows the screenshot of the output interface dot file. Based on the actual results that were produced, the size of the dot file is 296 Kb.

4.4 Call Graph Visualization

To show the visualization of the call graph, several services from the sample used are chosen. The graphs presented in the following section are edited because of the large number of nodes. The details will be shown in sections 4.3.5.1 and 4.3.5.2.

4.4.1 Sample 1: JUnit 4.7

a) Dot format for getName service from TestClass interface.

```
digraph "C:\JUnit.dot" {
   graph [
fontsize = "6"
fontname = "Times-Roman"
fontcolor = "black"
bb = "0,0,794,236"
color = "black"
nodesep = "1"
ranksep = "0.25"
]
edge [
width="0"
height="0"
label=""
]
start [ fixedsize=false label="(S)" ]
start [ fixedsize=false label="(S)\nalphabet={
org.junit.internal.runners.TestClass#getName
org.junit.runners.BlockJUnit4ClassRunner#withAfters
org.junit.runners.BlockJUnit4ClassRunner#validateTestMethods
org.junit.runners.BlockJUnit4ClassRunner#validatePublicVoidNoArgMethods
org.junit.matchers.JUnitMatchers#JUnitMatchers
org.junit.runners.model.FrameworkField#get
junit.framework.TestCase#getName
}\nsymbol= {
org.junit.internal.runners.TestClass#getName
org.junit.runners.BlockJUnit4ClassRunner#withAfters
org.junit.runners.BlockJUnit4ClassRunner#validateTestMethods
org.junit.runners.BlockJUnit4ClassRunner#validatePublicVoidNoArgMethods
org.junit.matchers.JUnitMatchers#JUnitMatchers
org.junit.runners.model.FrameworkField#get
junit.framework.TestCase#getName
}
finalStates ={EOS}
errorState={errorState_16695559}
states={
CS_-1709246797(24093812) CS_329389440 (26335425) CS_-372915219(2989062) CS_-1821556123(6183504)
CS_1857468700 (8087689) CS_2112419304 (798709) CS_-1527259317(2145913)}" ]

start -> "CS_-2052662688Start_24993066"
[label="CS_-2052662688Start_24993066\n(start)" ]
"CS_-1709246797_24093812"
[label=" CS_-1709246797_24093812 "]
"CS_329389440_26335425"
[label=" CS_329389440_26335425 "]
"CS_-372915219_2989062"
[label=" CS_-372915219_2989062 "]
"CS_-1821556123_6183504"
[label=" CS_-1821556123_6183504 "]
"CS_1857468700_8087689"
[label=" CS_1857468700_8087689 "]
"CS_2112419304_798709"
[label=" CS_2112419304_798709 "]
"CS_-1527259317_2145913"
[label=" CS_-1527259317_214591 "]
"CS_-2052662688Start_24993066" -> "CS_-1709246797_24093812"
[label="org.junit.internal.runners.TestClass#getName" ]
"CS_-1709246797_24093812"                    ->                    "CS_329389440_26335425"
[label="org.junit.runners.BlockJUnit4ClassRunner#withAfters"]
"CS_329389440_26335425" -> "CS_-372915219_2989062"
[label="org.junit.runners.BlockJUnit4ClassRunner#validateTestMethods"]
"CS_-372915219_2989062" -> "CS_-1821556123_6183504"
[label="org.junit.runners.BlockJUnit4ClassRunner#validatePublicVoidNoArgMethods"]
"CS_-1821556123_6183504"                    ->                    "CS_1857468700_8087689"
[label="org.junit.matchers.JUnitMatchers#JUnitMatchers"]
"CS_1857468700_8087689" -> "CS_2112419304_798709"
[label="org.junit.runners.model.FrameworkField#get"]
"CS_2112419304_798709" -> "CS_-1527259317_2145913"
[label="EOS "]
}
"CS_-2052662688Start_24993066" -> "CS_-1709246797_24093812"
[label="org.junit.internal.runners.TestClass#getName" ]
"CS_-1709246797_24093812"                    ->                    "CS_329389440_26335425"
[label="org.junit.runners.BlockJUnit4ClassRunner#withAfters"]
"CS_329389440_26335425" -> "CS_-372915219_2989062"
[label="org.junit.runners.BlockJUnit4ClassRunner#validateTestMethods"]
"CS_-372915219_2989062" -> "CS_-1821556123_6183504"
[label="org.junit.runners.BlockJUnit4ClassRunner#validatePublicVoidNoArgMethods"]
"CS_-1821556123_6183504"                    ->                    "CS_1857468700_8087689"
[label="org.junit.matchers.JUnitMatchers#JUnitMatchers"]
"CS_1857468700_8087689" -> "CS_2112419304_798709"
[label="org.junit.runners.model.FrameworkField#get"]
"CS_2112419304_798709" -> "CS_-1527259317_2145913"
[label="EOS "]
```

