P2306

Final Report

USM Short Term Grant
304/PKOMP/636009

15 March 2005 – 14 June 2007

Title:
A User-Defined Approach for Reverse Engineering Tool to Visualize, Understand and Re-document Existing Software Systems (UDARE)

Project Leader:
Shahida Binti Sulaiman, PhD

Submission Date:
20.07.2007

LAPORAN AKHIR PROJEK PENYELIDIKAN JANGKA *PENDEK*

FINAL REPORT OF SHORT TERM RESEARCH PROJECT

Sila kemukakan laporan akhir ini melalui Jawatankuasa Penyelidikan di Pusat Pengajian dan Dekan/Pengarah/Ketua Jabatan kepada Pejabat Pelantar Penyelidikan

1. **Nama Ketua Penyelidik:** Shahida Binti Sulaiman
   Name of Research Leader

   ☐ Profesor Madya/ *Assoc. Prof.* [✓ Dr./ *Dr.*] ☐ Encik/Puan/Cik *Mr/Mrs/Ms*

2. **Pusat Tanggungjawab (PTJ):** Pusat Pengajian Sains Komputer
   School/Department

3. **Nama Penyelidik Bersama:** Fazilah Haron, Nur'Aini Abdul Rashid, Rosalina Abdul Salam, Rosni Abdullah.
   Name of Co-Researcher

4. **Tajuk Projek:** *Title of Project*
   A User-Defined Approach for Reverse Engineering Tool to Visualize, Understand and Re-document Existing Software Systems (UDARE)

5. **Ringkasan Penilaian**/*Summary of Assessment*:	Tidak Mencukupi *Inadequate* 1	2	Boleh Diterima *Acceptable* 3	Sangat Baik *Very Good* 4	5
i) **Pencapaian objektif projek:** *Achievement of project objectives*	☐	☐	☐	√	☐
ii) **Kualiti output:** *Quality of outputs*	☐	☐	☐	√	☐
iii) **Kualiti impak:** *Quality of impacts*	☐	☐	√	☐	☐
iv) **Pemindahan teknologi/potensi pengkomersialan:** *Technology transfer/commercialization potential*	☐	☐	√	☐	☐
v) **Kualiti dan usahasama :** *Quality and intensity of collaboration*	☐	☐	√	☐	☐
vi) **Penilaian kepentingan secara keseluruhan:** *Overall assessment of benefits*	☐	☐	☐	√	☐

6. Abstrak Penyelidikan

(Perlu disediakan di antara 100 - 200 perkataan di dalam **Bahasa Malaysia dan juga Bahasa Inggeris**. Abstrak ini akan dimuatkan dalam Laporan Tahunan Bahagian Penyelidikan & Inovasi sebagai satu cara untuk menyampaikan dapatan projek tuan/puan kepada pihak Universiti & masyarakat luar).

Abstract of Research
(An abstract of between 100 and 200 words must be prepared in Bahasa Malaysia and in English). This abstract will be included in the Annual Report of the Research and Innovation Section at a later date as a means of presenting the project findings of the researcher/s to the University and the community at large)

Bahasa Malaysia:

Pemahaman sesuatu perisian sedia ada khususnya sistem legasi ialah satu tugas rumit. Pembangun atau pengemaskini perisian perlu mempelajari kod sumber sebelum menukar program terlibat dengan bantuan mana-mana dokumen atau tanpa dokumen. Banyak produk CASE (Kejuruteraan Perisian Berbantukan Komputer) atau alat telah muncul untuk membantu pengemaskini perisian yang menghadapi ketiadaan dokumentasi atau ia tidak terkemaskini terutamanya dokumen reka bentuk yang menyediakan maklumat paling terperinci mengenai sistem perisian. Alat-alat seumpama ini kebanyakannya dikenali sebagai alat kejuruteraan terbalik (RE). Alat-alat RE juga dipanggil sebagai alat visualisasi perisian (SV) dalam sesetengah kajian kerana ia bukan sahaja membolehkan pengguna menterbalikkan sistem perisian sedia ada untuk mengekstrak komponen-komponen perisian tetapi ia juga membolehkan para pengguna untuk menvisualisasi kebergantungan artifak atau komponen perisian. Alat-alat sedia ada menggunakan pelbagai teknik RE dan pendekatan SV. Kebanyakan alat ditujukan khas untuk bahasa-bahasa tertentu dan proses RE berhenti jika ia tidak memenuhi sesetengah peraturan RE yang ditentukan oleh alat teresebut. Dalam penyelidikan ini kami mencadangkan pendekatan berdefinisi pengguna untuk persekitaran RE yang membolehkan para pengguna menentukan sintaks bahasa pengaturcaraan berkenaan, jenis-jenis komponen perlu diekstrak dan kebergantungan yang mereka hendak kaji atau visualkan serta panduan dokumen yang diperlukan. Kemudiannya maklumat akan dikemaskini dalam pangkalan data untuk diguna semula atau dikemaskini pada masa hadapan sekiranya pengguna-pengguna perlu mengkaji pelbagai jenis komponen dan kebergantungannya atau bahasa pengaturcaraan yang berbeza. Maka penggunaan pendekatan tersebut dipercayai boleh mengelakkan kelimpahan maklumat dan mampu membantu pemahaman pengemaskini perisian dengan menyediakan persekitaran kejuruteraan terbalik yang lebih fleksibel. Pendekatan ini digelar UDARE yang bermaksud pendekatan berdefinisi pengguna untuk alat kejuruteraan terbalik.

Bahasa Inggeris:

Understanding an existing software system particularly a legacy system is a tedious task. Software developers or maintainers need to study the source codes prior to changing the affected programs with the aid of any documents or even without any document. Many CASE (Computer-Aided Software Engineering) products or tools have emerged to assist software maintainers who are confronted with absence of documentation or out-dated documentation particularly design document that provides the most detail information about a software system. Such tools are dominantly known as reverse engineering (RE) tools. RE tools are also called software visualization (SV) tools in some studies because they do not only enable users to reverse engineer existing software systems to extract software components but they also enable users to visualize the dependencies of software artifacts or components. Existing tools apply diverse RE techniques and SV approaches. Most tools are dedicated for certain languages and the RE process halts if it does not meet some RE rules set by the tools. In this research we propose a user-defined approach for the RE environment that enables users to indicate the syntaxes of the concerned programming language, types of components to be extracted and the dependencies they want to study or visualize and also the document template required. Then the information will be updated in a database to be re-used or edited in the future in case users need to study different types of components and their dependencies or even different programming languages. Hence by using this approach it is believed that we can avoid information overload and is capable to better assist software maintainers' software understanding by providing a more flexible reverse engineering environment. The approach is called UDARE that stands for a User-Defined Approach for Reverse Engineering tool.

UNIVERSITI SAINS MALAYSIA
JABATAN BENDAHARI
KUMPULAN WANG PENYELIDIKAN USM (304)
PENYATA PERBELANJAAN PADA 30 JUN 2007

09/07/2007

JEK
ELINED APPROACH FOR REVERSE ENGINEERING TOOL TO
UNDERSTAND AND RE-DOCUMENT EXISTING SOFTWARE SYSTEM

IA SULAIMAN
NGAJIAN SAINS KOMPUTER

PTJ	PROJEK	DONOR	PERUNTUKAN PROJEK	PERBELANJAAN TERKUMPUL SEHINGGA THN LALU	PERUNTUKAN THN SEMASA	TANGGUNGAN SEMASA	BAYARAN THN SEMASA	BELANJA THN SEMASA	BAKI PROJEK
KOMP	636009		7,594.80	6,152.45	1,442.35	0.00	3,331.00	3,331.00	-1,888.65
KOMP	636009		4,200.00	1,542.40	2,657.60	0.00	340.00	340.00	2,317.60
KOMP	636009		1,193.20	0.00	1,193.20	0.00	0.00	0.00	1,193.20
KOMP	636009		2,400.00	1,291.50	1,108.50	0.00	337.15	337.15	771.35
KOMP	636009		0.00	2,249.20	-2,249.20	0.00	20.00	20.00	-2,269.20
KOMP	636009		3,500.00	3,500.00	0.00	0.00	0.00	0.00	0.00
			18,888.00	14,735.55	4,152.45	0.00	4,028.15	4,028.15	124.30

DISEDIAKAN :

DISEMAK : [illegible]

NOOR BAHIRAH MOHD ALI
Pembantu Tadbir Kewangan (W17)
Jabatan Bendahari
Universiti Sains Malaysia

Lampiran A (disediakan dalam Bahasa Inggeris)

Technical Report

A User-Defined Approach for Reverse Engineering Tool to Visualize, Understand and Re-document Existing Software Systems (UDARE)

1.0 Introduction

Software systems evolve and need to be maintained or enhanced to satisfy new requirements. Without proper documentation it would be difficult for software maintainers to update the software. Hence reverse engineering technique may assist the understanding of existing source codes by software maintainers by extracting software artifacts and represent it in a higher level of abstractions. Most of the representations are in graphical views.

Existing tools apply diverse RE techniques and SV approaches. Most tools are dedicated for certain languages and the RE process halts if it does not meet some RE rules set by the tools. Besides documents generated are either too general that is they are useful only in the initial approach of software understanding but they are ignored during implementation or they are too detailed (Canfora *et al.*, 1991).

In this project we propose a user-defined approach for the RE environment that enables users to indicate the syntaxes of the concerned programming language, types of components to be extracted and the dependencies they want to study or visualize and also the document template required. Then the information will be updated in a database to be re-used or edited in the future in case users need to study different types of components and their dependencies or even different programming languages. Hence by using this approach it is believed that we can avoid information overload and manage to better assist software maintainers' software understanding by providing a more flexible reverse engineering environment. The project is called UDARE that stands for a User-Defined Approach for Reverse Engineering Tool.

Objectives

The objectives of the project are:

(i) To build a reverse engineering prototype tool called UDARE that assists software engineers particularly software maintainers who need to maintain existing software systems without software documentation or with out-dated software documentation.

(ii) To improve the methods or approaches in analyzing, viewing and re-documenting existing software systems employed by existing tools of reverse engineering environment.

(iii) To implement the proposed user-defined approach that provides a more flexible and effective reverse engineering environment in the aspects of determining the syntaxes of programming languages, the types of components and their dependencies, parameter passing and also the document templates required.

Outcome

The outcomes of this project are listed as follows:

(i) A more flexible and effective reverse engineering tool and environment called UDARE.

(ii) Enhanced methods or approaches in analyzing, visualizing and re-documenting existing software systems especially legacy systems.

(iii) A technical paper that has been submitted or published in an International journal or a national conference proceeding.

Importance and Benefit

Maintaining existing software systems without proper documentation is costly. Thus this project should eliminate the problem by providing a better environment to analyze, visualize, understand

and re-document existing software systems. Malaysian software engineers and any software-related departments or research groups in the university could gain the benefit of the outcomes.

In the following sections we will discuss the background, the methodology, and the proposed approach called UDARE followed by its evaluation, conclusion and future work.

2.0 Background

Understanding an existing software system particularly a legacy system is a tedious task. This is due to some reasons such as unstructured code, maintenance programmers having insufficient knowledge of the system or application domain, documentation being absent, out-of-date or at best insufficient and software maintenance had a bad image (van Vliet, 2000). Thus in these cases software maintainers need to study the source codes prior to changing the affected programs with the aid of any documents or even without any document. According to Sulaiman *et al.* (2002b) the problem related to the absence of or out-dated documentation occurs in both software development and maintenance process.

Many CASE (Computer-Aided Software Engineering) products or tools have emerged to assist software maintainers who are confronted with absence of documentation or out-dated documentation particularly design document that provides the most detail information about a software system. Such tools are dominantly known as reverse engineering (RE) workbenches or tools. Some examples are CIA (Chen *et al.*, 1990), Rigi (Rigi, 2004)(Muller *et al.*, 1994), SNiFF+ (Wind River, 2004) and CodeCrawler (Lanza, 2003). The RE technology is also incorporated into some analysis and design tools such as Rational Rose (Rational, 2007). However we focus on the former that is targeted to assist the understanding of existing software systems particularly legacy systems. On the other hand, the latter focuses more on the analysis and design aspect thus it is not within the scope of our interest. Reverse engineer an existing software system using the tool will only produce the class diagram, which may not be so informative to software maintainers. RE tools are also called software visualization (SV) tools in some studies because they do not only enable users to reverse engineer existing software systems to extract software components but they also enable users to visualize the dependencies of software artifacts or components. In addition majority of the tools can also re-document existing software systems therefore they are also known as document generators.

3.0 Methodology

The research methodology comprises the following components that is illustrated in **Figure 1**:
(i) Initiate project with the recruitment of a student assistant.
(ii) Conduct a thorough literature study on existing methods or approaches related to the 4 main modules of CI, CA, CV and DG. Each module should be improved in the related aspects that should complement the proposed user-defined approach for the whole project. This activity will verify the capability of the proposed approach in overall.
(iii) Select and compare existing tools of RE environment. Find their weaknesses and strengths and then produce the analysis of the tools to be incorporated into the UDARE project.
(iv) Design the new modules of CI and CA.
(v) Design the database that will be the software repository for the UDARE project.
(vi) Develop and test the CI and CA modules by employing their own enhanced methods and also the user-defined approach.
(vii) Study DocLike Viewer prototype tool (the output of the applicant's PhD work) to be the foundation of both CV and DG modules. Maintain the tool and integrate it with the other new modules (CI and CA). Enhance the DMG method employed in DocLike Viewer to reflect the proposed user-defined approach.
(viii) Integrate the whole modules of the project and conduct the integration testing.
(ix) Conduct a usability study of UDARE prototype tool among software engineers in USM or other companies that maintain software systems in-house.

(x) Produce a paper to an international journal or a national conference proceeding.
(xi) Compile the documentation for the whole components of UDARE project.
(xii) Close down project.

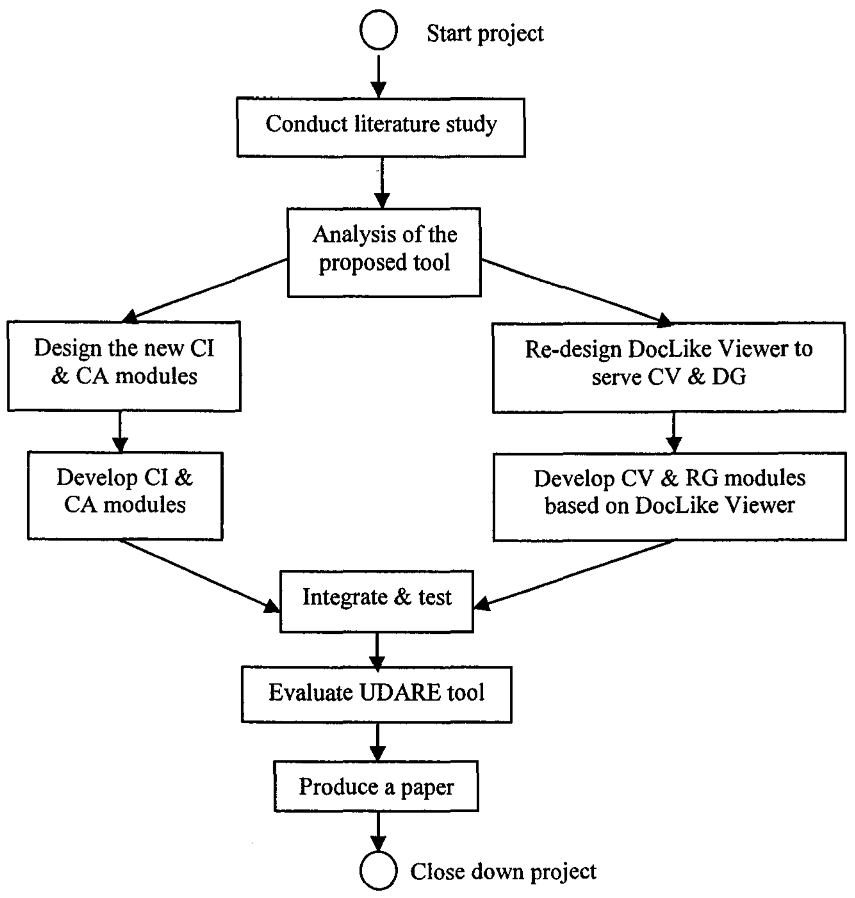
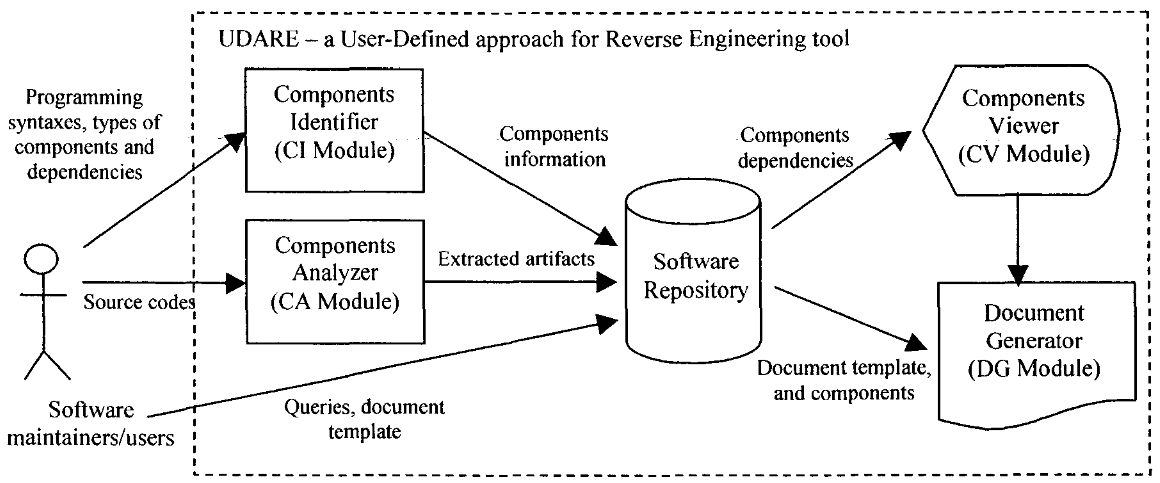

