Enhancing Cognitive Aspects of Software Visualization Using DocLike Modularized Graph

Shahida Sulaiman[1], Norbik Bashah Idris[2], and Shamsul Sahibuddin[3]
[1]Faculty of Computer Science, University Sains Malaysia, Malaysia
[2]Center for Advanced Software Engineering, University Technology Malaysia, Malaysia
[3]Faculty of Computer Science and Information System, University Technology Malaysia, Malaysia

Abstract: *Understanding an existing software system to trace possible changes involved in a maintenance task can be time consuming especially if its design document is absence or out-dated. In this case, visualizing the software artefacts graphically may improve the cognition of the subject system by software maintainers. A number of tools have emerged and they generally consist of a reverse engineering environment and a viewer to visualize software artefacts such as in the form of graphs. The tools also grant structural re-documentation of existing software systems but they do not explicitly employ document-like software visualization in their methods. This paper proposes DocLike Modularized Graph method that represents the software artefacts of a reverse engineered subject system graphically, module-by-module in a document-like re-documentation environment. The method is utilized in a prototype tool named DocLike viewer that generates graphical views of a C language software system parsed by a selected C language parser. Two experiments were conducted to validate how much the proposed method could improve cognition of a subject system by software maintainers without documentation, in terms of productivity and quality. Both results deduce that the method has the potential to improve cognitive aspects of software visualization to support software maintainers in finding solutions of assigned maintenance tasks.*

Keywords: *Software maintenance, software visualization, program comprehension.*

Received July 21, 2003; accepted March 8, 2004

1. Introduction

Visualization for software, or Software Visualization (SV), is a method in program comprehension, which is vital in the costly software maintenance. SV is the use of interactive computer graphics, typography, graphic design, animation and cinematography to enhance interface between the software engineers or the computer science student and their programs [7]. The objective is to use graphics to enhance the understanding of a program that has already been written.

Computer-Aided Software Engineering (CASE) workbench in the class of maintenance and reverse engineering such as CIA [3], Rigi [8, 17], PBS [6] and SNiFF+ [16] are normally incorporated with editor window in which the extracted software artifacts will be visualized graphically besides their textual information. These tools aid and optimize software engineers' program comprehension or cognitive strategies, particularly when there is an absence of design level documentation that is still a major problem in software engineers' practice [14]. Existing methods of the tools focus on visualizing the software artifacts whilst structural re-documentation as another aspect provided. Nevertheless, they do not explicitly grant the environment to re-document software systems via their viewers.

Another type of CASE tool of class analysis and design such as Rational Rose is also incorporated with reverse engineering utility. However it should be highlighted that this tool focuses more on forward engineering, while reverse engineering as part of its utilities. Thus reverse engineering an existing software system using this tool without proper forward engineering will only produce the relationships of classes that might not be so meaningful to software maintainers who are confronted with out-dated or absence of documentation. Hence such tool is not within the scope of our work.

This paper proposes DocLike Modularized Graph (DMG) method employed in DocLike viewer prototype tool that represents the existing software architectures graphically in a modularized and standardized document-like manner. The discussion and evaluation of our DMG method in DocLike viewer was based on Storey's work [10] that provides the cognitive framework to describe and evaluate software exploration tools, or in our context we refer them as SV tools. The method was also empirically evaluated based on productivity and quality of program comprehension.

The remainder of the paper is organized as follows. Sections 2 and 3 briefly discuss DocLike Modularized Graph method and DocLike viewer prototype tool, respectively. The tradeoff issues of the method and the aspects of visualizing, understanding and re-

documenting software systems can be found in our previous work [15]. Section 4 includes the evaluation conducted, in addition to illustrating the analysis and inferring the findings. Section 5 discusses some related work. Finally, section 6 draws the conclusion and future work.

2. DocLike Modularized Graph Method

DMG method employs graph to visualize software abstraction. A graph G = (V, E) consists of a set of vertices V and a set of edges E, such that each edge in E is a connection between a pair of vertices in V [9]. DMG uses a directed graph described as directed edge $e_n = (v_i, v_j)$. A vertex in G can be of different types. Currently DMG only considers the types as in structured programming, which are symbolized as module (M), program (P), procedure or function (F) and data (D).

We provide five types of DMG representations, defined as the follows:

1. Module decomposition: $DMG_1 = (V_i, E_i)$ where the set $V_i \subseteq M$ represents all modules in set M and E_i represents relationship (calls, m_1, m_2).
2. Module m_i description: $DMG_2 = (V_i, E_i)$ where the set $V_i \subseteq P$ represents all programs of set P associated to module m_i and E_i represents relationship (calls, p_1, p_2) in module m_i only.
3. Module mi interface: DMG_3 = (Vi, Ei) where the set Vi $\subseteq$ F represents all procedures or functions of set F associated to module mi and Ei represents relationship (calls, f1, f2) in module mi only.
4. Module mi dependencies: DMG_4 = (Vi, Ei) where the set Vi $\subseteq$ F represents all procedures or functions of set F associated to module mi and Ei represents relationship (calls, f1, f2) in module other than mi including the compiler standard library.
5. Module mi data dependencies: DMG_5 = (Vi, Ei) where the set Vi $\subseteq$ F and Vi $\subseteq$ D represent all procedures or functions Fi of set F in program Pi of module mi and all associated global data of set D defined in program Pi or header file .h, while Ei represents the use of data (either read or write or both read and write) by Fi.

3. DocLike Viewer Prototype Tool

DocLike viewer is initially based on the C language parser provided by Rigi tool [8]. We filter the software artifacts extracted by selecting only the required artifacts that are going to be visualized via DocLike viewer. DocLike viewer consists of three main panels: Content Panel, Graph Panel and Description Panel (see Figure 1).

Based on the cognitive framework of Storey [10], the two major elements to describe and evaluate SV tools such as DocLike viewer are:

1. Improve program comprehension (enhance bottom-up comprehension: E1 to E3, enhance top-down comprehension: E4 and E5, integrate bottom-up and top-down approaches: E6 and E7)
2. Reduce the maintainer's cognitive overhead (facilitate navigation: E8 and E9, provide orientation cues: E10 to E12, reduce disorientation: E13 and E14).

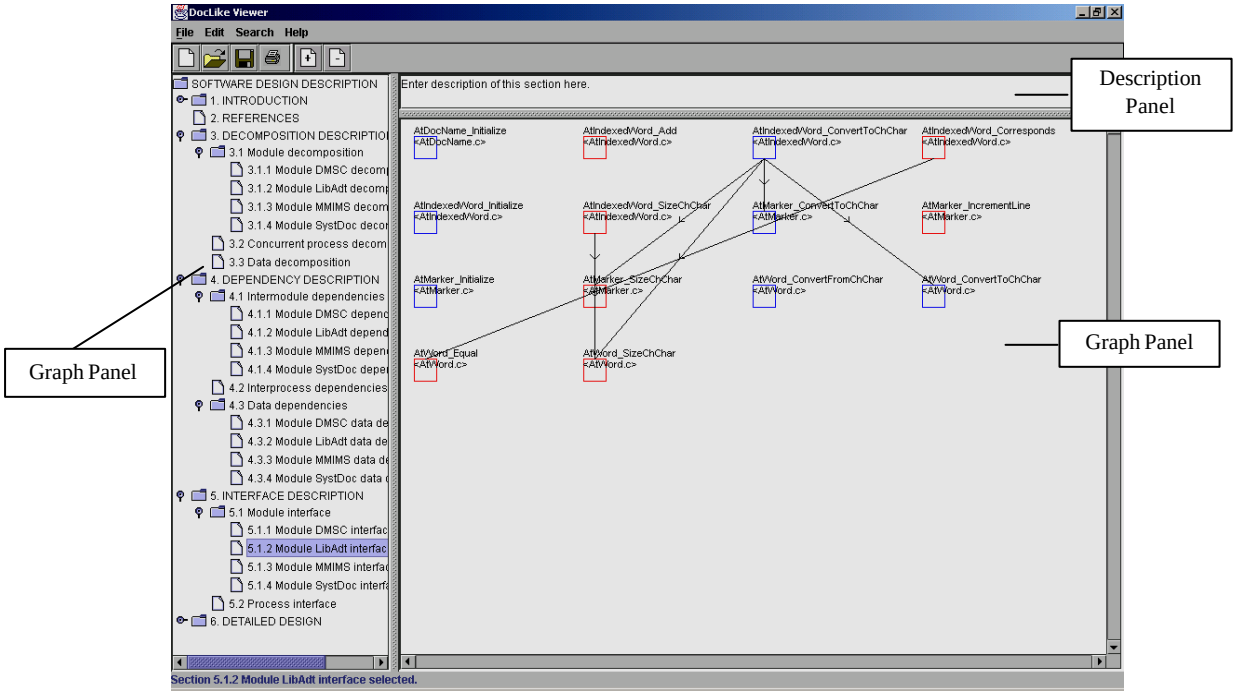

Figure 1. DocLike viewer consists of content panel, graph panel and description panel.

Refer [10] for the details of the activity code E1 to E14 mentioned above. From Table 1, it is observed that DocLike viewer does not support any feature for E4, E11 and E13 activity code. The rest of the activities are supported at least by one feature in DocLike viewer.

Table 1. Formulation of criteria to be evaluated based on Storey's cognitive framework.

Criteria (C1 to C12)	Activity Code (refer [10])	Does DocLike Viewer Support? (Yes/No)
C1: Easy to identify affected components	E1, E6, E10	Yes
C2: Easy to identify dependencies in a module	E3, E5	Yes
C3: Easy to identify dependencies among modules	E3, E5	Yes
C4: Easy to navigate among windows	E7, E12	Yes
C5: Easy to navigate the components link	E8	Yes
C6: Easy to trace back previous navigation	E11	No
C7: Easy to trace link between graphical representation and source code	E2	Yes
C8: Good tool to assist re-documenting system	-	Yes
C9: Information provided is well organized	E14	Yes
C10: Graphical information provided is sufficient	-	Yes
C11: Textual information provided is sufficient	-	Yes
C12: Search utility provided is efficient	E9	Yes

4. The Evaluation

Two controlled experiments were conducted to study the significance of improvement in software understanding or program comprehension. The selected subjects who mostly had programming experience studied the subject system using DocLike Viewer (DV) and they were compared to those using Rigi (RG) and Microsoft Visual C++ (MV).

4.1. Hypothesis and Goal/ Question/ Metric

As described in section 1, SV has the objective to use graphics in order to enhance the understanding of a program that has already been written [7]. A number of studies applied experiments to measure this factor such as in [2, 4, 11], which measure program comprehension by providing a list of maintenance tasks to be solved by the selected subjects. Our experiment used the same variables as in [2, 4]. The null hypothesis can be described as:

H_0: The DMG method will not significantly improve program comprehension or software understanding.
Based on the Goal/ Question/ Metric (GQM) paradigm [1, 5], we indicate the goals, questions and metrics for the study as the followings:

1. The goal: the main goal was to statistically analyze how much the proposed DMG method could improve program comprehension in order to solve maintenance tasks. From the main goal, two sub-goals derived involving productivity and quality as shown in Table 2.
2. The questions: the questionnaire had three sections:
   - *Section A*: Expertise-related questions that can determine the expertise of the subjects.
   - *Section B*: Program comprehension improvement-related questions comprised 6 maintenance task questions that were formulated in such a way to simulate a change (corrective or adaptive) or a new requirement (perfective), which may need different levels of information abstraction [13] including system hierarchy view, call graphs and data flow graphs.
   - *Section C*: Usefulness-related questions that were usefulness of the tool used in overall and also by criteria as formulated within the cognitive framework (see Table 1). Refer Table 3 for the list of questions.
3. The metrics: The metrics used in our study are shown in Table 4.

Table 2. The goal of study.

Goal of Study	Purpose: Analyze for the Purpose of	Perspective: with Respect to	Perspective: from the Point of View
Goal	Improvement of program comprehension	Programmers' cognition	Programmers
Sub-goal 1: Productivity	Productivity of program comprehension	Programmers' speed to solve maintenance tasks	Software manager
Sub-goal 2: Quality	Quality of program comprehension	The correctness of solution given	Software manager
Sub-goal 3: Usefulness	Usefulness of the tool and its criteria	Programmers' needs	Programmers

Table 3. The questions formulated.

Section A: Expertise-Related Questions.
A1: Last job before joining Master program e.g. programmer.
A2: Software development or maintenance experience in previous companies (if any) e.g. less 1 year.
A3: Grade in C language module e.g. grade A.
Section B: Program Comprehension Improvement-Related Questions.
1. System hierarchy view (high level of abstraction).
B1: Which module might have no change if the MMIMS module in GI system needs to be maintained?
B2: Which program has the highest number of procedures or functions?
2. Call graph (low level of abstraction).
B3: List the procedures or functions in other module that are called by index_Record not including those from standard library (if any).
B4: What procedure or function calls processWordToIndex?
3. Data flow graph (low level of abstraction).
B5: Which procedure accumulates the value of data from AtMarker_Tmarker?
B6: Identify the function that checks whether a word exists in dictionary or not.
Section C: Usefulness-Related Questions.
C1: Specify the usefulness of the tool provided to understand GI system.
C2: Specify your opinion on the criteria of the tool. The 12 criteria given shown in Table 1. The evaluation based on Likert scale 1. Strongly Disagree, 2. Disagree, 3. Normal, 4. Agree, 5. Strongly Agree.

The three tools are the independent variables or factors whilst the dependent variables are time taken (T) and number of correct answers (S). The attribute variables are related to expertise of programmers and usefulness of tools (see Table 4).

Table 4: The metrics used.

Related to Expertise of Programmers.
M1.1: Last job before doing Master program.
M1.2: Year of experience in software development or maintenance.
M1.3: Grade of C language.
Related to Productivity – Based on Time (T).
M2.1: Time taken to answer each question regardless of correctness (T_1).
M2.2: Time taken to answer each question correctly (T_2).
Related to Quality.
M3.1: Score or sum of correct answers (S) for question (B1 to B6 – see Table 3).
Related to Usefulness of Tool Used.
M4.1: Mean of the usefulness of the tool used in overall (M_1).
M4.2: Mean of the usefulness of the tool used for each criteria (C1 to C12 – see Table 1) provided (M_2).

4.2. Experiment

We chose Rigi, the latest version available [8] and Microsoft Visual C++ 6.0 programming editor as the controls of our experiment. Rigi was chosen because it is quite a representative tool within the scope of our study and has the most criteria needed to compare with our tool. We believed in some ways using program editors with the search text utility could be sufficient enough to understand a subject system but in some ways these tools might not be able to challenge SV tools. Thus we chose the most unanimous programming editor Microsoft Visual C++ as another control of our experiment. Although Visual C# is the latest technology of Visual.net, the tool is still new and not widely used compared to its predecessor.

4.2.1. Subjects and Subject System

The subjects of the first and second experiment involved 33 and 27 of Master students in Software Engineering, respectively. Both experiments were conducted after a Maintenance Module taught. In consequence, subjects were exposed with the issues in software maintenance including the tools that can assist static analysis during program comprehension and the concepts of maintenance tasks and ripple effects.

The subject system used in the experiment was Generate Index (GI) system written in C language consisted of approximately 900 lines of codes (not including comments). The GI was a word processing system that could generate the index of the text file created and edited by a user. The system was introduced to the subjects to perform their minor project assignment and they also had taken C language module in the previous semester. Consequently, the subjects had some ideas of what the system all about and the C language itself. Their previous experience could eliminate our effort to brief on subject system because they already had some domain and application knowledge. This enabled us to focus on training the subjects to use the tools.

4.2.2. Procedures

The subjects were divided into 3 groups consisted of 11 individuals in the first experiment and 9 individuals in the second experiment. The grouping was supervised in such a way that all the groups had a fairly equal level of expertise, which were based on their previous job (if any), experience in software line and also grade in C language module. Each group was required to use different tool that was DocLike viewer, Microsoft Visual C++ or Rigi and each group was identified as DV, MV and RG respectively. All subjects were briefed for 5 to 10 minutes on the use of the dedicated tool to find solutions for the maintenance tasks given (see section B in Table 3) without changing the source codes. For the second experiment, the subjects were given a brief user manual handout of the dedicated tool and a better training. They were provided with stopwatch to indicate the time taken for each question. They were allowed to answer all questions without any time limit. Then they were required to evaluate the tool used by answering section C (see Table 3).

4.2.3. Possible Threats

There were a few factors that could be possible threats to our study. The level of expertise might be a threat; hence we studied subjects' experience and expertise via section A of the questionnaire (see Table 3). When grouping the subjects we considered all the three attributes: last job position, years of experience in previous job and grade in C language module. During the analysis of the two experiments, we tested the correlation of subjects' expertise with time and score. We found no significant correlation between the expertise factor and the two dependent variables. Thus this factor was not a threat.

Another factor could be the leak of questions on maintenance tasks among the subjects. Due to lack of computers, the subjects took turns to perform the experiment. Besides, they were not quarantined and sat next to each other in the lab. Therefore some subjects might have some hints from their friends and when their turn came for the experiment they most probably had prepared with some answers and cues, which indirectly could affect the time taken to answer and correctness of the answers given. We attempted to eliminate the threat by reminding the tested subjects not to leak the questions because they were going to be evaluated individually for 5% assessment of Maintenance Module taught earlier without informing them that DocLike viewer was a tool of the researcher to avoid any Hawthorne effect.

There could be a bias on the actual capabilities of Rigi and Microsoft Visual C++ tools that might have been hindered during the two experiments. For example for Rigi, we did not manage to link the node clicked with Notepad source codes editor as what Rigi claimed. Due to time constraint we could not verify the problem with Rigi developers hence we just trained RG group to open existing Notepad tool to view the source codes we attempted to eradicate the threat by opening Notepad application by the side of Rigi tool and opening a program from GI system from the physical folder. We projected this alternative could minimize the threat particularly on time factor. But for the second experiment we managed to overcome the problem and this matter was not a threat anymore. Better training was also provided in the second experiment.

4.3. Analysis

The analysis of the experiment was based on the metrics and variables described in Table 4. Using the first metric of M2.1 that was related to productivity (see Table 4), we found that the DV group took the shortest time T_1 to answer question 1 (128 seconds) but the longest in 50% of the questions (see Figure 2), which the results were not so conclusive. Nevertheless after the speed of DocLike viewer was improved, the DV group was the fastest in answering all the six questions in the second experiment (see Figure 3).

We performed Oneway Anova to test the significance on the time consumed T_1 by all the groups based on $\alpha/2$ (two-tailed) that is 0.025. In the first experiment, the probability for the phenomena to occur was only significant for the time taken to answer question 3 with the difference 0.016. We used Post-Hoc Anova Tukey and LSD to test the significance of difference among the three groups. Only the pair of the DV versus RG group had significant time mean difference to answer question 3 with the value 0.013 (Tukey) and 0.005 (LSD) at the 0.05 level.

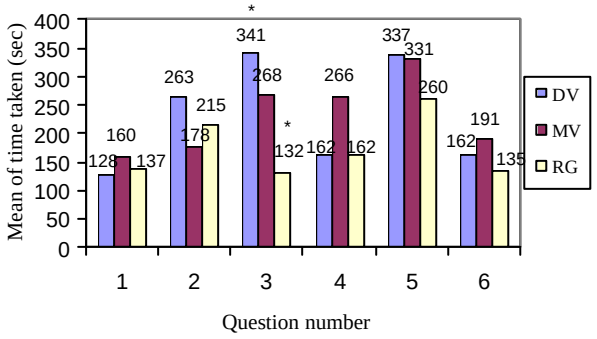

Figure 2. Mean of time taken (regardless of correctness) T_1 in the first experiment. The asterisk (*) shows the significant mean difference.

For the second experiment, by using Oneway Anova test, we found half of the questions had significant difference of T_1 value. Based on Post-Hoc Anova Tukey and LSD test, the time taken by the DV group was significant in question 1, 2 and 5 compared to the other two groups. For question 1, both pair of DV versus MV group and pair of DV versus RG group had significant mean difference of time T_1 with the values 0.023 (Tukey) and 0.009 (LSD); 0.024 (Tukey) and 0.009 (LSD) respectively. For question 2, only the pair of DV versus RG group had the significant mean difference of time with the value 0.000 for both tests. Finally, for question 5, the significant mean difference was only for the pair of DV and MV group with the value 0.021 (Tukey) and 0.008 (LSD).

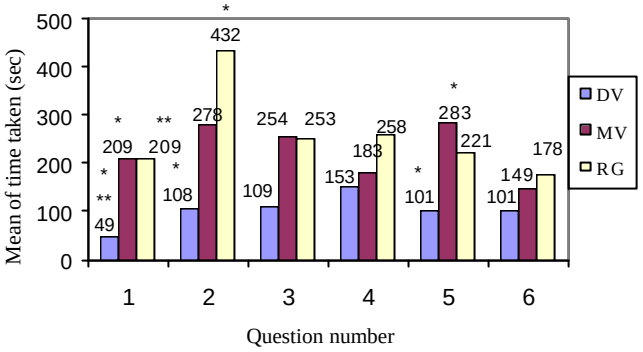

Figure 3. Mean of time taken (regardless of correctness) T_1 in the second experiment. The asterisk (*) shows the significant mean difference.

The metric M3.1 that was related to quality (see Table 4) indicated the sum of score S for each question. In the first experiment Figure 4 illustrates that the value of S is the highest by the DV group in question 1, 4 and 5 (half of the questions). The DV group scored the least for question 2 and 3. Using the same test of Oneway Anova, we identified that only the score for question 2 and 4 were significant ie. 0.019 and 0.001 respectively (< 0.025). While comparing the difference of scores among pairs of groups at 0.05 level, we discovered that the difference was significant in question 2 for the DV versus MV group by 0.016 (Tukey) and 0.006 (LSD). For question 4 we found all the pairs had significant score difference DV versus RG by 0.002 (Tukey) and MV versus RG by 0.008 (Tukey) while 0.001 and 0.003 respectively in LSD test. Comparing Figure 2 and Figure 4, we discovered that for question 2 and 3, the RG group took the longest time but the least score.

On the other hand, the results were more encouraging in the second experiment. Although the DV group scored the highest in question 4 only, the rest of the questions were scored well (see Figure 5). Based on Oneway Anova and Post-Hoc Anova Tukey and LSD test, we indicated the significant score difference was in question 4 only for the pair DV versus RG (0.000 for both tests) and MV versus RG (0.001 for Tukey and 0.000 for LSD). Regarding the total of S for the whole six questions, in the first

experiment it was scored the highest by the MV group (48 out of 66 i. e. 73%) followed by the DV group (65%) and the RG group (61%). However, for the second experiment, the total of S was scored the highest by the DV group (47 out of 54 i. e. 87%) followed by the MV group (81%) and the RG group (80%).

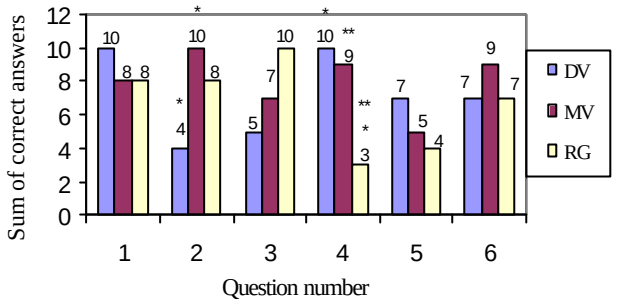

Figure 4. Score S in the first experiment. The asterisk (*) indicates the significant score difference.

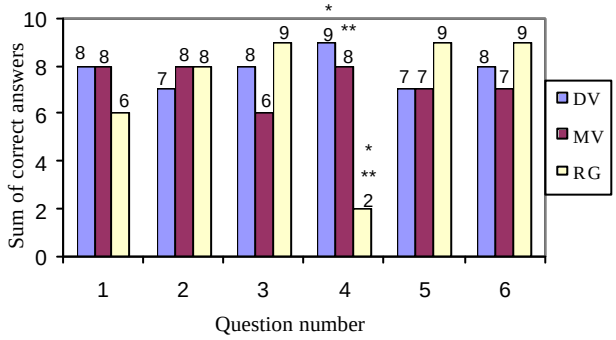

Figure 5. Score S in the second experiment. The asterisk (*) shows the significant score difference.

By measuring using the metric M2.2 related to productivity, the mean of time T_2 consumed by DV group to answer correctly in the first experiment was the shortest for question 1, 4 and 6 (135, 171 and 80 seconds respectively) compared to the control groups (see Figure 6). By comparing to the values in Figure 2, we observed that for the first four questions the values of T_2 were more than T_1 but for the last two questions the values of T_2 were less than T_1. Using Univariate Analysis of Variance test, we indicated that only the time taken to answer question 3 correctly had significant difference for the pair of DV and RG group with the value 0.015 (Tukey) and 0.005 (LSD).

For the second experiment, Figure 7 deduces that the DV group took slightly longer time to answer correctly compared to the MV group in question 4. Thus the DV group did not take the shortest time in all questions in order to answer correctly compared to Figure 3 in which the group took the shortest time for all questions. However, in overall the values of T_1 and T_2 for the DV group in the second experiment had very little difference.

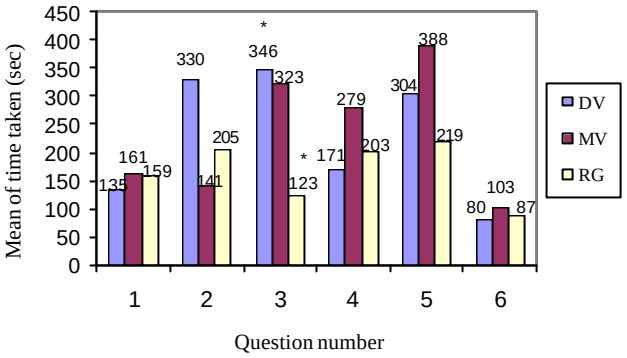

Figure 6. Mean of time taken to answer correctly T_2 in the first experiment. The asterisk (*) indicates the significant mean difference.

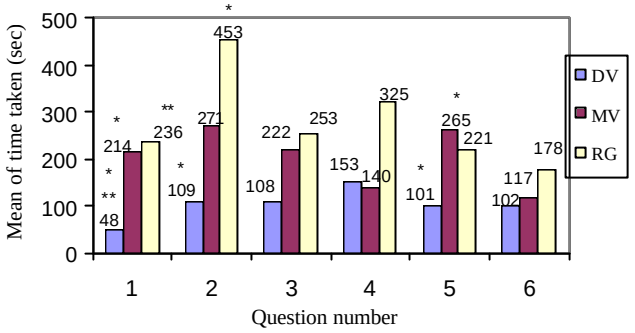

Figure 7. Mean of time taken to answer correctly T_2 in the second experiment. The asterisk (*) indicates the significant mean difference.

For the value of variable M_1 of metric M4.1, usefulness of the tools in overall, Figure 8 depicts that the DV group gave the most positive opinion towards the tool in the first and second experiment (4.27 and 4.44 respectively) followed by RG group (4.00) and MV group (3.45) in the first experiment. However, in the second experiment, the MV group had more positive opinion (3.33) compared to the RG group (3.22). The mean values given were based on Likert scale:

1. Strongly disagree.
2. Disagree.
3. Normal.
4. Agree.
5. Strongly agree.

Based on the metric M4.2 (see Table 4), Figure 9 portrays that DocLike viewer derived the most positive opinion or mean value M_2 towards each criterion (C1 to C12) provided by the tool compared to the other two groups in both experiments. But the MV group gave more positive opinion towards the criteria in the second experiment compared to that of the first experiment. Whereas, the RG group gave more positive opinion in the first experiment but not that of second experiment.

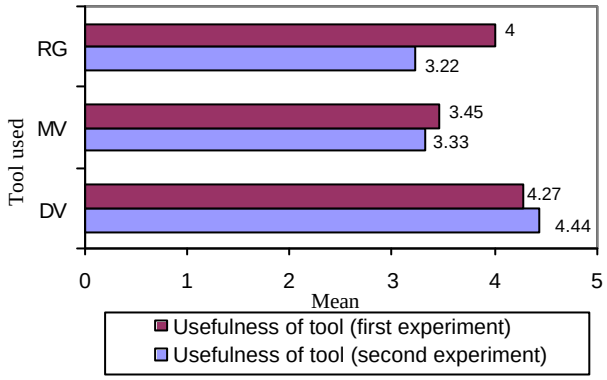

Figure 8. Usefulness of tool from the perspective of programmers in the first and second experiment.

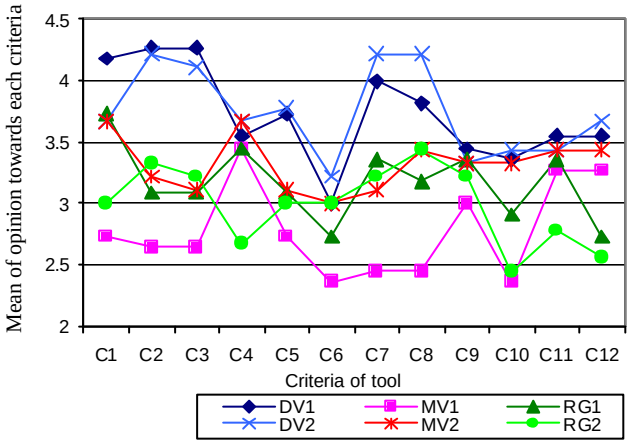

Figure 9. Subjects' opinion towards criteria of the tool used in the first and second experiment (indicated as 1 and 2 respectively in the legend), mean-- values based on Likert scale (see Table 5), see Table 1 for description of criteria code.

Table 5. The mean values for Figure 10 based on Likert scale (1. Strongly disagree, 2. Disagree, 3. Normal, 4. Agree, 5. Strongly agree), see Table 1 for description of criteria code.

Criteria/ Tool	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12
DV1	4.18	4.27	4.27	3.55	3.73	3.00	4.00	3.82	3.45	3.36	3.55	3.55
MV1	2.73	2.64	2.64	3.45	2.73	2.36	2.45	2.45	3.00	2.36	3.27	3.27
RG1	3.73	3.09	3.09	3.45	3.09	2.73	3.36	3.18	3.36	2.91	3.36	2.73
DV2	3.67	4.22	4.11	3.67	3.78	3.22	4.22	4.22	3.33	3.44	3.44	3.67
MV2	3.67	3.22	3.11	3.67	3.11	3.00	3.11	3.44	3.33	3.33	3.44	3.44
RG2	3.00	3.33	3.22	2.67	3.00	3.00	3.22	3.44	3.22	2.44	2.78	2.56

4.4. Findings

From the analysis we observed that in the first experiment the DV group took longer time to answer half of the questions regardless of correctness (T_1) and one-third of the questions with regard to correctness (T_2). However after DocLike viewer was improved, the speed was much better in the second experiment in which the DV group took the least time to answer all the questions regardless of correctness (T_1) and five of the six questions with regard to correctness (T_2). Better training also contributed to the speed of the groups. Referring back to the goal of the study (see Table 2) the improved DocLike viewer managed to achieve the first sub-goal that was related to productivity of program comprehension based on the speed or time taken to solve maintenance tasks. Thus productivity is important from the point of view of software managers towards their programmers' performance. If productivity can be improved, a maintenance project most probably needs a shorter time therefore it incurs lower cost.

In term of quality of program comprehension indicated as the second sub-goal of this study (see Table 2), if more correct answers or solutions (S) given, thus fewer errors occur. Consequently, this will directly lessen the debugging activities after source codes have been changed. Although in the first experiment the DV group scored the highest in half of the questions, the overall score was the highest from the MV group. Thus by providing better training that was insufficient in the first experiment, the DV group managed to have the highest overall score in the second experiment. Surprisingly, we observed that the RG group scored significantly less in question 4 (What procedure or function calls processWordToIndex?) in both experiments. After checking the view provided by Rigi for this question, we suspected that the incoming and outgoing arcs for the concerned node confused the RG group. Hence with better training delivered in the second experiment, the quality was improved not only on the DV group but also the other groups as long as the tools did not mislead the programmers with wrong solution such as in the case of question 4 answered by the RG group.

Concerning the null hypothesis described in section 4.1. as the statistical benchmark of this study, DV group did not manage to totally reject the null hypothesis because not all the values of variables measured in terms of productivity and quality of program comprehension had significant difference compared to the other two groups. However, the group gave the most positive opinion towards the usefulness of DocLike viewer in overall and each criterion provided. This fact portrays that our tool has the potential to enhance understanding or cognition of software systems via its DMG method.

5. Related Work

There is a number of related work but we just discuss some examples only. An example is Rigi [8, 17] that provides SV in reverse engineering environment that applies two approaches to present software structures in its graph editor. The approaches are: 1. multiple, individual windows 2. fisheye views of nested graphs called SHriMP (Simple Hierarchical Multi-Perspective) [10]. While PBS [6] uses the approach of software landscape and represents the software abstractions in web pages. An example of commercial tool is SNiFF+ [16] that provides column-by-column

view of software artifacts. Some studies evaluated how SV method used could enhance software understanding of an existing software system in some aspects such as programmers' cognition strategies [11, 18] or program comprehension [2, 4]. Our previous work had identified the drawbacks and strengths of the graph methods used by SV tools (Rigi, PBS, SNiFF+ and Logiscope) [12] and also a comparative study on the features and analysis aspects of the four tools [13]. Based on the study we found that most SV methods used by the tools need user intervention to collapse the nodes into subsystems after software abstraction visualized except for PBS that optionally allow users to collapse components prior to generating of views. Even if source codes parsed are not very large in size, the graph presented will be quite complicated, with crossing of arcs except for SNiFF+ (because graph drawn column-by column). Besides, none of the tools employ an explicit document-like re-documentation environment in their SV methods.

Our work differs from existing methods by improving program comprehension and reducing cognitive overhead using DMG method that proposes a standardized, modularized and document-like SV.

6. Conclusion and Future Work

SV can improve cognition of an existing software system particularly when software engineers are confronted with out-dated or absence of design documents. However, current approaches in graph drawing of SV methods tend to produce overcrowded or confined graph even if source codes parsed are not very large and they do not provide better environment to structural re-documentation of the subject system. Hence we propose a document-like SV method called DocLike Modularized Graph that provides graph representation module-by-module in a document-like re-documentation environment. We realized the method in DocLike viewer tool and conducted two experiments to evaluate how much our DMG method can improve program comprehension in solving different types of maintenance tasks. Although in some maintenance tasks DocLike viewer could not significantly improve productivity and quality, generally programmers who used DocLike viewer could find solutions of maintenance tasks much faster thus enhancing the productivity and they could obtain more correct solutions or fewer errors thus enhancing the quality. On the other hand, the most positive opinions given by the users towards the usefulness of DocLike viewer in overall and each criterion provided by the tool reflect that DMG method has enhanced cognitive aspects of existing SV methods.

Future work should include the finding of weaknesses in the criteria with less positive opinions and then improve the criteria towards the maximum. In addition the future work should also consider the testing of DMG method of DocLike viewer on a larger software system.

Acknowledgement

We would like to thank the reviewers, participants of the experiment, Rigi researchers and also other individuals who indirectly contributed to this research.

References

[1] Basili V. R., "Software Modeling and Measurement: The Goal/ Question/ Metric Paradigm," *University of Maryland Technical Report*, UMIACS-TR-92-96, 1992.

[2] Binkley D., "An Empirical Study of the Effect of Semantic Differences on Programmer Comprehension," *in Proceedings of the 10th International Workshop on Program Comprehension*, IEEE Computer Society Press, USA, pp. 97-106, 2002.

[3] Chen Y. F., Nishimoto M. Y., and Ramamoorthy C. V., "The C Information Abstraction System," *IEEE Transactions on Software Engineering*, vol. 16, no. 3, pp. 325-334, 1990.

[4] Hendrix T. D., Cross J. H. II, and Maghsoodloo S., "The Effectiveness of Control Structure Diagrams in Code Comprehension Activities," *IEEE Transactions on Software Engineering*, vol. 28, no. 5, pp. 463-477, 2002.

[5] Mashiko Y. and Basili V. R., "Using the GQM Paradigm to Investigate Influential Factors for Software Process Improvement," *Journal of Systems and Software*, vol. 36, pp. 17-32, 1997.

[6] Parry T. III, Lee H. S., and Tran J. B., "PBS Tool Demonstration Report on Xfig," *in Proceedings of the 7th Working Conference on Reverse Engineering*, IEEE Computer Society Press, USA, pp. 200-202, 2000.

[7] Price B. A., Baecker R. M., and Small I. S., "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, vol. 4, pp. 211-266, 1993.

[8] Rigi, "Rigi Group Home Page," http://www.rigi.csc.uvic.ca, 2004.

[9] Shaffer C. A., *A Practical Introduction to Data Structures and Algorithm Analysis*, Prentice-Hall, New Jersey, pp. 12-21, 1997.

[10] Storey M. A. D., Fracchia F. D., and Muller H. A., "Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration," *Journal of Systems and Software*, vol. 44, pp. 171-185, 1999.

[11] Storey M. A. D., Wong K., and Muller H. A., "How Do Program Understanding Tools Affect How Programmers Understand Programs?," *in Proceedings of the 4th Working Conference on*

Reverse Engineering, IEEE Computer Society Press, USA, 1997.

[12] Sulaiman S. and Idris N. B., "An Enhanced Approach of Software Visualization in Reverse Engineering Environment," *in Proceedings of the National Conference on Computer Graphic and Multimedia (CoGRAMM'02)*, UTM Press, Malaysia, pp.459-464, 2002.

[13] Sulaiman S., Idris N. B., and Sahibuddin S., "A Comparative Study of Reverse Engineering Tools for Software Maintenance," *in Proceedings of the 2nd World Engineering Congress Information and Communications Technology (ICT)*, UPM Press, Malaysia, pp. 478-483, 2002.

[14] Sulaiman S., Idris N. B., and Sahibuddin S., "Production and Maintenance of System Documentation: What, Why, When and How Tools Should Support the Practice," *in Proceedings of the 9th Asia Pacific Software Engineering Conference (APSEC'2002)*, IEEE Computer Society Press, USA, pp. 558-567, 2002.

[15] Sulaiman S., Idris N. B., Sahibuddin S., and Sulaiman S., "Re-Documenting, Visualizing and Understanding Software Systems Using DocLike Viewer," *in Proceedings of the 10th Asia Pacific Software Engineering Conference (APSEC 2003)*, IEEE Computer Society Press, USA, pp. 154-163, 2003.

[16] Wind River, "Wind River: IDE: SNiFF+," http://www.windriver.com/products/html/sniff.html, 2004.

[17] Wong K., Tilley S. R., Muller H. A., and Storey M. A. D., "Structural Redocumentation: A Case Study," *IEEE Software*, vol. 12, no. 1, pp. 46-54, 1995.

[18] Zayour I. and Lethbridge T. C., "Adoption of Reverse Engineering Tools: a Cognitive Perspective and Methodology," *in Proceedings of the 9th International Workshop on Program Comprehension*, IEEE Computer Society Press, USA, pp. 245-255, 2001.

Shahida Sulaiman BSc in computer science, MSc in real-time software engineering, and PhD in computer science. She is a lecturer in the Faculty of Computer Science, University Sains Malaysia. Her field of interest is software visualization for software maintenance.

Norbik Bashah Idris is a professor and director of the Center for Advanced Software Engineering at University Technology Malaysia (KL Campus). He received his BSc in computer science from the University of New South Wales in Sydney, MSc from University of Queensland, Brisbane and PhD in ICT Security from the University of Wales, UK. He is a certified system security professional by The International Association for Computer System Security, USA and received a certificate in industrial software engineering from Universite Thales, France. He is a member of SIGSAC ACM, IEEE, New York Academy of Science and IFIP.

Shamsul Sahibuddin is an associate professor and the deputy dean (development) in the Faculty of Computer Science and Information Technology of University Technology Malaysia. He received his BSc in computer science from Western Michigan University, USA, MSc in computer science from Central Michigan University, USA, and PhD in computer science from Aston University, Birmingham, UK. He is member of the program committee for Asia-Pacific Conference in Software Engineering since 2003. His fields of specialization are computer supported cooperative work, computer network, and software quality. He is a member of ACM.