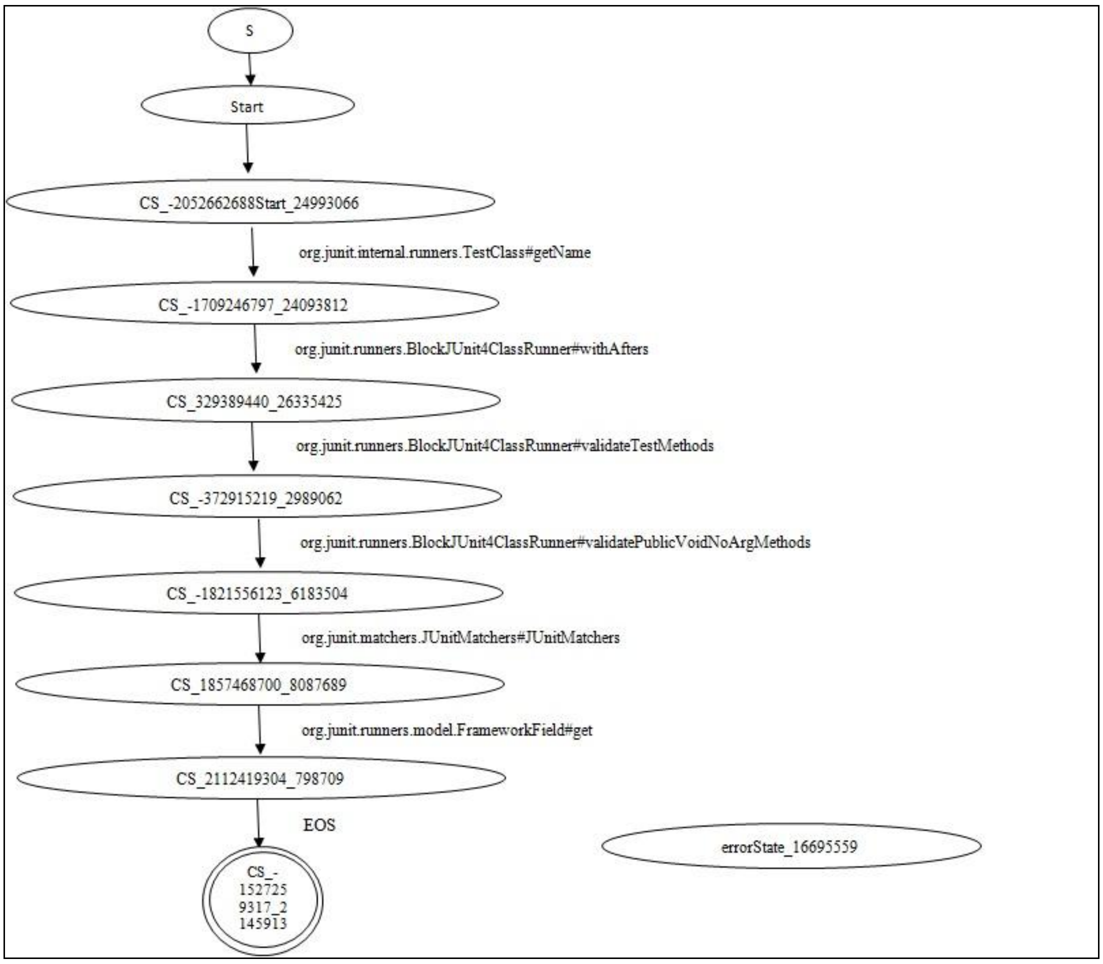

S
Start
CS_-2052662688Start_24993066
org.junit.internal.runners.TestClass#getName
CS_-1709246797_24093812
org.junit.runners.BlockJUnit4ClassRunner#withAfters
CS_329389440_26335425
org.junit.runners.BlockJUnit4ClassRunner#validateTestMethods
CS_-372915219_2989062
org.junit.runners.BlockJUnit4ClassRunner#validatePublicVoidNoArgMethods
CS_-1821556123_6183504
org.junit.matchers.JUnitMatchers#JUnitMatchers
CS_1857468700_8087689
org.junit.runners.model.FrameworkField#get
CS_2112419304_798709
EOS
errorState_16695559
CS_-152725 9317_2 145913

Figure 4.9. Call graph produced for getName service from TestClass interface for JUnit 4.7

Based on Figure 4.9, this call graph shows seven traces of interfaces invoked that are executed in this service. The traces of interfaces invoked are, org.junit.internal.runners.TestClass#getName, org.junit.runners.BlockJUnit4ClassRunner#withAfters, org.junit.runners.BlockJUnit4ClassRunner#validateTestMethods, org.junit.runners.BlockJUnit4ClassRunner#validatePublicVoidNoArgMethods, org.junit.matchers.JUnitMatchers#JUnitMatchers, org.junit.runners.model.FrameworkField#get and lastly is EOS for this service. For the states invoked are CS_-2052662688Start_24993066, CS_-1709246797_24093812, CS_329389440_26335425, CS_-372915219_2989062, CS_-1821556123_6183504, CS_1857468700_8087689, CS_2112419304_798709, and finally for end of states is CS_-1527259317_2145913. Last, for error state is errorState_16695559. The hash (#) symbol is used to separate the components interface and their method name.

4.4.2 Sample 2: UML 0.9

a) Dot file for newInstance service from LogMonitorAdapter interface.

```
digraph "C:\uml2.dot" {
   graph [
fontsize = "6"
fontname = "Times-Roman"
fontcolor = "black"
bb = "0,0,794,236"
color = "black"
nodesep = "1"
ranksep = "0.25"
]
edge [
width="0"
height="0"
label=""
]
start [ fixedsize=false label="(S)" ]
start [ fixedsize=false
label="(S)\nalphabet={
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.util.LogMonitorAdapter#LogMonitorAdapter”
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.LogLevel#getJdk14Levels”
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel”
“org.apache.log4j.lf5.util.LogMonitorAdapter#setSevereLevel”
“org.apache.log4j.lf5.LogLevel#getLog4JLevels”
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel”
“org.apache.log4j.lf5.util.LogMonitorAdapter#”
}
\nsymbols= {
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.util.LogMonitorAdapter#LogMonitorAdapter”
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.LogLevel#getJdk14Levels”
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel”
“org.apache.log4j.lf5.util.LogMonitorAdapter#setSevereLevel”
“org.apache.log4j.lf5.LogLevel#getLog4JLevels”
“org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance”
“org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel”
“org.apache.log4j.lf5.util.LogMonitorAdapter#”
}
\nfinalStates ={EOS} \nerrorState={errorState_30495813}
\nstates={
CS_1555958612Start(28637909)          CS_-211101784(12577309)          CS_-868287843(813251)
CS_668692069(11665455)  CS_-1939490076(2771331)  CS_1601600884(4219289)  CS_1913308791(13086732)
CS_877950025(32961174)  CS_1448178545(28349544)  CS_-922782094(31414927)  CS_-554546748(11707096)
CS_-2133694150(31538514) CS_-2133694150(31538514) CS_-998238848(8452719) errorState(30495813) }”]
Start-> "CS_1555958612Start_28637909"
[label="CS_1555958612Start_28637909\n(start)" ]
"CS_-211101784_12577309"
[label="CS_-211101784_12577309" ]
"CS_-868287843_813251"
[label="CS_-868287843_813251" ]
"CS_668692069_11665455"
[label="CS_668692069_11665455" ]
"CS_-1939490076_2771331"
[label="CS_-1939490076_2771331" ]
"CS_1601600884_4219289"
[label="CS_1601600884_4219289" ]
"CS_1913308791_13086732"
[label="CS_1913308791_13086732" ]
"CS_877950025_32961174"
[label="CS_877950025_32961174" ]
"CS_1448178545_28349544"
[label="CS_1448178545_28349544" ]
"CS_-922782094_31414927"
[label="CS_-922782094_31414927" ]
"CS_-554546748_11707096"
[label="CS_-554546748_11707096" ]
"CS_-2133694150_31538514"
[label="CS_-2133694150_31538514" ]
"CS_-998238848_8452719"
[label="CS_-998238848_8452719" ]
"errorState_30495813"
[label="errorState_30495813\n(error)"]

"CS_1555958612Start_28637909" -> "CS_-211101784_12577309"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance"]
"CS_-211101784_12577309" -> "CS_-868287843_813251"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#LogMonitorAdapter"]
"CS_-868287843_813251" -> "CS_668692069_11665455"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance"]
"CS_668692069_11665455" -> "CS_-1939490076_2771331"
[label="org.apache.log4j.lf5.LogLevel#getJdk14Levels"]
"CS_-1939490076_2771331" -> "CS_1601600884_4219289"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance"]
"CS_1601600884_4219289" -> "CS_1913308791_13086732"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel"]
"CS_1913308791_13086732" -> "CS_877950025_32961174"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#setSevereLevel"]
"CS_877950025_32961174" -> "CS_1448178545_28349544"
```

```
[label="org.apache.log4j.lf5.LogLevel#getLog4JLevels"]
"CS_1448178545_28349544" -> "CS_-922782094_31414927"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance"]
"CS_-922782094_31414927" -> "CS_-554546748_11707096"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel"]
"CS_-554546748_11707096" -> "CS_-2133694150_31538514"
[label="org.apache.log4j.lf5.util.LogMonitorAdapter#"]
"CS_-2133694150_31538514" -> "CS_-998238848_8452719"
[label="EOS"]
}
```