Start project
Conduct literature study
Analysis of the proposed tool
Design the new CI & CA modules
Re-design DocLike Viewer to serve CV & DG
Develop CI & CA modules
Develop CV & RG modules based on DocLike Viewer
Integrate & test
Evaluate UDARE tool
Produce a paper
Close down project

Figure 1: The flow chart of the research methodology

4.0 User-Defined Approach for Reverse Engineering (UDARE)

UDARE provides a more flexible approach that allows users to define the syntaxes of programming language to be parsed before extracting the concerned source codes. **Figure 2** depicts the four (4) main modules that should be incorporated into the reverse engineering environment. They are: Components Identifier (CI), Components Analyzer (CA), Components Viewer (CV) and Document Generator (DG). Once software maintainers or users have indicated the syntaxes of programming languages, types of components and dependencies required via the CI module, the information is updated into the repository. Then users need to input the source codes files into the CA module in order to analyze the source codes and output the extracted artifacts into the software repository as required and indicated by the users via the CI module. From the data in the repository, the CV module should be able to generate the dependencies of components to be viewed by the users while the DG module should generate the documentation of the software artifacts extracted according to the template required by the users. For both CV and DG modules, a part of the functionalities will be based on the previous PhD research (Sulaiman, 2004a). The outcome of the research is a tool called DocLike Viewer that employs a document-like and modularized SV method known as DMG to visualize the artifacts and dependencies of a subject system using graph representations. DocLike Viewer and its method have been discussed from different perspective in a number of papers including Sulaiman and Idris (2002), Sulaiman *et al.* (2002a), Sulaiman *et al.* (2002b), Sulaiman *et al.* (2003), Sulaiman *et* *al.* (2004) and Sulaiman (2004b). One of the journal papers on DMG and DocLike Viewer is enclosed in Appendix B. Currently DocLike Viewer depends on the existing parser of Rigi from University Victoria of Canada (Rigi, 2007). Both CI and CA modules in UDARE is being integrated with DMG method in CV and DG modules.



UDARE – a User-Defined approach for Reverse Engineering tool
Programming syntaxes, types of components and dependencies
Components Identifier (CI Module)
Components information
Components dependencies
Components Viewer (CV Module)
Components Analyzer (CA Module)
Extracted artifacts
Software Repository
Source codes
Document Generator (DG Module)
Document template, and components
Software maintainers/users
Queries, document template

Figure 2: UDARE environment and the components to be incorporated into the tool

A user-defined approach is user-centered in which users need to determine the syntaxes of the language they want to deal with before inputting the source codes to be analyzed. The approach requires the following information from the users for the first time of parsing the specified language:

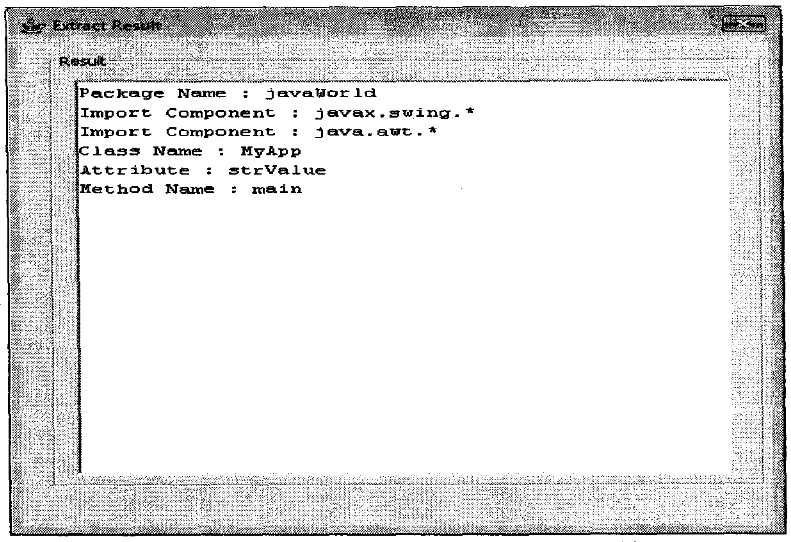
(i) Programming language concerned: Let *L* is the language inputted.
(ii) Identifier type of the language specified: Let *ID* is the identifier.
(iii) Syntax to recognize the identifier: Let *S* is the syntax specified.
(iv) Parsing or RE rules: Let *R* is the rule specified.

Thus for *L* language concerned it will consist of a set of *ID* that corresponds to the specified *S*. Based on the user-defined set of values for *ID* and *S*, the analysis of a program file *P* can be done. In this project we apply parsing-based technique to analyze the program file or source codes. Each *P* consists of a set of tokens *T*. While parsing each line, each T_i value is compared with each S_i value of set *S*. If they are equal, the corresponding ID_i value and other required details such as modifier and type of concerned component would be retained based on the parsing rules R_i.

For example consider the top segment of source code shown in Figure 3(a). Let L = java, P = classA, ID_i = class, S_i = class. To identify a class, a token value T_i must be equal to S_i. Thus in this case once the syntax value is parsed, the concerned token that is classA is the identifier name to be retained under the *class* identifier. In order to identify a method the *ID* and *S* value should be determined including the RE rules to extract the concerned artifacts.

For data, the users can specify whether to consider the variables that hold persistent data only or any variables. For instance in classA the data is data1. In order to consider an association between the two classes as shown by highlighted texts in Figure 3(a), the dot operator may be used as the syntax in Java language. Then the corresponding object instantiation is traced to determine the class associated with it. In this case myClass is the instantiation of classA hence methodA belongs to classA. This relationship is retained in the repository as one of class dependencies.

The extracted artifacts include package, import component, class, method, and attribute. The sample of extracted artifacts is shown in **Figure 5**. In the sample given the package name is javaWorld, while two import components are available named javax.swing.* and java.awt.*. The class name is extracted that is MyApp together with the attribute strValue and method main.



```
Package Name : javaWorld
Import Component : javax.swing.*
Import Component : java.awt.*
Class Name : MyApp
Attribute : strValue
Method Name : main
```

Figure 5: A list of extracted software artifacts

6.0 Conclusion and Future Work

UDARE provides an automatic environment that allows software maintainers to understand written source codes faster and better before actually changing them. The users can define the syntaxes of the source codes that they want to analyze. This provides more flexibility to the users. More details of UDARE can be referred in the system document (Appendix C) and its user manual (Appendix D). UDARE is currently being integrated with existing tool called SoVis that will generate the graphical views and later re-document the extracted artifacts.

In future UDARE will be tested and compared with other existing reverse engineering tools to measure the efficiency of the approach in parsing existing source codes.

References:

Canfora, G., Cimitile, A. and Carlini, U. (1991). A Logic-Based Approach to Reverse Engineering Tools Production. *IEEE Transactions on Software Engineering.* 18(12): 1053-1064.

Chen, Y. –F., Nishimoto, M. Y. and Ramamoorthy, C. V. (1990). The C Information Abstraction System. *IEEE Transactions on Software Engineering.* 16(3): 325-334.

Lanza, M. (2003). *Lessons Learned in Building a Software Visualization Tool.* Proceedings of the 7th European Conference On Software Maintenance and Reengineering (CSMR'03). *USA: IEEE Computer Society Press. 1-10.*

Muller, H. A., Wong, K. and Tilley, S. R. (1994). Understanding Software Systems Using Reverse Engineering Technology. Proceedings of 62nd Congress of L'Association Canadienne Francaise pour L'Avancementdes des Sciences (ACFAS). In: Alagar, V. S. and Missaoui, R. eds. (1995). *Object-oriented Technology for Database and Software Systems.* Singapore: World Scientific. 240-252.

Rational (2007). *Rational Software Corporation.* http://www.rational.com/products/

Rigi (2007). *Rigi Group Home Page.* http://www.rigi.csc.uvic.ca/

van Vliet, H. (2000). *Software Engineering Principles and Practice*. England: John Wiley.

Wind River (2004). *Wind River: IDE: SNiFF+*. Wind River Systems Inc. http://www.windriver.com/products/html/sniff.html

Sulaiman, S., Idris, N. B. and Sahibuddin, S. (2002a). A Comparative Study of Reverse Engineering Tools for Software Maintenance. *2nd World Engineering Congress*. Sarawak, Malaysia. 22-25 July 2002. Malaysia: UPM Press. 478-483.

Sulaiman, S. and Idris, N. B. (2002). Software Visualization Tools for Software Maintenance. *National Conference on Computer Graphics and Multimedia*. Melaka, Malaysia. 7-9 October 2002. Malaysia: UTM Press. 459-464.

Sulaiman, S., Idris, N. B. and Sahibuddin, S. (2002b). Production and Maintenance of System Documentation: What, Why, When and How Tools Should Support the Practice. *9th Asia Pacific Software Engineering Conference*. Queensland, Australia. 4-6 December 2002. USA: IEEE Computer Society Press. 558-567.

Sulaiman, S., Idris, N. B., Sahibuddin, S. and Sulaiman, S. (2003). Re-documenting, Visualizing and Understanding Software Systems Using DocLike Viewer. *10th Asia Pacific Software Engineering Conference*. Chiang Mai, Thailand. 10-12 December 2003. USA: IEEE Computer Society Press. 154-163.

Sulaiman, S., Sarkan, H., Azmi, A. and Mahrin, N. (2004). Visualizing Software Systems Using a Document-Like Software Visualization Method: a Case Study. *International Conference on Computer Graphics, Imaging and Visualization (CGiV04)*. Penang, Malaysia. 26-29 July 2004. Malaysia: USM Press. 191-197.

Sulaiman, S. (2004a). A Document-Like Software Visualization Method for Effective Cognition of C-Based Software Systems. *PhD Thesis*. Malaysia: UTM.

Sulaiman, S., Abdullah, R. and Sulaiman, S. (2004). Exploiting Software Visualization to Re-document Existing Software Systems. *Malaysian Science and Technology Congress (MSTC2004)*. Kuala Lumpur, Malaysia. 5-7 October 2004. Malaysia: COSTAM. 469-476.

Sulaiman, S. (2004b). Viewing Software Artifacts for Different Software Maintenance Categories Using Graph Representations. *Malaysian Journal of Computer Science (MJCS)*. Vol. 17, No. 2, December 2004. Malaysia: UM. 55-67.

Lampiran B: Senarai Penerbitan

Shahida Sulaiman, Rosalina Abdul Salam, Sarina Sulaiman, "A User-Defined Approach for Reverse Engineering to Support Software Understanding", *Proceedings of The First Malaysian Software Engineering Conference (MySEC'05)*, Penang, USM: Malaysia, 246-250, 2005.

Shahida Sulaiman, Norbik Bashah Idris, Shamsul Sahibuddin, "Enhancing cognitive aspects of Software Visualization Using DocLike Modularized Graph", *The International Arab Journal of Information Technology*, Vol. 2, No. 1, Zarka University: Jordan, 1-9, January 2005.

Shahida Sulaiman, Sarina Sulaiman, "A Tutor-Based Software Visualization Approach (TubVis) for Novice Software Engineers", *Proceedings of The Second Malaysian Software Engineering Conference (MySEC'06)*, Kuala Lumpur, UTM: Malaysia, 323-328, 2006.

PROCEEDINGS OF MySEC'05

The First Malaysian Software Engineering Conference

"Maturing Software Engineering Research and Practice"

12 – 13 December 2005 • Penang, Malaysia

Edited by:
Abdullah Zawawi Talib, Ahamad Tajudin Khader and Shahida Sulaiman

Organised by:
School of Computer Sciences, USM

Co-organised by:
Centre for Advanced Software Engineering (CASE), UTM

Sponsored by:
InvestPenang Bhd. & Software Consortium of Penang (SCoPe)

A User-Defined Approach for Reverse Engineering to Support Software Understanding

Shahida Sulaiman, Rosalina Abdul Salam
School of Computer Sciences
Universiti Sains Malaysia
11800 USM
Pulau Pinang
shahida@cs.usm.my, rosalina@cs.usm.my

Sarina Sulaiman
Faculty of Computer Science & Information System
Universiti Teknologi Malaysia
81310 Skudai, Johor
sarina@fsksm.utm.my

ABSTRACT

*Reverse engineering (RE) tools or workbenches have been developed to assist software maintainers who are confronted with absence of documentation or out-dated documentation particularly design document that provides the most detail information about a software system. RE tools are also called software visualization (SV) tools because they facilitate users to visualize the dependencies of software artifacts besides the utility to re-document existing software systems. Such tools apply diverse RE techniques and SV approaches. Most tools are dedicated for certain languages and the RE process halts if it does not meet some RE rules set by the tools. In this paper we propose a user-defined approach for a RE environment that enables users to indicate the syntaxes of the concerned programming language, the RE rules, types of components to be extracted and the dependencies they want to study or visualize and also the document template required. Then the information will be updated in a database to be re-used or edited in the future in case users need to study different types of components and their dependencies or even different programming languages. Hence by using this approach it is believed that we can avoid information overload and manage to better support software maintainers' software understanding by providing a more flexible reverse engineering environment. The approach is called UDaRE that stands for a **U**ser-**D**efined **a**pproach for **R**everse **E**ngineering Tool. We provide an example of how the approach may support software understanding.*

KEYWORDS

Software maintenance, software documentation, reverse engineering, software understanding.

1. Introduction

Software engineers or programmers perceive software maintenance as uninteresting and daunting tasks because they normally need to study the programs written by previous programmers prior to changing the source codes. Without documentation or out-dated documents, the process of software understanding can be more cumbersome. According to Sulaiman *et al.* [7] the problem related to the absence of or out-dated documentation occurs in both software development and maintenance process. This depicts that the first released version of newly developed software systems might have confronted with documentation problem.

Despite of the emerging commercialized or prototypes of CASE (Computer-Aided Software Engineering) tools that suppose to assist software engineers particularly in documenting their software design, software systems are still produced without proper documentation. Hence reverse engineering (RE) workbenches or tools have become the alternative to solve the problem by automating source code analysis and represent the analyzed software artifacts into a highler level of abstraction such as using graph representation. Some examples are Rigi [3][5], SNiFF+ [10] and CodeCrawler [2]. The RE technology is also incorporated into CASE tools for analysis and design such as Rational Rose [4]. Our work focuses on the tool that is targeted to assist the understanding of existing software systems. Such RE tools also provide representations of software extracted to better support software understanding and the utility to re-document the extracted artifacts.

Existing tools are mostly dedicated for certain languages and the RE process halts if it does not meet some RE rules set by the tools. This causes the tools to be too rigid and inflexible to meet users' need in supporting software understanding. Besides, such tools are strictly set with predefined properties in generating the representations of extracted artifacts causing the graphical representations to be cluttered with unnecessary information. Thus in this paper we propose a user-defined approach for the RE tool that enables users to indicate the syntaxes of the concerned programming language, the RE rules, types of components to be extracted, the dependencies they want to study or visualize and also the document template required. Then the information will be updated in a database to be re-used or edited in the future in case the users need to study different types of components and their dependencies or to study software systems of different programming languages. Hence by using this approach it is believed that we can avoid information overload and manage to better assist software maintainers' software understanding by providing a more flexible reverse engineering environment. The project is called UDaRE that stands for a **U**ser-**D**efined approach for **R**everse **E**ngineering Tool. In Section 2 we will discuss some related work, followed by the description of UDaRE project in Section 3 and the proposed user-defined approach. Finally we conclude the work and discuss some possible future work in Section 4.

