Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

Using software analysis tools to understand military applications

A qualitative study

P. Charland
D. Dessureault
M. Lizotte
D. Ouellet
C. Nécaille
DRDC Valcartier

Defence R&D Canada – Valcartier
Technical Memorandum
DRDC Valcartier TM 2005-425
August 2006

Canada

Using software analysis tools to understand military applications

A qualitative study

P. Charland
D. Dessureault
M. Lizotte
D. Ouellet
C. Nécaille
DRDC Valcartier

Defence R&D Canada - Valcartier

Technical Memorandum

DRDC Valcartier TM 2005-425

August 2006

Author

Philippe Charland

Approved by

Guy Turcotte
Head System of Systems

Approved for Release by

Gilles Bérubé
Chief Scientist

Abstract

Although some studies have already been conducted to evaluate the effect of reverse engineering and visualization tools on programmers' understanding, most of them were conducted under conditions which do not prevail in the industry. They involved undergraduate and graduate students performing comprehension tasks on relatively small scale programs. Also, they either focused exclusively on the static or dynamic aspect of the software under examination. This technical memorandum describes the design and reports the observations of a qualitative study conducted to assess the value added by one reverse engineering and two dynamic analysis tools. The software examined were three large scale military applications written in C++ and Java. In this study, five participants had to perform 31 comprehension tasks, taking into consideration both the static and dynamic aspects of the applications under examination. The tasks were intended to be as close as possible to the ones performed during an understanding effort at the architectural level on large scale software. Although it was observed that the tools aided the participants to understand the applications under examination, some deficiencies were observed. These stem from the fact that the tools do not always provide the appropriate viewpoints, abstraction levels, and filters needed to understand the architecture of applications of considerable size. This is especially true in the case of the dynamic tools.

Résumé

Bien qu'un certain nombre d'études aient déjà été menées afin d'évaluer l'effet des outils de rétro-ingénierie et de visualisation sur la compréhension des programmeurs, la plupart d'entre elles ont été accomplies dans des conditions qui ne prévalent pas dans l'industrie. Elles impliquaient des étudiants de premier et de deuxième cycle exécutant des tâches de compréhension sur des programmes relativement de petite taille. De plus, ces études portaient exclusivement soit sur l'aspect statique ou dynamique des logiciels examinés. Le présent mémorandum technique décrit la conception et rend compte des observations d'une étude qualitative qui a été menée afin d'évaluer la valeur ajoutée d'un outil de rétro-ingénierie et de deux outils d'analyse dynamique. Les logiciels examinés par ceux-ci étaient trois applications militaires de grande taille écrites en C++ et en Java. Lors de cette étude, cinq participants ont eu à accomplir 31 tâches de compréhension, prenant en considération tant l'aspect statique et dynamique des applications sous observation. Ces tâches avaient comme intention d'être aussi près que possible de celles qui sont exécutées lors d'un effort typique de compréhension au niveau de l'architecture sur des logiciels de grande taille. Bien qu'il fût observé que les outils aient aidé les participants à comprendre les applications à l'étude, quelques points faibles ont été observés. Ces derniers découlent du fait que les outils ne fournissent pas toujours les points de vue appropriés ainsi que les niveaux d'abstraction et filtres requis pour comprendre l'architecture d'applications de taille considérable. Ceci est particulièrement vrai dans le cas des outils dynamiques.

This page intentionally left blank.

Executive Summary

Over the years, the needs of the Canadian Forces (CF) for systems interoperability have significantly increased. As the CF demand greater systems interoperability, their software architects need techniques and tools to comprehend the architecture of existing systems before making them interoperate in order to build a system of systems. There already exist a variety of commercial and academic tools that can be used to assist architects in recovering the architecture of existing systems. However, no study has been conducted to assess the impact of these tools on the understanding of users performing comprehension tasks under conditions that prevail in the industry. This technical memorandum describes the design and reports the observations of a qualitative study conducted in such conditions. Its objective was to assess the value added by one reverse engineering and two dynamic analysis tools on the understanding of architects performing comprehension tasks on large scale software. In this study, five participants with experience in software development were observed performing 31 high level comprehension tasks on three military applications written in C++ and Java. The observations that were made are as follows:

1. Although the three tools helped to understand the different applications, the level of comprehension achieved by the participants at the end of the study was not only due to their use. It was also attributable to the fact that the participants studied the applications domain and performed the comprehension tasks by first using an Integrated Development Environment (IDE).

2. The three tools provided information which would have been very difficult to obtain otherwise. For example, the dependency hierarchy in Headway reView conveyed to the participants a mental map of the applications under examination. Also, the suite of software metrics provided with Headway reView allowed participants to obtain accurate numbers characterizing properties of the source code very quickly.

3. In spite of their advantages, the tools have some deficiencies. Their biggest drawback is that they do not always provide the appropriate viewpoints, abstraction levels, and filters needed to understand the architecture of an application. This is especially true in the case of the dynamic tools. The participants were quickly swamped by a mass of irrelevant low level details.

4. Meaningful names chosen for components, classes, and methods had a considerable positive impact on the comprehension of the participants.

5. A consistent naming convention used throughout the applications source code for components and classes also facilitated greatly their understanding.

6. Some tasks were more useful than others to achieve the objective of the study and comprehend the different applications at a high level. These were the ones involving clustering, abstractness, and the overall structure of the applications.

7. One approach which ended to be very useful for the comprehension consisted of combining static and dynamic analysis as well as information about the operation of the applications and the naming conventions.

8. The study allowed to validate some assumptions which were made at the time the comprehension tasks were designed. These related to the number of dependencies between components in an application.

9. Some problems experienced with the tools are inherent to the programming languages. Unlike Java, the directory structure of a C++ program does not always correspond to its logical structure. This complicates the understanding and was confirmed in the study. It has been more difficult to achieve the same comprehension level for the C++ application than for the ones in Java.

Following this qualitative study and using the theoretical and practical knowledge acquired through it, the next step will consist of developing a prototype. This prototype will address the limitations identified concerning the dynamic aspect. It will therefore provide the appropriate viewpoints, abstraction levels, and filters required for the visualization of dynamic information at the architectural level. These dynamic views will be integrated into an IDE providing static views of the source code. In addition, the prototype will offer functionalities to facilitate the mapping of source code elements to their corresponding concept of the application domain.

Charland, P., Dessureault, D., Lizotte M., Ouellet, D., and Nécaille, C. 2006. Using Software Analysis Tools to Understand Military Applications: A Qualitative Study. DRDC Valcartier TM 2005-425. Defence R&D Canada - Valcartier.

Sommaire

Au cours des années, les besoins des Forces canadiennes (FC) en matière d'interopérabilité de systèmes ont augmenté de façon significative. Alors que les FC exigent plus d'interopérabilité entre les systèmes, leurs architectes logiciels ont besoin de techniques et d'outils pour comprendre l'architecture des systèmes existants avant de les faire interopérer pour construire un système de systèmes. Il existe déjà un grand nombre d'outils commerciaux et universitaires qui peuvent être utilisés afin d'aider les architectes à récupérer l'architecture de systèmes existants. Cependant, aucune étude n'a été menée afin d'évaluer l'impact de ces outils sur des usagers accomplissant des tâches de compréhensions dans des conditions qui prévalent dans l'industrie. Le présent mémorandum technique décrit la conception et rend compte des observations d'une étude qualitative qui a été menée dans de telles conditions. Son objectif était d'évaluer la valeur ajoutée d'un outil de rétro-ingénierie et de deux outils d'analyse dynamique sur la compréhension d'architectes exécutant des tâches de compréhension sur des logiciels de grande taille. Lors de cette étude, cinq participants avec de l'expérience en développement logiciel ont été observés en train d'accomplir 31 tâches de compréhension à haut niveau sur trois applications militaires écrites en C++ et Java. Les observations qui ont été faites vont comme suit :

1. Bien que les trois outils aient aidé à comprendre les différentes applications, le niveau de compréhension atteint par les participants à la fin de l'étude n'était pas seulement dû à leur utilisation. Il était aussi attribuable au fait que les participants aient étudié le domaine des applications et aient accompli les tâches de compréhension en utilisant d'abord un environnement de développement intégré (EDI).

2. Les trois outils ont fourni de l'information qui aurait été très difficile d'obtenir autrement. Par exemple, la dépendance hiérarchique d'Headway reView a donné aux participants une carte mentale des applications à l'étude. De plus, la suite de métriques logicielles fournie avec Headway reView a permis aux participants d'extraire des chiffres exacts caractérisant les propriétés du code source très rapidement.

3. En dépit de leurs avantages, les outils ont un certain nombre de points faibles. Leur plus gros inconvénient est qu'ils ne fournissent pas toujours les points de vue appropriés ainsi que les niveaux d'abstraction et filtres requis afin de comprendre l'architecture d'une application. Ceci est surtout vrai dans le cas des outils dynamiques. Les participants ont été rapidement noyés par une grande quantité de détails non pertinents de très bas niveau.

4. Des noms significatifs choisis pour les composantes, classes et méthodes ont eu un impact positif considérable sur la compréhension des participants.

5. Une nomenclature constante utilisée dans le code source des applications pour les composantes et les classes a également facilité considérablement leur compréhension.

6. Un certain nombre de tâches ont été plus utiles que d'autres pour atteindre les objectifs de l'étude et comprendre les différentes applications à un haut niveau. Celles-ci étaient les tâches impliquant des regroupements, des abstractions ainsi que la structure globale des applications.

7. Une approche qui s'avéra être très utile pour la compréhension consistait à combiner l'analyse statique et dynamique ainsi que l'information sur le fonctionnement des applications et la nomenclature utilisée.

8. L'étude a permis de valider certaines hypothèses qui avaient été formulées lors de l'élaboration des tâches de compréhension. Celles-ci étaient reliées au nombre de dépendances entre les composantes d'une application.

9. Certains problèmes rencontrés avec les outils sont inhérents aux langages de programmation. Contrairement à Java, la structure des répertoires d'un programme C++ ne correspond pas toujours à sa structure logique. Ceci complique la compréhension et fut corroboré lors de l'étude. Il fut plus difficile d'atteindre le même niveau de compréhension pour l'application C++ que pour celles développées en Java.

En utilisant les résultats théoriques et pratiques obtenus suite à cette étude qualitative, la phase suivante consistera à développer un prototype. Ce prototype abordera les limitations identifiées concernant l'aspect dynamique. Par conséquent, il fournira les points de vue appropriés ainsi que les niveaux d'abstraction et filtres requis pour la visualisation d'information dynamique au niveau de l'architecture. Ces vues dynamiques seront intégrées dans un EDI fournissant des vues statiques du code source. De plus, le prototype offrira des fonctionnalités afin de faciliter l'association des éléments du code source à leur concept correspondant du domaine d'application.

Charland, P., Dessureault, D., Lizotte M., Ouellet, D. et Nécaille, C. 2006. Using Software Analysis Tools to Understand Military Applications: A Qualitative Study. DRDC Valcartier TM 2005-425. R&D pour la défense Canada - Valcartier.

Table of Contents

List of Figures

List of Tables

Acknowledgements

For this qualitative study, two of the participants were members of the OASIS research project at DRDC Valcartier. Without the involvement of additional people, the realization of this study would not have been possible. As a result, the authors would like to thank the following participants for their involvement: Philippe Plante (DRDC Valcartier), Maxime Tardif (Thales Systems Canada), and Marco Savard (Neosapiens). Thales Systems Canada and Neosapiens are two consulting firms doing contractual work for DRDC Valcartier. The authors are also indebted to René Proulx (Thales Systems Canada), who assisted in organizing and running the study. Furthermore, the comments from Allan Gibb (DRDC Valcartier) and François Lemieux (DRDC Valcartier) were much appreciated and helped to improve this technical memorandum.

1. Introduction

Over the years, the needs of the Canadian Forces (CF) for systems interoperability have significantly increased. For example, to improve the automation of the Command, Control, Communications, Computers, Intelligence, Surveillance, and Reconnaissance (C4ISR) process, a large number of software intensive systems must interact together to handle a massive amount of information. The CF also require systems interoperability when they collaborate with the allied nations to achieve common objectives.

As the CF demand greater systems interoperability, their software architects need techniques and tools to comprehend the architecture of existing systems and make them interoperate in order to build a system of systems (SoS). A SoS is an assemblage of components which individually may be regarded as systems and which possess two additional properties: operational and managerial independence of the components [1]. Each component system must be able to operate independently if the SoS is disassembled. Furthermore, even though the component systems are separately acquired and integrated, they maintain a continuing operating existence independent of the SoS. An example of a SoS is a system built for a coalition operation, where each participating nation brings its own operational planning system.

Before existing systems can interoperate, their architectures first need to be understood. The architecture of a system can be defined as the structure of its components, their interrelationships, as well as the principles and guidelines governing their design and evolution over time [2]. However, understanding the architecture of systems can prove to be quite a complex task. These systems have most probably undergone several code revisions without a real concern about maintaining their architectural design documentation up to date [3]. As a result, architecture recovery has to be performed to regenerate coherent abstractions and guide architects during their comprehension task. Architecture recovery can be described as the process of retrieving up-to-date architectural information from existing source code artefacts [4, 5, 6, 7, 8, 9, 10]. The rational of system architectural recovery is to provide reasoning behind the software architecture or high-level organization of a system [11, 12].

To support the effort of developing methodologies, techniques, and tools needed for the recovery and comprehension of existing systems' architecture, the SoS section of Defence Research and Development Canada (DRDC) Valcartier started a project called Opening up Architecture of Software-Intensive Systems (OASIS) [13]. Its objective is to develop technical solutions in order to reduce the time needed to comprehend systems to be integrated into a SoS.