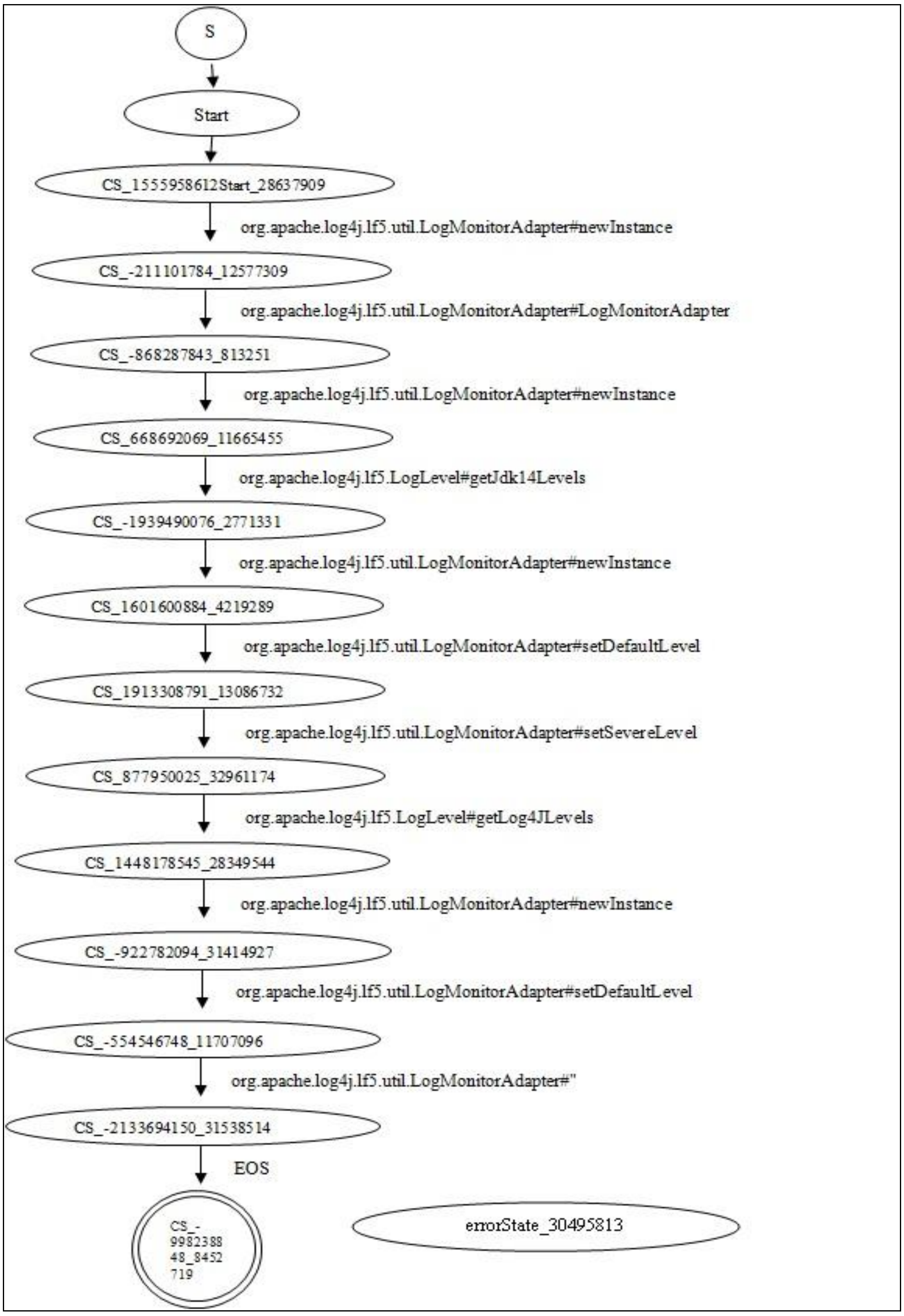

S
Start
CS_1555958612Start_28637909
org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance
CS_-211101784_12577309
org.apache.log4j.lf5.util.LogMonitorAdapter#LogMonitorAdapter
CS_-868287843_813251
org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance
CS_668692069_11665455
org.apache.log4j.lf5.LogLevel#getJdk14Levels
CS_-1939490076_2771331
org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance
CS_1601600884_4219289
org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel
CS_1913308791_13086732
org.apache.log4j.lf5.util.LogMonitorAdapter#setSevereLevel
CS_877950025_32961174
org.apache.log4j.lf5.LogLevel#getLog4JLevels
CS_1448178545_28349544
org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance
CS_-922782094_31414927
org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel
CS_-554546748_11707096
org.apache.log4j.lf5.util.LogMonitorAdapter#"
CS_-2133694150_31538514
EOS
CS_-998238848_8452719
errorState_30495813

Figure 4.10. Call graph produced for new Instance service from Log Monitor Adapter interface for UML2 9.0