2. Related Work

There are a number of related works that attempt to produce a generic or more flexible RE tools or workbenches. Tadonki [9] presents Universal Report, a generic source code documentation tool or RE tool. The tool applies heuristic and pattern matching algorithms that can indicate standard programming statements for a wide range of programming languages. However this tool is still limited to the most common programming languages pre-defined by the tool developers.

Another example is Moose [1], a language independent reengineering environment for object-oriented software systems. It can be extended in order to allow language plugins for tools that require specific information. Rigi [5] is a RE research prototype tool that provides quite a comprehensive level of software abstraction from program level up to local variables. The parsing components are pre-defined based on the languages supported by the tool such as C and COBOL. With this approach, users can only choose the level of software abstractions after the graphical representations have been produced. This method can cause the graphs become clutter and very difficult to collapse the nodes of the graph. It also can lead to information overload among the users.

SNiFF+ [10] is a commercial reverse engineering tool that provides source code analysis environment with code visualization and navigation tool. Besides it provides graphical views of include files, class hierarchy and cross referencer. SNiFF+ can be integrated with other tools and allows edition and compilation of source codes in its working environment. SNiFF+ is also an extensive tool. Yet it is not flexible to be extended by the users since the tool provides only specific and predefined parsers. Another ubiquitous commercial CASE tool is Rational Rose [4] that is 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 fully benefited if software engineers have designed and developed a software system using the tool and the integrated software development environment. This also promotes the round-trip engineering using Rational Rose. Otherwise, reverse engineering an exiting software system will only produce a high level of abstraction such as a class diagram of UML (Unified Modeling Language) notation.

3. UDaRE Project

Figure 1 depicts the four main modules that should be incorporated into the reverse engineering environment. They are: Components Identifier (CI), Components Analyzer (CA), Components Viewer (CV) and Document Generator (DG). Once software maintainers or users have indicated the syntaxes of programming languages, the RE rules, types of components and dependencies required via the CI module, the information is updated into the repository. Then users need to input the source codes files into the CA module in order to analyze the source codes and output the extracted artifacts into the software repository as required and indicated by the users via the CI module.

From the data in the repository, the CV module should be able to generate the dependencies of components to be viewed by the users while the DG module should generate the documentation of the software artifacts extracted according to the template required by the users. For both CV and DG modules, a part of the functionalities will be based on the previous work of Sulaiman *et al.* [6][8]. The outcome of the research is a tool called DocLike Viewer that employs a document-like and modularized SV method known as DMG to visualize the artifacts and dependencies of a subject system using graph representations. Currently DocLike Viewer depends on the existing parser of Rigi from University Victoria of Canada [5]. We are in the process of integrating DocLike Viewer with the CI and CA modules of UDaRE project.

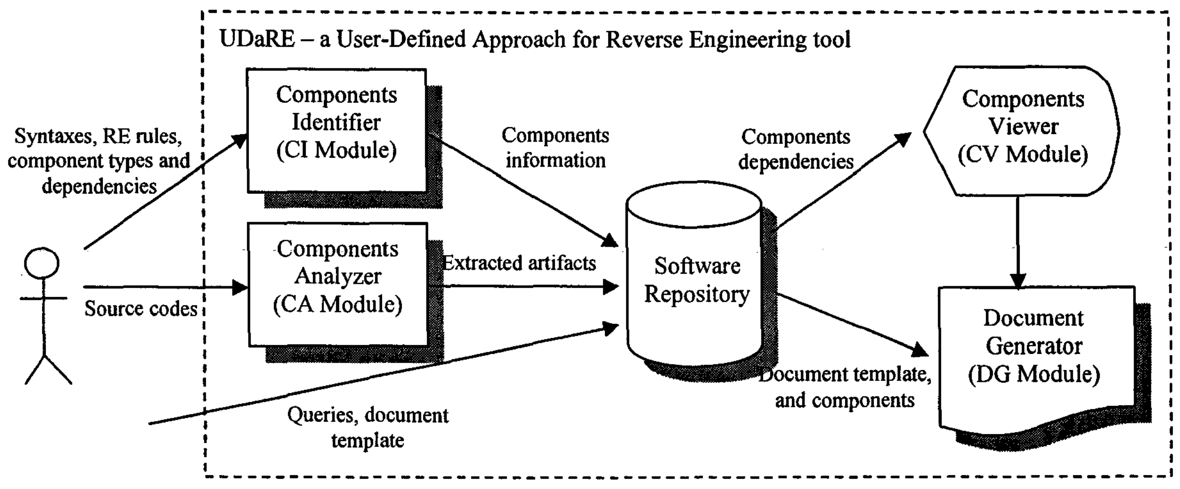

UDaRE – a User-Defined Approach for Reverse Engineering tool
Syntaxes, RE rules, component types and dependencies
Components Identifier (CI Module)
Components information
Components dependencies
Components Viewer (CV Module)
Components Analyzer (CA Module)
Extracted artifacts
Software Repository
Source codes
Document Generator (DG Module)
Document template, and components
Queries, document template

Figure 1: The proposed RE environment and the modules to be incorporated into the tool

3.1 A User-Defined Approach

A user-defined approach is user-centered in which users need to determine the syntaxes of the language they want to deal with before inputting the source codes to be analyzed. The approach requires the following information from the users for the first time of parsing the specified language:

(i) Programming language concerned: Let L is the language inputted.
(ii) Identifier type of the language specified: Let ID is the identifier.
(iii) Syntax to recognize the identifier: Let S is the syntax specified.
(iv) Parsing or RE rules: Let R is the rule specified.

Thus for L language concerned it will consist of a set of ID that corresponds to the specified S. Based on the user-defined set of values for ID and S, the analysis of a program file P can be done. In this project we apply parsing-based technique to analyze the program file or source codes. Each P consists of a set of tokens T. While parsing each line, each T_i value is compared with each S_i value of set S. If they are equal, the corresponding ID_i value and other required details such as modifier and type of concerned component would be retained based on the parsing rules R_i.

For example consider the top segment of source code shown in **Figure 2**. Let L = java, P = classA, ID_i = class, S_i = class. To identify a class, a token value T_i must be equal to S_i. Thus in this case once the syntax value is parsed, the concerned token that is `classA` is the identifier name to be retained under the *class* identifier. In order to identify a method the ID and S value should be determined including the RE rules to extract the concerned artifacts.

```
public class classA
{
    private int data1;
    methodA()
    {
        int data1;
    }
    methodA (String x) {}
    private int methodB() {}
}
```

```
public class classB
{
    int data1;
    int data2;
    classA myClass = new classA();
    public methodA()
    {
        int data2;
        myClass.methodA();
    }
    methodC() {}
    methodC(int x, int y) {}
}
```

Figure 2: Sample of two program files of Java classes

For data, the users can specify whether to consider the variables that hold persistent data only or any variables. For instance in `classA` the data is `data1`. In order to consider an association (see highlighted texts in **Figure 2**) between the two classes, the dot operator may be used as the syntax in Java language. Then the corresponding object instantiation is traced to determine the class associated with it. In this case `myClass` is the instantiation of `classA` hence `methodA` belongs to `classA`. This relationship is retained in the repository as one of class dependencies.

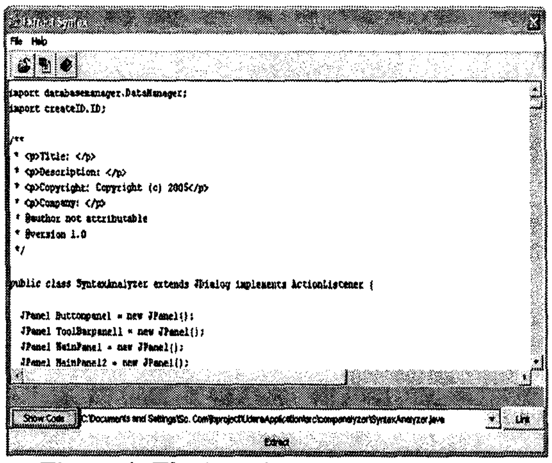
In **Figure 3** we illustrate two different languages, which are C++ and Java. The figure depicts the difference in the syntaxes used for the two languages in implementing inheritance. From this example it is observed that the reserved word `extends` shows inheritance in Java while the syntax double colon ':' after naming the identifier of a class shows inheritance usage in C++. In this case as long as users identify the correct syntaxes and identifiers, the artifacts required for any languages can be extracted accordingly using the user-defined approach.

```
class myClass: public BaseClass {
  public:
     MyClass(int x): BaseClass(x+1)
     { }
};
```

```
class myClass extends BaseClass {
  public MyClass(int x) {
       super(x+1);
  }
}
```

Figure 3: Sample of two program files of C++ (top) and Java (bottom)

The initial prototype for CA module of UDaRE project is shown in **Figure 4**. The users can choose multiple of source code files to be parsed. Furthermore users can also view the parsed source codes.



```
import databasemanager.DataManager;
import createID.ID;

/**
 * <p>Title: </p>
 * <p>Description: </p>
 * <p>Copyright: Copyright (c) 2005</p>
 * <p>Company: </p>
 * @author not attributable
 * @version 1.0
 */

public class SyntaxAnalyzer extends JDialog implements ActionListener {

  JPanel Buttonpanel = new JPanel();
  JPanel ToolBarpanel1 = new JPanel();
  JPanel MainPanel = new JPanel();
  JPanel MainPanel2 = new JPanel();
```

Show Code
C:\Documents and Settings\Sc. Com\project\UdareApplication\src\companalyzer\SyntaxAnalyzer.java
Link
Extract

Figure 4: The interface to analyze and to optionally view the source code

3.2 An Example

Figure 5 shows an example of a Java class that is parsed and some of the extracted artifacts are illustrated in **Figure 6**. The bold texts in **Figure 5** highlight the concerned artifacts to be extracted and to be considered for visualization.

```
package javaWorld;

/* An example of java program */
import javax.swing.*;
import java.awt.*;

public class MyApp extends JFrame {
  String strValue = "";
  Container pane = getContentPane();
  JLabel myLabel1 = new JLabel();

  public MyApp() { //default constructor
    setTitle("My First Application");
    setSize(200, 100);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    pane.setLayout(new GridLayout(1,1));
    pane.add(myLabel1);
    strValue = "Hello!";
    myLabel1.setText(strValue);
    setVisible(true);
  }
  public static void main(String[] args) {
    MyApp myApp = new MyApp();
  }
}
```

Figure 5: An example of source code to be parsed

strFileName:	MyApp.java
strPackageName:	javaWorld
strClassName:	MyApp
strClassModifier:	public
strMethodName:	MyApp
strMethodModifier:	public
strReturnType:	void
strDataName:	strValue
strDataType:	String
strDataName:	myLabel1
strDataType:	Object
strMethodName:	main
strMethodModifier:	public static
strReturnType:	void
strParamName:	args
strParamType:	String[]
strDataName:	myApp
strDataType:	Object

Figure 6: The list of some extracted artifacts

The extracted artifacts in **Figure 6** lists all the concerned artifacts as defined by users prior to parsing the source codes. In this example users consider local variables as the required artifacts. In future, the users can delete the syntaxes to identify local variables in order to simplify the scope of parsing results. The extracted artifacts are retained in a database and will be accessed by CV and DG modules of UDaRE environment (see also **Figure 1**) to generate the graphical view and to re-document the software artifacts extracted. Thus this method is perceived to be able to avoid information overload and generate a simplified, less clutter graphical views of software artifacts to support software understanding.

4. Conclusion

Understanding existing software systems without proper system or design documents is a cumbersome task. Existing RE tools are quite rigid and inflexible because they set predefined rules in the RE process. Thus the process will halt if the rules are violated. Although there is some tools attempt to be more flexible and generic, they are still limited to certain common programming languages. Most tools also do not allow users determine their own level of abstractions prior to RE process. Hence their approaches are not able to simplify the graphical and textual information generated.

Thus we propose a user-defined approach that allows users to set their own syntaxes rules of the concerned programming languages and indicate the level of abstraction required for the graphical representation. This approach is expected to avoid the view to be clutter and also to prevent information overload. We believe a simple view with sufficient information will be able to support software understanding more effectively among software engineers.

The future work may include the enhancement of existing modules of CV and DG to enable them to be integrated with CI and CA modules developed. The whole integrated modules will be further evaluated to determine the effectiveness of the approach proposed in UDaRE project to improve software engineers' software understanding.

5. Acknowledgements

We would like to thank USM to support the UDaRE research project under the USM Short Term Grant (304/PKOMP/636009) and also the student assistant, Shahriza Khairudin who has been implementing the project using Java language.

6. References

[1] Ducasse, S., Lanza, M. and Tichelaar, S., "Moose: An Extensible Language-Independent Environment for Reengineering Object-Oriented Systems", *Proceedings Second International Symposium Constructing Software Engineering Tools (CoSET 2000)*, June 2000.

[2] Lanza, M., "Lessons Learned in Building a Software Visualization Tool", *Proceedings of the 7th European Conference On Software Maintenance and Reengineering (CSMR'03)*, IEEE Computer Society Press, USA, 2003, pp. 1-10.

[3] Muller, H. A., Wong, K. and Tilley, S. R., "Understanding Software Systems Using Reverse Engineering Technology", Proceedings of 62nd Congress of L'Association Canadienne Francaise pour L'Avancementdes des Sciences (ACFAS), in Alagar, V. S. and Missaoui, R. eds., *Object-oriented Technology for Database and Software Systems*, World Scientific, Singapore, 1995, pp. 240-252.

[4] Rational, "Rational Software Corporation", http://www.rational.com/products/, 2005.

[5] Rigi, "Rigi Group Home Page", http://www.rigi.csc.uvic.ca/, 2005.

[6] Sulaiman, S., Idris, N. B. and Sahibuddin, S., "Enhancing Cognitive Aspects of Software Visualization Using DocLike Modularized Graph (DMG)", *International Arab Journal of Information Technology (IAJIT)*, 2(1), 2005, Zarka Private University, Jordan, pp. 1-9.

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

[8] Sulaiman, S., Idris, N. B., Sahibuddin, S. and Sulaiman, S., "Re-documenting, Visualizing and Understanding Software Systems Using DocLike Viewer", *10th Asia Pacific Software Engineering Conference*, IEEE Computer Society Press, USA, 2003, pp. 154-163.

[9] Tadonki, C., "Universal Report: A Generic Reverse Engineering Tool", *Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC'04)*, IEEE Computer Society Press, USA, 2004.

[10] Wind River, "Wind River: IDE: SNiFF+", http://www.windriver.com/products/html/sniff.html, 2005.

IAJIT Vol. 2, No. 1
January 2005

Contents

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).