There already exist a variety of commercial and academic tools that can be used to assist architects in recovering the architecture of existing systems. Most of them were identified in [14] as part of a previous phase of the OASIS project. However, one can ask if the readily available tools for the CF, whether open source or commercial, can address the needs of the current research project, i.e., to recover and comprehend the

architecture of military applications written in C++ or Java and consisting of more than 1,000 classes. For this reason, a study of the existing tools has to be conducted. Its purpose would be to assess their added value on the understanding of architects performing comprehension tasks on large scale software. This would help in determining which existing approaches appear to be more effective. Furthermore, it would provide some elements of response to the question that a military client could ask: To what extent the existing tools can assist him with his software comprehension and maintenance needs?

This technical memorandum describes the design and reports the observations of a qualitative study in which five participants performed various high level comprehension tasks on object-oriented software written in C++ and Java. The characteristics of the selected applications are indicated in Table 1.

Table 1. *Applications Used for the Study*

APPLICATION	LANGUAGE	NO. OF CLASSES	LINES OF CODE
HCI_CASE_ATTI	Java	565	74 K
COPlanS	Java	1600	120 K
ATS	C++	1650	670 K

To perform the assigned tasks, the participants used three commercial software tools: Headway reView, Rational PureCoverage, and Rational Quantify. These were selected based on a survey of existing tools. Headway reView 3.4 [15] is a reverse engineering and static analysis tool used for source code comprehension. It can parse C++ and Java programs to reverse engineer a visual representation of the composition and dependencies of an application. PureCoverage and Quantify are part of Rational PurifyPlus 6.13 [16], a set of automated runtime analysis tools for improving the reliability and performance of applications. PureCoverage is a code coverage tool, while Quantify is a performance analysis tool. Both of them can analyze C++ and Java programs.

The remainder of this technical memorandum is organized as follows: Section 2 reviews previous studies conducted on software comprehension. In Section 3, the tools used as part of the present qualitative study are described. Section 4 details its design while Section 5 reports observations that were made. Section 6 discusses the limitations of the study and finally, Section 7 identifies the conclusions and future work.

2. Software Comprehension

Several studies have been conducted to determine which strategies programmers use when trying to understand unfamiliar code. The results have demonstrated that different cognitive models are applied to create mental representations of programs under examination. But before these models can be reviewed, their terminology first needs to be defined.

2.1 Concepts and Terminology

A programmer's mental representation of a program under study is referred as the mental model [17]. The cognitive processes and temporary information structures used by the programmer to form the mental model are described by a cognitive model [17].

Programming plans are generic fragments of source code which represent typical programming scenarios. An example of a programming plan is a sorting algorithm [18].

Beacons are familiar features in the source code which act as cues to the presence of certain structures [19]. An example of a beacon is the swapping of two variables in a sorting algorithm. Rules of programming discourse are the programming conventions and algorithm implementations [18].

2.2 Cognitive Models

Following is a review of some of the influential cognitive theories in program comprehension as reviewed in [17].

2.2.1 Bottom-Up

Shneiderman [20, 21] proposed that programs are understood bottom-up, i.e., by first reading the source code and then mentally grouping lower level software artifacts into higher level abstractions that are more meaningful. These abstractions are further aggregated until a high level comprehension of the program is obtained. The cognitive framework of Shneiderman and Mayer [20] makes a distinction between the syntactic and semantic knowledge of a program. The syntactic knowledge is language dependent and relates to the statements of a program, while the semantic knowledge is language independent and is formed in progressive layers until a mental model of the application domain is built.

In [22], Pennington also observed that programmers use a bottom-up strategy when trying to understand a program. They first produce a control flow abstraction, referred as the program model, which represents the sequence of operations of the program.

This model is generated by grouping source code microstructures (statements, predicate statements, dependencies) into macrostructures (source code structure abstractions) and then by cross-referencing them. After the program model has been assimilated, the situation model is generated. This model incorporates knowledge about the data flow and the functional abstractions, e.g., the program goals hierarchy.

2.2.2 Top-Down

Brooks formed a theory that programs are understood in a top-down manner, where the knowledge about the application domain is first reconstructed and then mapped on the source code [19]. This process starts with the formulation of a hypothesis about the general nature of the program. This global hypothesis is then refined into a hierarchy of secondary hypotheses, which are evaluated in a depth-first manner. The validation of rejection of a hypothesis depends heavily on the presence or absence of beacons [19].

Soloway and Ehrlich [18] observed that a top-down strategy is used when the source code or type of source code is familiar. They also noted that experienced programmers use beacons, programming plans, as well as rules of programming discourse in order to decompose goals and plans to a lower level. Furthermore, it was observed that delocalized plans complicate program comprehension.

2.2.3 Knowledge-Based

Letovsky [23] suggested that programmers are opportunistic processors, capable of understanding programs using either a bottom-up or top-down approach, depending on the cues available. His theory has three components: a knowledge base, which encodes the programmer's expertise and knowledge about the application; a mental model, which represents the programmer's current understanding of the program; and an assimilation process, which explains how the mental model evolves using the knowledge base and information about the program.

Inquiry episodes are an essential part of the assimilation process. During such an episode, a programmer asks a question, forms a hypothesis, and searches through the source code and documentation to validate or reject the hypothesis. Inquiry episodes often happen as a result of delocalized plans.

2.2.4 Systematic and As-Needed

In [24], Littman et al. observed programmers enhancing a personnel database program. They noted that the programmers either read the source code systematically, tracing the control and data flow dependencies in order to acquire a general understanding, or used an as-needed approach, focusing only on the source code related to the task to achieve. The subjects using a systematic approach gained information about the structure of the program and the interactions between its components at run-time. The ones who used an as-needed approach only acquired static knowledge, resulting in a

weaker mental model compared to the one of the other subjects. They also made more errors, as they did not identify the dynamic interactions between the components.

Soloway et al. [25] combined these two theories as macro-strategies in order to understand programs at a more global level. Using this strategy, the programmer traces the flow dependencies for the whole program and performs simulations as the source code and documentation is read. However, this method is not applicable for programs of considerable size. In the more commonly used approach, programmers examine only what they consider relevant. The drawback of this approach is that more mistakes can be made, since important interactions can be missed.

2.2.5 Integrated Metamodel

Based on the results of experiments, Von Mayrhauser and Vans combined the previous approaches into a single metamodel [26]. They suggested that understanding is built at several levels of abstractions, by freely switching between the different comprehension strategies. Their model is composed of four components. The first three detail the comprehension processes used to create the metal representations at different levels of abstractions. The fourth component describes the knowledge base used to carry out the comprehension process. In their integrated metamodel [26]:

- The top-down approach is invoked as an as-need strategy, when the source code or programming language is familiar. It uses the domain knowledge as a starting point for the formulation of hypotheses.

- The program model, which is a control flow abstraction, is invoked when the source code and application is completely unfamiliar.

- The situation model, which describes the data flow and functional abstractions in a program, is developed after a partial program model has been formed using systematic or opportunistic strategies.

- The knowledge contains the information required to build these three cognitive models. It stores the programmer's current knowledge as well as the one acquired and inferred during the comprehension process.

2.2.6 Factors Influencing the Selected Approach

The wide variety of comprehension strategies discussed above stems from the fact that certain factors will affect the approach selected by a programmer [27, 28]. These factors are the program under study, the comprehension task to achieve as well as the programmer's past experience, ability, and creativity [17].

2.3 Related Studies

Different studies [29, 30] and other evaluations [31, 32] have also been conducted to explore the question as to whether or not reverse engineering and visualization tools

enhance programmers' understanding. However, they were performed on relatively small scale programs. Therefore, their results cannot be directly mapped to larger ones, since the nature of the software used in a comprehension experiment affects the comprehension process [33]. Also, all of them took into consideration either the static or dynamic aspect of the applications under study, but not both and focused on only one programming language (C or Java). Furthermore, the training provided to the participants was somewhat limited. For example, in [29, 30], the training time lasted between 30 and 40 minutes. As a result, all the features provided by the tools were probably not fully exploited. The present study attempted to address these limitations and tried to reproduce the conditions under which a person working in the industry must be subjected to when trying to understand unfamiliar source code.

3. Static and Dynamic Analysis Tools

The majority of current tools that can be used for architecture recovery are built around traditional reverse engineering ones, as they use the source code of an application as their starting point [14]. Examples of reverse engineering tools used frequently to recover architectures are [34, 35, 36, 37, 38, 39].

Figure 1, adapted from [40], illustrates the four major components of a traditional reverse engineering tool. The parser extracts the artifacts of the system. This static information is stored in the knowledge base or repository. It is then analyzed by the analysis component which derives information not explicitly available from the extracted facts. Finally, the visualization component provides high level views of the extracted and analyzed data. Some reverse engineering tools have an additional filtering component which allows the user to specify the result set to be displayed.

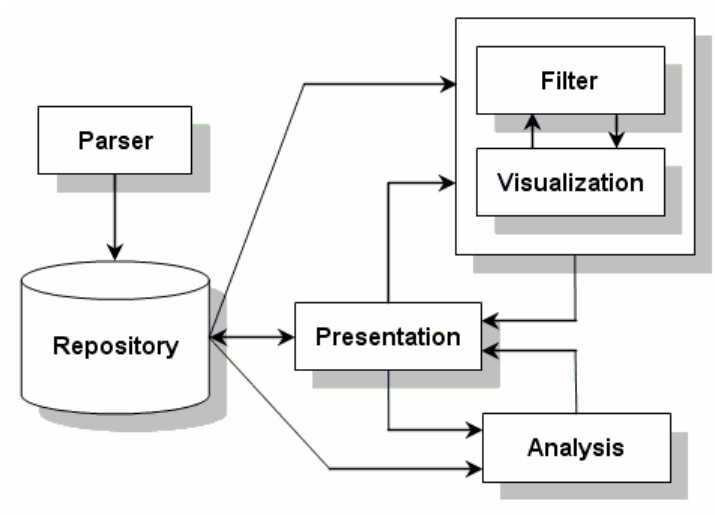


Figure 1. *Components of a Traditional Reverse Engineering Tool*

Today's software makes extensive use of polymorphism and dynamic binding. Therefore, architectural recovery cannot rely only on static information. It must be complemented by dynamic analysis, such as the exchange of control and data between the various components at run time. This information increases the level of precision provided by the static analysis and as a result, improves understanding.

3.1 Selection Process

The selection of the static and dynamic analysis tools for the present qualitative study comprised three stages. The first one consisted of performing a state of the art survey

of the current architecture recovery tools. This was part of a study [14] carried out by Dr. Juergen Rilling, an Assistant Professor at Concordia University. The tools surveyed are listed in Tables 2 and 3.

Table 2. *Static and Dynamic Tools Surveyed*

TOOL NAME	ORGANIZATION	TYPE OF ANALYSIS	SUPPORTED LANGUAGES
Refine/C - Illuma	Reasoning	Static	C, C++
SNiFF+	**Wind River**	**Static**	**Ada, C, C++, CORBA IDL, Java, FORTRAN**
Columbus/CAN	**FrontEndART**	**Static**	**C++**
Understand for C++/Java	**Scientific Toolworks**	**Static**	**C, C++, Java**
Datrix	Bell Canada Entreprises	Static	C, C++, Java
CodeCrawler	University of Bern	Static	External parser required
CodeSurfer	GrammaTech	Static	C
ArgoUML	Open Source	Static	Java
Visual Paradigm for UML	Visual Paradigm	Static	Java
SWAG Kit	**University of Waterloo**	**Static, Dynamic**	**C, C++**
Rigi	**University of Victoria**	**Static**	**C, C++, COBOL**
reView	**Headway Software**	**Static**	**Ada83/95, C, C++, Java**
Klocwork	inSight	Static	C, C++
RIVA	Nokia	Static	C, C++
Bauhaus	University of Stuttgart	Static, Dynamic	C, C++, Java, third-party parsers
URCA	University of Belgrade	Dynamic	C++
CONCEPT	Concordia University	Static, Dynamic	Java
Aladdin	University of Colorado	Static	Rapide
PROMON	Technical University of Vienna	Dynamic	Java
DocGen	Software Improvement Group	Static	C++, Java
Ciao	AT&T Bell Laboratories	Static	C, C++, Java

Table 3. *Static and Dynamic Tools Surveyed (Continued)*

TOOL NAME	ORGANIZATION	TYPE OF ANALYSIS	LANGUAGES SUPPORTED
G[see]	University of Grenoble	Static	Java
Lemma	IBM	Static	Assembler, C, C++, Java, Pascal, PL/X, PL/I, Rexx
Red Hat Source-Navigator	Red Hat	Static	C, C++, COBOL, Java, Tcl, FORTRAN
SoftArch	University of Auckland	Static, Dynamic	Java
ARMIN	Software Engineering Institute	Static	C++, Java, third-party parsers
ManSART	MITRE	Static	C, C++
Imagix 4D	**Imagix**	**Static**	**C, C++**
Rose	**Rational**	**Static**	**Ada83/95, C++, Java, CORBA IDL**

Following the state of the art survey, a practical evaluation of a selected subset of the tools reviewed was performed. The tools which were evaluated are highlighted in bold in Tables 2 and 3. While performing this evaluation, other static and dynamic analysis tools which could potentially address the needs of the current research project were identified. These additional tools, listed in Tables 4 and 5, were also evaluated. The selection of the tools for the practical evaluation was based on the characteristics of the targeted applications to be analyzed. These characteristics are listed in Table 6.