Based on Figure 4.10, this call graph shows nine traces of interfaces invoked that are executed and ten states that were involved in this services. The traces of interfaces invoked are,

org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance,

org.apache.log4j.lf5.util.LogMonitorAdapter#LogMonitorAdapter,

org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance,

org.apache.log4j.lf5.LogLevel#getJdk14Levels,

org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance,

org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel,

org.apache.log4j.lf5.util.LogMonitorAdapter#setSevereLevel,

org.apache.log4j.lf5.LogLevel#getLog4JLevels,

org.apache.log4j.lf5.util.LogMonitorAdapter#newInstance,

org.apache.log4j.lf5.util.LogMonitorAdapter#setDefaultLevel,

org.apache.log4j.lf5.util.LogMonitorAdapter#, lastly is EOS for this service. For the states invoked are start from CS_1555958612Start(28637909), followed by CS_-211101784_12577309, CS_-868287843_813251, CS_668692069_11665455, CS_-1939490076_2771331, CS_1601600884_4219289, CS_1913308791_13086732, CS_877950025_32961174, CS_1448178545_28349544, CS_-922782094_31414927, CS_-554546748_11707096, CS_-2133694150_31538514, CS_-2133694150_31538514, and finally for end of states is CS_-998238848_8452719. Last, for error state is errorState_30495813. The hash (#) symbol is used to separate between the components interface and their method name.

4.5 Summary

This chapter is a discussion of the actual processes involved in the model representing a call graph. The technique that was applied in developing the prototype tool has been presented. A section explaining in detail the methods to represent a component in a call graph has been presented. The results in the form of a dot file was produced and can be viewed using any visualization tools to show the interactions between components in the software. The size of the call graph captured also was discussed.

Two existing forms of software were chosen as samples. The existing tool is used to extract software information from JAR files. The output file produced from the existing tool will be used by the program component extractors, which are ExtractJavaGenerator, ExtractTestDataNew and CreateGraph. These three programs were used to extract the component services software and implementation traces. To view the call graph in the visualization format, an existing tool is used which is Graphviz.

Graphviz provides several graphing possibilities, but for this research, the evaluation is only focused on its directed graph capabilities using the dot language. In this code, the dot file that was created by our program is used to generate test.dot graph specifications and produce a PDF format in the file test.pdf. This visualization uses the PDF format. The dot tool also supports other image formats including JPG, GIF, PNG, and postscript. The next section is a presentation of the evaluation of the model of call graph representation.

CHAPTER FIVE

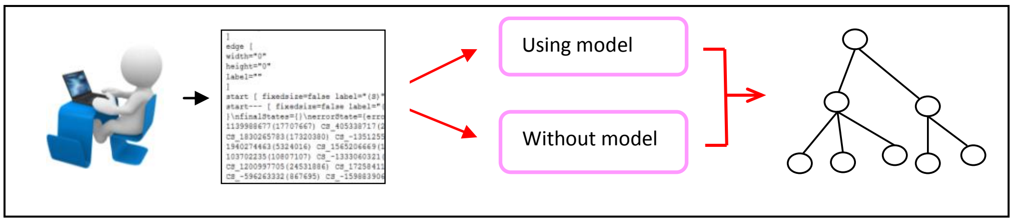
EVALUATION

5.1 Introduction

This chapter is a discussion of the procedure and the results taken from the laboratory experiment and the comparative study to evaluate the technique that was applied in the models of representation that are proposed in this research. As there is no model similar to the proposed model, a comparison was made between the technique that is proposed by this model and the manual code review method. This comparison was conducted by experts.

5.2 Evaluation processes

Figure 5.1 below shows the evaluation process for this research.



```
]
edge [
width="0"
height="0"
label=""
]
start [ fixedsize=false label="(S)"
start--- [ fixedsize=false label="(
]\nfinalStates=[]\nerrorState=[erro
1139988677(17707667) cs_405338717(2
cs_1830265783(17320380) cs_-1351255
1940274463(5324016) cs_1565206669(1
103702235(10807107) cs_-1333060321(
cs_1200997705(24531886) cs_17258411
cs_-596263332(867695) cs_-159883906
```

Figure 5.1. Evaluation process

Figure 5.1 is an illustration of the process of evaluation. First, the source code from the existing sample software is required. To visualize a call graph, experts are compulsory to use raw material, a source code that was extracted by an existing tool, JUnit. The component was extracted using a program that was proposed in the model and extracted using manual extraction. The result can be measured based on 10 criteria proposed by Baecker (1988), which will be compared based on using the proposed model and without using the model (manual method).

5.3 Evaluation procedure

This section is an explanation of the procedure taken to evaluate the proposed model. The activities in this phase are designing a questionnaire, identifying the right respondents, collecting data procedures, and analyzing the collected data.

The tests were conducted on a Pentium IV, 3.20Ghz With 2GB RAM. The model was run as a Java project under the Eclipse 3.3 platform, running a Java 1.6 Virtual Machine. The tests were iterated three times to ensure that the results were consistent.

The laboratory experiment conducted and the adopted questionnaires were based on Beacker (1998). A set of questionnaires were used during the data collection, so that the experts could provide their feedback systematically. The data collection procedure was conducted at a computer laboratory in Universiti Teknologi MARA Pulau Pinang. These experiments were conducted in the middle of the semester. The results will be summarized based on the questions that were distributed to the respondents during the experiment.

5.3.1 Respondent’s profile

The experiment was conducted at a local university in Malaysia with three respondents. All of them are female and work as lecturers at a local university. The first respondent, who is age 39, has eight years teaching experience at a local university in programming subjects, which are C++ and Java. She graduated with a Bachelor of Science in Computer Science. Her major study is software tools and system programming. She completed her Master of Science in the area of information technology. She has had four years of industrial experience as a system developer. Her expertise is in the area of JavaScript, Linux administrator, and PHP/MySQL programming.

The second respondent, who is 37, has eight years’ experience in teaching at a local university in programming subjects, which are C++ and Java. The graduated with a Bachelor of Computer and Information Sciences. Her major study is software development. She completed her Master of Science in the area of information technology. She has four years’ industrial experience as a project manager. Her expertise is in the area of project management, Java, Microsoft and LIPS programming.

The third respondent, who is age 37, has six years’ experience in lecturing at a local university for programming subject, which are C++ and Java. She has a Bachelor of Science in Computer Science. The major of her study is software engineering. She completed her Master of Science in the area of information technology. She has four years’ industrial experience as a Software Test Engineer. Her expertise is in the area of software testing with experience in JavaScript, C++, C shell, and PHP/MySQL programming. The respondents’ profile form is attached in Appendix A.

5.4 Result and discussion

This section is a discussion of the results of the comparative study. Table 5.1 shows the results from each respondent. A detailed description of the table is mentioned below.