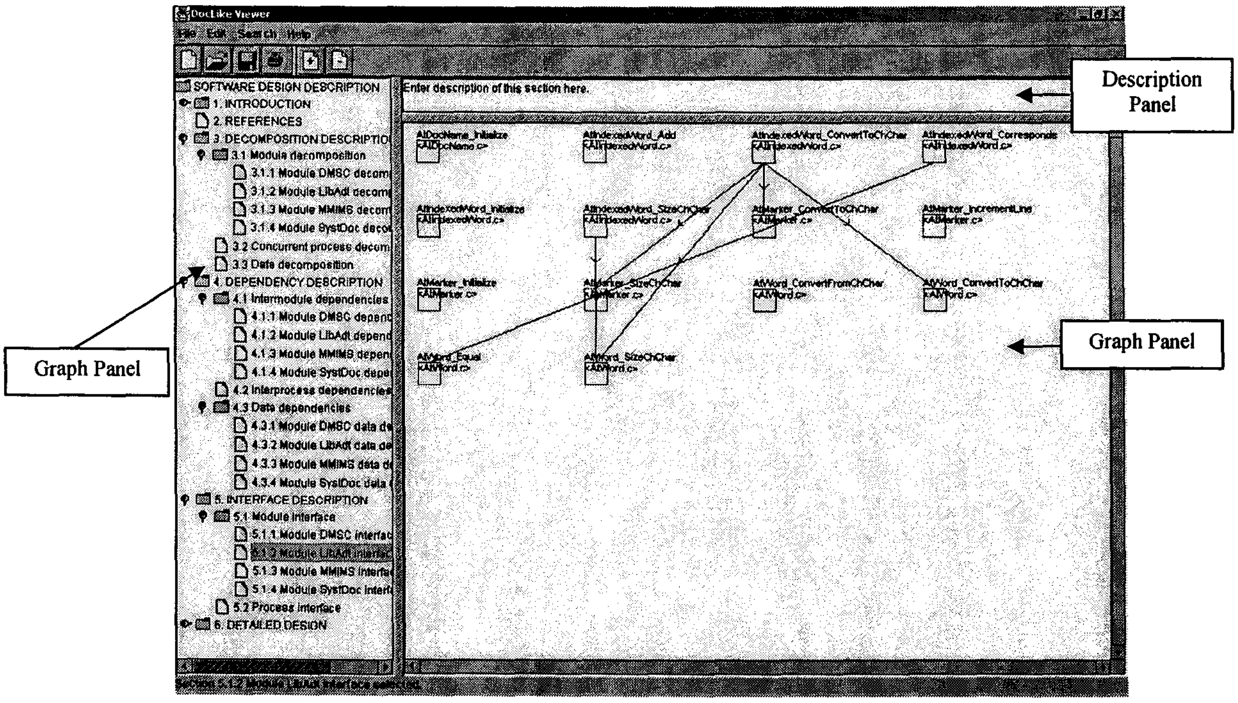

DocLike Viewer
File Edit Search Help
SOFTWARE DESIGN DESCRIPTION
1. INTRODUCTION
2. REFERENCES
3. DECOMPOSITION DESCRIPTIO
3.1 Module decomposition
3.1.1 Module DMSC decom
3.1.2 Module LibAdt decom
3.1.3 Module MMIMS decom
3.1.4 Module SystDoc deco
3.2 Concurrent process decom
3.3 Data decomposition
4. DEPENDENCY DESCRIPTION
4.1 Intermodule dependencies
4.1.1 Module DMSC depend
4.1.2 Module LibAdt depend
4.1.3 Module MMIMS depen
4.1.4 Module SystDoc depe
4.2 Interprocess dependencies
4.3 Data dependencies
4.3.1 Module DMSC data de
4.3.2 Module LibAdt data de
4.3.3 Module MMIMS data d
4.3.4 Module SystDoc data
5. INTERFACE DESCRIPTION
5.1 Module interface
5.1.1 Module DMSC interfac
5.1.2 Module LibAdt interfac
5.1.3 Module MMIMS interfa
5.1.4 Module SystDoc interf
5.2 Process interface
6. DETAILED DESIGN
Enter description of this section here.
AlDocName_Initialize <AlDocName.c>
AlIndexedWord_Add <AlIndexedWord.c>
AlIndexedWord_ConvertToChChar <AlIndexedWord.c>
AlIndexedWord_Corresponds <AlIndexedWord.c>
AlIndexedWord_Initialize <AlIndexedWord.c>
AlIndexedWord_SizeChChar <AlIndexedWord.c>
AlMarker_ConvertToChChar <AlMarker.c>
AlMarker_IncrementLine <AlMarker.c>
AlMarker_Initialize <AlMarker.c>
AlMarker_SizeChChar <AlMarker.c>
AlWord_ConvertFromChChar <AlWord.c>
AlWord_ConvertToChChar <AlWord.c>
AlWord_Equal <AlWord.c>
AlWord_SizeChChar <AlWord.c>
Section 5.1.2 Module LibAdt interface selected.
Description Panel
Graph Panel
Graph Panel

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.

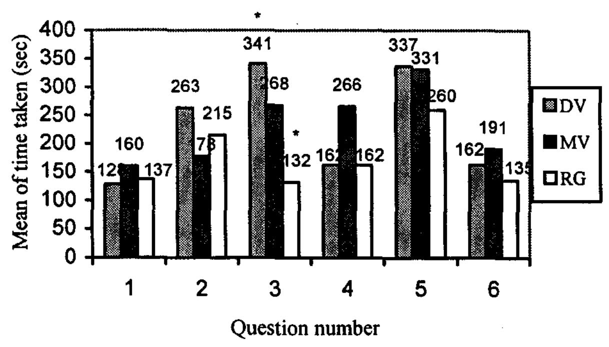


Figure 2. Mean of time taken (regardless of correctness) T_1 in the first experiment. The asterisk (*) shows the significant mean difference.

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).

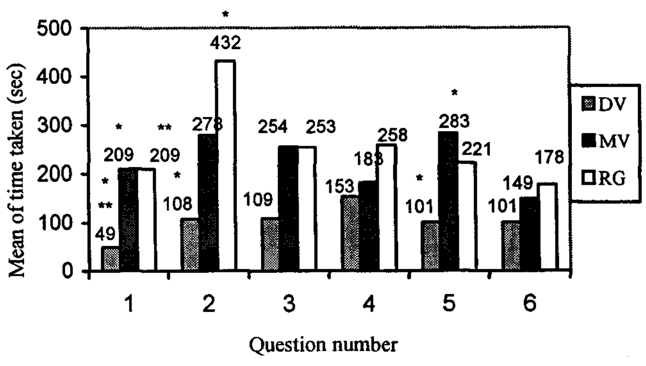


Figure 3. Mean of time taken (regardless of correctness) T_1 in the second experiment. The asterisk (*) shows the significant mean difference.

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 i.e. 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%).

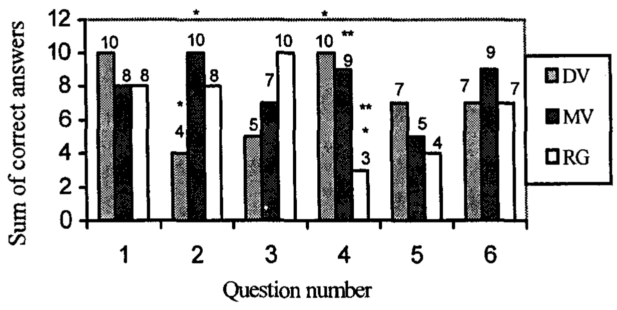


Figure 4. Score S in the first experiment. The asterisk (*) indicates the significant score difference.

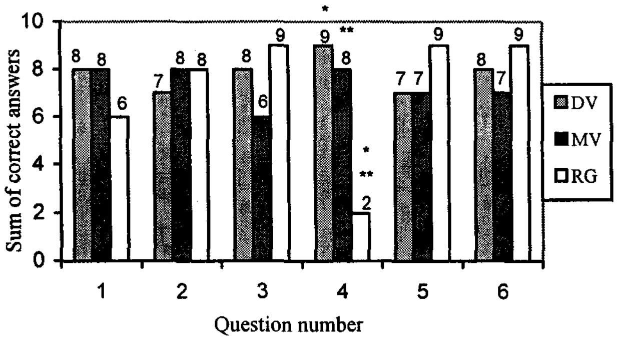


Figure 5. Score S in the second experiment. The asterisk (*) shows the significant score difference.

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.

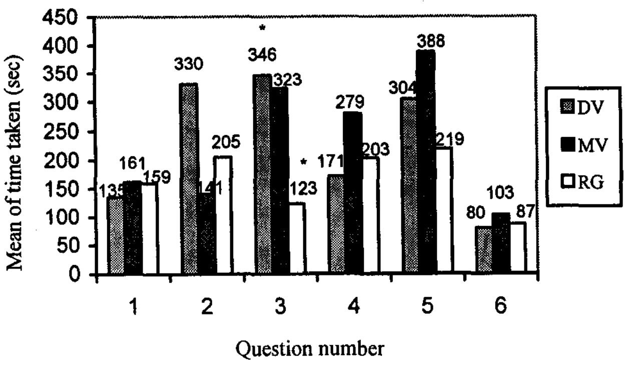


Figure 6. Mean of time taken to answer correctly T_2 in the first experiment. The asterisk (*) indicates the significant mean difference.

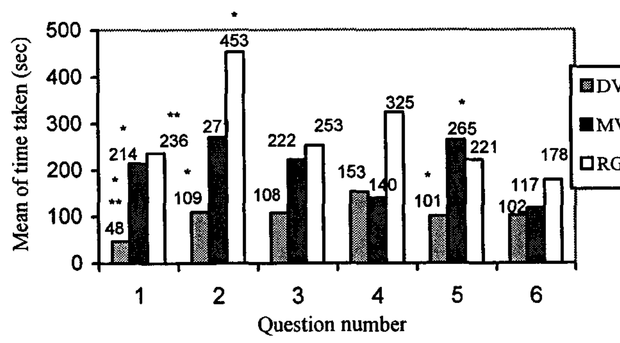


Figure 7. Mean of time taken to answer correctly T_2 in the second experiment. The asterisk (*) indicates the significant mean difference.

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.

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.

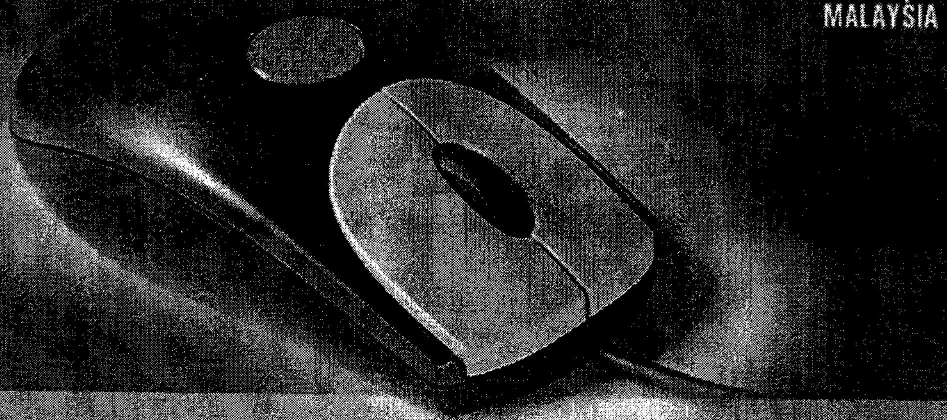
[10] Storey M. A. D., Fracchia F. D., and Muller H. A., "Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration," *Journal of Systems and Software*, vol. 44, pp. 171-185, 1999.

[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*

The 2nd Malaysian MySEC'06 Software Engineering Conference

The 2nd Malaysian MySEC'06 Software Engineering Conference

11TH & 12TH DECEMBER 2006
CROWN PRINCESS Kuala Lumpur
MALAYSIA



Towards Practical and Maintainable Software

Proceedings

Organised by :

Faculty of Computer Science & Information System
Universiti Teknologi Malaysia

Jointly Organised:

CASE Centre For Advanced Software Engineering

Centre for Advanced Software Engineering (CASE)
Universiti Teknologi Malaysia

SPACE UTM
ISO 9001:2000 CERTIFICATE NO.:KLR 0500186

School of Professional and Continuing Education (SPACE)
Universiti Teknologi Malaysia

A Tutor-Based Software Visualization Approach (TubVis) for Novice Software Engineers

Shahida Sulaiman
School of Computer Sciences, Universiti Sains Malaysia, 11800 USM, Penang, Malaysia
shahida@cs.usm.my

Sarina Sulaiman
Faculty of Computer Science & Information System, Universiti Teknologi Malaysia, 81310 Skudai, Johor, Malaysia
sarina@fsksm.utm.my

Abstract

A number of software visualization tools either research prototypes or commercial computer-aided software engineering (CASE) products are available. Besides, existing Integrated Development Environments (IDEs) mostly provide visualization utility to view software artefacts being developed. In order to visualise software artefacts in such tools, reverse engineering is required. Existing tools employ various methods and approaches for software visualization with the main goal to improve program comprehension of written software systems. However, existing methods or approaches are limited to generating the views or component dependencies that is focusing on 'what' the output of reverse engineering process. The online help provided by the tools only indicate 'how' to use the tools to generate the views. Since existing tools mostly target for experienced software engineers, they tend to overlook the need of explaining 'why' the output is recommended or not recommended. Hence a tutor-based software visualization approach (TubVis) is proposed that analyse software artefacts pertaining to software engineering best practices and generate a set of recommendations regarding design and coding for a novice software engineer or a computer science student. The work is anticipated to improve better quality and understanding of software by combining both practical and theoretical aspects of software engineering education in a software visualization tool.

1. Introduction

A lot of software visualization tools either research prototypes such as Rigi [11] or commercial computer-aided software engineering (CASE) products such as Rational Rose [10] are available. Besides, existing Integrated Development Environments (IDEs) for instance Borland JBuilder [2] mostly provide visualization utility to view software artefacts that are being developed. In order to visualise software artefacts in such tools, reverse engineering is required.

Reverse engineering is the process of analysing a subject system to identify the system's components and their interrelationships, and create representations of the system in another form or at a higher level of abstraction [4]. By having a software visualization tool in a reverse engineering environment, extracted software artefacts and their interrelationships can be visualised in a more meaningful way to aid software engineers' program comprehension.

Existing tools employ various methods and approaches [3, 7, 15] for software visualization with the main goal to improve program comprehension of written software systems. However existing methods or approaches are limited to generating the views or component dependencies that is focusing on 'what' the output of reverse engineering process. The online help provided by the tools only indicate 'how' to use the tools to generate the views. Since existing tools mostly target for experienced software engineers, they tend to overlook the need of explaining 'why' the output is recommended or not recommended.

For instance computer science students or novice software engineers need to be guided whether the programs they have written are well designed or not. Existing tools provide the automation of software visualization in practical but they are lack of theoretical aspects. Normally, computer science students learn theories of software engineering during their study. By integrating the theoretical aspects in a tutor-based approach of software visualization, the students will be able to balance practical and theoretical aspects during software development and maintenance. This integration will make the tool more beneficial and vital in giving them theoretical guidance even to novice software engineers. However it is crucial to highlight that experienced or expert software engineers may find this approach relatively useful if they want to ensure their software designs are conformed to software engineering best practices all the time.

Novice software engineers described in this paper refer to both computer science students and software practitioners who develop software systems but do not fully practise software engineering discipline. No specific definition given by existing work because most of them refer their subjects as either computer science students or software engineers. However in our study, novice software engineers do not only refer to computer science students but also practitioners who do not fully adhere to software engineering practices and disciplines while developing software.

Based on the observations and evaluations of reports produced by undergraduates' or even post graduates' software development projects, the problems faced by computer science students or novice software engineers including how to create a good requirement analysis, how to transform the requirement into a proper design using certain modelling notation, how to convert the design into source codes and how to relate the diagrams produced during different stages of software development or maintenance with the source codes. In addition, they can hardly understand the smooth transition among different diagrams generated. Hence the diagrams produced mostly do not correspond with what they code during implementation phase. This problem has been scrutinised by some work focusing on the ability of students or novice designers to understand object-oriented software [5, 6, 12, 13], comprehend programs using animation [8, 9] and understand software for maintenance [1, 14, 18]. Based on the literature study conducted so far, none of existing work attempts to provide a tutor-based approach as the guidance for the novice while developing software system through out the phases.

In the following sections we will discuss the proposed tutor-based visualization approach, the prototype tool, related work and conclusion.

2. Tutor-based visualization approach (TubVis)

We propose a tutor-based approach for software visualization tool (TubVis) to support novice software engineers. The methodology employed in this research includes: (i) Examine the best practices in software engineering specifically for design and coding stage, (ii) archive the output of (i) into a database that will be the rules to be checked, (iii) develop a software visualization tool that will employ the tutor-based approach (TubVis), (iv) integrate TubVis tool with the existing UDaRE environment and (v) evaluate the tool on the subjects of computer science students who have studied software engineering. The evaluation will apply the empirical study used by Shull [12]. For the scope of this paper we will discuss the proposed TubVis approach and an example of how the prototype works.

The variables involved include software engineers or computer science students who develop a software system, the software system to be reverse engineered, the extracted software artefacts, rules of best practices in software engineering, recommendations produced and program comprehension. Attributes of the variables include the novice's level of expertise, the size of software systems, the quality of software developed. The main research question is: *How to produce a software visualization tool that can provide both practical and theoretical guidance while designing and coding software systems?* The null hypothesis to be rejected is H_0: *A tutor-based approach for software visualization tool does not significantly improve the quality of software written by novice software engineers.* The proposed TubVis will be integrated with existing reverse engineering environment (UDaRE) that is being developed by the researchers (refer Figure 1). UDaRE is currently under the process of finalising the implementation and integration with the enhancement of researchers' existing method of software visualization [16, 17].

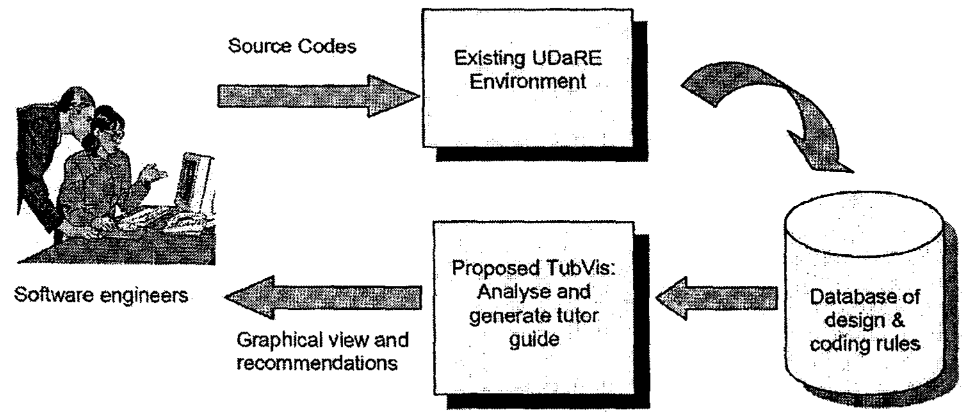

Source Codes
Existing UDaRE Environment
Software engineers
Graphical view and recommendations
Proposed TubVis: Analyse and generate tutor guide
Database of design & coding rules

Figure 1: The proposed tutor-based approach for software visualization in UDaRE environment

UDaRE allows a software system to be inputted and then software artefacts will be extracted. Based on the extracted artefacts, the proposed TubVis will analyse software dependencies and state some recommendations. Then a graphical view using graphical notation will be generated together with the recommendations. The tool will indicate 'why' the design is not recommended. For instance a circular class diagram is not a good design. A good design should follow high cohesion and low coupling principle. Thus TubVis will be able to detect any un-recommended features to be highlighted to the users. The tutor-based approach will indicate best practices in software analysis and design hence it indirectly promotes better coding of software systems. The focus of this research limited to providing guidance or tutoring to novice software engineers involving the transformation of views in design stage to source codes and vice versa. Other stages of software development or maintenance will be considered in the future work.