Table 4. *Additional Static and Dynamic Tools Surveyed*

TOOL NAME	ORGANIZATION	TYPE OF ANALYSIS	LANGUAGES SUPPORTED
Together	Borland	Static	C++, C#, Java
SHriMP	University of Victoria	Static	C, C++, Java
CodeLogic	Logic Explorers	Static	C#, Java
Eclipse	Open Source	Static	Java
Sun One Studio	Sun Microsystems	Static	Java
Rational	PureCoverage	Dynamic	C++, C#, Java

Table 5. *Additional Static and Dynamic Tools Surveyed (Continued)*

TOOL NAME	ORGANIZATION	TYPE OF ANALYSIS	LANGUAGES SUPPORTED
Rational	Quantify	Dynamic	C++, C#, Java
JProbe	Quest Software	Dynamic	Java
OptimizeIT	Borland	Dynamic	Java
JProfiler	ej-technologies	Dynamic	Java
AQtime 4	AutomatedQA	Dynamic	C++, C#

Table 6. *Characteristics of the Targeted Applications*

CHARACTERISTIC	DESCRIPTION	MANDATORY	OPTIONAL
Programming Language	The application should be written in an object-oriented language: C++ and/or Java (J2SE).	x	
Operating System	The application should run on Windows 2000 or XP.	x	
External Dependencies	The application should interact with others.		x
Application Domain	The domain of the application should be military (e.g., tactical mission planning, decision aid).	x	
Multi-Process	The application should use interprocess communication mechanisms such as sockets.		x
Multithreading	The application should be multithreaded.		x
Source Code Available	The source code should be available to perform reverse engineering using static tools.	x	
Executables Available	The executables should be available to observe the behaviour of the application using dynamic tools.	x	
Application Type	The application should be event-based.	x	
Application Size	The size of the application should be between 50,000 to 500,000 LOC and between 100 classes to 1,000 classes.	x	

Following the state of the art survey and the practical evaluation, a final selection of the tools to be used for the present study was made. The selection consisted of Headway reView, Rational PureCoverage, and Rational Quantify.

Headway reView was selected since it has been identified as a reverse engineering tool supporting architecture recovery [14]. It was also chosen because of its visualization and analysis features, described in more detail below, and the fact that it could parse both C++ and Java source code. The latter was a mandatory requirement, since the applications to be examined as part of the present study were written in either C++ or Java. Also, the tests carried out prior to the beginning of the study indicated that it was robust and could parse applications consisting of more than 1,000 classes. Furthermore, Headway reView is used by numerous organizations, such as Sun Microsystems and the NASA.

Headway reView provides static information only. As a result, it had to be complemented by a dynamic analysis tool, since the examined applications were object-oriented and made use of polymorphism as well as of dynamic binding. For this reason, two tools of the Rational PurifyPlus suite were selected: PureCoverage and Quantify. The features of these tools provide information needed to analyze the run-time behaviour of C++ and Java applications. Also, the information produced by Quantify had been used in the past by other program understanding tools such as Software Emancipation's DISCOVER [41].

Other static analysis tools such as ARMIN (Architecture Reconstruction and MINing) [5] and SHriMP (Simple Hierarchical Multi-Perspective) [42] could also have been selected for the present study, as they can analyze the source code of applications developed in C++ and Java.

ARMIN is an architecture reconstruction tool developed by the Software Engineering Institute and the Robert Bosch Corporation. Although its use has been reported in [43, 44] to recover the architecture of C++ and Java applications, ARMIN was not selected because of its cost: 15,000 USD, in addition to the traveling expenses of the person who would come to DRDC Valcartier for a two-day tutorial on its use. The OASIS project could not afford to purchase it, since the allocated budget for the acquisition of computers and software for the study was 10,000 CAD.

SHriMP is an application developed at the University of Victoria to visualize and explore software architectures. It is also a domain-independent visualization technique designed to enhance how people browse and explore complex information spaces. SHriMP has been integrated into the open source software development project Eclipse [45] through the Creole plug-in [46]. One should use this plug-in and not the stand-alone version of SHriMP to visualize Java source code, as the Java fact extractor of the stand-alone version is now obsolete [47].

The reason why SHriMP was not selected is because to parse C++ source code, one needs another fact extractor generating files in the Rigi Standard Format (RSF), a format that SHriMP can read. The organizers of the study did not want to support two different parsers, one for each programming language, due to time constraints.

Furthermore, the authors of the present technical memorandum believed that potential military clients for the OASIS project would be more interested by commercial tools than by academic projects, as commercial products tend to be more robust and able to analyze large scale applications. This is very important, since most military applications are quite large. The Creole plug-in has some known issues with programs of a considerable size [48]. It can take some time to manage and display large amounts of data and can run out of memory while trying to visualize large working sets. It also has some memory leaks and may require the restart of Eclipse. Furthermore, the fact extraction does not find some relationships between anonymous inner classes.

For the dynamic analysis of the applications under study, other tools could have been selected: AutomatedQA AQtime 4 [49], Quest JProbe [50], Borland Optimizeit [51], and ej-technologies JProfiler [52]. The reason that none of them were chosen for the study is that they only support one programming language: C++ in the case of AQtime 4 and Java for the other ones.

As already mentioned, the training provided to the participants in previous studies [29, 30] was minimal. To address this limitation, it was decided to provide what was believed to be appropriate training, so that the participants could use efficiently the features of the tools needed to perform the comprehension tasks. By limiting the choice of tools to three, the training as well as the completion of all comprehension tasks could be carried out in a reasonable amount of time.

3.2 Headway reView

Headway reView uses reverse engineering and static analysis techniques to provide software architects with code comprehension, code review, and source code visualization tools. It can display all the dependencies within an application at all levels and between all levels: method, data member, class, package, and application. These dependencies are displayed in a hierarchical directed graph, using a number of intelligent layout algorithms. This allows users to drill top-down from component to methods in order to discover sub-system relationships, while retaining the ability to understand the overall context of the component. reView can also show the code base using a number of different views, all synchronized within the code base. Moreover, it provides a suite of analysis and software metrics integrated with its visualization tool. Figure 2 and 3 show two screenshots of reView. Figure 2 illustrates a packages hierarchy and their dependencies displayed as a directed graph. The application used was JUnit [53], a Java framework for unit testing. The adornments on the edges represent the number of relationships between two packages. Figure 3 gives an overview of the set of metrics provided in reView. In the present example, the abstractness is displayed for each of the JUnit packages.

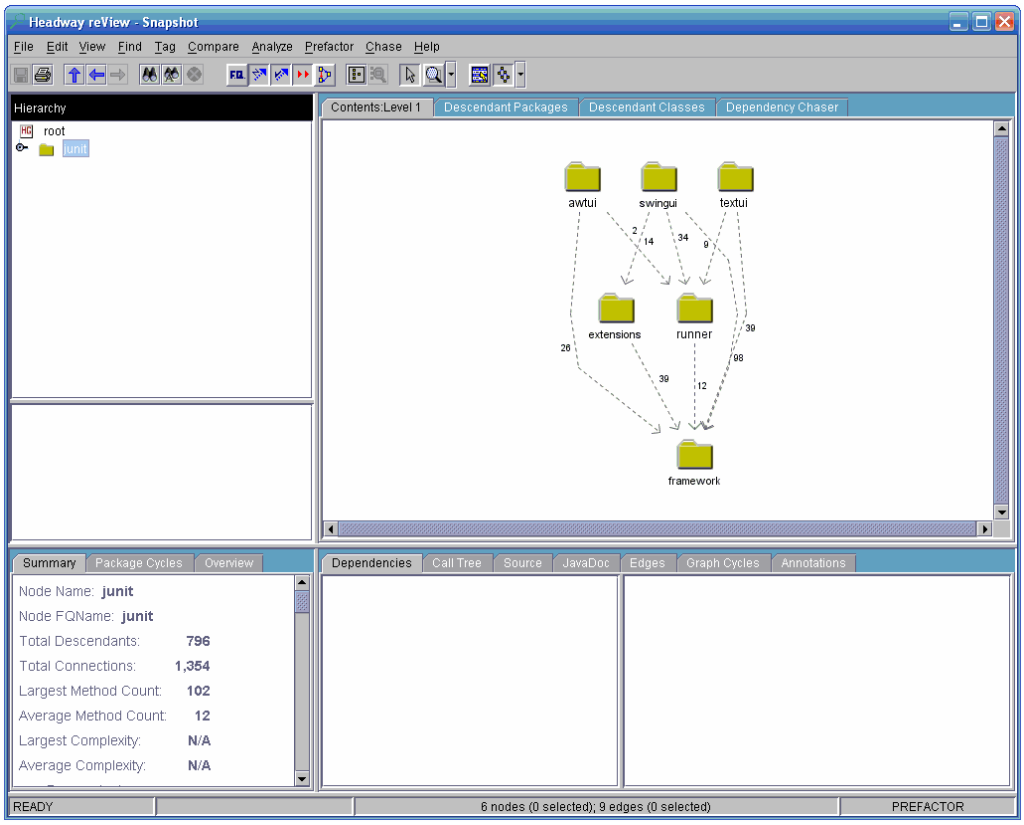


Figure 2. *A Packages Hierarchy and their Dependencies in Headway reView*

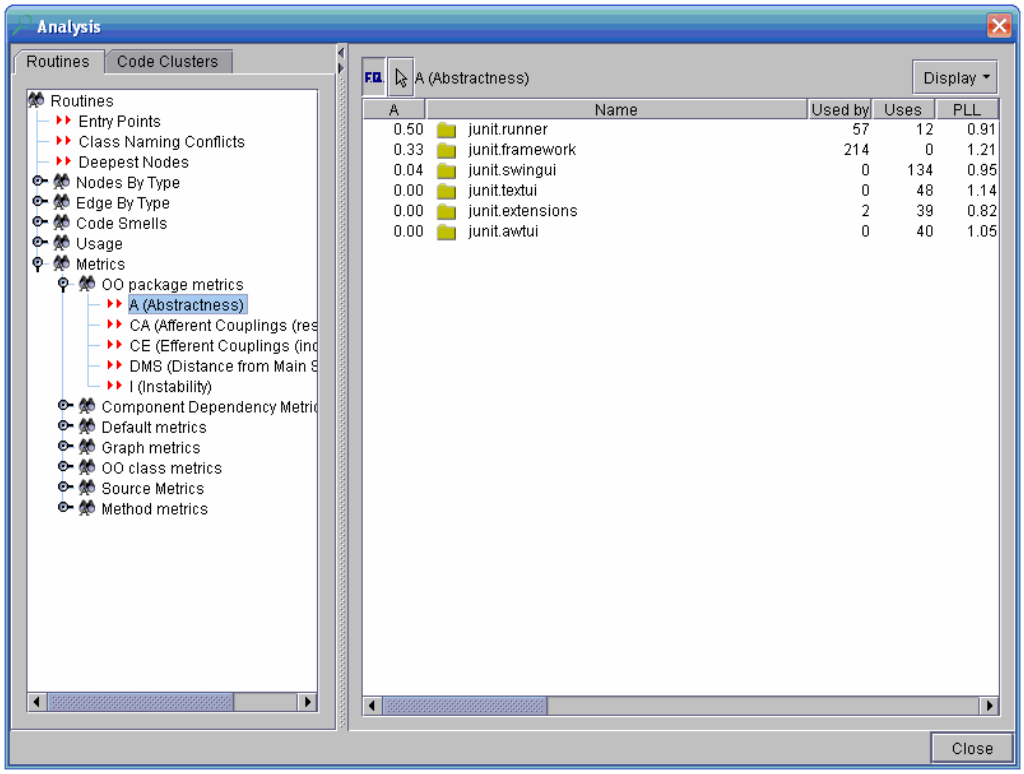


Figure 3. *An Example of the Metrics Provided in Headway reView*

3.3 Rational PureCoverage

Rational PureCoverage is a customizable code coverage analysis tool. It can automatically pinpoint executed and non-executed areas of code and visually display application analysis data. It can also merge coverage data from multiple runs of the same executable for an aggregate view of coverage data. Figure 4 displays the Coverage Browser window in PureCoverage for a particular execution of JUnit.

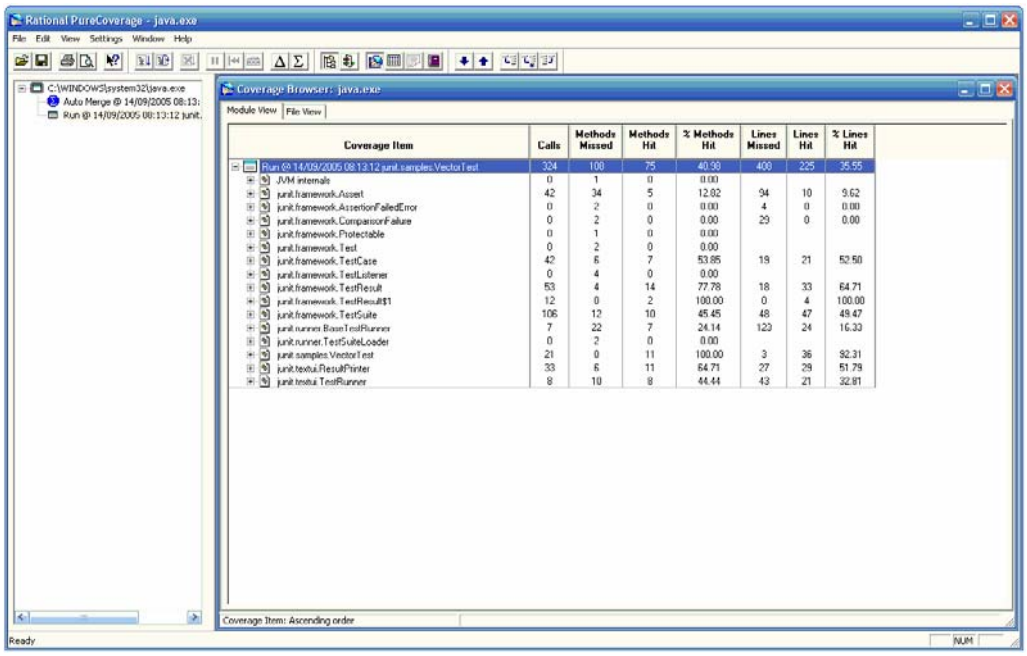


Figure 4. *The Coverage Browser in Rational PureCoverage*

3.4 Rational Quantify

Rational Quantify is a profiling tool that automatically pinpoints application performance bottlenecks. Its graphical performance data views shows how a program is executed in terms of function call architecture. It also highlights which functions contributed the most time to the program execution by line thickness and position on the Quantify's Call Graph and indicates the importance of any function to the program's overall performance. Its filter features allow one to concentrate on the parts of the application that are of most interest. Furthermore, its Diff and Merge capacities can respectively compare the execution time between two runs or merge the execution time for multiple runs. Figure 5 shows a section of a call graph in Quantify resulting from an execution of JUnit, with the most expensive path highlighted in bold.