Table 5.1

Result of comparative study

	Descriptions	Proposed Model			Manual Method		
		R1	R2	R3	R1	R2	R3
1.	Does the technique maintain a repository of architectural and/or behavioural information about a program?	√	√	√	√	√	√
2.	Does the technique provide a presentation model for visualizing information about programs in various ways (e.g. graph, code)?	√	√	√	X	X	X
3.	Does the technique provide transformations for going from one kind of representation to another (e.g., text to graph)?	√	√	√	X	X	X
4.	Does the technique increase the speed by which the users are able to read code?	√	√	√	X	X	X
5.	Do users prefer the technique representation?	√	√	√	X	X	X
6.	Do users remember more of what they read when they use the technique?	√	√	√	X	X	X
7.	Do users learn more easily or more quickly if they use the technique?	√	√	√	X	X	X
8.	Does the technique use basic symbols, which are simple, elegant, and easier to be understood in the flow of a program?	√	√	√	X	X	X
9.	Does the technique use special graphical symbols to indicate the task of the program?	√	√	√	X	X	X
10.	Does the technique clearly describe the flow of program?	√	√	√	X	X	X

Note: Agree (√) or Disagree (X)

Table 5.1 above shows the results, which have been gathered for the three respondents who are involved in evaluating the proposed model. R1, R2, and R3 represent respondent 1, respondent 2, and respondent 3. The question to evaluate the criteria refers to section 3.2.4.2 in Chapter 3. The symbol √ shows that the respondents agree with the question and symbol X shows the respondents do not agree to the questions.

The summaries of the comparative study are as follow:

i. By using the proposed model and the manual method, the information can be maintained that is shown in the presentation where the original source codes are still displayed with the same results. These two techniques also provide the same information at the end of experiment. This aspect is covered by question number 1.

ii. The proposed model helps users in increasing reading and comprehending a source code. By presenting the model, a source code can be shown in the form of a graphical view. The user can remember more of what he or she reads and learn easily and quickly. This technique can improve a user's memory and learnability. This aspect is covered by questions 2, 3, 4, 5, and 6.

iii. The technique proposed uses basic symbols to show the flow of programs. By using the basic symbol, it is more simple, elegant, and easier for users to understand the program flow. Because of respondents' familiarity with the ANSI flowchart when presenting the program flow, it is easier for them to understand the flow quickly. The ANSI flowchart attracts the users with its various forms of diagrams to show the program task. By using the manual method, respondents need to draw the graph by themselves; it is quite difficult because each respondent has his or her own opinions in representing the program flow. This aspect is covered by questions 7, 8, and 9.

iv. By using the proposed method, the program flow provides clearer descriptions as the flow shows the integration of its component in a program. By using the manual method, the presentations will be in various ways, because the respondents need to analyse the source code that is text-based and draw the component interaction information into graphical format using their own opinions. This aspect is covered by question 10.

From the comparative study, it can be concluded that the proposed technique can give a greater contribution to the users in enhancing the program comprehension in representing component integration.

5.5 Summary

This chapter is a discussion of the result from analysis of data that evaluate the effectiveness and efficiency by using the proposed method as compared to the manual method in gathering component interaction information. The result for evaluation of respondents' preferences shows that most respondents prefer the proposed technique compared to the manual method. This is because by using the proposed method, time taken is reduced and it is easy to understand the flow of the program. The results can be concluded based on the data that were analysed from the questionnaire given when the experiment was conducted. Moreover, this study also shows that the proposed method can be a broad contribution in improving program comprehension.

CHAPTER SIX

CONCLUSION

In this research work, a model of software component interactions using the call graph technique is proposed. The technique used in capturing the components interaction information has been investigated. To evaluate the program extraction, several existing tools have been used to understand better how to provide valuable information related to software component interaction. This chapter is organised as follows: section 6.1 is a presentation of the summary of the research findings, section 6.2 is a discussion of the achievement of the objectives, section 6.3 is a presentation of the contributions of this research, and finally, section 6.4 is a discussion of limitations of the study. Finally, section 6.5 is a discussion of the directions for future works.

6.1 Achievement of Objectives

As stated in chapter 1, this research is intended to achieve several objectives to solve the problem statement. Therefore, this sub-section will discuss the achievement of the research objectives.

6.1.1 Research Objective 1

The first objective is to propose a model for representing software component interactions. This objective is achieved by conducting a literature review in chapter 2. The model is reviewed in detail by identifying the techniques to extract software component interaction information and several model representations of software. As a result, a decision was made to use the call graph to visualize component interactions in software. This call graph is used especially for novices who want to improve their understanding of the component interactions. In the maintenance request, the call graph will help the programmer to identify directly which part of code that he or she wants to modify. Furthermore, this model also can help the users to understand a program by translating it into a graphical view to show the program code and component interaction flow.

6.1.2 Research Objective 2

The second objective is to extract traces of component interactions. The objective is to capture the component interactions, where programs were developed successfully and executed properly to achieve this objective. With the use of an existing reverse engineering tool, three programs were developed to extract component interaction information from the software. These programs are *ExtractJavaGenerator*, *ExtractTestDataNew* and *createGraph*. The results will produce a call graph to show the component services implementation trace in dot file format. The processes of the implementation were discussed in Chapter 4.2.3. The programs were implemented in the simulation environment using Java. The results from the programs are successful where they met the objective.

6.1.3 Research Objective 3

The third objective is to define the process in representing software components interaction model by using a suitable visualization tool for software representation. The purpose of this objective is to evaluate the result that has been made in objective two, to ensure that the call graph produced is following the right graph format. To evaluate the text, it is translated and displayed in call graph format with a combination of text; it becomes more translucent to understand the information presented. Graphviz is used as a suitable tool to visualize the component interactions. Based on the results from the extractor programs, which have been developed in objective two, Graphviz is able to visualize the created dot file format in the form of a call graph. Because of the large number of components in the software, the call graph produced in this thesis has been modified to avoid complexity in the graph view. Thus, objective three to evaluate a call graph produced is achieved successfully.

In addition, a group of three experts have evaluated the proposed model process of a call graph. They must answer the questions to evaluate the ease of use by comparing the proposed model to the manual method in gathering component integration. The result for evaluation of respondents' preference shows that most respondents prefer the proposed technique compared to the manual method. This is because, by using the proposed method, the time taken is reduced and it is easy to understand the flow of the program. Therefore, objective three is achieved.

After discussing the objectives that were successfully achieved, the research contributions are described in the next section.

6.2 Research Contributions

The major contributions from this research can be summarized as follows:

1. From the theory point of view, the produced model of software component interactions provides a systematic process to extract and represent the information of components related to their interactions, positions, and the names of the components that are involved in the interaction. The information captured in this model can be used to manage and test the components involved in integration testing.

2. From the practical point of view, in presenting a large amount of software component interaction in a scalable way, the proposed model gives a benefit for people who are in the area of software comprehension, which requires them to extract the information in many purposes of study. For example, they can use the information to upgrade software, to detect weaknesses of the new system, to enhance their business processes, and others.

3. The produced model of software component interactions provides effective ways for those who are unfamiliar with the location of software components by providing them in the model, which makes it easier for them to

comprehend the flow of software systems when compared to code review. By using this model, the user will increase the ability to view the interactions of all components.

6.3 Limitation of study

The execution of the experiment in this study was very dependable on two existing tools, which are Ptidej and Graphviz. Therefore, the type and size of input and output files used are limited to the one accepted by these two tools.

6.4 Future Work

This work can be continued in the following directions for future research:

1. In this research, component interaction is represented in single software. In the future, the work can be extended by representing components from different versions of software. This information is useful to identify the changes in software within different versions.

2. The technique used in this research is static analysis, which involves extracting static properties of program source code without actually executing the program. This can be enhanced by using a dynamic analysis technique, which allows extracting component interaction information while the program is executing.

3. For extended study, other graphical software can be used to make a comparison to obtain more accurate study results.

REFERENCES

Abran, A., Moore, J., Bourque, P., Dupuis, R., & Tripp, L. (2004). Guide to the software engineering body of knowledge: 2004 version: *IEEE Computer Society*, Los Alamitos, CA. Retrieved 23 July 2010, from http://www.swebok.org

Acharya, R. (2013). *Object-oriented design pattern extraction from Java source code*. Master's thesis. UPPSALA University.

Ackermann, C. & Lindvall, M. (2007). *Understanding change requests to predict software impact.* Paper presented at the Software Engineering Workshop, 2006. SEW'06.

Aldrich, J., Chambers, C., & Notkin D. (2002). Architectural reasoning in ArchJava. In Proceedings ECOOP 2002, volume 2374 of LNCS, pp 334-367. Berlin, DE: Springer Verlag.

Ali, Q. (2008). *Static program visualization within the ASF+ SDF meta-environment.* Master's thesis. University of Amsterdam, Holland, Netherlands.

Arafa, Y., Boldyreff, C., Tawil, A. H., & Liu, H. (2012). *A high level service-based approach to software component integration.* Paper presented at Sixth International Conference on the Complex, Intelligent and Software Intensive Systems.

Arboleda, H., Royer, J. C (2011). *A comparison of two Java component extraction approaches*. Paper presented at Proceedings of the 4th India Software Engineering Conference (ISEC '11), New York, NY, USA. pp. 155-164.

Ardakan, M. A. & Mohajeri, K. (2009). Applying design research method to IT performance management: Forming a new solution. *Journal of Applied Sciences, 9*(7), 1227-1237.

Arrington, C. T. (2001). *Enterprise Java with UML*. New York, NY: John Wiley & Sons.

Ayewah, N., Hovemeyer, D., Morgenthaler, J., Penix, J., & Pugh, W. (2008). Using static analysis to find bugs. *Software, IEEE, 25*(5), 22-29.

Baecker, R. (1988). *Enhancing program readability and comprehensibility with tools for program visualization*. Proceedings of the 10th international conference on Software engineering. pp. 356-366.

Becker, B. (2011). Modeling and verification of self-adaptive service-oriented systems. *Proceedings of the 5th Ph. D. Retreat of the HPI Research School on Service-oriented Systems Engineering, 5,* 149.

Bergeron, J., Debbabi, M., Desharnais, J., Erhioui, M., Lavoie, Y., & Tawbi, N. (2001). Static detection of malicious code in executable programs. *International Journal of Requirements Engineering*, 184-189.

Bollobas, B. (1998). *Modern graph theory*. New York, NY: Springer-Verlag.

Briand, L., Labiche, Y., & Sówka, M. (2006). *Automated, contract-based user testing of commercial-off-the-shelf components.* Paper presented at the Proceedings of the 28th international conference on Software engineering.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*. 18, 543-554.

Broy, M. & Kruger, I. (1998). *Interaction interfaces-towards a scientific foundation of a methodological usage of message sequence charts.* Proceedings of the Second International Conference on Formal Engineering Methods.

Brusilovsky, P., Grady, J., Spring, M. & Lee, C. (2006) What should be visualized? Faculty perception of priority topics for program visualization. In: *ACMSIGCSE Bulletin*, *38*(2), 44-48.

Bures, T., Hnetynka, P., & Malohlava, M. (2009). *Using a product line for creating component systems*. Paper presented at Proceeding of the 2009 ACM symposium on Applied Computing. pp. 501-508.

Caserta, P. & Zendra, O. (2011). Visualization of the static aspects of software: A survey. *IEEE Transactions Journal on Visualization and Computer Graphics, 17*(7), 913-933.

Chatzigeorgiou, A., Tsantalis, N., & Stephanides, G. (2006). *Application of graph theory to OO software engineering*. Paper presented at the Proceedings of the 2006 international workshop on interdisciplinary software engineering research, Shanghai, China.

Chen, H., Dean, D., & Wagner, D. (2004). *Model checking one million lines of C code.* Paper presented at the Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS).

Chen, K. & Rajlich, V. (2000). Case study of feature location using dependency graph. Paper presented at the Proceedings of Intern. Workshop on Program Comprehension (IWPC'00), pp. 241-249.

Chen, K. & Rajlich, V. (2001). *RIPPLES: Tool for change in legacy software*. Paper presented at the Proceedings of IEEE International Conference on Software Maintenance.

Chikofsky, E. J. & Cross II, J. H. (1990). Reverse engineering and design recovery: A taxonomy in IEEE software. *IEEE Computer Society*, 13-17.Chouambe, L., Klatt, B., & Krogmann, K., (2008). Reverse engineering software-models of component-based systems. European Conference on Software Maintenance and Reengineering, pp. 93-102. IEEE.

Claub, M. (2001). *Generic modeling using UML extensions for variability*. Paper presented at the Workshop on Domain Specific Visual Languages at Object-Oriented Programming, Systems, Languages, & Applications.

Corritore, C. L. & Wiedenbeck, S. (2001) An exploratory study of program comprehension strategies of procedural and object-oriented programmers. *International Journal of Human-Computer Studies, 54*(1). pp. 1-23.

Crnkovic, I. (2003). *Component-based software engineering-new challenges in software development*. Paper presented at Proceedings of the 25th International Conference on Information Technology Interfaces (ITI).

Cross, J. H., Hendrix, T. D., & Maghsoodloo, S. (1998). The control structure diagram: An overview and initial evaluation. *Empirical Software Engineering*, *3*(2), 131-158.

Deo, N. (2004). *Graph theory with applications to engineering and computer science*. New Delhi, India: PHI Learning Pvt. Ltd.

Détienne, F. & Bott, F. (2001) Software design—Cognitive aspects. New York, NY: Springer-Verlag.

Devadas, S., & Lehman, E. (2005). *Mathematics for computer science*. Retrieved 12 May 2010, from http://ocw.mit.edu/NR/rdonlyres/Electrical-Engineering-and-Computer-Science/6-042JSpring-2005/FF95BA8F-6457-4592-B473-349A5C1CA277/0/l8_graphs1.pdf

Diehl, S. (2007). *Software visualization: Visualizing the structure, behaviour, and evolution of software, Secaucus,* NJ. New York, NY: Springer-Verlag.

Eick, S., Graves, T., Karr, A., Mockus, A., & Schuster, P. (2002). Visualizing software changes. *IEEE Transactions on Software Engineering, 28*(4), 396-412.

El-Ramly, M. (2006). *Experience in teaching a software reengineering course.* Paper presented at the Proceedings of the 28th International Conference on Software Engineering, New York.

Emanuelsson, P., & Nilsson, U. (2008). A comparative study of industrial static analysis tools. *Electronic Notes in Theoretical Computer Science, 217,* 5-21.

Evalguide Online.org (n.d). Retrieved 18.06.2012, from: http://www.evalguide.ethz.ch/project_evaluation/introduction/goalbased_evaluation_EN

Ferrante, J., Ottenstein, K., & Warren, J. (1987). The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS), 9*(3), 319-349.

Fiege, L. (2005). *Visibility in event-based systems.* (PhD. Thesis). Technische Universität Darmstadt, Darmstadt, Germany.

Gansner, E. R. (2012). *Drawing graphs with Graphviz* technical report. Murray Hill, NJ: AT&T Bell Laboratories.

Garousi, V. & Koochakzadeh, N. (2010). An empirical evaluation to study benefits of visual versus textual test coverage information. *Testing—Practice and Research Techniques*, 189-193.

Garrison, R. (2000). Theoretical challenges for distance education in the 21st century: A shift from structural to transactional issues. *The International Review of Research in Open and Distance Learning, 1*(1).

Gelber, N. (2006). *Bridging component models and integration problems*. Växjö University, Sweden. Retrieved 20 February 2010, from http://urn.kb.se/resolve?urn=urn:nbn:se:vxu:diva-677.

Gibbons, A. (1985). *Algorithmic graph theory*. Cambridge, UK: Cambridge University Press.

Grove, D. & Chambers, C. (2001). A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems, 23*(6), 746.

Grove, D., DeFouw, G., Dean, J., & Chambers, C. (1997). Call graph construction in object-oriented languages. *ACM SIGPLAN Notices, 32*(10), 108-124.

Grubb, P. & Takang, A. (2003). *Software maintenance*. Singapore, Singapore: World Scientific Publishing Company, Incorporated.

Gueheneuc, Y. G. (2005). *Ptidej: Promoting patterns with patterns*. Paper presented at European Conference on Object Oriented Programming, workshop on Building a System with Patterns, Glasgow, Scotland.

Hopkins, J. (2000). Component primer. *Communications of the ACM, 43*(10), 27-30.

Inverardi, P. & Tivoli, M. (2003). Software architecture for correct components assembly. *Formal Methods for Software Architectures*, 92-121.

Iwanari, Y., Tasaki, M., Yokoo, M., Iwasaki, A., & Sakurai, Y. (2009). *Introducing communication in Dis-POMDPs with finite state machines.* Paper presented at IEEE/WIC/ACM International Joint Conferences on Web Intelligence and Intelligent Agent Technologies WI-IAT'09.

Jackson, C. & Pascual, R. (2008). Optimal maintenance service contract negotiation with aging equipment. *European Journal of Operational Research, 189*(2), 387-398.

Jin, Y. & Han, J. (2005). Specifying interaction constraints of software components for better understandability and interoperability. *COTS-Based Software Systems Journal*, 54-64.

Kaeli, A. & Calder, B. (1997). *Procedure mapping using static call graph estimation.* Paper presented at the Proceeding of Workshop Interaction between Compiler and Computer Architecture.

Koehler, J. & Vanhatalo, J. (2007). Process anti-patterns: How to avoid the common traps of business process modelling. *IBM WebSphere Developer Technical Journal, 10*(2), 4.

Koivulahti-Ojala, M. & Käkölä, T. (2010). *Framework for evaluating the version management capabilities of a class of UML modeling tools from the viewpoint of multi-site, multi-partner product line organizations.* Paper presented at the Proceedings of 43rd Hawaii International Conference on Systems Sciences (HICSS-43). IEEE.

Kothari, J., Denton, T., Shokoufandeh, A., Mancoridis, S., & Hassan, A. (2006). *Studying the evolution of software systems using change clusters.* Paper presented at the Proceedings of the 14th International Conference on Program Comprehension. IEEE Computer Society.

Kuck, D., Kuhn, R., Padua, D., Leasure, B., & Wolfe, M. (1981). *Dependence graphs and compiler optimizations.* Paper presented at the Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of programming languages.

Lanza, M. (2001). *The evolution matrix: Recovering software evolution using software visualization techniques.* Paper presented at the Proceedings of the 4th international workshop on principles of software evolution.

Lavagno, L., Martin, G. E., & Selic, B. (2003). *UML for real: Design of embedded real-time systems*. Alphen aan den Rijn: Netherlands, Kluwer Academic Publishers.

Lee, D. & Yannakakis, M. (1996). Principles and methods of testing finite state machines. *PROCEEDINGS-IEEE, 84,* 1090-1123.

Letovsky, S. (1986a) Cognitive processes in program comprehension. In *Empirical Studies of Programmers*. 58-79. New York, NY: Ablex Publishing Corporation.

Lilius, J. & Paltor, I. (1999). *Formalising UML state machines for model checking.* Paper presented at the Unified Modeling Language: UML'99: Beyond the Standard: Second International Workshop, Fort Collins, CO, October 28-30, 1999: Proceedings.

Lin, Y., Zhang, S., & Zhao, J. (2009). *Incremental call graph reanalysis for AspectJ software*. Paper presented at the International Conference on Software Maintenance. (ICSM 2009). IEEE.

Lindvall, M. & Publica, F. (2003). Impact analysis in software evolution. *Advances in computers, 59*, 130-211.

McMillan, J. H. & Schumacher, S. (1984). *Research in education: A conceptual introduction*. Boston, MA: Little, Brown.

Muller, H. & Klashinsky, K. (1988). *Rigi—A system for programming-in-the-large.* Paper presented at the Proceedings of the 10th International Conference on Software Engineering.

Murata, M., Tozawa, A., Kudo, M., & Hada, S. (2006). XML access control using static analysis. *ACM Transactions on Information and System Security, 9(3),* 324.

Najumudheen, E., Mall, R., & Samanta, D. (2009). A dependence graph-based representation for test coverage analysis of object-oriented programs.

SIGSOFT Software Engineering Notes, 34(2), 1-8. 100-113. Norwood, NJ: Ablex.

O'Brien, M. (2003) Program comprehension—A review & research direction technical report UL-CSIS-03-3, University Of Limerick.

Ore, O. (1967). *Theory of graphs*. Providence, RI: American Mathematical Society.

Pennington, N. (1987). Comprehension strategies in programming. G. M. Olson, S. Sheppard, & E. Soloway, (eds.). Empirical Studies of Programmers: Second Workshop.

Piel, E. & Gonzalez-Sanchez, A. (2009). *Data-flow integration testing adapted to runtime evolution in component-based systems.* Paper presented at the Proceedings of the ESEC/FSE workshop on Software integration and evolution@ runtime.

Price, B. A., Small, I. S., & Baecker, R. M. (1992). *A taxonomy of software visualization*. Proceedings of the Twenty-Fifth Hawaii International Conference on System Sciences, vol 2, pp. 597-606.

Prowell, S. (2005). *Using Markov chain usage models to test complex systems*. Paper presented at the Proceeding 38th Annual Hawaii International Conference. System Sciences (HICSS'05), Big Island, Hawaii.

Rader, J. (1997). *Mechanisms for integration and enhancement of software components.* Paper presented at the Proceedings Fifth International Symposium on Assessment of Software Tools and Technologies.

Rajan, H. & Sullivan, K. (2005). *Classpects: Unifying aspect- and object-oriented language design.* Paper presented at the Proceedings of 27th International Conference on Software Engineering (ICSE).

Rajlich, V. (1997). *A model for change propagation based on graph rewriting.* Paper presented at the 13th International Conference on Software Maintenance (ICSM'97), October 1-3 2007, Wayne State University (pp. 84), Bari, ITALY.

Rajlich, V., Damaskinos, N., Linos, P., & Khorshid, W. (2006). VIFOR: A tool for software maintenance. *Software: Practice and Experience, 20*(1), 67-77.

Reekie, H. & Lee, E. (2002). *Lightweight component models for embedded systems*. Electronics Research Laboratory, College of Engineering, University of California.

Royer, J. C. (2010). *Short draft about the Java component extractor*. Academic Society For Competition Law, INRIA Nantes, France: Mines de Nantes.

Ruiz, M., Espana, S., & Gonzalez, A. (2012). *Model-driven organisational reengineering: A framework to support organisational improvement.* Paper presented at the Informatica (CLEI), 2012 XXXVIII Conferencia Latinoamericana En.

Saidi, H. (2008). Logical foundation for static analysis: Application to binary static analysis for security. *ACM SIGAda Ada Letters, 28*(1), 96-102.

Schilling, J. W. (2007). A cost effective methodology for quantitative evaluation of software reliability using static analysis. The University of Toledo.

Schilling, W. & Alam, M. (2008). *A methodology for quantitative evaluation of software reliability using static analysis.* Paper presented at the Proceedings of the 2008 Annual Reliability and Maintainability Symposium. volume 00.

Scholz, B., Zhang, C., & Cifuentes, C. (2008). *User-input dependence analysis via graph reachability*. Mountain View, CA: Sun Microsystems, Inc.

Sherburne, P. & Fitzgerald, C. (2004). You don't know jack about VoIP. *Queue, 2*(6), 30-38.

Shneiderman, B. & Mayer, R. (1979) Syntactic semantic interactions in programmer behavior: A model and experimental results. In *International Journal of Computers & Information Sciences*, *8*(3), 219-238.

Soloway, E. & Ehrlich, K. (1984) Empirical studies of programming knowledge. In *IEEE Transactions on Software Engineering*, SE-10, 595-609.

SoMoX SOftware MOdel eXtractor, Retrieved 10.02.2013, from http://www.somox.org

Suman, R. R. & Mall, R. (2009). State model extraction of a software component by observing its behaviour. *SIGSOFT Software Engineering Notes, 34*(1), 1-7.

Szyperski, C. (1998). *Component software: Beyond object-oriented programming*, New York, NY: ACM Press/Addison-Wesley Publishing Co.

The JCE checker. (2013). Retrieved 2 November 2013, from http://www.emn.fr/z-info/jroyer/JCE/index.html.

Travis, D. (2007). Retrieved 24 July 2013. http://www.userfocus.co.uk/articles/expertreviews.html

Vaishnavi, V. K. & Kuechler, W. (2008). *Design science research methods and patterns* (1st Ed.). Boca Raton, FL: Auerbach Publications.

Von Mayrhauser, A. & Vans, A. M. (1995). Program understanding: Models and experiments. *Advances in Computers, 40*(4), 25-46.

Wen, L. & Dromey, R. (2004). *From requirements change to design change: A formal path.* Paper presented at SEFM 2004. Proceedings of the Second International Conference on Software Engineering and Formal Methods.

Winslow, L. E. (1996). Programming pedagogy—A psychological overview. *ACM SIGCSE Bulletin*, *28*(3), 17-22.

Wu, X. & Woodside, M. (2004). Performance modelling from software components. *ACM SIGSOFT Software Engineering Notes, 29*(1), 301.

Xue, J., Hu, C., Wang, K., Ma, R., & Leng, B. (2009). *Constructing a knowledge base for software security detection based on similar call graph.* Paper presented Second International Conference on Computer and Electrical Engineering.

Zhang, L., Marinov, D., Zhang, L., & Khurshid, S (2011). *An empirical study of JUnit test-suite reduction*. Paper presented at 22nd IEEE International Symposium on Software Reliability Engineering.

Zwillinger, D. (2002). *CRC standard mathematical tables and formulae.* Boca Raton, FL: CRC Pr I Llc.