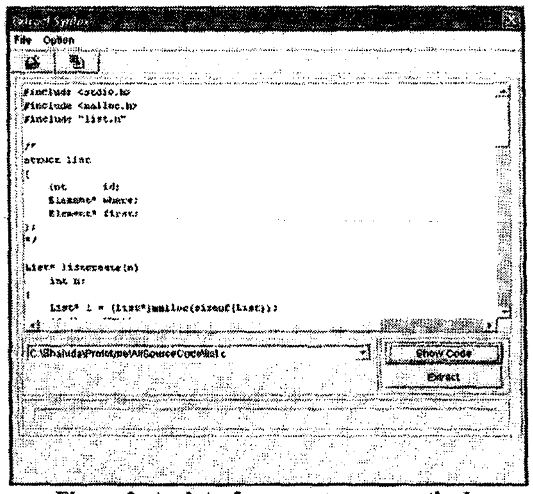
Let S is the source code parsed by UDaRE that output a set of X artefacts. X consists of x_1 ... x_n that can be packages, classes, methods or data. Let R is a set of rules archived by software engineering experts for specific stages of software development. Upon users' selections on type of checking required, TubVis will make an analysis of the relevant artefacts of set X in order to generate a tutor guide or experts' recommendations. Finally the graphical view generated is accompanied with a set of recommendations or guidance G. Software engineers need to decide whether to change their design as required or proceed. If they proceed, the "defects" will be further accumulated in the next stage of checking in order to highlight their design or coding deficiencies.

3. TubVis prototype tool

In this section we give an example of how TubVis will generate tutor guide to a particular source code and design after UDaRE has parsed and generate the view via a visualization tool.

In order to start checking the design, software engineers or users need to feed in the source codes into UDaRE parser such as in Figure 2. This is C language source code derived from Rigi [11] web site.

Once the source codes have been parsed, the artifacts will be visualized as in Figure 3. In this example the 3 modules are interconnected with each other. Based on the rules, circular relationship occurs when A calls B, B calls C and C calls A. Circular design is not a good practice in software engineering discipline. Looking at the navigability among the modules, it is observed that *MAIN* calls *LIST*, *LIST* calls *ELT* and *MAIN* calls *ELT*. Hence this is an accepted design.



```
#include <stdio.h>
#include <malloc.h>
#include "list.h"

/*
struct list
{
    int      id;
    Element* where;
    Element* first;
};
*/

List* listcreate(n)
    int n;
{
    List* l = (List*)malloc(sizeof(List));
```

C:\Shahida\Prototype\AllSourceCode\list.c

Figure 2: An interface to extract a particular source code in UDaRE

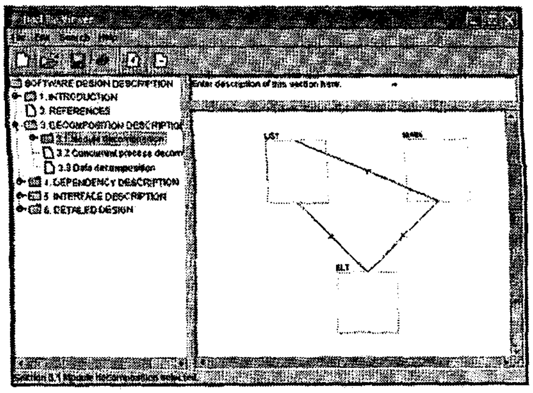
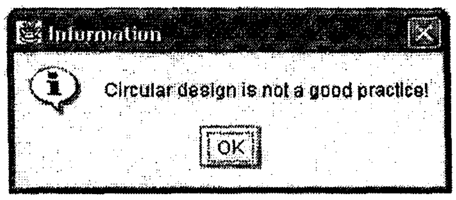

File Edit Search Help
SOFTWARE DESIGN DESCRIPTION
1. INTRODUCTION
2. REFERENCES
3. DECOMPOSITION DESCRIPTION
3.1 Module decomposition
3.2 Concurrent process decom
3.3 Data decomposition
4. DEPENDENCY DESCRIPTION
5. INTERFACE DESCRIPTION
6. DETAILED DESIGN
Enter description of this section here.
LIST
MAIN
ELT
Section 3.1 Module decomposition selected.

Figure 3: Software artifacts parsed are visualized

After all the rules set by software engineering experts in the repository have been checked, TubVis tool will be able to prompt message of recommendation or guidance based on the current design viewed (see Figure 4). Hence users need to edit their source codes in order to ensure no more circular design produced when they generate the graphical view of the changed source codes. In order to give a flexible approach to the users, they may be allowed to proceed with the design. However the TubVis tool may be set to be mandatory by the experts in order to avoid novice software engineers to ignore the recommendations and correct their design. Using this approach, users are forced to rectify their design before they proceed to the next stage. Hence this will ensure software engineering discipline is practiced at the very beginning of software development or maintenance.


Information

Circular design is not a good practice!

OK

Figure 4: A message box displays the relevant recommendation or guidance

4. Related work

As we discussed earlier in the introduction, commercial CASE tools and IDEs provide a comprehensive environment for software engineers or computer science students to draw diagrams during analysis and design stage and then transform the design into corresponding source codes. Rational Rose [10] is a ubiquitous commercial tool that provides both forward and reverse engineering so-called roundtrip engineering. However the tool is very expensive causing the academic institutions or small software departments opt to employ open source tools available or do not even use such tools at all. In this case novice software engineers will use drawing tool to design their software and then transform manually into source codes using any available IDEs for the software languages they use. For instance they might draw a class diagram using Microsoft Word and then implement the coding using Borland JBuilder to develop a Java-based software system.

In addition, commercial tools like Rational Rose [10] are quite complicated to be used by novice software engineers particularly students. Such tools are more appropriate to be used by experienced or expert software engineers if they are fully equipped with best practices in software engineering in particular software design and coding. Users of the tools will be able to generate diagrams such as use case diagram for analysis and then create the sequence diagram during the design stage. The tool allows generation of source codes' skeleton from the sequence diagram. However the tool does not check whether the diagrams in both analysis and design stage are correct or conform to software engineering discipline or best practices. They do not provide direct guidance or tutoring tool to suggest to novice software engineers or computer science students how to analyze, design and write source codes conform to the best practices in order to produce a high quality software.

Hence this has motivated us to integrate tutor-based approach to visualise software systems in reverse engineering environment and then the extracted software artefacts will be analysed by comparing them with the analysis, design and coding rules. A set of recommendations will be generated once the rules have been checked. For instance, the rule of cohesion and coupling of classes can be archived in the database and then the rule is checked when analysing extracted artefacts consist of a number of interrelated classes.

Referring to the main research question indicated earlier: *How to produce a software visualization tool that can provide both practical and theoretical guidance while designing and coding software systems?* The main issues to be considered include what aspects to be considered in order to provide a software visualization tool that not only generate graphical views and provide the on-line help or user manual on how to use the tool but it should also provide a theoretical guidance to software engineers particularly novice such as students. This problem partly has been pondered by some research focusing on the ability of students to understand object-oriented software [6, 12, 13], comprehend programs using animation [8, 9] and understand software for maintenance [1, 14, 18].

Jimenez-Diaz *et al.* [6] proposed the use of virtual role-play by computer science students in a virtual 3D environment. Role-play is claimed to be an active learning approach in which the tool indicates the behaviour performed by the role-play including name, goal, actors and description. During the simulation, the students can observe the participating objects, their run-time classes, active objects, and their scope, control flow and method calls. This approach targets to assist understanding of object-oriented software system but it does not provide any recommendation or guidance whether the software written corresponds to the design and analyse the correctness of the design.

Shull *et al.* [12] reported how an object-oriented framework employing example-based techniques are more suitable to beginners compared to hierarchy-based techniques, which the learning curve is quite huge. The work indicated that example-based technique that guide students to explore an example by understanding a particular object in the source code is more effective for beginners of the framework. On the other hand, hierarchy-based techniques, which guide students starting to understand high level or broad classes of functionality towards a deeper level of classes and specific instantiation, are not suitable for beginners to understand software artefacts in an object-oriented framework. This study shows the best reading technique to be employed by beginners, it does not suggest any tutoring element to improve the novice's understanding in designing and coding object-oriented software.

The work of McWhirter [8] suggested [illegible] an animation system to assist students to understand the behaviour of programs. For instance in order to [illegible] students' understanding regarding a breadth first search algorithm, the tool called Algorithm[illegible] produces a graph-based animation that allows [illegible] to input values to the program and observe the changes via the animated graph. Another work of Ohki and Hosaka [9] promoted PAVI (Program Action Visualization Interpreter) that interprets and visualises program behaviour. The tool represents variables, arrays, and pointers as three-dimensional objects. Then it animates the actions of an object when there is an assignment operation. Both AlgorithmExplorer [8] and PAVI [9] focus on the understanding of source codes via animation without any checking element for the design and coding best practices.

More related work like that of Lowry [19] has proposed a computer-based tutorial resource to support students to understand complex commercial CASE tools in order to demonstrate software engineering concept. Wood and Danielson [20] suggest a web-based tutorial resource for introductory logic design course that enable students to learn the available topics and then discuss and review text interactively. Hacker and Sitte [21] developed WinLogicLab teaching suite that not only provide materials of digital logic design course but also allow students to interactively produce the design. However such work target more on educational aspects which differ from our proposed work that integrates both theoretical and practical aspects of software engineering into a CASE tool using both reverse engineering and software visualisation method in order to guide novice software engineers or computer science students.

Based on the literature study conducted so far, none of existing work has proposed a tutor-based approach as the guidance for the novice while developing software system using a CASE tool.

5. Conclusion and future work

We have proposed a tutor-based software visualization approach that is integrated with a reverse engineering environment. The software artifacts are extracted and visualized together with the recommendation of how good the artifacts produced compared with the rules or best practices archived by software engineering experts. The approach focuses on novice or beginners in software engineering including computer science students who need both theoretical and practical guidance while developing software system. Hence the integration of the proposed approach in CASE tool is anticipated to be able to improve quality of software process. Hence the end software product will probably have better quality too.

Our future work will be to study more extensively design metrics to be covered by TubVis and also to further expand the proposed approach to other stages such as analysis and testing. Current work focuses on design and coding stage only. Then we will further evaluate the significance of the proposed approach.

6. Acknowledgements

We would like to thank USM to support this study as part of UDaRE research project under the USM Short Term Grant (304/PKOMP/636009).

7. References

[1] Austin, M. A., III, and Samadzadeh, M. H. (2005). Software Comprehension/Maintenance: an Introductory Course. *18th International Conference on Systems Engineering, ICSEng 2005*. 414-419.

[2] Borland (2001). *Borland JBuilder 5 Enterprise 5.0.294.0.* Borland Software Corporation (1996-2001).

[3] Buchsbaum, A., Chen, Y-F., Huang, H., Koutsofios, E., Mocenigo, J., Rogers, A., Jankowsky, M. and Mancoridis, S. (2001). Visualizing and Analyzing Software Infrastructures. *IEEE Software*, September/October 2001. 62-70.

[4] Chikofsky, E. J. and Cross II, J. H. (1990). Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*. January 1990: 13-17.

[5] Din, J. Ali, N. M., Mohd Noah, S. A. and Idris, S. A Conceptual Framework of Object-Oriented Design for Novice Designers. *The First Malaysian Software Engineering Conference (MySEC'05)*. USM, Malaysia. 83-88.

[6] Jimenez-Diaz, G., Gomez-Albarran, M. Gomez-Martin, M. A. and Gonzalez-Calero, P. A. Understanding Object-Oriented Software through Virtual Role-Play. *IEEE International Conference on Advanced Learning Technologies, ICALT 2005*. 875-877.

[7] Lanza, M. (2003). Lessons Learned in Building a Software Visualization Tool. *Proceedings of the 7th European Conference On Software Maintenance and Reengineering (CSMR'03)*. 1-10.

[8] McWhirter, J. D. (1996). AlgorithmExplorer: a Student Centered Algorithm Animation System. *IEEE Symposium on Visual Languages*. 174-181.

[9] Ohki, M. and Hosaka, Y. (2003). •A Program Visualization Tool for Program Comprehension. *IEEE Symposium on Human Centric Computing Languages and Environments*. 263-265.

[10] Rational (2006). *Rational Software Corporation.* http://www.rational.com/products/

[11] Rigi (2006). *Rigi Group Home Page.* http://www.rigi.csc.uvic.ca/

[12] Shull, F., Lanubile, F. and Basili, V. R. (2000) Investigating Reading Techniques for Object-Oriented Framework Learning. *IEEE Transactions on Software Engineering*. 26(11) Nov. 2000. 1101-1118.

[13] Soga, M., Kashihari, A. and Toyoda, J. –I. (1996). Designing a Self-Explanation Environment for Multilayer Understanding. In Case of Program Understanding. *IEEE International Conference on Multi Media Engineering Education*. 49-57.

[14] Slimick, J. (1997). An Undergraduate Course in Software Maintenance and Enhancement. *Tenth Conference on Software Engineering Education & Training*. 61-73.

[15] Storey, M. –A. D., Fracchia, F. D. and Muller, H. A. (1999). Cognitive Design Elements to Support the Construction of a Mental Model during Software Exploration. *The Journal of Systems and Software*.

[16] Sulaiman, S., Idris, N. B. and Sahibuddin, S. (2005). Enhancing Cognitive Aspects of Software Visualization Using DocLike Modularized Graph (DMG). *International Arab Journal of Information Technology (IAJIT)*. 2(1). Zarka Private University, Jordan. 1-9.

[17] Sulaiman, S., Idris, N. B., Sahibuddin, S. and Sulaiman, S. (2003). Re-documenting, Visualizing and Understanding Software Systems Using DocLike Viewer. *10th Asia Pacific Software Engineering Conference*, IEEE Computer Society Press, USA. 154-163.

[18] van Deursem, A., Favre, J. –M. Koschke, R. and Rilling, J. (2003). Experiences in Teaching Software Evolution and Program Comprehension. *11th IEEE International Workshop on Program Comprehension*. 283-284.

[19] Lowry, G. R. (1996). CAL Support for Complex CASE Tutorials. Demonstrating Software Engineering Concepts through CASE. *International Conference on Software Engineering: Education and Practice*. IEEE Computer Society Press, USA. 292-299.

[20] Wood, S. and Danielson, R. (2000). Java-Based Instructional Materials for Introductory Logic Design Courses. *30th Annual Frontiers in Education Conference 2000*. S2D/10-S2D/16 vol.2.

[21] Hacker, C. and Sitte, R. (2004). Interactive Teaching of Elementary Digital Logic Design with WinLogiLab. *IEEE Transactions on Education*. Vol. 47, Issue 2, May 2004. 196-203.