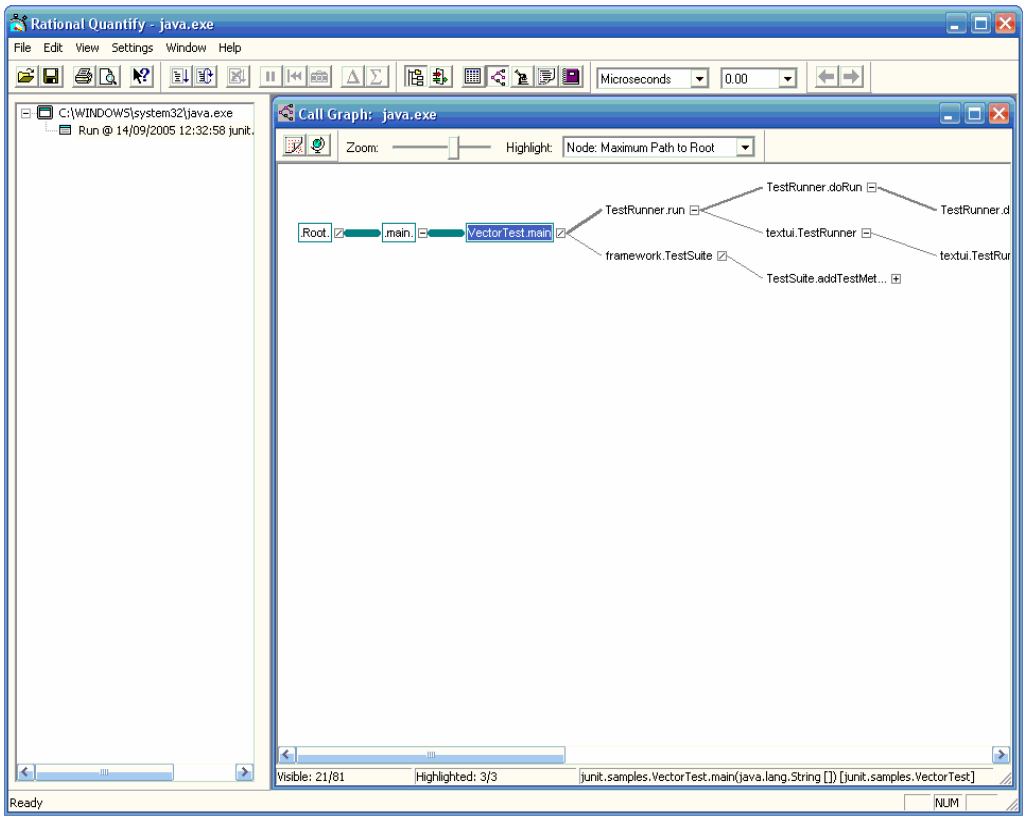


Figure 5. *A Call Graph Displayed in Rational Quantify*

4. Qualitative Study

The study to assess the value added by Headway reView, Rational PureCoverage, and Rational Quantify on the comprehension of the participants was conducted at DRDC Valcartier during the winter of 2005. It is based on a qualitative rather than a quantitative approach. This is because the study was performed within a specific context, the researchers observed the participants while they were executing the comprehension tasks, and they interacted with them through interviews.

Although most of the studies and other evaluations on program comprehension have been conducted following a quantitative approach, there is a shift occurring towards the qualitative paradigm. The cause of this move is that many researchers recognize the fact that studies conducted in controlled settings can be extremely revealing, since the conditions affecting program comprehension are complex and diverse, as mentioned in Section 2.2.6.

4.1 Objectives

There were several objectives involved in this study. First of all, it would allow members of the OASIS project to acquire a practical expertise in the discipline of architecture recovery and comprehension. It would also be an opportunity for them to experiment the limitations of the currently available tools. Furthermore, it could reveal strategies on how the comprehension of a large scale application can be improved and speeded up. Finally, this study would assess the added value of the selected tools and determine if they can assist architects in understanding unfamiliar systems at the architectural level. Such a high level of comprehension is needed when several existing systems need to interoperate together to form a SoS.

4.2 Participants

Five people agreed to participate to the study. Their backgrounds are listed in Table 7. Three of them (A, B, and C) were working for the SoS Section of DRDC Valcartier in the Computer Systems group or as Defence Scientist. They had respectively eight, five, and four years of experience. The two other participants (D and E) were consultants from Thales Systems Canada and Neosapiens working for DRDC Valcartier. They had twelve years of software development experience on large scale applications. None of the participants had previously used the three applications under examination nor participated in their development. Also, they had very limited knowledge about the applications domain.

Besides the participants, other individuals were involved in the study. These additional people played the roles of instructors and observers. These roles are described respectively in sections 4.4.1 and 4.4.3.

Table 7. *Participants' Backgrounds*

PARTICIPANT	DIPLOMA	YEARS OF EXPERIENCE	AREAS OF EXPERTISE
Participant A	B. Eng. (Comp. Eng.)	8	Systems Interoperability, Software Visualization
Participant B	B. Sc. A (Comp. Sc.)	5	Software Modelling, Software Simulation
Participant C	M. Comp. Sc.	4	Source Code Analysis, Reverse Engineering
Participant D	DEC (Comp. Sc.)	12	Software Development, Command and Control Systems
Participant E	B. Sc. A. (Comp. Sc.)	12	Software Development, Software Modelling

4.3 Applications under Study

As mentioned in the introduction, the study was performed on three military applications: HCI_CASE_ATTI, COPlanS, and ATS.

CASE_ATTI (Concept Analysis and Simulation Environment for Automatic Target Tracking and Identification) is a multi-sensor data fusion simulation test bed. It is used to analyze the performance of various multi-sensor data fusion architectures and algorithms for the Canadian Patrol Frigate. HCI_CASE_ATTI is the Human Computer Interface of CASE_ATTI and it was this component which was used for the study.

COPlanS (Collaborative Operations Planning System) is an integrated flexible suite of planning, decision-aid, and workflow management tools aimed at supporting the Military Operations Planning Process. It offers facilities such as: support a planning team involved in a distributed workflow; document the decision-making process; support decision analysis; and estimate readiness, operational cost as well as risk management.

ATS (Athene Tactical System) is a Command and Control Information System (C2IS) for the support of the Land Forces Command System. It was developed to support field commanders of all arms and services in the planning, directing as well as monitoring of their combat operations. ATS provides C2 tools for commanders and staff such as maneuver, combat support, and geographic information system functions.

4.4 Qualitative Study Design

Before the beginning of the study, the three tools were installed on two desktop computers in the OASIS lab. This was done in order to have simultaneously two participants performing the same comprehension tasks for each of the applications under examination. This would avoid reliance on the understanding of only one individual and as a result, provide more data for each of the studied application.

For each application examined, a predefined set of 31 comprehension tasks was performed. Completing all of them took one day of work. The participants started at 8:30 in the morning and finished at 16:00, with one hour for lunch. For each comprehension task, a time limit was set to make sure that the participants would at least try all of them. Also, to make the study more realistic, the participants did not have access to the applications' documentation. The reason is that in most cases, there is little or no documentation available and the one that exists probably does not reflect the current system implementation, due to drift and erosion [54, 55, 56]. In the cases where the documentation is up to date, it is usually too voluminous, very detailed, and does not provide the appropriate viewpoints and abstraction levels needed to understand the architecture. As a result, the source code is most of the time the only reliable source of information.

To maximize the results of this study, the participants were assigned to the different applications according to their level of experience with the two programming languages. The participants who felt more comfortable using C++ were assigned to ATS, while the ones who had more experience with Java performed the comprehension tasks on HCI_CASE_ATTI and COPlanS. Only one participant was assigned to more than one application: COPlanS and ATS.

Since the goal of the study was to assess the value added by the analysis tools, a point of comparison was needed. The point of comparison selected was an Integrated Development Environment (IDE). The reason it was chosen is because an IDE is the tool most commonly used these days for software development. Also, it supports unaided browsing of source code, without providing features especially targeted for software comprehension. For this reason, by first executing the comprehension tasks using only an IDE, the participants would be in a better position to evaluate the impact that the other tools had on their understanding. The selected IDEs were Eclipse 3.0 [45] and Visual C++ 6.0 [57]. Eclipse was used to perform the comprehension tasks for HCI_CASE_ATTI and COPlanS. Visual C++ was used in the case of ATS.

Table 8 summarizes the activities performed by the participants for each application under examination over a four-day period. Each of them is further detailed in the sections which follow.

***Table 8**. Schedule of Activities Performed for each Application*

DAY	ACTIVITIES PERFORMED
1	- Introduction of the application - Introduction of the application domain - Familiarization with the operation of the application
2	- Execution of the comprehension tasks using the IDE
3	- Execution of the comprehension tasks using the analysis tools
4	- Interview and debriefing of the participants with the observer

4.4.1 Training

The first step of the study consisted of providing group training to the participants to ensure that they had a sufficient working knowledge of the tools prior to the beginning of the study. This was necessary since only one participant had used previously two of the tools, i.e., Rational PureCoverage and Quantify. The instructors were people who had experience with these tools. There was one instructor for Headway reView and another one for Rational PureCoverage and Quantify. The training lasted one day in the case of reView and half a day for PureCoverage and Quantify. Since explaining all the functionalities of each tool would have taken too much time, the instructors focused on the ones they believed would be the more appropriate for performing the upcoming comprehension tasks. Group training was followed by sessions during which the participants practiced individually, but with the assistance of the instructors, what they had learned using two open source applications: JUnit and Notepad++. Their characteristics are indicated in Table 9. As mentioned previously, JUnit is a Java testing framework used to write and run repeatable tests. Notepad++ [58] is a source code editor which supports several programming languages. For the two IDEs, since the participants already had experience using them, only a brief refresher was provided.

***Table 9**. Applications Used for Training*

APPLICATION	LANGUAGE	NO. OF CLASSES	LINES OF CODE
JUnit	Java	90	5 K
Notepad++	C++	136	46 K

4.4.2 Familiarization

During the first day of the study, an architect who had participated in the development of the application under examination introduced it to the participants. He explained its domain to give them a general understanding of what the application did, without saying how it did it. For example, in the case of COPlanS, the person explained the operational planning process of the CF. This would help them later to map some source code elements to their corresponding concepts. It was believed to be a necessary first step due to the large size of the applications under examination and the fact that most of the participants had a very limited knowledge of their domains. Furthermore, it is expected to be a representative activity of many architecture reconstruction and comprehension efforts. The people who want to understand the application go see a user who knows its domain and operation.

This first part of the study lasted half a day. For the rest of the day, the participants made themselves familiar with the operation of the application. This was also believed necessary, since for some of the comprehension tasks, users had to generate execution traces. If they had questions, they could direct them to the architect.

4.4.3 Execution of the Comprehension Tasks

During the second day of the study, the comprehension tasks (see Section 4.6) were performed using only an IDE. The following day, the same tasks were performed, this time using both the static and dynamic analysis tools. During these two days, the architect who had provided the participants with information about the application domain was there, acting as an observer. There were three observers, one for each application. Their role was to clock the participants performing the comprehension tasks to make sure that they did not exceed the allocated time set for each of them. If the participants had questions concerning the tasks or tools, they would answer them. Also, to evaluate the level of comprehension achieved by the participants, the observers were asking them questions. They recorded their answers as well as any other observations. Due to their heavy responsibilities, people playing the role of observer needed to have a vast experience in software development. All of them had at least ten years of experience.

For each of the applications under study, the above steps were repeated. During the first day, an introduction of the application and its domain were presented to the participants. During the following day, they performed the comprehension tasks using an IDE. On the third day, they performed the same comprehension tasks, this time aided by the static and dynamic analysis tools.

While performing the comprehension tasks, the participants recorded their results for each of them, either manually or using a text editor and taking screen shots of the outputs generated by the tools.

4.4.4 Interview and Debriefing

After the third day of the study, the observer and the two participants met to conduct a post-mortem. To determine the level of comprehension achieved, the observer evaluated the results obtained by the participants for each of the comprehension tasks. The observer also asked them to provide a general appreciation of the comprehension gained by using the different functionalities provided by the tools.

4.5 Software Comprehension Charts

For each session of the study, the participants were provided with a software comprehension chart. This chart listed, for every task, its description, its expected output, as well as the time allocated for its completion. The participants were also given paper forms to record their results. In case they preferred to use a text editor instead and take screen shots of the outputs of the tools, they would save their results in folders specifically created for this purpose.

The observers were also given the same comprehension charts and forms to record their observations and remarks.

4.6 Comprehension Tasks

The comprehension tasks to be performed by the participants were the result of brainstorming sessions among the members of the OASIS project. During these sessions, a process to understand the architectures of existing systems to be integrated into a SoS was designed.

Although a number of software comprehension tasks are proposed in the software visualization and comprehension literature [59, 31, 29, 30, 60] a definitive set of typical comprehension tasks does not currently exist [61]. The proposed tasks were intended to be as close as possible to the comprehension process designed previously. They were designed without prior knowledge of the applications' source code. As a result, they could be reused in another study involving other applications and software analysis tools.

The purpose of some of the software comprehension tasks was to identify potential areas of interest for the understanding of the applications. Examples of such tasks are the metrics whose computation was requested, since they are useful when understanding software systems [60]. Once identified, these potential areas of interest could have been further analyzed if the participants had more time. This would have enhanced their comprehension.

Some of the tasks which follow are inspired by the large scale questions of Pacione et al. [32] and the overall understanding questions of Systä et al. [60]. The others are based on the experience of some of the authors who had to understand applications of considerable size in the past. The tasks take into consideration both the static and

dynamic aspects of the applications under examination. Also, they were intended to be as close as possible to the ones performed during an understanding effort at the architectural level on large scale software, with a focus on system interoperability.

