# A DOCUMENT-LIKE SOFTWARE VISUALIZATION METHOD FOR EFFECTIVE COGNITION OF C-BASED SOFTWARE SYSTEMS

SHAHIDA BINTI SULAIMAN

UNIVERSITI TEKNOLOGI MALAYSIA

# A DOCUMENT-LIKE SOFTWARE VISUALIZATION METHOD FOR EFFECTIVE COGNITION OF C-BASED SOFTWARE SYSTEMS

SHAHIDA BINTI SULAIMAN

A thesis submitted in fulfilment of the
requirements for the award of the degree of
Doctor of Philosophy

Faculty of Computer Science and Information System
Universiti Teknologi Malaysia

JULY 2004

ALHAMDULILLAH…

For my beloved mother and husband who have given the strength and courage,

my two energetic kids Erin and Ariff,

and the rest of Allahyarham Sulaiman’s family members

## ACKNOWLEDGEMENT

The researcher would like to thank Professor Dr Norbik Bashah bin Idris as the supervisor of this research for always delivering his support and sharing his knowledge with the researcher.  A lot of thanks go to Associate Professor Dr Shamsul bin Sahibuddin as the co-supervisor of this research who has given guidelines particularly in the empirical aspects of the research.

A great gratitude also goes to Universiti Sains Malaysia that has funded the three-year PhD study.  Not forgetting all software practitioners who had involved in the preliminary study and the usability study of the prototype tool and also the post-graduates of the Center for Advanced Software Engineering (CASE), UTM, Kuala Lumpur who participated in the experiment.  Their co-operation has made me feel motivated and supported in making this research successful in the future.

My gratitude also goes to other researchers of CASE, the examiners who gave their valuable feedbacks on the proposal of this research, the internal and external examiners who delivered their constructive comments for the final manuscript and other individuals who had contributed in this research indirectly until this thesis was finalized.

Last but not least, countless thanks to my beloved mother who has been supporting and blessing me from the day of registering my study until the day of submitting the final manuscript.  Also my appreciation goes to my husband who has been encouraging me all the time to further and complete my study.  Not forgetting my active kids Erin and Ariff who always wanted to “click on the butterfly” and also the rest of family members, in-laws and friends who have given their eternal moral support.

## ABSTRACT

It is clear that maintenance is a crucial and very costly process in a software life cycle. Nowadays there are a lot of software systems particularly ***legacy systems*** that are always maintained from time to time as new requirements arise. One important source to understand a software system before it is being maintained is through the ***documentation***, particularly ***system documentation***. Unfortunately, not all software systems developed or maintained are accompanied with their reliable and updated documents. In this case, source codes will be the only reliable source for programmers. A number of studies have been carried out in order to assist cognition based on source codes. One way is through tool automation via ***reverse engineering*** technique in which source codes will be parsed and the information extracted will be visualized using certain visualization methods. Most ***software visualization*** methods use graph as the main element to represent extracted software artifacts. Nevertheless, current methods tend to produce more complicated graphs and do not grant an explicit, document-like ***re-documentation*** environment. Hence, this thesis proposes a document-like software visualization method called ***DocLike Modularized Graph (DMG)***. The method is realized in a prototype tool named ***DocLike Viewer*** that targets on C-based software systems. The main contribution of the DMG method is to provide an explicit structural re-document mechanism in the software visualization tool. Besides, the DMG method provides more level of information abstractions via less complex graph that include inter-module dependencies, inter-program dependencies, procedural abstraction and also parameter passing. The DMG method was empirically evaluated based on the ***Goal/Question/Metric (GQM)*** paradigm and the findings depict that the method can improve productivity and quality in the aspect of cognition or program comprehension. A usability study was also conducted and DocLike Viewer had the most positive responses from the software practitioners.