Lampiran C:
Dokumentasi Sistem (dalam Bahasa Inggeris)

A User-Defined Approach for Reverse Engineering (UDARE) Tool Project

System Document

USM Short Term Grant: 304/PKOMP/636009

15 March 2005 – 14 June 2007

Compiled By:

UDARE Project Team

Dr Shahida Sulaiman (Client/Project Leader)

Shahriza Khairuddin (Developer)

Md Yamin Md Yusoff (Developer)

Software Engineering Research Group (SERG)
School of Computer Sciences
Universiti Sains Malaysia
11800 USM
Penang, Malaysia

20/07/2007

Document Revision History

Revision	Date	Updated by	Description

Table of Contents

List of Figures

List of Tables

1. Software Project Management Plan (SPMP)

1.1. System Overview

1.1.1. System Description and Function

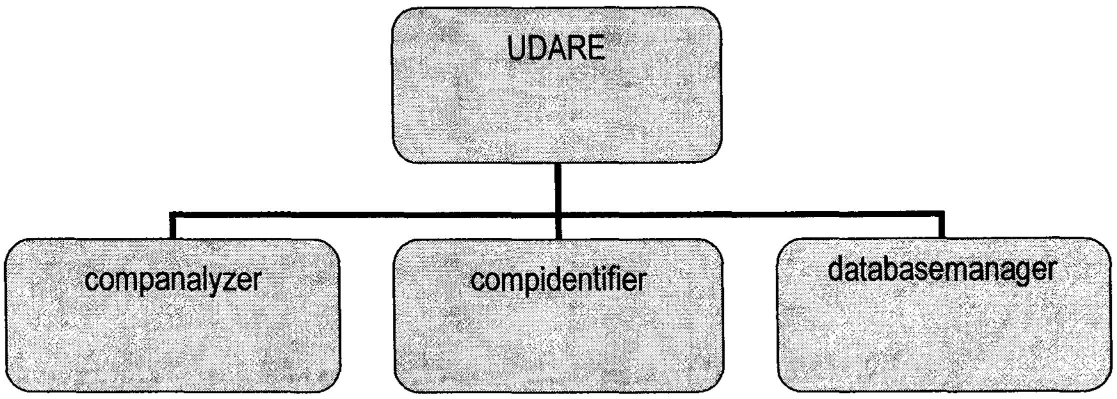
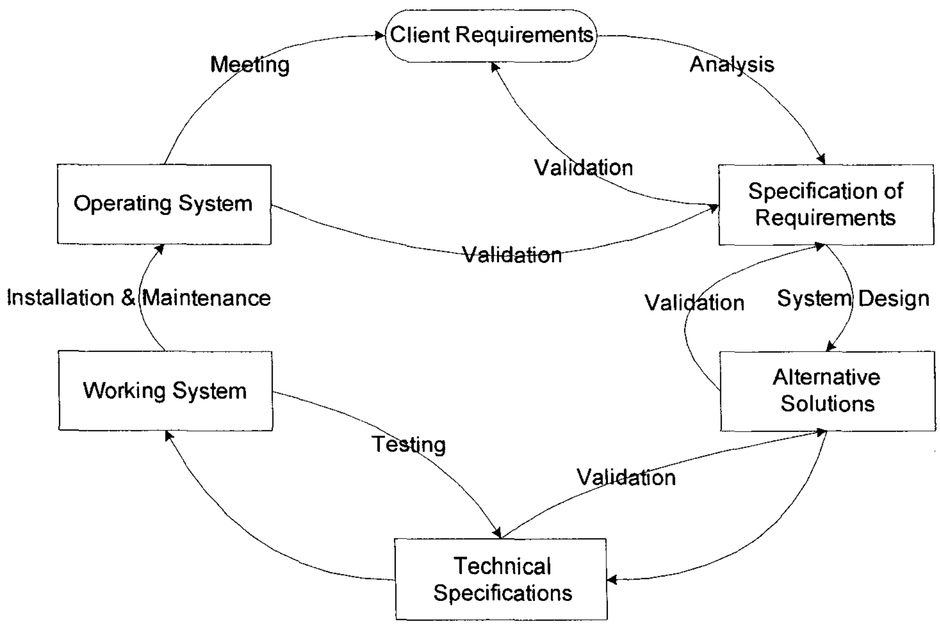


Figure 1.1: Complete System Module Breakdown

1.1.2. Development Methodology

We will be using OOAD as our development methodology for this project. Moreover, the existing system that will be integrated with our system used the same methodology.

1.1.3. Software Development Lifecycle (SDLC)



1.1.4. Modeling Notation

The project uses UML as the modeling notation, with emphasis on Use Cases, Class Diagram and Sequence Diagram.

1.1.5. Coding Standard

This project uses JAVA2 as the programming language and Borland JBuilder 2006 as the development environment.

1.2. Team Structure and Roles

This project is led by Dr. Shahida Sulaiman with support by Md Yamin bin Md Yusoff and Shahriza. Both Md Yamin and Shahriza will carry task as developer and each assigned with different module to work on.

1.2.1. Role Assignments

Each team member is a Developer. In addition, the following roles are assigned to the respective team members.

Table 1.1: Project Team Role Assignments

Role	**Team Member**
Project Leader	Dr Shahida Sulaiman
Quality Assurance	Dr Shahida Sulaiman
Developer	Md Yamin bin Md Yusoff (1 Aug. 2005 – 31 July 2006)
Developer	Shahriza bin Khairudin (1 Sept. 2006 – 29 Feb. 2007)

1.2.2. Development Responsibilities

The following team members have been assigned to the given Modules for the project.

Table 1.2: Module Development Responsibilities

Module	**Team Member**
udareapplication	Md Yamin bin Md Yusoff and Shahriza bin Khairudin
companalyzer	Md Yamin bin Md Yusoff and Shahriza bin Khairudin
databasemanager	Md Yamin bin Md Yusoff and Shahriza bin Khairudin
compidentifier	Md Yamin bin Md Yusoff and Shahriza bin Khairudin

1.3. Facilities and Computer Resources

1.3.1. Workspace Requirements and Allocation

This project was developed at the Artificial Intelligence Lab (410), Level 4, School of Computer Sciences, Universiti Sains Malaysia.

1.3.2. Computer and other Hardware Resources

This project will be developed using two computers, each of which is belongs to respective group members. Both computers run on Pentium 4 3.0 GHz, with the speed of RAM of 512Mb. Projected hard disk space are up to 10Gb for this project.

1.3.3. Software and Operating System Resource Specifications

This project runs under Microsoft Windows XP Professional Edition Service Pack 2 environment. Borland JBuilder 2006 is chosen as the software development tools. The edition of Java the we will be using is Java Standard Edition (J2SE).

1.4. Risk Management

1.4.1. Areas of Risk

Table 1.3: Areas of Risk

Area of Risk	**Constituents**
Resource	Software are too expensive to acquire
Client	Client keeps changing requirements too often
Communication	Meeting with client might difficult to arrange
Technical	The skills available might not be sufficient to develop the system
Security	Computer might vulnerable to theft attack

1.4.2. Monitoring Procedure and Contingency Plan

Table 1.4: Monitoring Procedure and Contingency Plan for Risks

Risk	**Priority** (1=high risk, 2=medium risk, 3=low risk)	**Monitoring Procedure**	**Contingency Plan**
Developer communication	2	Meeting and discuss	Conduct a meeting regularly depending on needs
Inexperience developer	2	Training	Schedule time for training purpose
System resist by end users	1	Make interview and discussion session before develop the system	Have a prototype review with end user
Hardware failure	1	Backup	Make a backup regularly. Provide a server

1.5. Reviews

1.5.1. Formal Reviews

Not applicable

1.5.2. Informal Reviews

Informal reviews were conducted between the Project Team and the Client.

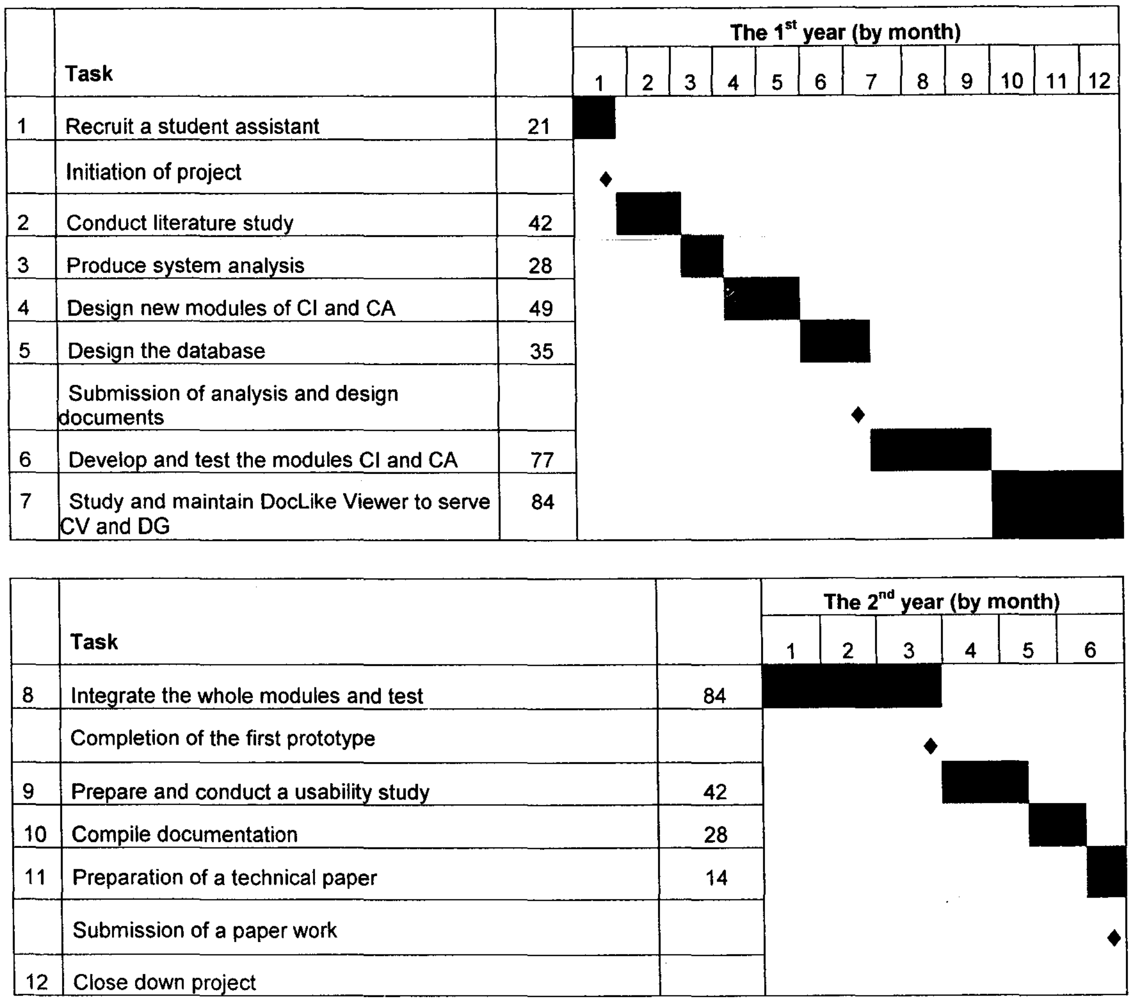
1.5.3. Review Progress

To be defined.

Table 1.5: Review Progress

Review Type	**Date**	**Reviewed Components**	**Remarks**
Requirements and Project Schedule			
Design and Progress Review			
Prototype Review			
Client Update			
Client/Management Demo	▨	▨	▨

1.6. Project Schedule and Milestones



	Task		The 1st year (by month)											
			1	2	3	4	5	6	7	8	9	10	11	12
1	Recruit a student assistant	21	■											
	Initiation of project		♦											
2	Conduct literature study	42		■	■									
3	Produce system analysis	28			■									
4	Design new modules of CI and CA	49				■	■							
5	Design the database	35						■	■					
	Submission of analysis and design documents								♦					
6	Develop and test the modules CI and CA	77							■	■	■			
7	Study and maintain DocLike Viewer to serve CV and DG	84										■	■	■

	Task		The 2nd year (by month)					
			1	2	3	4	5	6
8	Integrate the whole modules and test	84	■	■	■			
	Completion of the first prototype				♦			
9	Prepare and conduct a usability study	42				■	■	
10	Compile documentation	28					■	■
11	Preparation of a technical paper	14						■
	Submission of a paper work							♦
12	Close down project							

Figure 1.2: GANTT Chart shows the schedule for UDARE project

Table 1.6: Task Assignments

The following team members are responsible for the following Tasks in Phase I:

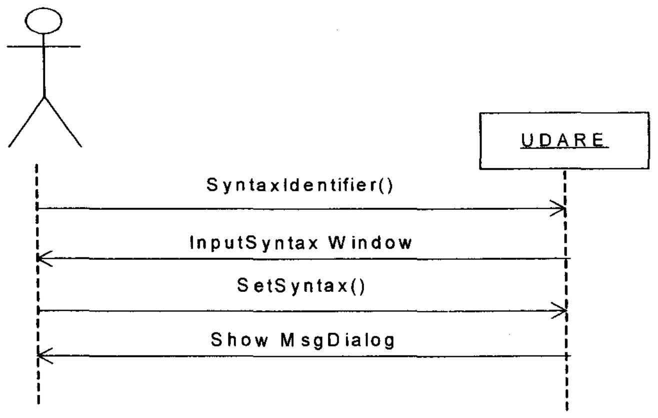
Task ID #	Responsibility	Remarks
1	Project leader	
2	All	Project leader consolidates research papers
3	All	
4	Project leader	
5	All	Project leader initiates and then refined by developers
6	Developers	Project leader leads reviews
7	Developers	
8	Developers	
9	All	Project leader guides developers
10	All	Project leader consolidates all reports
11	Project leader	
12	Project leader	

2.4. *Internal Interfaces Requirements for compidentifier*

2.4.1. Use-Case Input Syntax: SRS-0001

Use-case name : **Input Syntax**
Summary : **User input the syntax such as Java or C++ syntax.**
Dependency
Actor : **Software User**
Pre-Condition : **User need to input the syntax for analyzer.**
Description : 1. **Go to the "Tools" at the toolbar, select "Input Syntax".** 2. **User has to fill the form according to what system needs.** 3. **Press "add" if user done with it.** 4. **Include "UpdateDatabase" use-case.**
Alternatives :
Post-condition : **There are syntax that analyzer can use for analyzing the source code.**

2.4.2. System Sequence Diagram: Input Syntax

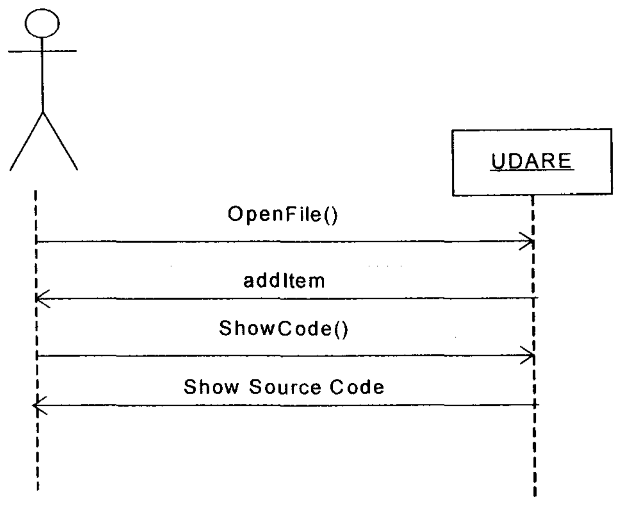


2.5. *Internal Interfaces Requirements for companalyzer*

2.5.1. Use-Case Input Source Code: SRS-0002

Use-case name : **Input Source Code**
Summary : **User input the source code that they want to analyze**
Dependency
Actor : **Software User**
Pre-Condition : **User want to upload source code into the system.**
Description : [SRS-0002-A1] 1. **User press open file button and select the file from local drive** 2. **Press "Link" button and all the file that user selected earlier will appear in combo box.** 3. **If user wants to know details, then include "ExtractArtifacts" use-case.**
Alternatives: **1a) User may select more than one file.** [SRS-0002-A2] 4. **Press "Show Code" button and the source code will appear in Text Area.** [SRS-0002-A3]
Post-condition : **Source code has been appeared in the system.**

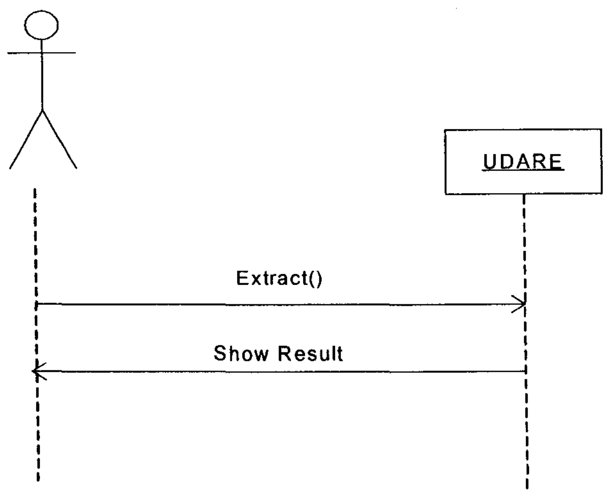
2.5.2. System Sequence Diagram: Input Source Code



2.5.3. Use-Case Extract Artifacts: SRS-0003

Use-case name : **Extract artifacts**
Summary : **User want to know the details about syntax that they added earlier.**
Dependency
Actor : **Software User**
Pre-Condition : **Syntaxes updated.**
Description : **1. Press "Extract" button then the analyzer will produce the details.** **2. If analyzer finished analyzing, then include "UpdateDatabase" use-case.**
Alternatives :
Post-condition : **Details of the extracted artifacts shown.**