Table 10. *Source Code Composition Tasks*

NO.	DESCRIPTION
1	Identify how the application is organized into components and sub-components. For each component, evaluate its size in terms of number of classes.
2	Identify a set of classes relevant to the application domain (e.g., mission, operation, country, and tracks).
3	Identify the classes containing an entry point. Among all the entry points found, identify the one which is most likely the main entry point of the application. Identify other important entry points if applicable.
4	Identify the components involved in interactions with end-users.
5	Identify the components involved in interactions with the file system.
6	Identify the components involved in interactions with external applications via network communications.
7	Identify the components accessing database management systems.
8	Identify the components involved in interactions with third party libraries.
9	Identify clusters of components which have high cohesion but low coupling.

Table 11. *Source Code Analysis Tasks*

NO.	DESCRIPTION
1	Find if there are dependency cycles between the components of the application.
2	Compute the abstractness of the components. Rank the largest components, in terms of number of classes, according to their abstractness, in decreasing order.
3	Compute the number of classes in the application.
4	Compute the coupling between object classes (CBO) of each major class.
5	Compute the afferent (Ca) and efferent (Ce) coupling of each major component.
6	Identify the components with the highest afferent and lowest efferent coupling.
7	Identify the components with the lowest afferent and highest efferent coupling.
8	Identify the components with the lowest afferent and efferent coupling.

Table 12. *Source Code Visualization Tasks*

NO.	DESCRIPTION
1	Show the overall structure of the application at the component level as well as the interaction dependencies between them.
2	Isolate a large group of classes (at least four) involved in an inheritance dependency and show the corresponding inheritance cluster of classes for each group.
3	Isolate a large group of classes (at least four) involved in an aggregation dependency and show the corresponding aggregation cluster of classes for each group.
4	Show a top-down component dependency hierarchy of the application.
5	Compute the layer of dependency of each component.

Table 13. *Execution Trace Visualization Tasks*

NO.	DESCRIPTION
1	Perform a representative run of the application and identify the creation/deletion of processes/threads using a call graph.
2	Using the call graph produced previously, describe the interactions between the different processes/threads.

Table 14. *Execution Trace Analysis Tasks*

NO.	DESCRIPTION
1	Execute a set of representative runs and identify the covered and non-covered areas of the application.
2	Execute a set of representative runs. Identify the most solicited areas of the application.
3	Identify the initialization hierarchy of the components.

Table 15. *Data Exchange Format Task*

NO.	DESCRIPTION
1	Analyze the data exchange format (e.g., binaries, serializable objects, and XML).

Table 16. *Reduction/Simplification Tasks*

NO.	DESCRIPTION
1	Extract a subset of information that is of interest for the user.
2	Identify the deepest inheritance tree in the application.
3	Identify the deepest composition/aggregation tree in the application.

5. Observations

As already mentioned, the purpose of some of the software comprehension tasks was to identify potential areas of interest for the understanding of the applications. Once these were identified, it would have been very beneficial for the comprehension of the participants to conduct further analysis. This analysis would have provided them with a much better understanding of the applications. However, due to time constraints, this was not possible. Despite the above limitation and the fact that some adjustments had to be made during the course of the study, it is still believed that it produced interesting results which will be useful for the upcoming phases of the OASIS project. The observations that can be drawn from the results of Appendix A are discussed next.

5.1 Cumulative Comprehension

As anticipated when the comprehension process was designed, the level of understanding achieved by the participants at the end of the study was not only due to the fact that they were aided by software static and dynamic analysis tools. It was also attributable to the fact that they studied the applications domain and performed the comprehension tasks by first using IDEs.

5.1.1 Familiarization

The introduction of the application provided to the participants during the first day as well as the information about its domain proved to be crucial for their comprehension. Without this initial background, the participants would not have been able to know the purpose of the different applications. This is because the examined applications were quite large, had been designed to fulfill very specific military needs, and the participants were not familiar with the applications domain.

5.1.2 IDEs

The searching capacities offered as part of the IDEs allowed the participants to perform several tasks. These were useful for top-down comprehension to find beacons while verifying hypotheses. For example, to locate the components interacting with end-users in COPlanS (Table 10, task 4), the participants only had to search for classes importing the java.awt and javax.swing packages. In cases similar to this one, the software analysis tools do not seem to provide much added value. Eclipse also offers other useful features such as the possibility to find all the occurrences in the source code which refer to a particular variable as well as get the type hierarchy of a class. It would have been very interesting to see if the participants would have achieved the same level of understanding with the software analysis tools without first having performed the comprehension tasks using an IDE.

5.1.3 Static and Dynamic Software Analysis Tools

Performing the tasks on the third day of the study was not only facilitated by the fact that participants were using static and dynamic software analysis tools. It was also easier since they could benefit from the knowledge acquired during the previous two days. In spite of this, the tools still provided information which would have been very difficult to obtain otherwise. For example, the dependency hierarchy in Headway reView was a very useful feature. As implied by its name, it provides the hierarchy of the high level components of an application as well as the dependencies between them. This allowed the participants to know instantly which components formed the core of an application and which ones were utility components. Figure 6 shows the dependency hierarchy of the high level packages in COPlanS. Furthermore, the suite of software metrics integrated with the visualization tool of Headway reView allowed participants to obtain accurate numbers characterizing properties of the source code very quickly.

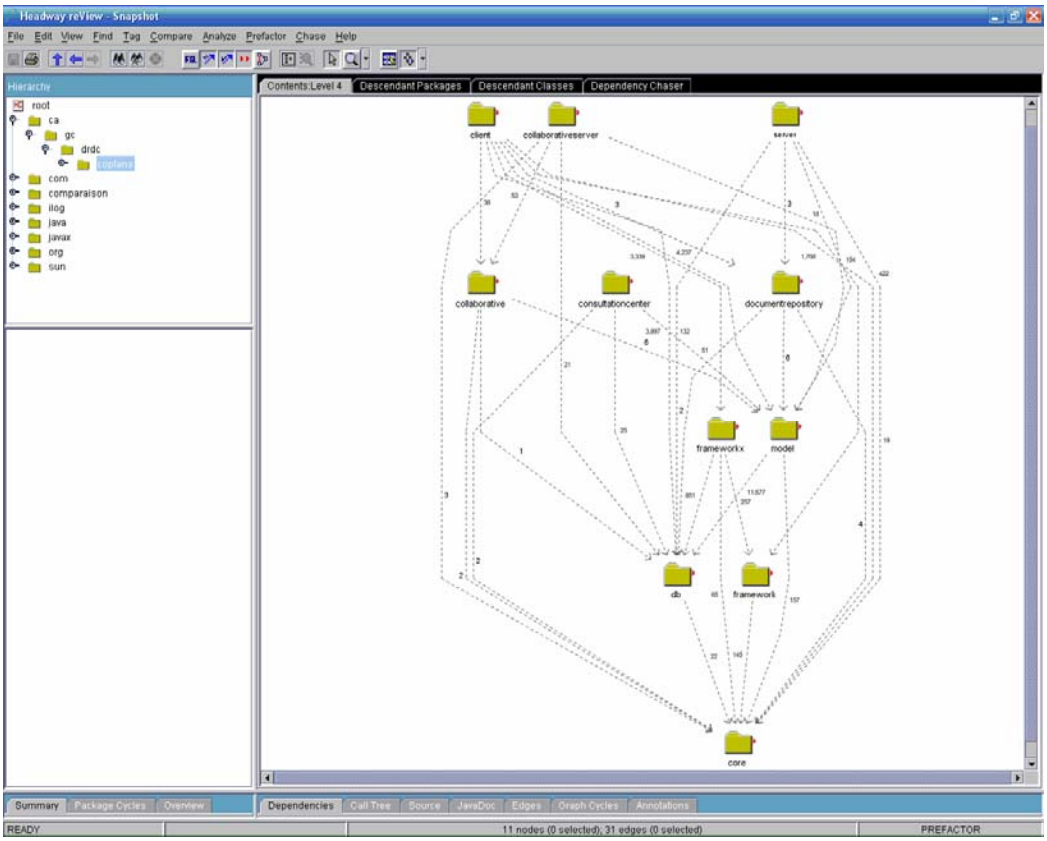


Figure 6. *COPlanS Packages Dependency Hierarchy in Headway reView*

The biggest drawback of the tools is that they do not always provide the appropriate viewpoints, abstraction levels, and filters needed to understand the architecture of an application. The participants were quickly swamped by a mass of irrelevant low level details. This is especially true in the case of Rational PureCoverage and Quantify, since most of the information provided is at the method level. However, one must take

into consideration that these tools were designed for software developers rather than people trying to recover and understand software architectures.

Figure 7 shows an example where the participants were overwhelmed by the large amount of information displayed in a single window. In the present Rational Quantify call graph, there is a large number of visible edges. Although Quantify offers some filters, as shown in Figure 8, for each desired filter, the participant had to find it first in the list box and then check it. Improving the filtering ability of the tools would increase their scalability. Also, applying more filters by default could improve the support for the as-needed comprehension strategy.

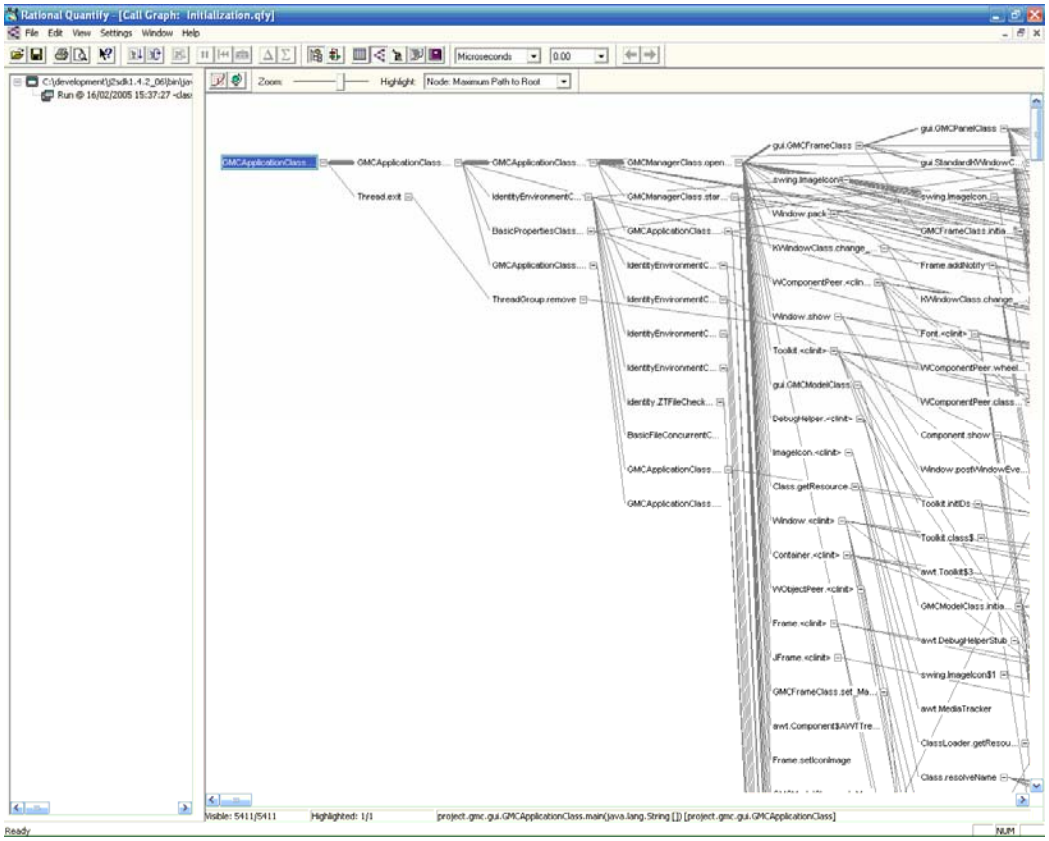


Figure 7. *A Rational Quantify Call Graph for HCI_CASE_ATTI*

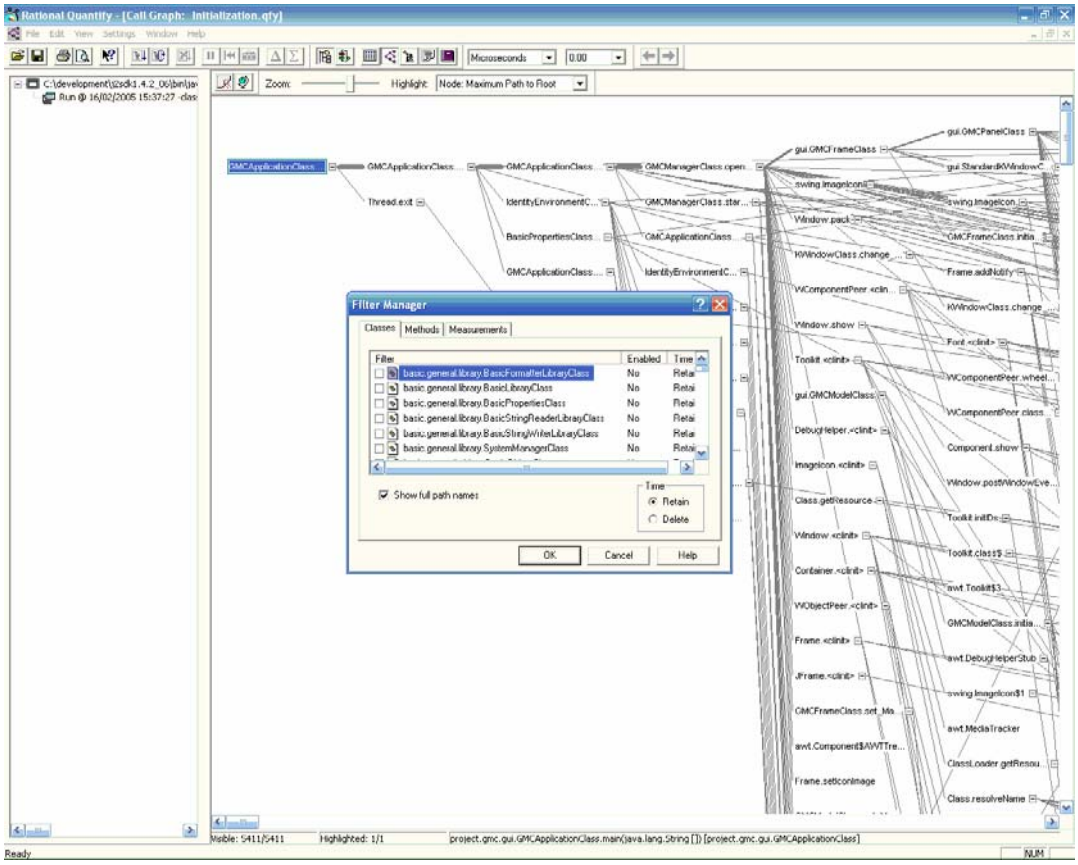


Figure 8. *Filter Manager in Rational Quantify*

5.2 Naming Conventions

The impact of identifier naming on real-life maintenance activities remains underestimated [62]. Naming rules do not go into much detail other than code formatting guidelines [63] or are not even treated at all in the context of code formatting and documenting [64].