# ABSTRAK

Adalah jelas bahawasanya pengubahsuaian merupakan proses yang penting dan sangat mahal dalam kitaran hidup perisian. Dewasa ini terdapat banyak sistem perisian terutamanya sistem legasi yang sentiasa diubahsuai daripada masa ke semasa apabila terdapat keperluan baru. Satu sumber penting untuk memahami perisian sebelum ia diubahsuai ialah melalui dokumentasi terutamanya dokumentasi sistem. Malangnya tidak kesemua sistem perisian dibangun atau diubahsuai dilengkapi dengan dokumen yang boleh dipercayai dan terkemaskini. Dalam kes ini, kod sumber akan menjadi satu-satunya sumber boleh dipercayai oleh pengaturcara. Terdapat beberapa kajian yang dijalankan untuk membantu kognisi berasaskan kod sumber. Satu cara ialah melalui teknik kejuruteraan berbalik dalam mana kod sumber akan diperiksa dan maklumat didapati akan divisualisasi menggunakan kaedah-kaedah visualisasi tertentu. Kebanyakan kaedah visualisasi perisian menggunakan graf sebagai elemen utama untuk mewakili artifak perisian didapati. Walau bagaimanapun, kaedah-kaedah masa kini menjurus kepada penghasilan graf yang lebih kompleks dan tidak menyediakan persekitaran pengdokumentasian semula ala dokumen secara jelas. Maka, tesis ini mencadangkan suatu kaedah visualisasi perisian ala dokumen dipanggil “DocLike Modularized Graph (DMG)”. Kaedah ini digunapakai dalam suatu alat prototaip dinamakan “DocLike Viewer” yang menumpukan kepada sistem perisian berasaskan C. Sumbangan utama kaedah DMG ialah untuk menghasilkan satu mekanisme dokumentasi semula berstruktur di dalam alat visualisasi secara langsung. Di samping itu, kaedah DMG menghasilkan lebih banyak peringkat abstraksi maklumat melalui graf yang kurang rumit termasuklah hubungan antara modul, antara program, abstraksi prosedur dan juga penyerahan parameter. Kaedah DMG telah dinilai secara empirikal berdasarkan paradigma “Goal/Question/Metric” (GQM) dan dapatan-dapatannya menunjukkan bahawa kaedah tersebut boleh menaiktaraf produktiviti dan kualiti dalam aspek kognisi atau pemahaman aturcara. Suatu kajian kebolehgunaan juga dilaksanakan dan DocLike Viewer mendapat maklumbalas paling positif daripada pengamal-pengamal perisian.

# TABLE OF CONTENTS

| CHAPTER | SUBJECT | PAGE |
|---|---|---|
| | **DECLARATION** | ii |
| | **DEDICATION** | iii |
| | **ACKNOWLEDGEMENT** | iv |
| | **ABSTRACT** | v |
| | **TABLE OF CONTENTS** | vii |
| **1** | **INTRODUCTION** | 1 |
| | 1.1 Introduction to Software Maintenance | 1 |
| | 1.2 Background of the Problem | 3 |
| | 1.2.1 Incomplete, Out-dated, Non Standardized or Absence of Documentation | 4 |
| | 1.2.2 Understanding an Existing Software System is Tedious and Costly | 6 |
| | 1.2.3 CASE Tools or Environment are Not Used or Properly Used | 7 |
| | 1.3 Current Solutions and Proposed Solution | 8 |
| | 1.4 Statement of the Problem | 12 |
| | 1.5 Objectives of the Research | 13 |
| | 1.6 Theoretical Framework | 13 |
| | 1.7 Importance of the Research | 15 |
| | 1.8 Scope of the Research | 17 |
| | 1.9 Organization of Thesis | 17 |
| | 1.10 Summary | 20 |

**2 LITERATURE REVIEW 21**

2.1 Introduction 21
2.2 Software Engineering 21
2.3 Software Maintenance 25
2.4 Program Comprehension 27
2.5 Reverse Engineering 29
2.6 Software Visualization 35
2.7 System Documentation 38
2.8 Computer-Aided Software Engineering (CASE) Tools 41
2.9 Summary of Literature Review 42

**3 SOFTWARE VISUALIZATION METHODS IN REVERSE ENGINEERING TOOLS 43**

3.1 Introduction 43
3.2 Types of Information Required 44
3.3 Current Reverse Engineering Tools 45
3.3.1 Rigi 46
3.3.2 PBS (Portable Bookshelf) 47
3.3.3 SNiFF+ 48
3.3.4 Logiscope 49
3.3.5 Software Visualization Methods Used 49
3.4 Software Maintenance Cases 52
3.4.1 Case Study 1 – Corrective 52
3.4.2 Case Study 2 – Adaptive 52
3.4.3 Case Study 3 – Perfective 54
3.5 Type of Information versus Tool 55
3.6 Drawbacks and Strengths of Existing Reverse Engineering Tools 58
3.6.1 Software Visualization Methods 59
3.6.2 Features Provided 60

3.7 Enhancement of Software Visualization Method 63
3.8 Rigi – the Reverse Engineering Environment for DocLike Viewer 64
3.8.1 Rigi C Language Parser 65
3.9 Summary 67

**4 RESEARCH METHODOLOGY 68**

4.1 Introduction 68
4.2 Operational Framework or Research Procedures 68
4.3 Formulation of Research Problem 70
4.3.1 Importance of System Documentation 70
4.3.1.1 System Documentation in Software Life-Cycle 71
4.3.1.2 The Role of Standard in System Documentation 72
4.3.1.3 The Role of CASE Tools in System Documentation 72
4.3.2 Importance of Software Visualization 74
4.3.3 Justification of Research Problem 74
4.4 Research Design and Hypothesis 76
4.5 Subject and Source of Information 78
4.6 Instrumentation or Means of Gathering Data 79
4.6.1 Empirical Evaluation – a Controlled Experiment 79
4.7 Data Analysis 80
4.8 Limitation on the Research and List of Assumptions 81
4.9 Summary of Research Methodology 82

**5 PRODUCTION AND MAINTENANCE OF SYSTEM DOCUMENTATION IN PRACTICE 83**

5.1 Introduction 83
5.2 The Survey 83
5.3 Objectives of the Survey 87
5.4 The Analysis 87
5.4.1 The Characteristic Element 88
5.4.2 The Behavior Element 90
5.4.3 The Attitude Element 96
5.4.4 The Belief Element 98
5.5 The Findings 100
5.6 Summary 106

**6 DOCUMENT-LIKE SOFTWARE VISUALIZATION METHOD EMPLOYED IN DOCLIKE VIEWER 107**

6.1 Introduction 107
6.2 The Analysis 108
6.3 The Design 110
6.3.1 Filtering Software Artifacts Extracted by Rigi C Language Parser 111
6.3.2 List of Components 111
6.3.3 The Algorithm 114
6.3.3.1 Algorithm for the NewDialog Class 114
6.3.3.2 Algorithm for the GraphPanel Class 115
6.3.3.3 Algorithm for the GlobalSearchDialog Class 116
6.3.3.4 Algorithm for the SearchDialog Class 118
6.3.4 Cognitive Framework 119

6.4 DocLike Modularized Graph (DMG) Method 121
6.4.1 Graph Drawing 124
6.4.2 Grid Layout Algorithm of DMG 126
6.4.3 Tradeoff Issues 134
6.5 DocLike Viewer Prototype Tool 137
6.5.1 Documentation Environment for DocLike Viewer 139
6.5.2 Visualizing Graph and Searching Concerned Nodes 140
6.5.3 Understanding Source Codes 146
6.5.4 Re-documenting Software System 147
6.6 Summary 150

**7 THE EVALUATION 151**

7.1 Introduction 151
7.2 Research Question and Hypothesis 151
7.2.1 Structure of Analysis Using Goal/ Question/Metric (GQM) 153
7.3 Controlled Experiment 155
7.3.1 Subjects and Subject System 156
7.3.2 Questionnaire 157
7.3.3 Procedures 159
7.3.4 Possible Threats 159
7.4 Usability Study 161
7.4.1 Subjects and Subject System 161
7.4.2 Questionnaire 162
7.4.3 Procedures 164
7.4.4 Possible Threats 164
7.5 The Analysis 165
7.5.1 Analysis of the Controlled Experiment 165
7.5.2 Analysis of the Usability Study 170
7.6 The Findings 173

7.6.1 Findings of the Controlled Experiment 173

7.6.2 Findings of the Usability Study 183

7.7 Qualitative Evaluation 183

7.8 Summary 189

**8 CONCLUSION** 190

8.1 Summary of the Thesis 190

8.2 Contribution 195

8.3 Limitation of the Research 197

8.4 Future Work 197

**REFERENCES** 199-209

**APPENDIX**

Appendices A – L 210-255

# LIST OF TABLES

| TABLE NO. | TITLE | PAGE |
|---|---|---|
| 1.1 | Cognitive framework to describe and evaluate software visualization tools (Storey, 1998) | 16 |
| 2.1 | Five sections of activity grouping in Software Life Cycle Process (SLCP) (IEEE Inc., 1998) | 23 |
| 2.2 | Classification of CASE products (van Vliet, 2000) | 41 |
| 2.3 | Classes of CASE workbenches (Fuggetta, 1993) | 42 |
| 3.1 | The taxonomy of level of information abstraction | 45 |
| 3.2 | The information of the four tools studied | 46 |
| 3.3 | Program versus file cross-reference table | 54 |
| 3.4 | Maintenance category versus information type and tools' support | 58 |
| 3.5 | Drawbacks and strengths of software visualization methods | 59 |
| 3.6 | The features provided by the four tools studied | 62 |
| 5.1 | Information sought based on characteristic, behavior, belief and attitude | 85 |
| 5.2 | Characteristics of software engineers | 88 |
| 5.3 | Task preference versus experience | 89 |
| 5.4 | Usefulness of system documentation (SD) | 90 |
| 5.5 | Software development and maintenance tasks or projects | 91 |
| 5.6 | SDLC phases, tools and diagramming techniques in relation with SD | 91 |
| 5.7 | Average number of SD produced or modified and maintenance project without SD | 92 |
| 5.8 | The use of CASE tool versus number of SD and maintenance project without SD | 92 |

| | | |
|---|---|---|
| 5.9 | The use of standard/template versus number of SD and maintenance project without SD | 93 |
| 5.10 | Number of projects and SD produced or modified based on category of industry | 94 |
| 5.11 | Top three programming languages and databases used | 94 |
| 5.12 | The use of SD standard or template, CASE or reverse engineering tool | 95 |
| 5.13 | T test of textual versus graphical features | 98 |
| 5.14 | Importance of SD, standard and template | 99 |
| 6.1 | Features in DocLike Viewer described within cognitive framework | 120 |
| 7.1 | The goal of study | 153 |
| 7.2 | The questions formulated | 154 |
| 7.3 | The metrics used | 154 |
| 7.4 | Formulation of criteria to be evaluated based on cognitive framework | 163 |
| 7.5 | Last job of subjects | 165 |
| 7.6 | Cross tabulation of last job versus tool used | 166 |
| 7.7 | Cross tabulation of experience versus tool used | 166 |
| 7.8 | Cross tabulation of grade versus tool used | 166 |
| 7.9 | ANOVA table of time taken ($T_1$) for the six questions | 168 |
| 7.10 | ANOVA table of score of the six questions | 169 |
| 7.11 | Job title of respondents in the usability study | 170 |
| 7.12 | Years of experience of respondents in the usability study | 171 |
| 7.13 | ANOVA table of usefulness of the three tools compared | 172 |
| 7.14 | Qualitative comparison of software visualization tools | 188 |
| 8.1 | The contribution of the research work in this thesis compared to others | 196 |

## LIST OF FIGURES

| FIGURE NO. | TITLE | PAGE |
|---|---|---|
| 1.1 | An example of graph visualization via Rigi tool | 11 |
| 1.2 | A software maintenance support environment (Kwon *et al.*, 1998) | 14 |
| 1.3 | Flow of the thesis diagram | 20 |
| 2.1 | The simple staged model for a software life cycle consists of five distinct stages (Rajlich and Bennett, 2000) | 24 |
| 2.2 | Taxonomy of reverse engineering techniques (Gannod and Cheng, 1999) | 35 |
| 3.1 | One of the views produced by Rigi via RigiEdit | 47 |
| 3.2 | A “Software Landscape” of a system produced on a web page by PBS | 47 |
| 3.3 | Windows of file manager and project manager (left), hierarchy browser and its filter utility (top right) and log information in SNiFF+ (bottom right) | 48 |
| 3.4 | Cross Referencer window in SNiFF+ | 48 |
| 3.5 | A control graph produced by Logiscope, the dotted line shows Untested path in a ‘while’ instruction | 49 |
| 3.6 | A graph consists of nodes (vertices) and arcs (edges) | 49 |
| 3.7 | Multiple, individual window in Rigi editor | 50 |
| 3.8 | Nested Graph window in SHriMP | 50 |
| 3.9 | Web-based software landscape view in PBS | 51 |
| 3.10 | Column-by-column tree view in SNiFF+ | 51 |
| 3.11 | Hierarchy of modules | 53 |
| 3.12 | Information required in case study 1 | 55 |
| 3.13 | Information required in case study 2 | 56 |
| 3.14 | Information required in case study 3 | 56 |

3.15 An example of C program to be parsed (Rigi, 2004) 65
3.16 The extracted artifacts in RSF format (Rigi, 2004) 66
4.1 Flowchart of operational framework for the research 69
4.2 The relationship of all variables and their attributes 77
5.1 The need of graphical versus textual features and other features 97
5.2 The mean versus reasons for not producing or maintaining SD 99
6.1 Use Case Diagram of DocLike Viewer 110
6.2 Nodes or vertices are drawn in grid layout 127
6.3 The left figure shows the arc drawn in DMG method, while the right figure shows the arc mostly drawn in other graph layout algorithms 128
6.4 Spring layout produced via RigiEdit 130
6.5 Sugiyama layout produced via RigiEdit 130
6.6 Grid layout produced via RigiEdit 131
6.7 Nested graph of SHriMP layout 132
6.8 The figure on the left shows how the labeling of vertices is made by the DMG method compared to other methods like Rigi as in the right figure 133
6.9 DMG method optimizes the Sugiyama layout based on the number of vertices to be drawn in the display area 134
6.10 DocLike Viewer consists of Content Panel, Graph Panel and Description Panel 136
6.11 Modularization process done prior to generating a new view 137
6.12 The best stage to implement DocLike Viewer within software life cycle 139
6.13 Module decomposition of section 3.1 is selected and the associated graph representation of type $DMG_1$ is viewed besides its textual description 140
6.14 Display Information dialog box lists the details of a node clicked 141
6.15 Search utility provided in DocLike Viewer 143
6.16 Search Node dialog box 143
6.17 The sections associated to DMG representations in which the Concerned function is found are listed 144

6.18 The concerned function is highlighted when the section is clicked 145

6.19 Source Code Window displaying highlighted procedure name based on the vertex clicked 147

6.20 An example of design document generated by DocLike Viewer 148

6.21 Document generation mechanism via DocLike Viewer 149

7.1 The modularized view of subject system generated by DocLike Viewer 155

7.2 The File View displays the subject files via Microsoft Visual C++ 156

7.3 The collapsed view of subject system in Rigi 156

7.3 One of the views generated from the List System via DocLike Viewer 161

7.5 Mean of time taken ($T_1$) versus question by DV, MV and RG group (regardless of correctness) – the asterisk (*) indicates the significant mean difference 167

7.6 Sum of correct answers (S) versus question for DV, MV and RG group - the asterisk (*) indicates the significant score difference 168

7.7 Mean of time taken to answer correctly ($T_2$) versus question for DV, MV and RG group – the asterisk (*) indicates the significant mean difference 169

7.8 Bar chart – category of industry of respondents in the usability study 171

7.9 Usefulness of tool (mean values based on Likert scale 1: Strongly disagree, 2: Disagree, 3: Normal, 4: Agree, 5: Strongly Agree) 172

7.10 The usability criteria as ranked by the respondents – RG: Rigi, MV: Microsoft Visual C++ and DV: DocLike Viewer 173

7.11 DMG1 representation shows the hierarchy view of the subject system 174

7.12 The DMG3 representation in DocLike Viewer highlights the nodes searched by program in LibAdt module 175

7.13 The collapsed nodes in LibAdt module viewed via Rigi 176

7.14 The graph view is re-arranged using Sugiyama layout in Rigi 176

7.15 List of sections in which index_Record function in found 177

7.16 Details of index_Record function is displayed in a dialog box 178

7.17 Global search provided by Microsoft Visual C++ 179

7.18 The searched processWordToIndex nodes are highlighted via Rigi 180

7.19 The $DMG_5$ representation of data flow graph in DocLike Viewer 181

7.20 A graph view of Gawk System via DocLike Viewer 184

# LIST OF ACRONYMS AND SYMBOLS

| | |
|---|---|
| ADADL | - Ada Design and Documentation Language |
| ADC | - Aide-De-Camp for C language |
| ANOVA | - Analysis of Variance |
| *C* | - Set of concerned components |
| CASE | - Computer-Aided Software Engineering |
| CIA | - C Information Abstraction System |
| CodeCrawler | - A software visualization tool |
| *const* | - Vertical gap factor of graph drawn |
| CPARSE | - Rigi C-Language Parser |
| *D* | - Set of all data |
| DERE | - Evaluating Reverse Engineering Framework |
| df | - difference |
| DFD | - Data Flow Diagram |
| DIF | - Document Integration Facility |
| DTD | - Specific Document Type Definitions |
| DMG | - DocLike Modularized Graph proposed in this thesis |
| $DMG_1$ | - First type of DMG (system hierarchy view) |
| $DMG_2$ | - Second type of DMG (program dependencies) |
| $DMG_3$ | - Third type of DMG (function dependencies in a module) |
| $DMG_4$ | - Fourth type of DMG (function dependencies inter-module) |
| $DMG_5$ | - Fifth type of DMG representation (data usage within a module) |
| DMSC | - The abstraction of document management system of GI system |
| DocLike Viewer | - A software visualization tool developed by the researcher |
| DRE | - Database Reverse Engineering |
| DV | - DocLike Viewer used in the experiment and usability study |
| *E* | - Set of edges of the graph |
| E-R | - Entity-Relationship |

| | |
|---|---|
| $E_i$ | - Directed edge or arc |
| F | - Frequency |
| $F$ | - Set of functions |
| $F_i$ | - Concerned function |
| $G$ | - Graph |
| GI | - Generate Index |
| GQM | - Goal/Question/Metric paradigm |
| GUI | - Graphical User Interface |
| $h_1$ | - Height of display area of graph layout |
| $h_2$ | - Height of vertex |
| $h_2$’ | - Height of vertex when $N_v$ increases |
| IEEE | - The Institute of Electrical and Electronics Engineers, Inc. |
| *inset* | - Barrier between the edges of the display area and the vertices |
| $j$ | - Horizontal gap factor |
| $k$ | - Relative gap factor for the first vertex with the following vertices in the same row |
| $label_i$ | - Label of vertex number $i$ where $i = 0,1,2\ldots N_v$ |
| LibAdt | - Library used to model the data in the data flows of GI system |
| $M$ | - Set of all modules |
| $M_i$ | - Concerned module |
| MMIMS | - Man Machine Interface Management module of GI system |
| *MT* | - Maintenance Tasks |
| MV | - Microsoft Visual C++ used in the experiment and usability study |
| $N_m$ | - Number of modules |
| $N_v$ | - Number of vertices |
| $P$ | - Set of all programs |
| $p$ | - Relative vertical gap factor of vertices |
| $P_i$ | - Concerned program |
| PBS | - Portable Bookshelf |
| Rational Rose | - A commercial analysis and design tool |
| RE | - Reverse Engineering |
| RG | - Rigi used in the experiment and usability study |
| Rigi | - A reverse engineering prototype tool |
| RMCS | - Research Materials Collection System |

| | |
|---|---|
| RQ | - Research question |
| RSF | - Rigi Standard Format |
| RUP | - Rational Unified Process |
| $S$ | - Score or number of correct solution for MT |
| $S_i$ | - Set of sections associated with any concerned DMG type |
| $s$ | - side |
| SCM | - Software Configuration Management |
| SD | - System Documentation |
| SDLC | - System Development Life Cycle |
| SE | - Software Engineers |
| *Sec* | - Set of all default sections in the Content Panel |
| SHriMP | - Simple Hierarchical Multi-Perspective |
| Sig. | - Significant value |
| SLC | - Software Life Cycle |
| SLCM | - Software Life Cycle Model |
| SLCP | - Software Life Cycle Process |
| SLEUTH | - Software Literacy Enhancing Usefulness To Humans |
| *SM* | - Software Maintainer |
| SNiFF+ | - A commercial reverse engineering tool |
| SoDA | - Rational SoDA for Word, a document tool of Rational Software |
| SODOS | - Software Documentation Support Environment |
| *SS* | - Software System |
| *SU* | - Software Understanding (measured based on $S$, $T_1$ and $T_2$) |
| *SV* | - Software Visualization |
| SystDoc | - The module comprises all document image items of GI system |
| $T_1$ | - Time taken to find solution for *MT* |
| $T_2$ | - Time taken to find correct solution for *MT* |
| $T_d$ | - Search graph component by data |
| $T_f$ | - Search graph component by function |
| $T_m$ | - Search graph component by module |
| $T_p$ | - Search graph component by program |
| $T_v$ | - Search graph component by variable |
| $U$ | - Usefulness and usability of SV tools |
| UML | - Unified Modeling Language |

| | |
|---|---|
| $V$ | - Set of vertices in the graph |
| $V_i$ | - vertex or node in the graph where $i = 0,1,2\ldots N_v$ |
| View II | - A software visualization tool based on XML |
| $w_1$ | - Width of display area of graph layout |
| $w_2$ | - Width of vertex |
| $w_2'$ | - Width of vertex when $N_v$ increases |
| WWW | - World Wide Web |
| *xCoord* | - x coordinate of vertex to be drawn |
| XML | - Extended Markup Language |
| Y2K | - The year 2000 |
| *yCoord* | - y coordinate of vertex to be drawn |

# LIST OF APPENDICES

| APPENDIX | TITLE | PAGE |
|---|---|---|
| A | Definition of terms | 210 |
| B | Questionnaire of the survey | 214 |
| C | Class diagram of DocLike Viewer | 224 |
| D1 | List of components involved in the sequence diagrams | 225 |
| D2 | Sequence diagram for normal scenario of “Modularized Components” | 226 |
| D3 | Sequence diagram for normal scenario of “View DMG Representations” | 227 |
| D4 | Sequence diagram for normal scenario of “Make Global Search | 228 |
| D5 | Sequence diagram for normal scenario of “Make Local Search | 229 |
| E | User manual of DocLike Viewer | 230 |
| F | Questionnaire of the controlled experiment | 240 |
| G | User manual of DocLike Viewer for DV group | 242 |
| H | User manual of Microsoft Visual C++ for MV group | 245 |
| I | User manual of Rigi for RG group | 247 |
| J | Questionnaire on usability of software visualization tool | 249 |
| K | Experimenter’s Note | 254 |
| L | List of researcher’s paper | 255 |

# 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**.

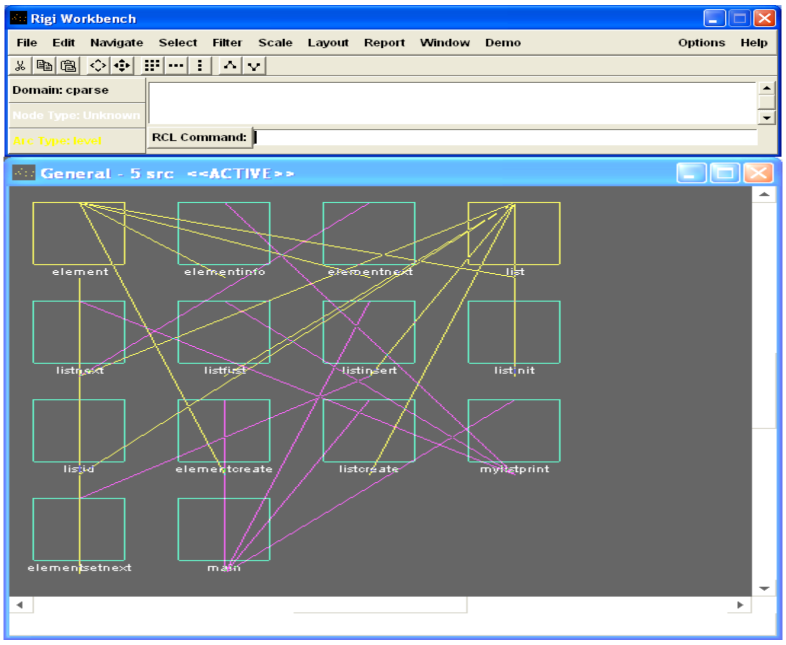

**Figure 1.1:** An example of graph visualization via Rigi tool

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

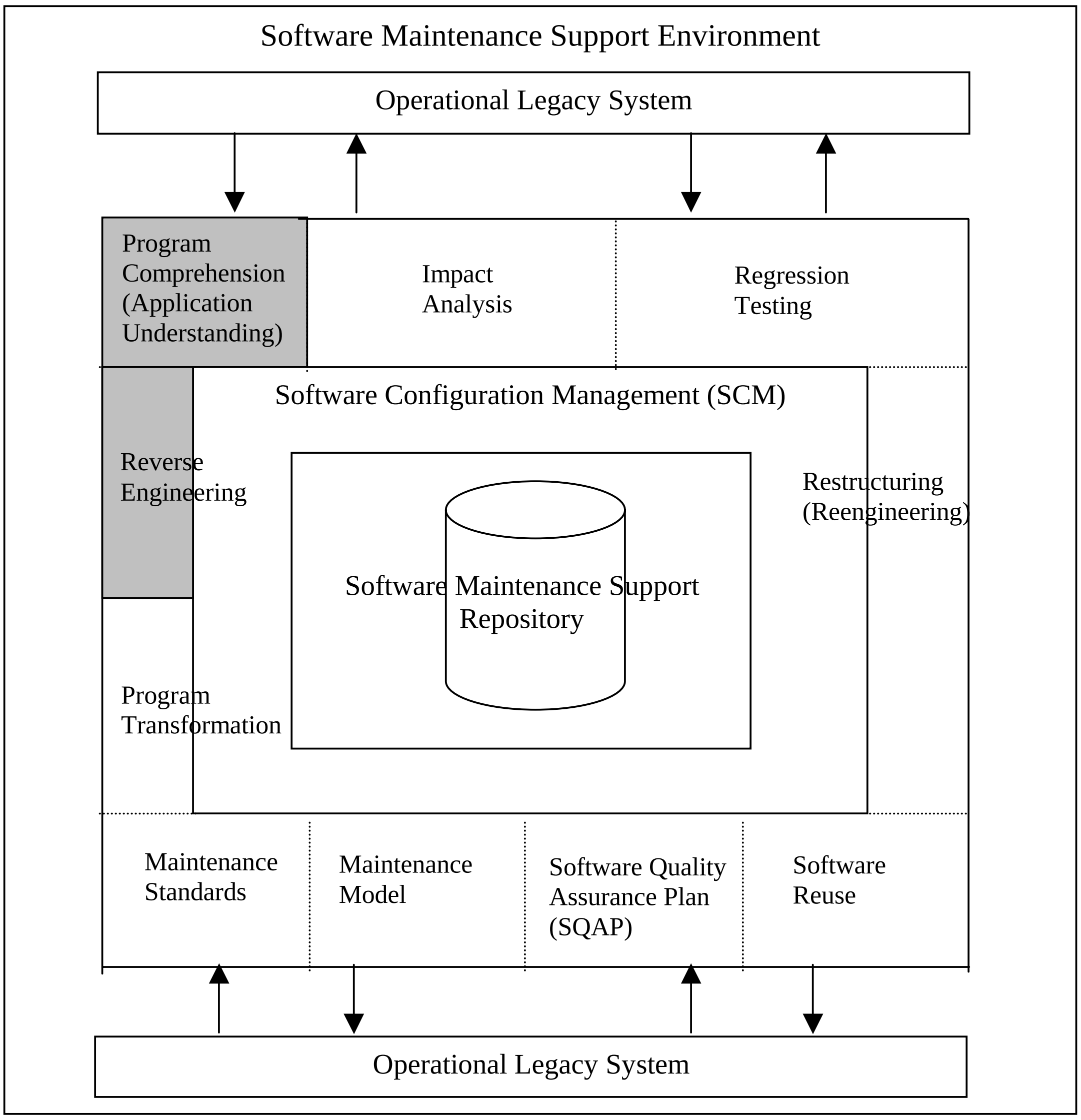
Software Maintenance Support Environment
Operational Legacy System
Program Comprehension (Application Understanding)
Impact Analysis
Regression Testing
Software Configuration Management (SCM)
Reverse Engineering
Restructuring (Reengineering)
Software Maintenance Support Repository
Program Transformation
Maintenance Standards
Maintenance Model
Software Quality Assurance Plan (SQAP)
Software Reuse
Operational Legacy System

**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.

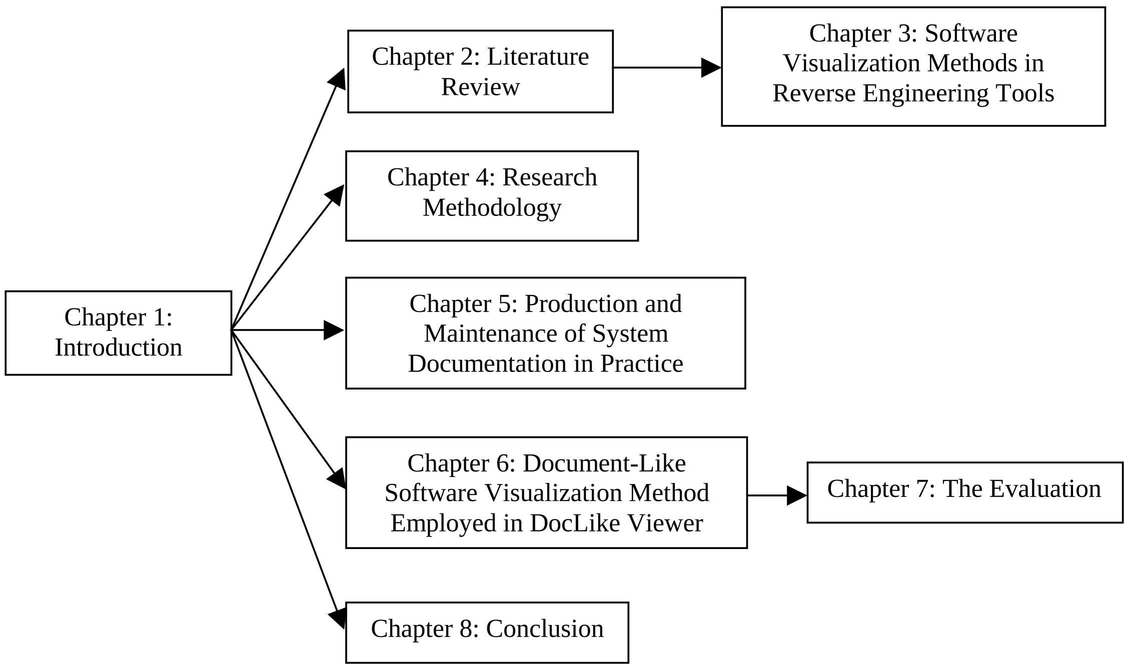
Chapter 1: Introduction
Chapter 2: Literature Review
Chapter 3: Software Visualization Methods in Reverse Engineering Tools
Chapter 4: Research Methodology
Chapter 5: Production and Maintenance of System Documentation in Practice
Chapter 6: Document-Like Software Visualization Method Employed in DocLike Viewer
Chapter 7: The Evaluation
Chapter 8: Conclusion

**Figure 1.3:** Flow of the thesis diagram