2.5.4. System Sequence Diagram: Extract Artifacts



2.6. Internal Interfaces Requirements for databasemanager

2.6.1. Use-Case UpdateDatabase: SRS-0004

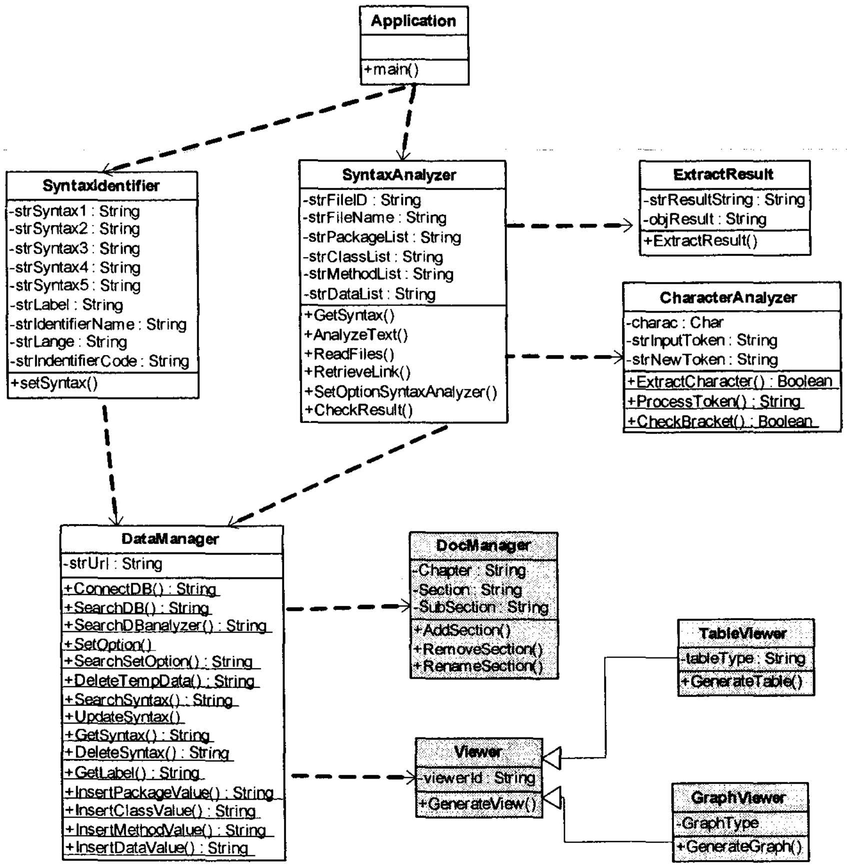
Use-case name : **UpdateDatabase**
Summary : **All the details that been analyze earlier will be store in database.**
Dependency
Actor : **Software (automatically)**
Pre-Condition : **All the extracted artifacts have been produced by the analyzer.**
Description : **1. The artifacts analyzed earlier will be stored in the database automatically.** **2. Then extend "ConnectionDown" use-case.**
Alternatives :
Post-condition : **Data stored in database.**

2.6.2. Use-Case ConnectionDown: SRS-0005

Use-case name : **ConnectionDown**
Summary : **Close the connection to the database**
Dependency
Actor : **Software (automatically)**
Pre-Condition : **Open for connection.**
Description : **1. If all the details stored into the database, connection will be closed automatically.**
Alternatives :
Post-condition : **Close the connection.**

3. Software Design Description (SDD)

3.1. System Architecture



Legend:

▨ SoVis Class (visualization and documenter tool to be integrated)

□ UDARE Class

Figure 3.1: Design Class Diagram for UDARE

3.1.1. System Packages

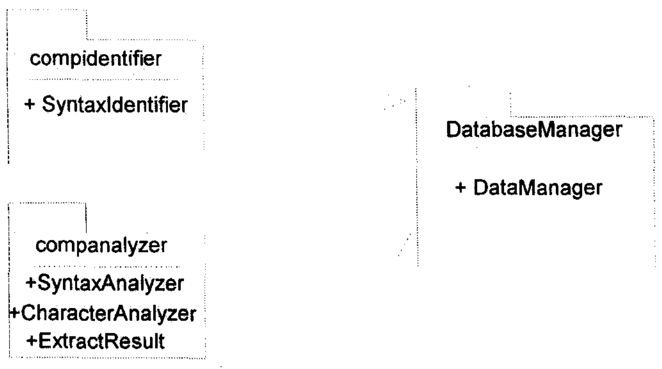
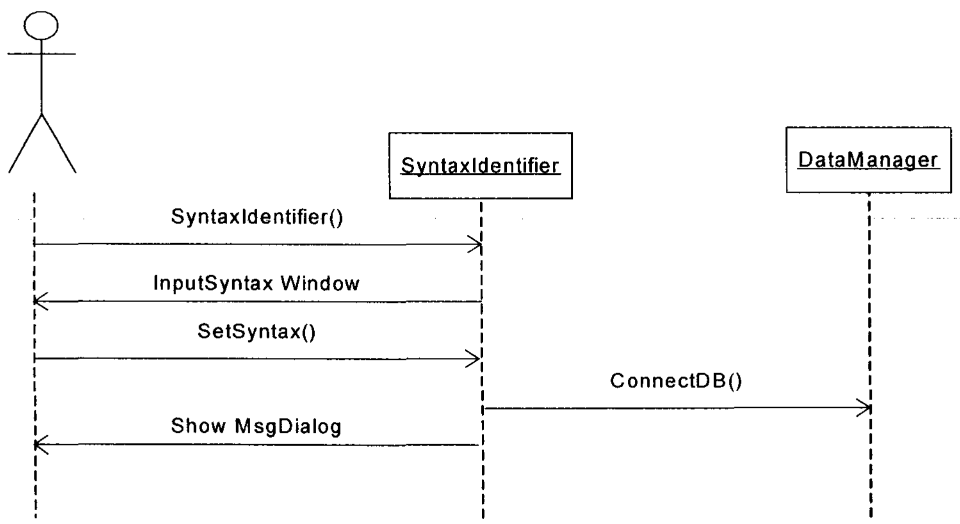


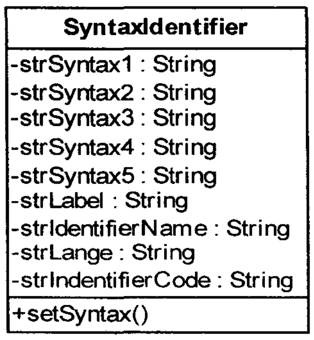
Figure 3.2: Package Diagram

3.3. Detailed Design for compidentifier

3.3.1. Detailed Sequence Diagram: Input Syntax [SRS-0001]



3.3.2. Class Design: SyntaxIdentifier



Stereotype:
Entity

Responsibility:
This class is responsible to manage the syntax input setup by the user.

Attributes:

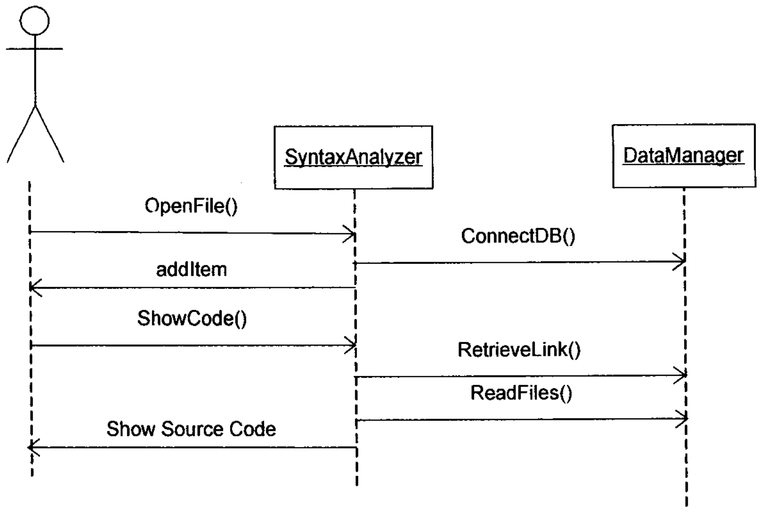
▪ strIdentifierName	- name to identify to artifacts
▪ strLang	- name of the language
▪ strIdentifierCode	- code for the artifact
▪ strLabel	- label name
▪ strSyntax1	- the first syntax
▪ strSyntax2	- the second syntax
▪ strSyntax3	- the third syntax
▪ strSyntax4	- the fourth syntax
▪ strSyntax5	- the fifth syntax

Operations/Methods:

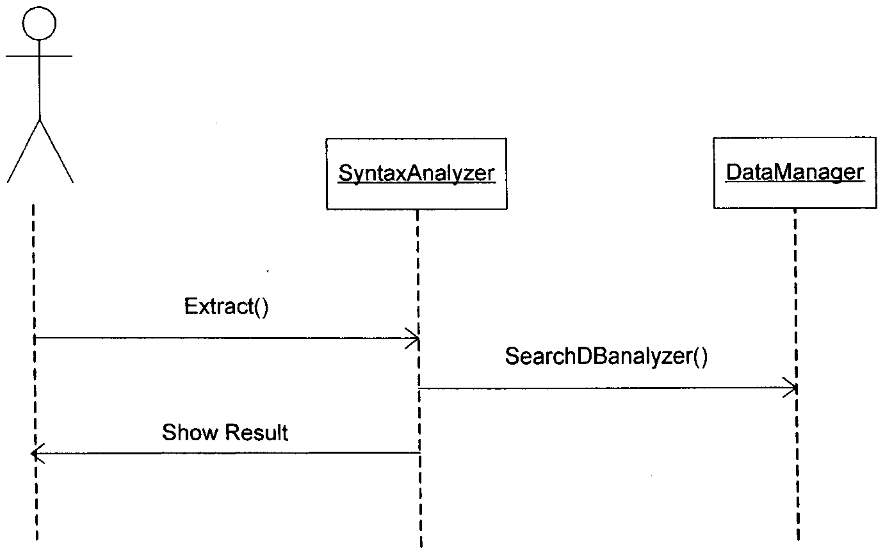
- setSyntax()

3.4. Detailed Design for companalyzer

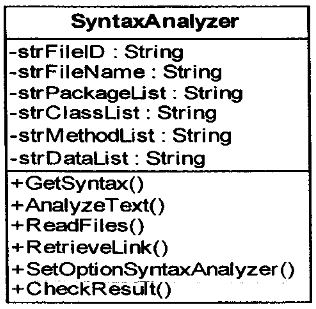
3.4.1. Detailed Sequence Diagram: Input Source Code [SRS-0002]



3.4.2. Detailed Sequence Diagram: Extract Artifacts [SRS-0003]



3.4.3. Class Design: SyntaxAnalyzer



Stereotype:
Entity

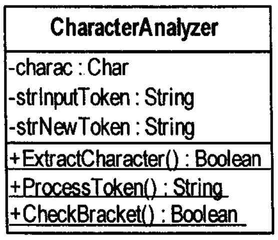
Responsibility:
This class is responsible in extracting artifacts from the source code.

Attributes:

▪ strFileID	- unique Id of the input file
▪ strFileName	- name of the input file
▪ strPackageList	- the package name extracted from the input source code
▪ strClassList	- the class name extracted from the input source code
▪ strMethodList	- the method name extracted from the input source code
▪ strDataList	- the attribute name extracted from the input source code

Operations/Methods:
- GetSyntax()
- AnalyzeText()
- ReadFiles()
- RetrieveLink()
- SetOptionSyntaxAnalyzer()
- CheckResult()

3.4.4. Class Design: CharacterAnalyzer



Stereotype:
Entity

Responsibility:
This class is responsible in processing the extracted artifacts character by character to remove unneeded characters.

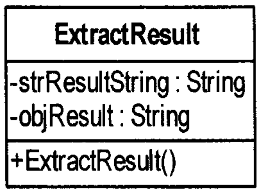
Attributes:

▪ charac	- character of the artifact
▪ strInputToken	- input artifact
▪ strNewToken	- output artifact

Operations/Methods:

- ExtractCharacter()
- ProcessToken()
- CheckBracket()

3.4.5. Class Design: ExtractResult



Stereotype:

Entity

Responsibility:

This class is responsible in to produce output to be presented to the user.

Attributes:

▪ strResultString	- result item
▪ objResult	- list of result item

Operations/Methods:

- ExtractResult()

4. Software Test Documentation (STD)

4.1. Test Cases for compidentifier

4.1.1. Test Case Input Syntax: STD-0001

Requirement Traceability Reference: SRS-0001

Use Case/ Scenario	Test Case	Initialization	Test Input	Expected Result	Test Procedure
SRS-0001	STD-0001		Syntaxes	Syntaxes updated	▪ Input syntax ▪ Click 'add' ▪ View syntax

4.2. Test Cases for companalyzer

4.2.1. Test Case Input Source Code: STD-0002

Requirement Traceability Reference: SRS-0002

Use Case/ Scenario	Test Case	Initialization	Test Input	Expected Result	Test Procedure
SRS-0002-A1	STD-0002-A1	Select a file	Open a file	File is linked	▪ Select a file ▪ Click 'link'
SRS-0002-A2	STD-0002-A2	Select multiple files	Open 2 files	Files are linked	▪ Select multiple files ▪ Click 'link'
SRS-0002-A3	STD-0002-A3	Click 'show code'	-	Source codes viewed	▪ Click 'show code'

4.2.2. Test Case Extract Artifacts: STD-0003

Requirement Traceability Reference: SRS-0003

Use Case/ Scenario	Test Case	Initialization	Test Input	Expected Result	Test Procedure
SRS-0003	STD-0003	Click 'Extract'	File(s) selected	Source codes extracted	▪ Click 'Extract'

4.3. Test Cases for databasemanager

4.3.1. Test Case UpdateDatabase: STD-0004

Requirement Traceability Reference: SRS-0004

Use Case/ Scenario	Test Case	Initialization	Test Input	Expected Result	Test Procedure
SRS-0004	STD-0004	-	File(s) extracted	Data updated	▪ Check database

5. Software Test Report (STR)

5.1. Test Reports

5.1.1. Test Report Input Syntax: STR-0001

Test Case Traceability Reference: STD-0001

Result:

Test Case	Success	Failure/Error	Remark
STD-0001	Yes	None	-

5.1.2. Test Report Input Source Code: STR-0002

Test Case Traceability Reference: STD-0002

Result:

Test Case	Success	Failure/Error	Remark
STD-0002-A1	Yes	None	-
STD-0002-A2	Yes	None	-
STD-0002-A3	Yes	None	-

5.1.3. Test Report Extract Artifacts: STR-0003

Test Case Traceability Reference: STD-0003

Result:

Test Case	Success	Failure/Error	Remark
STD-0003	Yes	None	-

5.1.4. Test Report UpdateDatabase: STR-0004

Test Case Traceability Reference: STD-0004

Result:

Test Case	Success	Failure/Error	Remark
STD-0004	Yes	None	-

5.1.5. Test Report ConnectionDown: STR-0005

Test Case Traceability Reference: STD-0005

Result:

Test Case	Success	Failure/Error	Remark
STD-0005	Yes	None	-

APPENDIX C: SOFTWARE DEVELOPMENT FILES (FINAL VERSION)

File name	Description	Soft/hardcopy location (PC/Room#)	Server	Medium
UDARE-SystemDocument-20070720-V2	System Document	C:\Shahida\UDaRE-ShortTermProject20070504\Documents (NEC VERSA E120 notebook)	-	Softcopy
UDARE SystemDocument	-	Room 627	-	Hardcopy
UdareApplication	JBuilder Project file	C:\Shahida\UDaRE-ShortTermProject20070504\UDARE	-	Softcopy
src	Source codes	C:\Shahida\UDaRE-ShortTermProject20070504\UDARE	-	Softcopy
UDaRE	Microsoft Access file	C:\Shahida\UDaRE-ShortTermProject20070504\UDARE	-	Softcopy

Lampiran D:
Manual Pengguna (dalam Bahasa Inggeris)

1.0 Introduction

1.1 Overview of the Window

Figure 1.1, "Overview of the UDARE window" shows the main UDARE window.

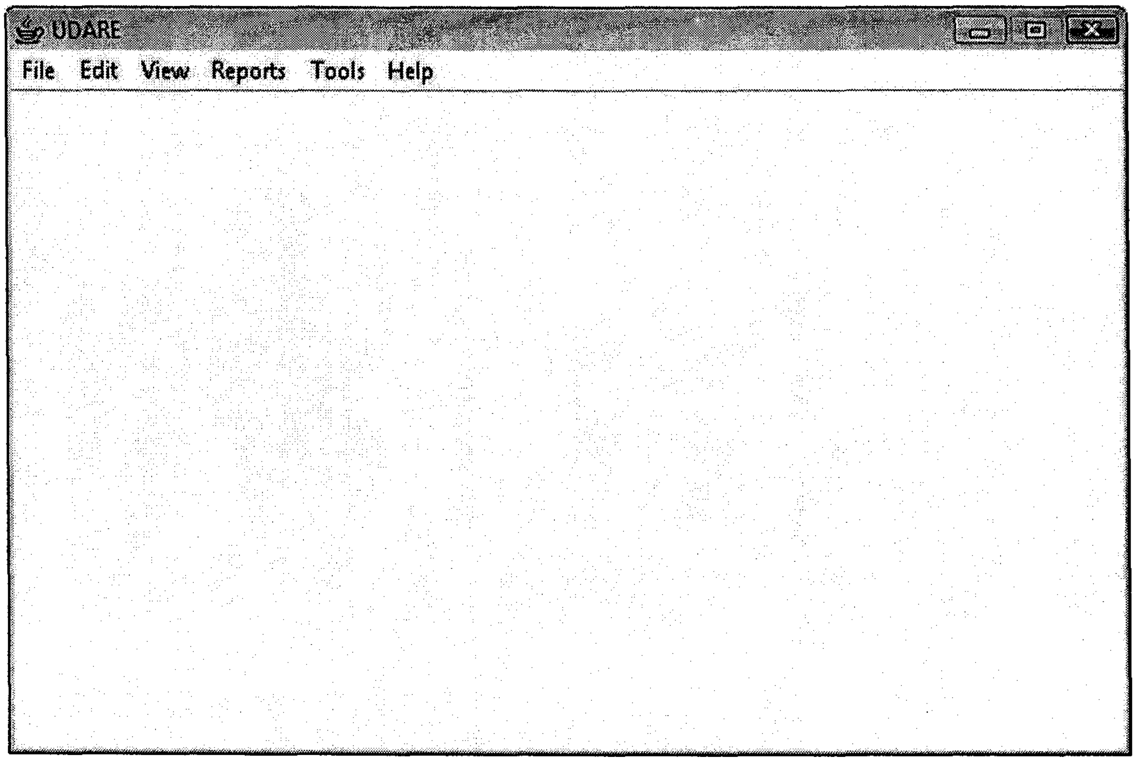


Figure 1.1: Overview of the UDARE window.

At the top of screen is a menu bar, which is described in 2.0, "The Menu Bar".

2.0 The Menu bar

2.1 Introduction

The menu bar in UDARE window allows user to point and click to access window function. Figure 2.1 shows the menu bar consists in UDARE window.

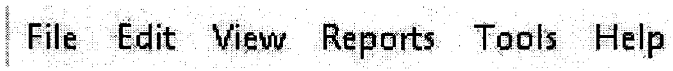


Figure 2.1: Menu bar in UDARE window.

- The **File** menu contains operations relating to the handling of files that affect on the whole project.
- The **Edit, View** and **Reports** menu are not functioning yet.
- The **Tools** menu is for input syntax and analyzing syntax.
- The **Help** menu contains about UDARE.

2.2 The File Menu

These are actions concerned on overall management of a project.

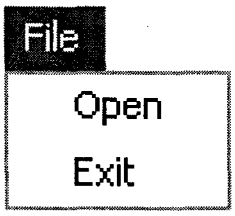


Figure 2.2: Sub-menus in File menu for UDARE window.

2.2.1 Open

This operation opens an existing document from a file.

2.2.2 Exit

This operation closes down UDARE.

2.3 The Tools Menu

This menu is used for input syntax and analyzes syntax.

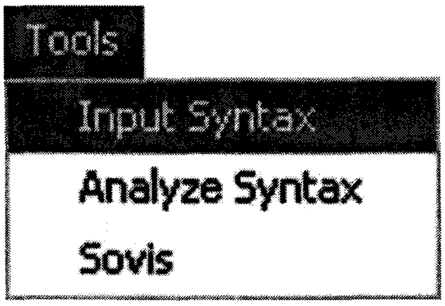


Figure 2.3: Sub-menus in Tools menu for UDARE window.

2.3.1 Input Syntax

This sub-menu provides for input syntax. A pop-up window (Figure 2.4) appears when you click at Input Syntax.

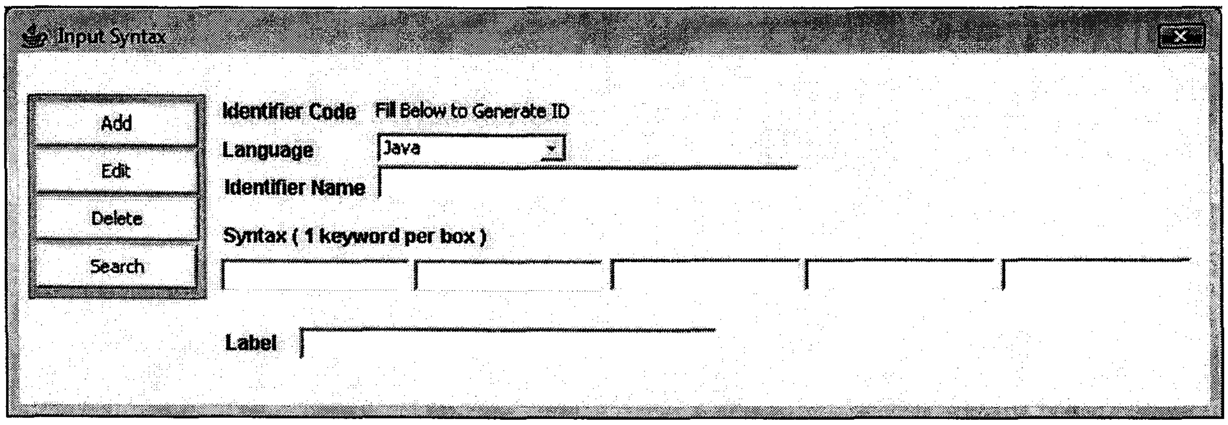
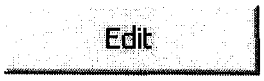


Figure 2.4: Window for Input Syntax.

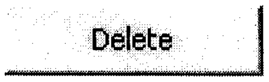
This section describes the operations for these buttons.

-

  Send record to the database.

- 

  Update records in the database.

- 

  Delete records in the database.

-

  Search records in the database to perform update or delete.

Input Syntax is used for identifying syntax. You have to choose the language of the programming language whether Java or C++. Then type in Identifier Name which is name used to identify the component that has been extracted in the database. Identifier Code will be generates automatically when you input both Language and Identifier Name.

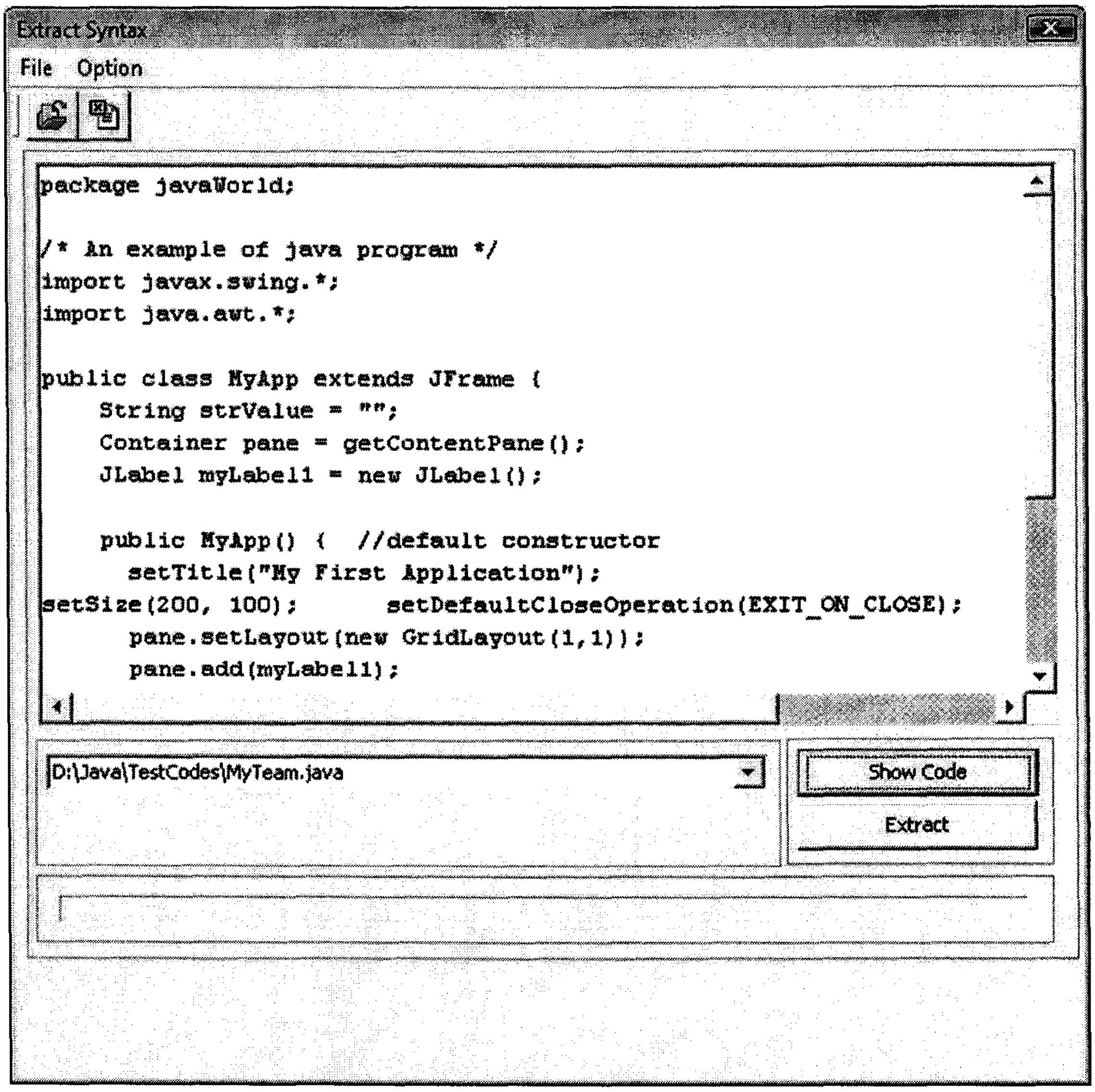
At the Syntax field, you are allowed to input up to 5 words as the syntax check. If the input contains the access modifier ('public', 'private' and 'protected'), you need to type in the software reserved word **'<access modifier>'**. If the input contains the data ('int', 'String' and etc.), you need to type in the software reserved word **'<data type>'**. If the input contains the name of the component to be extracted, you need to type in the software reserved word **'<name>'**.

At the Label field, it is used as the label name in the output of the Extracted Result window. Then click

Add button to send record to the database.

2.3.2 Analyze Syntax

This sub-menu provides for syntax analyzer. A pop-up window (Figure 2.5) appears when you click at Analyze Syntax.



```
package javaWorld;

/* An example of java program */
import javax.swing.*;
import java.awt.*;

public class MyApp extends JFrame {
    String strValue = "";
    Container pane = getContentPane();
    JLabel myLabel1 = new JLabel();

    public MyApp() {  //default constructor
      setTitle("My First Application");
setSize(200, 100);      setDefaultCloseOperation(EXIT_ON_CLOSE);
      pane.setLayout(new GridLayout(1,1));
      pane.add(myLabel1);
```

Figure 2.5: Window for syntax analyzer.

This section describes the operations for these buttons.

- Open.

  This operation opens a window for you to select the specific code file.

- Extract Result.

  This operation extracts results from code that has been analyzed by analyzer.

- Show Code

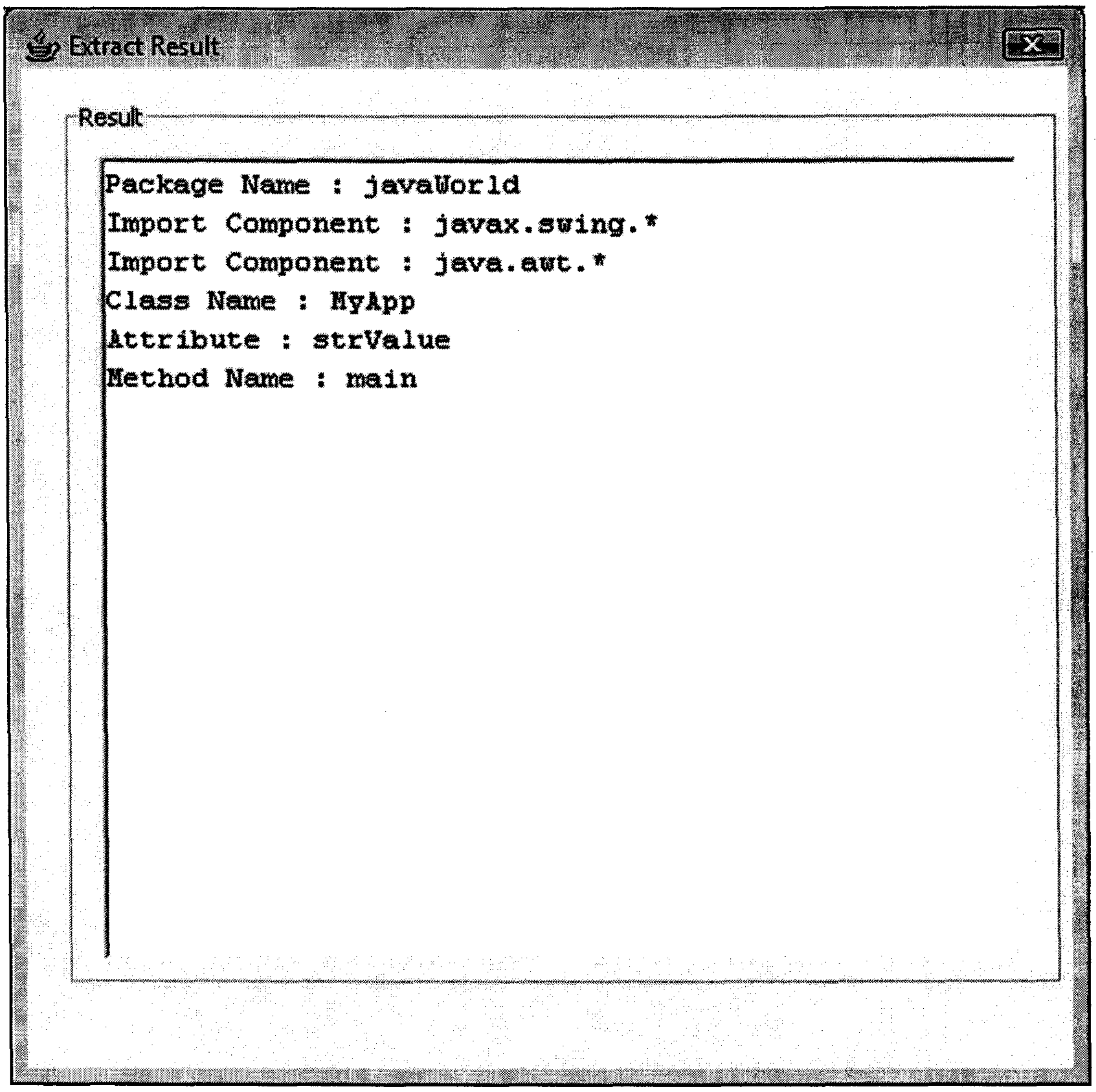
  This operation shows the code in jTextArea frame.

- Extract

  This operation extracts results from code that has been analyzed by analyzer.

Syntax analyzer is used for extracting code. Before extracting code, you have to show the code first by typing it at the space given or open it using open button. Default system will catch code from jTextArea in this frame.

Figure 2.6 shows the result for extracting code. This operation performs after you click Extract Extract button.



```
Package Name : javaWorld
Import Component : javax.swing.*
Import Component : java.awt.*
Class Name : MyApp
Attribute : strValue
Method Name : main
```

Figure 2.6: Result window for extracting code.

2.4 The Help Menu

This menu provides About UDARE.

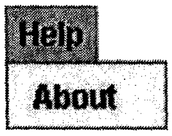
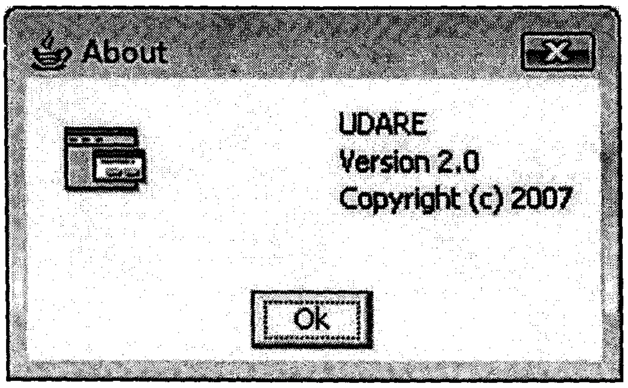


Figure 2.7: Sub-menu in Help menu for UDARE window.

2.4.1 About UDARE

This menu entry brings up the About window for UDARE. It tells the version and copyright of UDARE.



UDARE
Version 2.0
Copyright (c) 2007

Figure 2.8: About window for UDARE.