One result which was observed as part of the present study is the great impact that meaningful names chosen for components, classes, and methods had on the comprehension of the participants. The reason is that the meaning conveyed by the names allowed them to map elements of the source code to their corresponding elements of the application domain. A consistent naming convention used throughout the application also facilitated greatly their understanding.

In order to alleviate the burden of identifier names on program comprehension, rules such as the ones derived in [62] should be followed in order to consistently and concisely name identifiers. These rules are based on a formal analysis of the properties of identifiers, names, concepts, and source code, as well as their interrelationships.

5.3 High Level Comprehension

Some tasks were more useful than others to achieve the objective of the qualitative study and comprehend the different applications at a high level. Tasks involving clustering (tasks 2 and 3 of Table 12) allowed participants to group related classes together. The second task of Table 11 highlighted where the abstractions, i.e., the most general components, were in the source code. Also, tasks 1, 4, and 5 of Table 12 gave the participants an overview of how an application was structured. For example, they emphasized which components were the high level ones accessing the others at the lower levels.

5.4 Combination of Information

One approach proved to be very useful in identifying precisely which components were responsible for a particular interoperability function. It was based on a technique called software reconnaissance [65]. This approach consisted of combining dynamic analysis, static analysis, as well as information about the operation of the application and naming conventions.

If a participant wanted to know, for example, which components were involved in interactions with external applications via network communications, he would first execute the application without invoking the function of interest and record the resulting execution trace using Rational Quantify. The participant would know how to invoke the interoperability function when required using what he had learned during the first day of the study. He would then execute the application again and record its execution trace, but this time, the participant would invoke the function interacting with external applications via network communications. Afterwards, the two recorded execution traces would be compared and the differences identified.

Using Headway reView, the classes responsible for the differences would be visualized and using the naming conventions identified during the fist day of the study as a guide, the participants would be able to clearly locate, in most cases, the involved components.

Having a tool which would integrate the high level static aspects with the dynamic ones at the same level would definitely be an advantage. This capability would allow a user to start his comprehension using the static analysis and then refine it using dynamic information.

5.5 Assumptions Validated

When the software comprehension tasks were designed, some assumptions were made with respect to the afferent and efferent coupling metrics. At that time, it was believed that the components with the highest afferent and lowest efferent coupling would be utility components. It was also believed that the ones with the lowest afferent and highest efferent coupling would be application domain components. Furthermore, the components with the lowest afferent and efferent coupling should be unused

components, i.e., dead code. The study proved these three assumptions to be valid. This result could be later used in future comprehension efforts.

5.6 Programming Languages

Some problems experienced with the tools are inherent to the programming languages. For example, in C++, a package is defined as the container for all the entities in a namespace. However, unlike its similar concept in Java, a C++ namespace is not based on physical directory structures. Despite this, its name should reflect the logical hierarchy of the package, such as package names in Java. However, this coding standard is not always followed. This complicates the understanding of C++ programs, since the physical structure does not always correspond to the logical one. This was confirmed in the present study. It has been more difficult to achieve the same comprehension level for ATS than for HCI_CASE_ATTI and COPlanS.

6. Limitations

For this qualitative study, a non-negligible amount of time was spent to train the participants in advance. This was done to address a limitation of the previous studies [29, 30], which provided limited training and resulted in the fact that the features provided by the tools were probably not fully exploited by the participants.

In the present study, the participants were also asked to perform a large number of tasks. Furthermore, only two of them could perform the comprehension tasks at the same time: there were only two desktop computers available in the OASIS lab with the required software, due to budget constraints. As a result, for all of the above reasons, the organizers of the study could not afford to have a large sampling of participants and therefore, a statistical analysis of the results is not possible.

One direct consequence of the above limitation is that the participants who performed the tasks with and without the software analysis tools were the same. Therefore, the comprehension of the application gained by using only the IDE on the second day of the study could have affected their assessment of the value added by the software analysis tools. This is because some of the tasks could be performed using only an IDE. A larger sampling would have allowed one participant to perform the tasks using only the IDE and another one using only the software analysis tools. It would also be interesting to follow the same logic and have participants perform the tasks with and without the information about the application and its domain provided on the first day of the study. This approach would allow assessment of the impact of each of these activities on the global comprehension level achieved by the participants.

Even though Headway reView was believed to be a good tool with a large set of functionalities, in hindsight it would have been better to have used more than one architectural recovery tool. However, the additional training required precluded such an approach.

7. Conclusions and Future Work

This technical memorandum describes the design and reports the observations of a qualitative study conducted to assess whether the use of three commercial software analysis tools enhanced architects' understanding. In this study, five participants were performing 31 high level comprehension tasks on three large scale object-oriented applications written in C++ and Java.

This qualitative study is different from the other studies [29, 30] and evaluations [31, 32] which were conducted in the past to explore the question as to whether or not reverse engineering and visualization tools enhance programmers' understanding. It was not performed in an academic setting, but with people having several years of experience in software development, who were properly trained on using the analysis tools and familiarized with the applications domain. It did not examine relatively small scale programs but military applications of a considerable size, taking in consideration both the static and dynamic aspects, as well as focussing at the architectural level and on system interoperability. Furthermore, the objective of the study was not to compare tools [30, 31, 32] or approaches [29], but to try to assess whether the use of software analysis tools can assist in the understanding of unfamiliar systems at the architectural level.

Although it was observed that the tools aided the participants to understand the applications under examination, some deficiencies were observed. These stem from the fact that the software analysis tools do not always provide the appropriate viewpoints, abstraction levels, and filters needed to understand the architecture of applications consisting of more than 1,000 classes. This is especially true in the case of the dynamic tools.

Following this qualitative study and using the theoretical and practical knowledge acquired through it, the next step will consist of developing a prototype. This prototype will address the limitations identified concerning the dynamic aspect. It will therefore provide the appropriate viewpoints, abstraction levels, and filters required for the visualization of dynamic information at the architectural level. These dynamic views will be integrated into an IDE providing static views of the source code. In addition, the prototype will offer functionalities to facilitate the mapping of source code elements to their corresponding concept of the application domain. Ideally, once this tool is developed, another study with an improved design and set of comprehension tasks should be conducted. Its objective would be to assess the added value of the tool on the comprehension of participants.

8. References

1. The Technical Cooperation Program - Joint Systems and Analysis Group, "The Engineering and Acquisition of Systems of Systems in the United States DoD," *Tech. Report TR-JSA-TP4-1-2001*, Jan. 2001.

2. D. Garlan and D.E. Perry, "Introduction to the Special Issue on Software Architecture," *IEEE Trans. On Software Eng.*, vol. 21 no. 4, Apr. 1995, pp. 269-274.

3. R. Richardson, et al., "A Survey of Research into Legacy System Migration," *Tech. Report TCD-CS-1997-01*, Trinity College Dublin, Dublin, Ireland, Jan. 1997.

4. I.T. Bowman and R.C. Holt, "Software Architecture Recovery Using Conway's Law," *Proc. of the 1998 Conf. of the Centre for Advanced Studies on Collaborative research (CASCON'98)*, Mississauga, Ont., Nov. 1998, pp. 123-133.

5. L. O'Brien and C. Stoermer, "Architecture Reconstruction Case Study," *Tech. Note CMU/SEI-2003-TN-008*, Carnegie Mellon Univ., Pittsburgh, Pa., Apr. 2003.

6. R.L. Krikhaar, "Software Architecture Reconstruction," *Ph.D. Thesis*, Univ. of Amsterdam, Amsterdam, The Netherlands, 1999.

7. C. Riva, "Reverse Architecting: An Industrial Experience Report," *Proc. of the 7th Working Conf. on Reverse Eng. (WCRE'00)*, Brisbane, Australia, Nov. 2000, pp. 42-51.

8. M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.

9. A. Trevors and M.W. Godfrey, "Architectural Reconstruction in the Dark," *Workshop on Software Architecture Reconstruction held in conjunction with the 9th Working Conf. on Reverse Eng. (WCRE'02)*, Richmond, Va., Oct., 2002.

10. K. Wong, "Structural Redocumentation: A Case Study," *IEEE Software*, vol.12, no.1, Jan. 1995, pp.46-54.

11. R. Clayton, S. Rugaber, and L. Wills, "Dowsing: A Tool Framework for Domain-Oriented Browsing of Software Artifacts," *Proc. of the 13th IEEE Int'l Conf. on Automated Software Eng.*, Honolulu, Hawaii, Oct. 1998, pp. 204-207.

12. J. Grundy and J. Hosking, "High-Level Static and Dynamic Visualization of Software Architectures," *Proc. of the 2000 IEEE Int'l Symp. On Visual Languages (VL'00)*, Seattle, Wash. Sept. 2000, pp. 5-12.

13. M. Lizotte and J. Rilling, “OASIS: Opening-up Architecture of Software-Intensive Systems”, *Proc. of the 24th Army Science Conf. (ASC’04)*, Orlando, Fla., Nov. 2004.

14. J. Rilling, “State of the Art Report: System Architecture Recovery and Comprehension,” *Tech. Report*, DRDC Valcartier, Val-Bélair, Que., 2003.

15. Headway reView, “Headway Software,” Dec. 2005; http://www.headwaysoftware.com/.

16. Rational PurifyPlus, “IBM Software,” Dec. 2005; http://www-306.ibm.com/software/awdtools/purifyplus/win/.

17. M.-A.D. Storey, “Theories, Methods and Tools in Program Comprehension: Past, Present and Future,” *Proc. of the 13th Int’l Workshop on Program Comprehension (IWPC’05)*, St. Louis, Mo., May 2005, pp. 181-191.

18. E. Soloway and K. Ehrlich, “Empirical Studies of Programming Knowledge,” *IEEE Trans. On Software Eng*., vol. 10, no. 5, Sept. 1984, pp. 595-609.

19. R. Brooks, “Towards a Theory of the Comprehension of Computer Programs,” *Int’l J. of Man-Machine Studies*, vol. 18, no. 6, June 1983, pp. 543-554.

20. B. Shneiderman and R. Mayer, “Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results,” *Int’l J. of Computer and Information Sciences*, vol. 8, no. 3, 1979, pp. 219-238.

21. B. Shneiderman, *Software Psychology: Human Factors in Computer and Information Systems*, Winthrop Publishers, 1980.

22. N. Pennington, “Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs,” *Cognitive Psychology*, vol. 19, 1987, pp. 295-341.

23. S. Letovsky, “Cognitive Processes in Program Comprehension,” *Proc. of the 1st Workshop on Empirical Studies of Programmers, Ablex Publishing*, 1986, pp. 58-79.

24. D.C. Littman, et al., “Mental Models and Software Maintenance,” *1st Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, Washington, D.C., 1986, pp. 80-98.

25. E. Soloway, et al., “Designing Documentation to Compensate for Delocalized Plans,” *Comm. Of the ACM*, vol. 31, no. 11, Nov. 1988, pp. 1259-1267.

26. A. von Mayrhauser and A.M. Vans, “From Code Understanding Needs to Reverse Engineering Tool Capabilities,” *Proc. of the 6th Int’l Conf. on Computer-Aided Software Engineering (CASE’93)*, Singapore, Jul. 1993, pp. 230-239.

27. M.-A.D. Storey, F.D. Fracchia, and H.A. Mueller, “Cognitive Design Elements to Support the Construction of a Mental Model during Software Visualization,” *Proc. of the 5th Int’l Workshop on Program Comprehension (IWPC’97)*, Dearborn, Mich., May 1997, pp. 17-28.

28. S.R. Tilley, S. Paul, D.B. Smith, “Towards a Framework for Program Understanding,” *Proc. of the 4th Int’l Workshop on Program Comprehension (IWPC’96)*, Berlin, Germany, Mar. 1996, pp. 19-28.

29. M.-A.D. Storey, et al., “On Designing an Experiment to Evaluate a Reverse Engineering Tool,” *Proc. of the 3rd Working Conf. on Reverse Eng. (WCRE’96)*, Monterey, Calif., Nov. 1996, pp. 31-40.

30. M.-A.D. Storey, K. Wong, and H.A. Müller, “How Do Program Understanding Tools Affect How Programmers Understand Programs?,” *J. Science of Computer Programming*, vol. 36, No. 2-3, Mar. 2000, pp. 183-207.

31. S. Elliott Sim and M.-A. D. Storey, “A Structured Demonstration of Program Comprehension Tools,” *Proc. of the 7th Working Conf. on Reverse Eng. (WCRE’00)*, Brisbane, Australia, Nov. 2000, pp. 184-193.

32. M.J. Pacione, M. Roper, and M. Wood, “A Comparative Evaluation of Dynamic Visualisation Tools,” *Proc. of the 10th Working Conf. on Reverse Eng. (WCRE’03)*, Victoria, B.C., Nov. 2003, pp. 80-89.

33. K.S. Mathias, “The Role of Software Measures and Metrics in Studies of Program Comprehension,” *Proc. of the 37th Ann. Southeast Regional Conf.*, Mobile, Ala., Apr. 1999.

34. ArgoUML, “Tigris.org: Open Source Software Engineering,” Dec. 2005; http://argouml.tigris.org/.

35. CodeSurfer, “GrammaTech,” Dec. 2005; http://www.grammatech.com/products/codesurfer/.

36. Columbus/CAN, “FrontEndART,” Dec. 2005; http://www.frontendart.com/.

37. Fujaba Tool Suite, “Fujaba,” Dec. 2005; http://wwwcs.uni-paderborn.de/cs/fujaba/.

38. Rational Rose, “IBM Rational Software,” Dec. 2005; http://www-306.ibm.com/software/rational/.

39. SniFF+, “Wind River,” Dec. 2005; http://www.windriver.com/products/development_tools/ide/sniff_plus/.

40. R. Kazman, L. O’Brien, and C. Verhoef, “Architecture Reconstruction Guidelines,” *Tech. Report CMU/SEI-2001-TR-026*, Carnegie Mellon Univ., Pittsburgh, Pa., Aug. 2001.

41. S.R. Tilley and D.B. Smith, “Coming Attractions in Program Understanding,” *Tech. Report CMU/SEI-96-TR-019*, Software Engineering Institute, Pittsburgh, Pa., 1996.

42. M.-A.D. Storey, C. Best, and J. Michaud, “SHriMP Views: An Interactive Environment for Exploring Java Programs,” *Proc. of the 9th Int’l Workshop on Program Comprehension (IWPC’01)*, Toronto, Ont., May 2001, pp. 111-112.

43. L. O’Brien, “Architecture Reconstruction to Support a Product Line Effort: Case Study,” *Tech. Note CMU/SEI-2001-TN-015*, Carnegie Mellon Univ., Pittsburgh, Pa., Jul. 2001.

44. L. O’Brien and V. Tamarree, “Architecture Reconstruction of J2EE Applications: Generating Views from the Module Viewtype,” *Tech. Note CMU/SEI-2003-TN-028*, Carnegie Mellon Univ., Pittsburgh, Pa., Nov. 2003.

45. Eclipse, “Eclipse.org Main Page,” Dec. 2005; http://www.eclipse.org/.

46. Creole, “Creole - Home | the CHISEL group,” Dec. 2005; http://www.thechiselgroup.org/creole.

47. Java Extractor, “Java Extractor | the CHISEL group,” Dec. 2005; http://www.thechiselgroup.org/?q=stand-alone-shrimp/extractor.

48. Creole, “Creole - Download and Get Started | the CHISEL group,” Dec. 2005; http://www.thechiselgroup.org/creole/getting-started.

49. Aqtime 4, “Aqtime 4 - Automated Profiling and Debugging,” Dec. 2005; http://www.automatedqa.com/products/aqtime/.

50. Quest Jprobe, “Quest : Products : Jprobe : Overview,” Dec. 2005; http://www.quest.com/jprobe/.

51. Borland Optimizeit, “Borland : Optimizeit Enterprise Suite,” Dec. 2005; http://www.borland.com/us/products/optimizeit/.

52. ej-technologies Jprofiler, “Java Profiler - Jprofiler,” Dec. 2005; http://www.ej-technologies.com/products/jprofiler/overview.html.

53. JUnit, “JUnit, Testing Resources for Extreme Programming,” Dec. 2005; http://www.junit.org/.

54. P. Clements, et al., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley, 2002.

55. A. van Deursen and T. Kuipers, “Building Documentation Generators,” *Proc. of the Int’l Conf. on Software Maintenance (ICSM’99)*, Oxford, United Kingdom, Sept. 1999, pp. 40-49.

56. A. van Deursen, “Software Architecture Recovery and Modelling,” *ACM SIGAPP Applied Computing Rev.*, vol. 10, no. 1, Spring 2002, pp. 4-7.

57. Visual C++, “Visual C++ Developer Center,” Dec. 2005; http://msdn.microsoft.com/visualc/.

58. Notepad++, “NOTEPAD++,” Dec. 2005; http://notepad-plus.sourceforge.net/.

59. D. Kirk, M. Roper, and M. Wood, “Understanding Object-Oriented Frameworks - An Exploratory Case Study,” *Tech. Report EfoCS-42-2001*, Univ. of Strathclyde, Glasgow, Scotland, 2001.

60. T. Systä, K. Koskimies, and H. Müller, “Shimba - An Environment for Reverse Engineering Java Software Systems,” *Software Practice & Experience*, vol. 31, no. 4, Apr. 2001, pp. 371-394.

61. M.J. Pacione, M. Roper, and M. Wood, “A Novel Software Visualisation Model to Support Software Comprehension,” *Proc. of the 11th Working Conf. on Reverse Eng. (WCRE’04)*, Delft, the Netherlands, Nov. 2004, pp. 70-79.

62. F. Deißenböck and M. Pizka, “Concise and Consistent Naming,” *Proc. of the 13th Int’l Workshop on Program Comprehension (IWPC’05)*, St. Louis, Mo., May 2005, pp. 97-106.

63. P.W. Oman and C.R. Crook, “Typographic Style is More than Cosmetic,” *Comm. Of the ACM*, vol. 33, no. 5, May 1990, pp. 506-520.

64. M. Arab, “Enhancing Program Comprehension: Formatting and Documenting,” *ACM SIGPLAN Notices*, vol. 27, no. 2, Feb. 1992, pp. 37-46.

65. N. Wilde and C. Casey, “Early Field Experience with the Software Reconnaissance Technique for Program Comprehension,” *Proc. of the Int’l Conf. on Software Maintenance (ICSM’96)*, Monterey, Calif., Nov. 1996, pp. 312-318.

9. Appendix A

Following are the results which were obtained as part of this study for each of the application under examination and for every comprehension task. The results are presented in the form of bar charts with two dependent variables.

In the bar charts, the independent variable on the horizontal axis is the task performed. Below each graph, there is a caption which refers each task to its corresponding description in Section 4. The first dependent variable is the percentage of the comprehension task which was achieved. The values for this variable were set according to what the observers noticed during the study and to what the participants said during the interview afterwards. The second dependent variable is the time spent on this task. For example, for the first bar chart of section 9.1, the participants spent 20 minutes for the first task (Table 10 task 1). With only an IDE, they were able to complete only 70% of the task, while using the analysis tools, they were able to complete it in its entirety.

At the time the comprehension tasks were designed, a predefined amount a time was set for each of them. This was done to ensure that the participants did not spent too much time on a particular task and had enough time to at least try all of them. Unfortunately, during the course of the study, the observers realized that the time allocated for some tasks was too short. As a result, a few comprehension tasks, which were considered less important, had to be skipped so that all the work could be carried out within one day of work.

In the bar charts, the tasks in white were performed using an IDE, while the ones in dark gray were performed using the static and dynamic analysis tools.

9.1 HCI_CASE_ATTI Results

Table 17. *Source Code Composition Tasks*

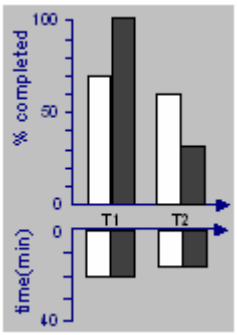
BAR CHART	DESCRIPTION
	**T1:** Identify how the application is organized into components and sub-components. For each component, evaluate its size in terms of number of classes. **T2:** Identify a set of classes relevant to the application domain (e.g., mission, operation, country, and tracks).

Table 18. *Source Code Composition Tasks (Continued)*

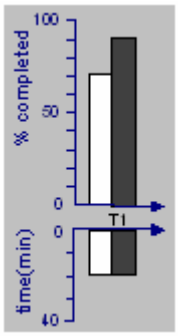
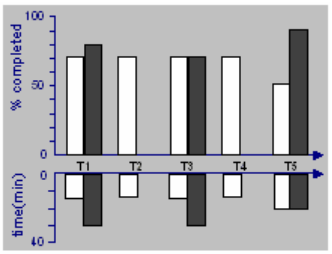
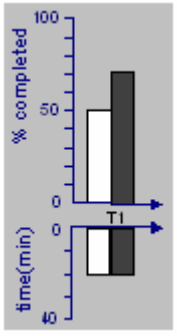
BAR CHART	DESCRIPTION
	**T1:** Identify the classes containing an entry point. Among all the entry points found, identify the one which is most likely the main entry point of the application. Identify other important entry points if applicable.
	**T1:** Identify the components involved in interactions with end-users. **T2:** Identify the components involved in interactions with the file system. **T3:** Identify the components involved in interactions with external applications via network communications. **T4:** Identify the components accessing database management systems. **T5:** Identify the components involved in interactions with third party libraries. **Note:** T2 and T4 were not performed using the analysis tools because they are not applicable in the case of HCI_CASE_ATTI. These tasks respectively require to identify the components involved in interactions with the file system and accessing database management systems. Unfortunately, HCI_CASE_ATTI relies on another component developed in C++ for its interactions with the file system. Also, it does not access any databases. This was realized after having first performed these tasks using the IDE.
	**T1:** Identify clusters of components which have high cohesion but low coupling.

Table 19. *Source Code Analysis Tasks*

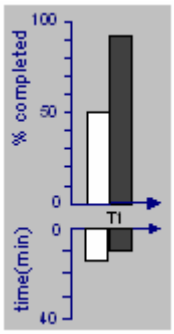
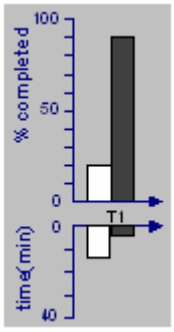
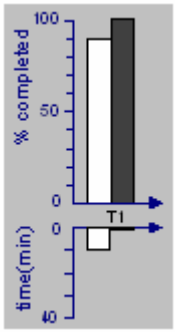
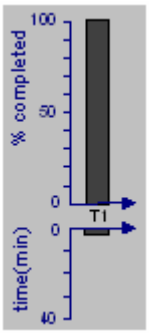
BAR CHART	DESCRIPTION
	**T1:** Find if there are dependency cycles between the components of the application.
	**T1:** Compute the abstractness of the components. Rank the largest components, in terms of number of classes, according to their abstractness, in decreasing order.
	**T1:** Compute the number of classes in the application.
	**T1:** Compute the coupling between object classes (CBO) of each major class. **Note:** T1 was not performed using the IDE. It requires the computation of a metric for each major class. Given the size of the application and the time constraints, it would not have been possible to compute this metric using only an IDE.

Table 20. *Source Code Analysis Tasks (Continued)*

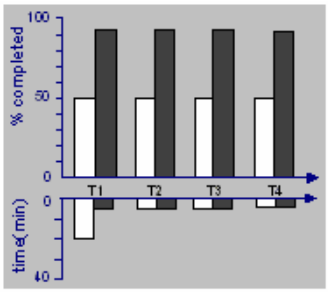
BAR CHART	DESCRIPTION
	**T1:** Compute the afferent (Ca) and efferent (Ce) coupling of each major component. **T2:** Identify the components with the highest afferent and lowest efferent coupling. **T3:** Identify the components with the lowest afferent and highest efferent coupling. **T4:** Identify the components with the lowest afferent and efferent coupling.

Table 21. *Source Code Visualization Tasks*

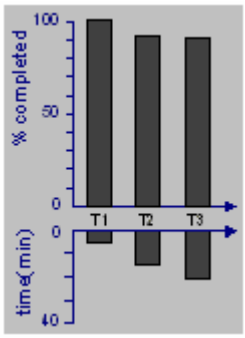
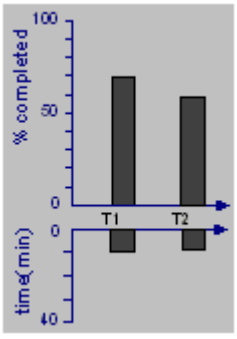
BAR CHART	DESCRIPTION
	**T1:** Show the overall structure of the application at the component level as well as the interaction dependencies between them. **T2:** Isolate a large group of classes (at least four) involved in an inheritance dependency and show the corresponding inheritance cluster of classes for each group. **T3:** Isolate a large group of classes (at least four) involved in an aggregation dependency and show the corresponding aggregation cluster of classes for each group. **Note:** T1, T2, and T3 were not performed using the IDE. They require the analysis of a large quantity of source code and it would not have been possible to perform them given the time constraints.
	**T1:** Show a top-down component dependency hierarchy of the application. **T2:** Compute the layer of dependency of each component. **Note:** T1 and T2 were not performed using the IDE. They require the analysis of a large quantity of source code and it would not have been possible to perform them given the time constraints.

Table 22. *Execution Trace Visualization Tasks*

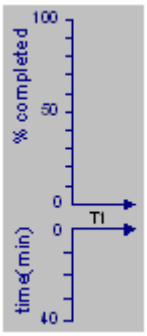
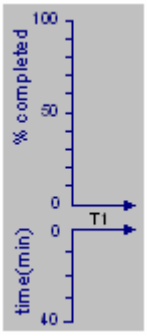
BAR CHART	DESCRIPTION
	**T1:** Perform a representative run of the application and identify the creation/deletion of processes/threads. **Note:** T1 was skipped during the course of the study as it seemed less important and to allow the participants to put their efforts on other ones.
	**T1:** Describe the interactions between the different processes/threads. **Note:** T1 was skipped during the course of the study as it seemed less important and to allow the participants to put their efforts on other ones.

Table 23. *Execution Trace Analysis Tasks*

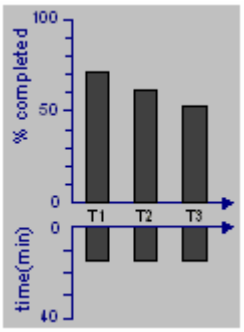
BAR CHART	DESCRIPTION
	**T1:** Execute a set of representative runs and identify the covered and non-covered areas of the application. **T2:** Execute a set of representative runs. Identify the most solicited areas of the application. **T3:** Identify the initialization hierarchy of the components. **Note:** T1, T2, and T3 were not performed using the IDE. They require dynamic analysis features that an IDE cannot provide.

Table 24. *Data Exchange Format Task*

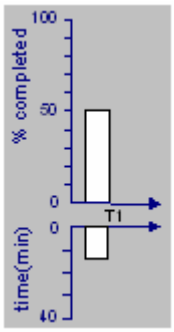
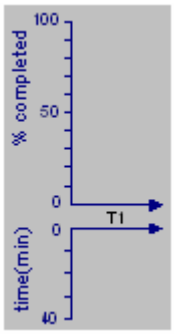
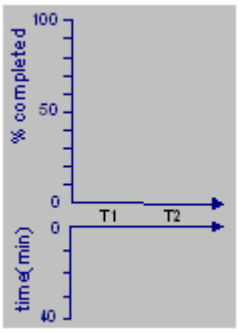
BAR CHART	DESCRIPTION
	**T1:** Analyze the data exchange format (e.g., binaries, serializable objects, and XML). **Note:** T1 was not performed using the analysis tools due to time constraints.

Table 25. *Reduction/Simplification Tasks*

BAR CHART	DESCRIPTION
	**T1:** Extract a subset of information that is of interest for the user. **Note:** T1 was skipped during the course of the study as it seemed less important and to allow the participants to put their efforts on other tasks.
	**T1:** Identify the deepest inheritance tree in the application. **T2:** Identify the deepest composition/aggregation tree in the application. **Note:** T1 and T2 were skipped during the course of the study as they seemed less important and to allow the participants to put their efforts on other tasks.

9.2 COPlanS Results

Table 26. *Source Code Composition Tasks*

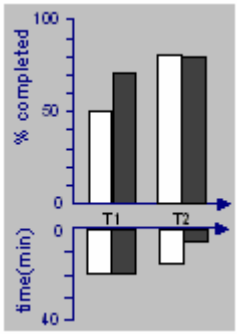
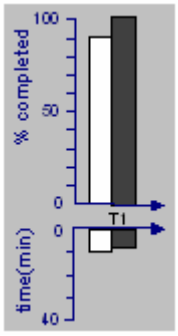
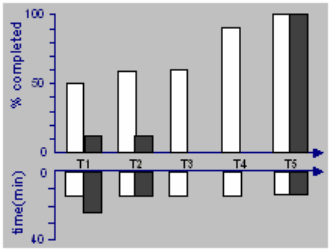
BAR CHART	DESCRIPTION
	**T1:** Identify how the application is organized into components and sub-components. For each component, evaluate its size in terms of number of classes. **T2:** Identify a set of classes relevant to the application domain (e.g., mission, operation, country, and tracks).
	**T1:** Identify the classes containing an entry point. Among all the entry points found, identify the one which is most likely the main entry point of the application. Identify other important entry points if applicable.
	**T1:** Identify the components involved in interactions with end-users. **T2:** Identify the components involved in interactions with the file system. **T3:** Identify the components involved in interactions with external applications via network communications. **T4:** Identify the components accessing database management systems. **T5:** Identify the components involved in interactions with third party libraries. **Note:** For T1 and T2, several problems were experienced with the dynamic analysis tools. T3 was not performed using the dynamic analysis tools, as it was discovered that the network could not be accessed. T4 was also not performed using the dynamic analysis tools. COPlanS uses a Tomcat servlet service to access a database and PureCoverage cannot track calls inside Tomcat.

Table 27. *Source Code Composition Tasks (Continued)*

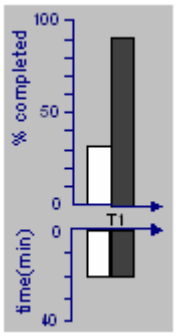
BAR CHART	DESCRIPTION
	**T1:** Identify clusters of components which have high cohesion but low coupling.

Table 28. *Source Code Analysis Tasks*

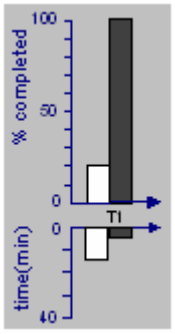
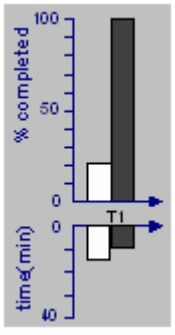
BAR CHART	DESCRIPTION
	**T1:** Find if there are dependency cycles between the components of the application.
	**T1:** Compute the abstractness of the components. Rank the largest components, in terms of number of classes, according to their abstractness, in decreasing order.

Table 29. *Source Code Analysis Tasks (Contiued)*

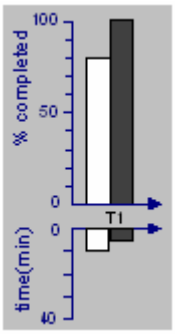
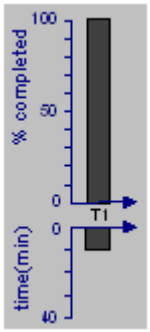
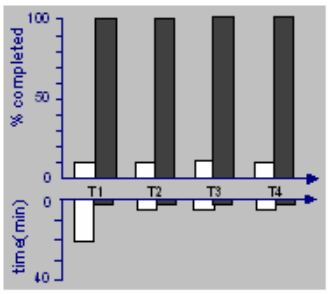
BAR CHART	DESCRIPTION
	**T1:** Compute the number of classes in the application.
	**T1:** Compute the coupling between object classes (CBO) of each major class. **Note:** T1 was not performed using the IDE. It requires the computation of a metric for each major class. Given the size of the application and the time constraints, it would not have been possible to compute this metric using only an IDE.
	**T1:** Compute the afferent (Ca) and efferent (Ce) coupling of each major component. **T2:** Identify the components with the highest afferent and lowest efferent coupling. **T3:** Identify the components with the lowest afferent and highest efferent coupling. **T4:** Identify the components with the lowest afferent and efferent coupling.

Table 30. *Source Code Visualization Tasks*

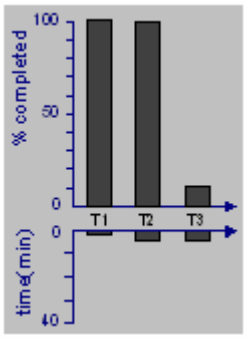
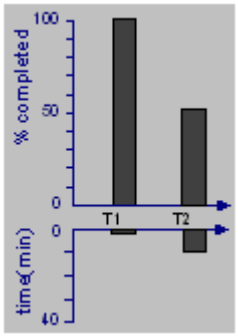
BAR CHART	DESCRIPTION
	**T1:** Show the overall structure of the application at the component level as well as the interaction dependencies between them. **T2:** Isolate a large group of classes (at least four) involved in an inheritance dependency and show the corresponding inheritance cluster of classes for each group. **T3:** Isolate a large group of classes (at least four) involved in an aggregation dependency and show the corresponding aggregation cluster of classes for each group. **Note:** T1, T2, and T3 were not performed using the IDE. They require the analysis of a large quantity of source code and it would not have been possible to perform them given the time constraints.
	**T1:** Show a top-down component dependency hierarchy of the application. **T2:** Compute the layer of dependency of each component. **Note:** T1 and T2 were not performed using the IDE. They require the analysis of a large quantity of source code and it would not have been possible to perform them given the time constraints.

Table 31. *Execution Trace Visualization Tasks*

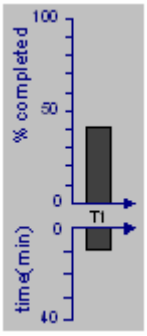
BAR CHART	DESCRIPTION
	**T1:** Perform a representative run of the application and identify the creation/deletion of processes/threads. **Note:** T1 was not performed using the IDE. They require dynamic analysis features that an IDE cannot provide.

Table 32. *Execution Trace Visualization Tasks (Continued)*

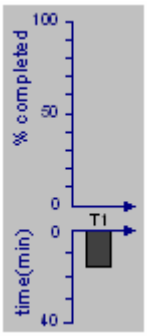
BAR CHART	DESCRIPTION
	**T1:** Describe the interactions between the different processes/threads. **Note:** T1 was aborted during the study, as the participants were not able to identify the classes involved.

Table 33. *Execution Trace Analysis Tasks*

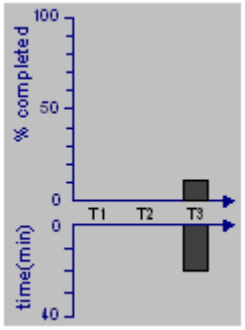
BAR CHART	DESCRIPTION
	**T1:** Execute a set of representative runs and identify the covered and non-covered areas of the application. **T2:** Execute a set of representative runs. Identify the most solicited areas of the application. **T3:** Identify the initialization hierarchy of the components. **Note:** T1, T2, and T3 were not performed using the IDE. They require dynamic analysis features that an IDE cannot provide. Also, T1 and T2 were not performed using the dynamic analysis tools. It would have required a complete execution of COPlanS, something which can take up to several hours.

Table 34. *Data Exchange Format Task*

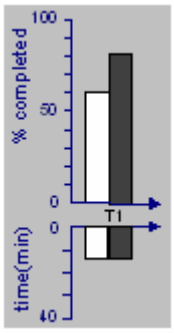
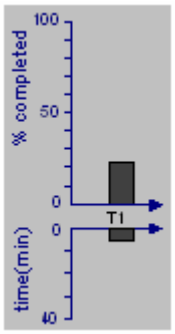
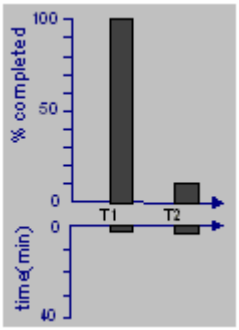
BAR CHART	DESCRIPTION
	**T1:** Analyze the data exchange format (e.g., binaries, serializable objects, and XML).

Table 35. *Reduction/Simplification Tasks*

BAR CHART	DESCRIPTION
	**T1:** Extract a subset of information that is of interest for the user. **Note:** T1 was not performed using the IDE. It requires the analysis of a large quantity of source code and it would not have been possible to perform it given the time constraints.
	**T1:** Identify the deepest inheritance tree in the application. **T2:** Identify the deepest composition/aggregation tree in the application. **Note:** T1 and T2 were not performed using the IDE. They require the analysis of a large quantity of source code and it would not have been possible to perform them given the time constraints.

9.3 ATS Results

Table 36. *Source Code Composition Tasks*

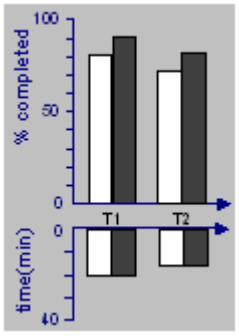
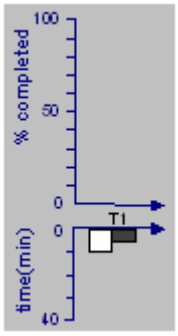
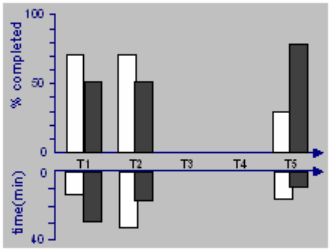
CHART	DESCRIPTION
	**T1:** Identify how the application is organized into components and sub-components. For each component, evaluate its size in terms of number of classes. **T2:** Identify a set of classes relevant to the application domain (e.g., mission, operation, country, and tracks).
	**T1:** Identify the classes containing an entry point. Among all the entry points found, identify the one which is most likely the main entry point of the application. Identify other important entry points if applicable.
	**T1:** Identify the components involved in interactions with end-users. **T2:** Identify the components involved in interactions with the file system. **T3:** Identify the components involved in interactions with external applications via network communications. **T4:** Identify the components accessing database management systems. **T5:** Identify the components involved in interactions with third party libraries. **Note:** T3 and T4 were not performed. The interactions with external applications via network communications as well as the interactions with databases are management by a COM object, the Data Service Layer (DSL).

Table 37. *Source Code Composition Tasks (Continued)*

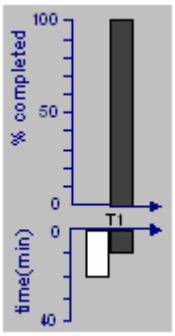
CHART	DESCRIPTION
	**T1:** Identify clusters of components which have high cohesion but low coupling.

Table 38. *Source Code Analysis Tasks*

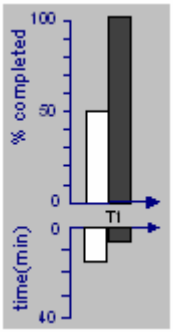
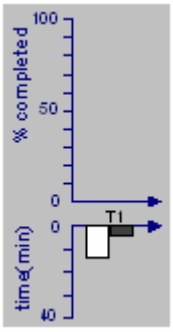
CHART	DESCRIPTION
	**T1:** Find if there are dependency cycles between the components of the application.
	**T1:** Compute the abstractness of the components. Rank the largest components, in terms of number of classes, according to their abstractness, in decreasing order.

Table 39. *Source Code Analysis Tasks (Contiuned)*

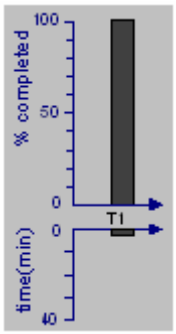
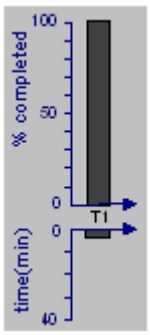
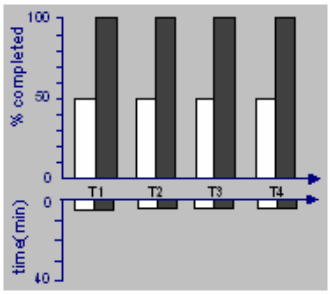
CHART	DESCRIPTION
	**T1:** Compute the number of classes in the application. **Note:** T1 was not performed using the IDE due to time constraints.
	**T1:** