CHAPTER 1

INTRODUCTION

1.1 Introduction to Software Maintenance

All objects in the world need to be maintained throughout their life cycles in order to survive and remain useful; and so does computer software. From time to time a software system evolves to meet new requirements and changes. When the technology or language of the software system is obsolete, it becomes a legacy system yet it is still in use to serve users' needs. Maintaining a software system particularly a legacy system definitely needs a lot of software engineers' time and effort especially when there is no document available or the existing documents are not updated.

According to Erlikh (2000), 85 to 90 percent of Information System budgets go to legacy system operation and maintenance, which is extremely high. In addition, Sommerville (1997) cites cost in maintenance could be double to that of development, unless extra development costs are invested in making a system more maintainable that can decrease maintenance costs including overall system costs. Hence software maintenance should be seen as an important stage in a software life cycle and need a lot of attention from researchers and practitioners.

Whenever software maintainers plan to modify or enhance a software system, they need to refer to ***system documentation (SD)*** in order to understand the software better and determine which components are affected in a maintenance task. Nevertheless the system documentation available is almost always out-of-date and unreliable due to some reasons including time constraint and commercial pressures (Sulaiman *et al.*,

2002b). As a result software maintainers need to read through all the source codes, which are the most reliable and updated source of the running system. This is very tedious task and often impractical. In this case, the use of ***CASE (Computer-Aided Software Engineering)*** products such as reverse engineering tool will be able to automatically extract components in existing source codes and provide graphical representations of the components to assist software engineers' ***program comprehension*** or ***software understanding*** (von Mayrhauser and Vans, 1993).

Program comprehension or software understanding involves cognition of software that is the mental process of knowing, learning and understanding the software system (Collins, 1988). A number of research and studies have been conducted in order to assist the ***cognition*** aspect of a software system based on source codes. One way is through reverse engineering technique in which source codes will be parsed and the information extracted will be visualized. This requires a software visualization method to represent the extracted information graphically (Kwon *et al.*, 1998). There are several approaches in software visualization with the target to improve cognition of software system. Most approaches use graph as the main element to visualize extracted software artifacts in a reverse engineering environment. According to the survey by Koschke (2003), 49 percent of the respondents used graph to visualize software artifacts besides text (14 percent) and ***Unified Modeling Language (UML)*** diagram (13 percent).

However, based on the literature review (to be discussed in **Chapter 2**) and also the comparative study conducted on the diverse software visualization methods employed by the existing reverse engineering tools (to be discussed in **Chapter 3**), it was discovered that the tools had some weaknesses such as the information provided is too detail causing the graphs generated are quite complex and they do not grant an explicit software re-documentation mechanism. Some of the weaknesses make the tools become less effective to be employed by software maintainers. Thus the main problem includes the issue of how to produce a more effective method in re-documenting and visualizing software artifacts that can enhance cognition of existing software systems in software maintenance. Hence, this thesis has attempted to eliminate the problem by proposing a document-like software visualization method named as ***DocLike Modularized Graph (DMG)*** employed in a prototype tool called DocLike Viewer (to be described in **Chapter 6**). The research also suggests the best stage to implement the tool in a reverse

engineering environment to capture the most updated software artifacts from the existing source codes.

This chapter will discuss more on the background of the problem followed by a brief discussion on current solutions and the proposed solution, statement of the problem, objectives of the research, theoretical framework, importance and scope of the research. Definition of terms and organization of thesis will be provided in the last two sections of the chapter.

1.2 Background of the Problem

Problems in software maintenance process are always related to how much system documentation of a software system can assist software engineers particularly programmers in understanding the architecture of the software system and identifying what are the components affected. As stated by van Vliet (2000), some of the major causes of maintenance problems are as follows:

(i) Unstructured code.

(ii) Maintenance programmers having insufficient knowledge of the system or application domain.

(iii) Documentation being absent, out-of-date or at best insufficient.

(iv) Software maintenance had a bad image.

The findings depict that the problems do not involve technical issues only but also managerial issues such as the bad image of software maintenance as seen by some software engineers.

In relation, the survey by Sousa and Moreira (1998) in Portugal shows that 3 biggest problems related to software maintenance process include:

(i) The lack of software maintenance process models.

(ii) The lack of documentation of applications.

(iii) The lack of time to satisfy the requests.

Both studies deduce that out-dated, incomplete or absence of documentation contribute to software maintenance problems. It will be worse if the documentation problem involves design level document, which is the most vital source to identify components of a system and their interrelationships. Indirectly the documentation problem leads to the problem in understanding or comprehending existing source codes. These problems will be the major focus of this thesis.

1.2.1 Incomplete, Out-dated, Non Standardized or Absence of Documentation

It is understood that each phase in a software development life cycle should have its generic document (Hoffer *et al.*, 1999). Nevertheless, during the initial development itself, documentation often comes off badly because of deadlines and other time constraints (van Vliet, 2000). The preliminary study of this thesis (Sulaiman *et al.*, 2002b) also found that the two main reasons for not producing system documentation were time constraint and commercial pressures. In addition most programmers tend to forget about the documentation when they need to start with a new software development project. In this case, unless the software managers or customers insist to have the documentation, then only the documentation will be produced otherwise the documentation will not exist at all. This factor was identified as the third reason why software engineers did not produce system documentation (Sulaiman *et al.*, 2002b).

In addition, software engineers face in average two maintenance projects without system documentation and they do not produce system documentation at least for one development project yearly. Furthermore, most programmers are product-oriented in which their main target is to make the system available on time and documentation comes later (Hoffer *et al.*, 1999). Besides, programmers dislike documenting their systems because it is perceived as a rather boring task compared to the excitements of creation in design and implementation (Macro, 1990). This is also shown in the survey of this research (Sulaiman *et al.*, 2002b) in which less than 4.1 percent of software engineers were interested in writing system documentation compared to developing (79.6 percent) and maintaining software (16.3 percent). Consequently, the quality of documentation is low because either it is done in a hurry or not with interest.

Software evolves, thus it always needs to be maintained due to the existence of new requirements, change in environment, transform to new language or many other factors. Thus software maintainability is extremely important to be considered in order to produce long lasting software. Some important factors in software maintainability are: the development process used, documentation and program comprehensibility (Pigoski, 1997). Therefore documentation is also vital to ensure maintainability. Nevertheless, only the documentation for the first maintenance could be reliable. The link between a program and its associated documentation is sometimes vanished during the maintenance process and this may be the consequence of poor configuration management or due to adopting a ***quick fix*** approach to maintenance (Sommerville, 1997). Therefore the existing documents are unreliable and out-of-date to be referred to for the following maintenance processes. Maintainers can only rely on the documents to derive the overall picture of the current system.

Thus software maintenance costs are increased due to the time required for a maintainer to understand the software design prior to being able to modify the source codes. Furthermore, most system documentation is produced in a hard copy to be kept as future reference. The soft copy of documentation is hardly being archived properly with the software version or revision. If a maintenance process does not involve major changes, it will be tedious and not practical to redo the documentation (especially if the soft copy has been poorly archived) and reprint the whole document together with the changes made. Hence, the following software maintainers probably will not bother to modify the documentation.

Documentation standard is important to be enforced in all software projects because its purpose is to communicate only necessary and critical information, not to communicate all information (Pigoski, 1997). However, most organizations do not employ any documentation standard (Sulaiman *et al.*, 2002b). Thus programmers who produce the documentation do not follow any formal guidelines. Consequently, documents produced in such organizations are of different format among software projects and it may not produce sufficient information as required by future software maintainers.

1.2.2 Understanding an Existing Software System is Tedious and Costly

Program comprehension is the most expensive task of a software maintenance process because it includes reading documents, scanning its source codes and understanding the change to be made (Kwon *et al.*, 1998). For instance, changing an existing variable or introducing a new variable in a function or method will cause a ripple effect towards other functions or methods, or even other programs or classes, or may be other modules or packages, or other integrated system. Furthermore, a concerned change involves the tracing of other de-localized components in the source codes. It could be in the same or different program or class, module or package, or even system. It will be more costly if the existing system does not have any design document or updated document. In this case, source codes are the only available source of information and the most reliable one. If the size of the software system is very large, the cognition of the software system will be more complex. Thus there is a need to assist the process of program comprehension in order to reduce the cost particularly when design document is absence.

Cognition of a software system involves different program comprehension strategies. According to Storey *et al.* (1999), these strategies can be influenced by the characteristics of maintainers, programs or maintenance tasks. Maintainer characteristics involve application and programming domain knowledge, maintainer expertise or creativity, familiarity with program and support tools expertise. Program characteristics involve application and programming domain, program size, complexity and quality, documentation and support tool availability. Tasks characteristics include task type and purpose, task size and complexity, time and cost constraints and environmental factors. For instance, an expert programmer might need less time to solve a quite difficult task compared to that of a novice programmer. Indirectly, the former incurs less cost than the latter. Therefore, to reduce cost by supporting programmers' cognitive strategies, the characteristics mentioned above should be considered.

1.2.3 CASE Tools or Environment are Not Used or Properly Used

Nowadays, there are a lot of CASE tools of different classes (see **Section 2.8**) that can assist software developers either in project planning, modeling in analysis and design, testing, producing documentation up to implementation that involves the whole ***System Development Life Cycle (SDLC).*** As for the maintenance all CASE tools might be used again while repeating the SDLC. Such fully integrated CASE tools or environment will normally produce a self-generated system documentation that will be very useful to maintainers. However most organizations do not use CASE tools to support all phases of SDLC (Sulaiman *et al.*, 2002b). Without a CASE environment or any CASE tools, software engineers need to use a word processor application to document the textual part and import all graphical representations drawn in other graphic application.

Theoretically, design document should be produced while designing a software system either by using structural modeling such as data flow diagrams and entity-relationship diagrams or object-oriented modeling such as UML and then the document is updated based on the final written and tested source codes. Otherwise, programmers need to read through the source codes again and transform them into graphical notation.

> This can be a tedious job, which requires considerable concentration. It is all too easy for concentration to lapse, for even a very short period, and miss some vital piece of information.
>
> (Lincoln, 1993)

There are a variety of reasons why organizations choose to adopt CASE partially or not to use it at all:

> These reasons range from a lack of vision for applying CASE to all aspects of the SDLC to the belief that CASE technology will fail to meet an organization's unique system development needs.
>
> (Hoffer *et al.*, 1999)

Software developers might just use the CASE tool for instance ***Rational Rose*** (Rational, 2004) to draw the diagrams that capture the user requirements in analysis and

design phase of SDLC. However, they do not use code generators or reverse engineering utility provided and also the document generator called ***SoDA*** (Rational SoDA for Word) that is integrated with Microsoft © Word, word processor and allows the importing of diagrams done during analysis and design into a specified document template. Therefore, there is a lack of integration between system documents and source codes causing the documents produced by the document generator is not reliable anymore. As a result, the developers still need to produce documentation manually in order to customize it to the particular system. In addition, to employ a CASE environment in an organization (particularly a software house) that produces software of different range of languages could be expensive and unrealistic. Only certain projects can implement the CASE tools since most CASE tools are dedicated for certain languages only. This is why most organizations are not able to bear the costs and just use any facilities provided by development tools they use.

1.3 Current Solutions and Proposed Solution

This section will briefly discuss on current solutions regarding the problems related to documentation as conversed in previous sections. At the end of this section, the proposed solution will be described.

Since the early 1990s, CASE vendors discovered that the "software maintenance crisis" would be resolved by ***reengineering*** or redeveloping with CASE tools (Pigoski, 1997). The software maintenance crisis includes problems in documentation as one of the factors. In order to reengineer software the technique of reverse engineering is required. Hence, a lot of tools for software reverse engineering were introduced especially when the computer world faced the ***Year 2000 (Y2K)*** bugs (Ragland, 1997). Some commercial products are ***Ada Design and Documentation Language (ADADL)*** for ADA language, ***Aide-De-Camp (ADC)*** for C language and Application Browser for COBOL language. For a more complete list see Ragland (1997). Some instances of research prototypes are Reverse Engineering tool of ***Rigi*** (Muller *et al.*, 1994) (Wong *et al.*, 1995) and ***Portable Bookshelf, PBS*** (Finnigan *et al.*, 1997).

A number of studies evaluated the existing reverse engineering or software visualization tools from different aspects. For instance Bellay and Gall (1997) compared four reverse engineering tools in term of four functional categories: analysis, representation, editing or browsing and general capabilities. Another study of Storey *et al.* (1997) compared three tools in terms of cognitive strategies used by the subjects to solve maintenance tasks. Gannod and Cheng (1999) described criteria that can be used to evaluate tool by-products that is any artifacts that is generated by the process of using the tool. The study of Sim and Storey (2000) observed that those tools without any or advanced search utility, forced users to use other alternative such as "grep" utility in UNIX tool in order to study source codes besides understanding the graph viewed. Thus they conclude that visualization and search tool are complementary to each other. The findings of the four comparative studies depict that no tool is "perfect" in reverse engineering, visualizing and re-documenting existing software systems, which normally involve different languages, platforms and types of information required.

In addition, CASE tools of environment class that serve the activity of the whole SDLC phases such as Oracle Designer/2000, Developer/2000 (Billings *et al.* 1997) and Rational Rose (OMG, 1999) are incorporated with a utility to reverse engineer the written source codes. However this type of CASE tool focuses more on the forward engineering. Reverse engineering utility can only be benefited if software engineers have designed and developed a software system using the tool. Otherwise, reverse engineering an exiting software system will only produce a high level of abstraction such as Class Diagram in Rational Rose (OMG, 1999).

Besides the massive development of reverse engineering tools, a number of research and studies have been carried out in order to discover and enhance methods and approaches related to reverse engineering. Reverse engineering process applies many methods in program comprehension. Weiser (1984) introduced the approach of program slicing in program comprehension and defined a slice *S* as a reduced, executable program obtained from a program *P* by removing statements such that *S* replicates parts of the behavior of *P*. Since then, a number of research and studies have been based on his work. Gallagher and Lyle (1991) improve that of program slicing by introducing decomposition slice of programs in which the computation on a given variable is

independent of line numbers. For example if a `main` program is decomposed, the program is broken into manageable pieces and directly assist a software maintainer in guaranteeing that there are no ripple effects induced by modifications in a sliced component. This gives the maintainer a technique to determine those statements and variables, which may be modified in a component and those, which may not. A research proposes the dynamic slicing and its application in program comprehension by developing novel dynamic slicing related concepts (Rilling, 1998). This approach exploits dynamic slicing for the purpose of program comprehension and comprehension of program executions on the source code level for instance executable dynamic slices, partial dynamic slicing, influencing variables and contributing nodes. More recent work (Villavicencio, 2001) promotes a technique based on the automatic comparison of slices that allows analyst to focus his attention on a meaningful code for the design of program plans.

Another important method in program comprehension besides program slicing is software visualization. In this method, source codes parsed via reverse engineering will be visualized graphically besides textual information (see **Figure 1.1**). Most software visualization methods apply graph technique as the main element to visualize a software system under study (Koschke, 2003). Rigi (Muller *et al.*, 1994) applies two approaches which are structural graph views in a multiple, individual window and nested graph view called ***SHriMP (Simple Hierarchical Multi-Perspective)*** of Storey (1998). PBS employs a web-based approach that generates "software landscape" view and ***SNiFF+*** (Wind River, 2004) generates column-by-column tree view. Another tool called ***C Information Abstraction System (CIA)*** of Chen *et al.* (1990) utilizes entity-relationship diagram to visualize software artifacts. ***CodeCrawler*** (Lanza, 2003) is also an instance of software visualization tool that applies the method of integrating metrics into its graph view. Rational Rose (Rational, 2004) has overlapped with this type of CASE workbench but it focuses more on analysis and design. Although it is also incorporated with a reverse engineering utility, without a proper forward engineering process the tool will only produce the relationships of classes that might not be so meaningful to software maintainers. In some studies it is regarded as software visualization tool but not in this research.

In addition, since most software systems involve databases, the study in ***database reverse engineering (DRE)*** is vital too in order to reverse engineer the data structures of a database including that of flat files. Some instances are as in the work of Ghannouchi *et al.* (1998) and Henrard and Hainaut (2001). A lot of software systems written in object-oriented programming have become legacy systems. Consequently, numerous research works conducted on object-oriented software understanding for instance the studies of Wilde and Huitt (1992), Lejter *et al.* (1992), Etzkorn and Davis (1994) and Systa (1998). More work will be elaborated in **Chapter 2** of Literature Review and the related work will be criticized in **Chapter 3**.

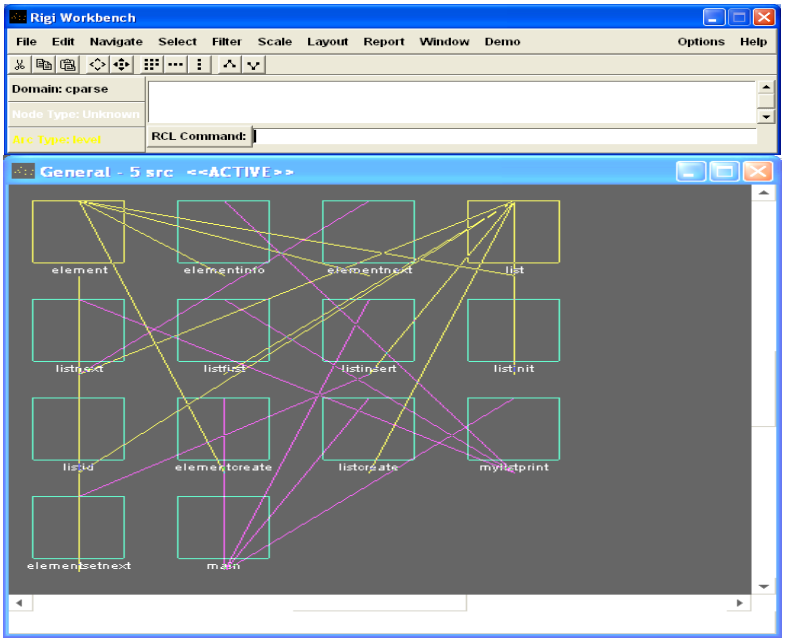

Figure 1.1: An example of graph visualization via Rigi tool

Despite all solutions previously mentioned, software maintenance still seems to face a lot of problems particularly those related to system documentation. Reverse engineering or software visualization tools have become the alternative to automate documenting and understanding of existing software systems with the hope to eliminate the problems related to documentation. A lot of commercial and prototype reverse engineering tools have been developed but none of the tools directly integrate its reverse engineering environment and viewer with a standardized documentation environment. Besides, current solutions mostly focus on legacy systems that normally consist of messy source codes also known as spaghetti source codes that are lack of documentation.

However it is believed that the best stage to use such tool is as soon as after software development process has completed and before the following maintenance processes. By applying the tool in early stage of software life cycle, the knowledge on how to cluster or modularize the software artifacts can be captured from software developers. Otherwise such tool should enforce modularization process prior to visualizing the software artifacts to ensure the visualization for example using the graph technique will not produce very crowded or confined graph representations. Thus this motivates the proposal of a software visualization method in a document-like way in which visualization is made in an explicit re-documentation environment consists of a standardized documentation template tagging with modularized software components. This approach will make software visualization can be fully optimized in re-documenting existing software systems studied. Besides to reduce complexity of graphs generated the proposed method enforce the modularization of software components prior to generating graph views. The proposed document-like software visualization method is called DocLike Modularized Graph, which will be described in **Section 6.4**. The method is employed in a document-like viewer known as ***DocLike Viewer*** that integrates a documentation environment in an existing reverse engineering environment of Rigi (Muller *et al.*, 1994).

1.4 Statement of the Problem

This research is intended to deal with the problems related to system documentation as discussed in **Section 1.2**. The main question *is "**How** to produce a **more effective method in re-documenting and visualizing** software artifacts that can **enhance cognition** of existing software systems for software maintenance?"*

The sub questions of the main research question are as follows:

(i) Why some CASE environment, which incorporated with document generator utility; and reverse engineering or software visualization workbenches are still not able to produce system documentation as needed by software maintainers?

(ii) What are the important information and features in system documentation, which must be captured by software visualization method and tool during re-documentation of software systems?

(iii) When software visualization tools should be employed in software life cycle in order to extract the most reliable and updated source codes?

(iv) How to validate that a particular method in a software visualization tool is effective and useful for software maintenance?

Sub-questions (i) and (ii) will be answered via preliminary studies and literature reviews which produce the input to design a more effective method in software visualization. Sub-question (iii) will be countered by the proposed solution. Lastly, sub-question (iv) will lead to the evaluation of the method employed in the prototype tool.

1.5 Objectives of the Research

The objectives of the research are:

(i) To enhance the method of software visualization using graph technique that can grant significant improvement in program comprehension or cognition of software systems.

(ii) To establish a more effective environment for structural re-documentation, which will capture the most reliable artifacts of a software system.

(iii) To produce a prototype tool that employs the enhanced method of software visualization and improves cognitive aspects of such tool.

1.6 Theoretical Framework

This research lies in the framework of a software maintenance environment by Kwon *et al.* (1998) as illustrated in the shaded area in **Figure 1.2**. Referring to the figure, maintenance support environment has three major parts:

(i) Implementation part of maintenance (Program Comprehension, Impact Analysis and Regression Testing)

(ii) Reverse Engineering part that can be supported by Program Transformation and Restructuring and enables Design Recovery

(iii) Maintenance Model and Standard part that support the process and guidelines of software maintenance

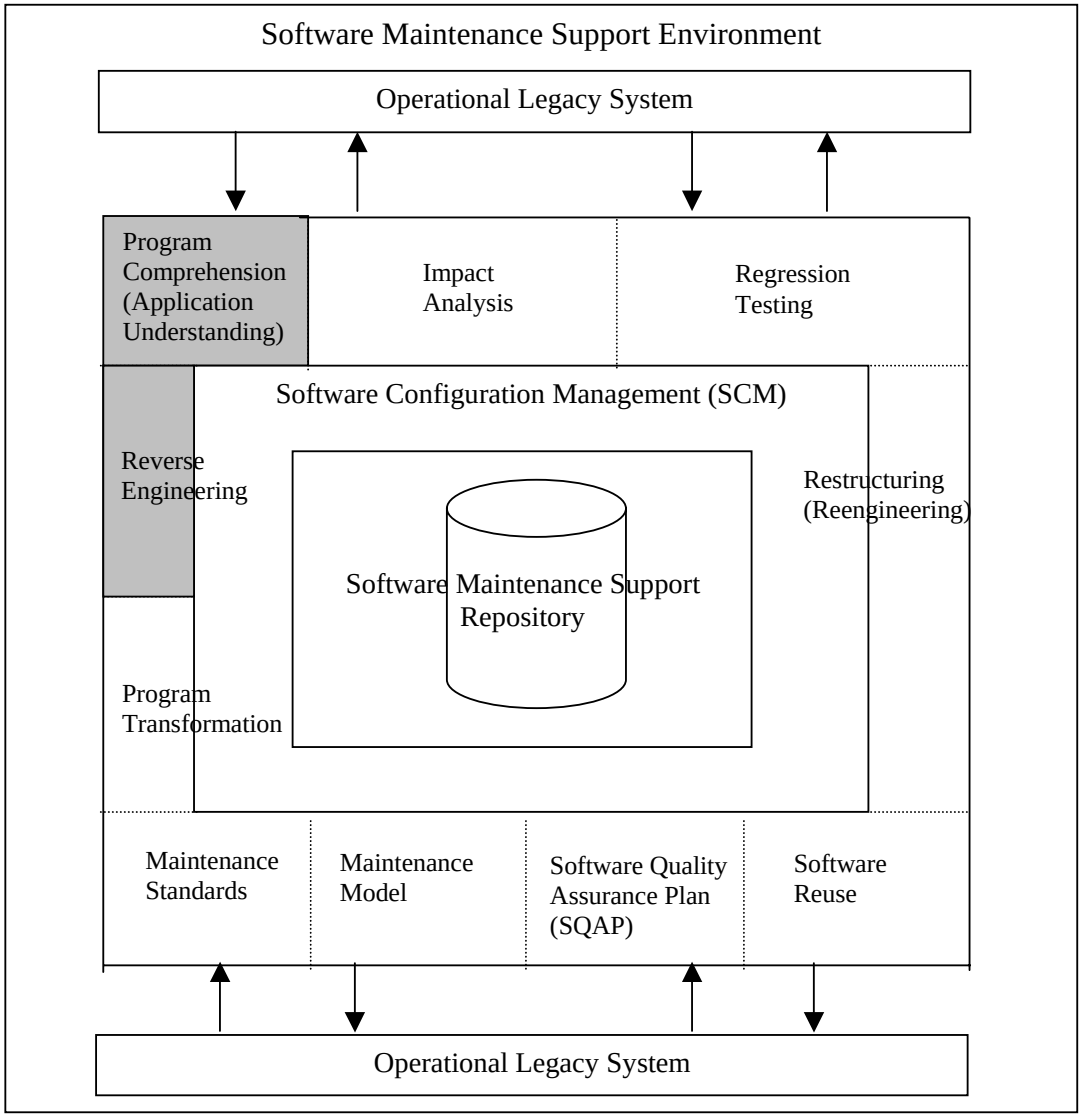

Figure 1.2: A software maintenance support environment (Kwon *et al.*, 1998)

Figure 1.2 shows a software maintenance support environment in which the repository is the central element of the environment. Software configuration management ensures the data in the repository is properly archived. Since software maintenance always deals with legacy systems, these kinds of systems become the major input in the environment. As described in the previous paragraph, the implementation part involves program comprehension, impact analysis and regression testing. The three

elements are integrated with reverse engineering part that is supported by program transformation and restructuring to enable design recovery of an existing legacy or software system. The maintenance environment also includes maintenance model and standard, quality assurance and software reuse. Without all these three major parts (see previous paragraph) software maintenance will not be conducted properly particularly when it involves legacy systems.

Thus this research will mainly cover program comprehension of implementation part while reverse engineering will be the supporting part of the research (see highlighted areas in **Figure 1.2**). For the program comprehension part the research will be specifically based on the work of Storey (1998) and Storey *et al.* (1999) that provides the cognitive framework to describe and evaluate software exploration tools or in this research context is referred as software visualization tools. The framework embraces two main elements that are improve program comprehension and reduce the maintainer's cognitive overhead (see **Table 1.1**). The two cognitive elements are divided into another three sub-elements respectively. In order to improve program comprehension, it is required to enhance bottom-up comprehension, top-down comprehension and integrate both approaches. In addition, to reduce cognitive overhead, it is necessary to facilitate navigation, provide orientation cues and reduce disorientation. Each cognitive element involves a number of activities and assigned with a code for ease of reference. The method proposed in this research will be partly evaluated based on this cognitive framework.

Hence, a software visualization tool should improve program comprehension at least to certain level (Price *et al.*, 1993) otherwise such tool will be useless to programmers or software maintainers. Software visualization tool should also reduce cognitive overhead in order to preserve the mental map and cognition of a subject system studied by software maintainers as described in the cognitive framework (Storey, 1998).

1.7 Importance of the Research

Erlikh (2000) states that 85 to 90 percent of Information System budgets go to legacy system operation and maintenance, which is extremely high. Hence this problem

merits serious investigation. The cost might be higher if software system is maintained without its system documentation because programmers consume more time to understand the existing source codes (Pigoski, 1997). As discussed in **Section 1.3**, software visualization is one of the vital methods to provide a solution related to program comprehension by visualize graphically existing software artifacts and their interrelationships. This was also proved in the preliminary study in which graphical visualization was significantly preferred by software engineers compared to its textual description (Sulaiman *et al.*, 2002b). Thus to overcome the problem related to program comprehension in the case of design documents are not up-dated or absence, there is a need of a proper and more effective tool to automate and assist program comprehension or cognition in order to eliminate the cost of the activities. It is expected that the deliverable of this thesis will be beneficial to other researchers in software maintenance field and also software maintainers who will use the prototype tool developed.

Table 1.1: Cognitive framework to describe and evaluate software visualization tools (Storey, 1998)

Cognitive elements		**Activities**	**Code**
Improve program comprehension	Enhance bottom-up comprehension	Indicate syntactic and semantic relations between software objects	E1
		Reduce the effect of de-localized plans	E2
		Provide abstraction mechanisms	E3
	Enhance top-down comprehension	Support goal-directed, hypothesis-driven comprehension	E4
		Provide an adequate overview of the system architecture at various levels of abstraction	E5
	Integrate bottom-up and top-down approaches	Support the construction of multiple mental models (domain, situation, program)	E6
		Cross-reference mental models	E7
Reduce the maintainer's cognitive overhead	Facilitate navigation	Provide directional navigation	E8
		Support arbitrary navigation	E9
	Provide orientation cues	Indicate the maintainer's current focus	E10
		Display the path that led to the current focus	E11
		Indicate options for further exploration	E12
	Reduce disorientation	Reduce additional effort for user-interface adjustment	E13
		Provide effective presentation styles	E14

1.8 Scope of the Research

This research focuses on how to enhance existing methods of software visualization in a selected existing reverse engineering environment by identifying their weaknesses and strengths via comparative study of existing tools and existing literatures. The scope of reverse engineering process is limited to source codes only but not include DRE. The proposed method should be able to improve the cognitive aspects in software visualization tool by providing a more effective ways to re-document, visualize and understand software system.

In order to justify the research problem and to find the answers of how, what, when and how software visualization tools should support the practice, a survey had been conducted in order to identify the current practice in producing and maintaining system documentation. The survey investigated the actual expectation towards reverse engineering or software visualization tools that could assist software engineers to solve their maintenance tasks.

Another scope of the research should include developing of a prototype tool that applies the proposed method of software visualization in a better documentation environment based on the findings from the preliminary studies and literature reviews. The usability of the prototype tool should be evaluated based on the cognitive framework (Storey, 1998). Besides, the controlled experiment to evaluate the significant of program comprehension or software understanding should be based on the identified metrics specified within ***Goal/Question/Metric (GQM)*** paradigm of Basili (1992).

1.9 Organization of Thesis

This thesis is organized into eight chapters: **Chapter 1**: Introduction, **Chapter 2**: Literature Review, **Chapter 3**: Software Visualization Methods in Reverse Engineering Tools, **Chapter 4**: Research Methodology, **Chapter 5**: Production and Maintenance of System Documentation in Practice, **Chapter 6**: Document-like Software Visualization Method Employed in DocLike Viewer Prototype Tool, **Chapter 7**: The Evaluation, and

finally **Chapter 8**: Conclusion. The flow of the thesis is illustrated in **Figure 1.3**. In addition, the definition of terms used in this thesis is listed in **APPENDIX A**.

Chapter 1 provides a general introduction of software maintenance and what is the background of problems to be tackled in this research, the current solutions and the proposed solution. It also describes in detail the problem to be solved or eliminated in this research, its objectives, the framework in which the research resides and lastly its importance and scope.

In **Chapter 2** readers are informed with the studies or research related to this thesis specifically on the topics of software engineering that indicates where software maintenance resides in a software life cycle, followed by an explanation on software maintenance issues such as the categories, techniques, methods and tools for software maintenance. A section on program comprehension highlights some studies related to comprehending source codes and how we apply the same technique to evaluate this research. The reverse engineering section describes some criteria to evaluate reverse engineering tools and some examples of existing comparative studies. The following section describes software visualization and how it links with reverse engineering and program comprehension areas. The chapter is ended with the studies related to system documentation and a brief discussion on CASE tools to highlight readers in which the deliverable of this research should be categorized.

The following chapter (**Chapter 3**) generally aims on finding what types of information required in different maintenance cases and how existing tools support the information needs. Thus this chapter is initiated with the description of types of information required by software maintainers. Based on the comparative study of the tools and literature review, the taxonomy of level of information abstraction is described. The section is followed by the explanation on the four reverse engineering tools studied (Rigi, PBS, SNiFF+ and Logiscope) and discussion on the three types of maintenance cases: corrective, adaptive and preventive. The final section discusses the findings of type of information required by software maintainers versus the information provided by the tools, which contributed the input to design the proposed method and prototype tool.

The fourth chapter (**Chapter 4**) describes how the research problem was formulated based on the issues of importance of system documentation, software visualization and the drawbacks and strengths of existing tools followed by a brief discussion on the proposed solution. Other sections comprise descriptions on research design, an explanation on subject or source of information and the means used to gather data in this research. Then the following section provides a step-by-step explanation on the research procedures conducted within the planned operational framework. The last two sections discuss on how data is analyzed and possible limitations on the research and the list of assumptions.

The fifth chapter (**Chapter 5**) covers the details of a survey on the software engineers' practice in production and maintenance of system documentation. The sections consist of the description on the analysis based on four elements: characteristic, behavior, beliefs and attitude. Then the findings are thoroughly described and the chapter is concluded. The chapter verifies and justifies the research problems addressed in this thesis.

In **Chapter 6** the analysis and design of the prototype tool that employs the proposed DMG method are described. Then the DMG method is defined thoroughly including the graph layout algorithm optimized. The tradeoff issues of the enhanced method are also discussed. The last part of this chapter describes about DocLike Viewer prototype tool that is developed to realize the method in the aspect of visualizing, understanding and re-documenting software systems.

The chapter on validation of the research is explained in **Chapter 7**. The details of the controlled experiment and usability study conducted, analysis and findings of the evaluation are discussed. In addition the qualitative evaluation of the prototype tool developed compared to other existing software visualization tools are also presented.

Finally, the thesis is concluded in **Chapter 8** that provides the summary of the thesis, highlights the contribution and limitation of the research and the possible future work.

1.10 Summary

This chapter has provided an introduction to software maintenance in general. It has also discussed on the background of the problems related to documentation, understanding source codes and the use of CASE tools. The current solutions of the problem and proposed solution have been explained briefly. It is followed by the description of statement of problem, objectives of the research and the theoretical framework. The importance and scope of the research has also been explained. The last section has described how this thesis is organized.

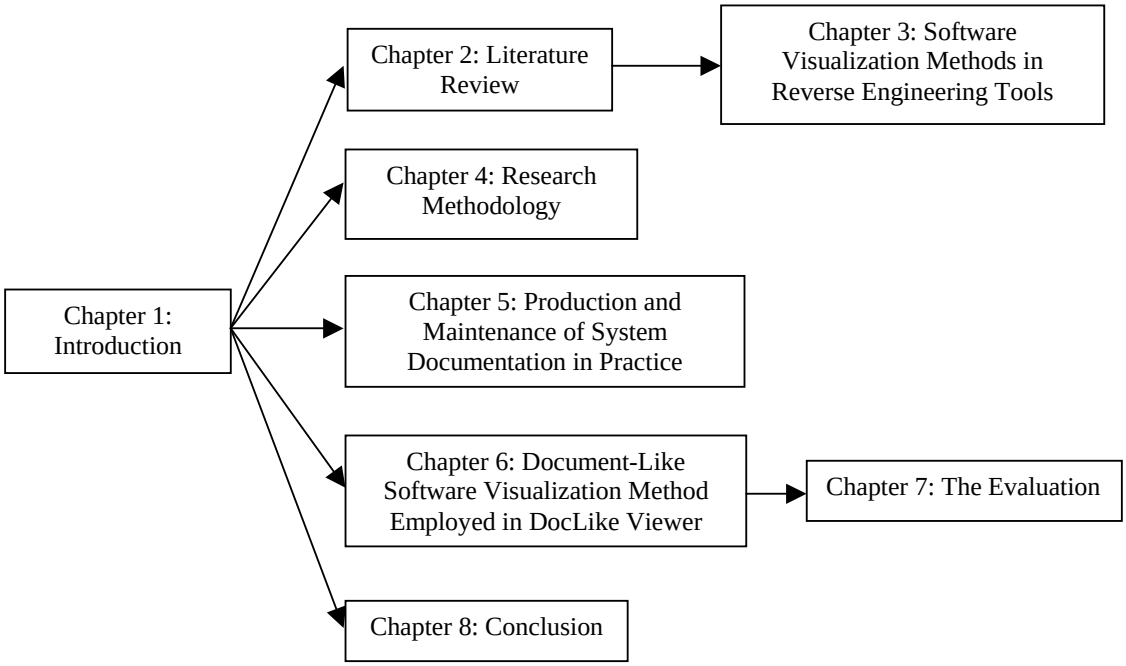

Figure 1.3: Flow of the thesis diagram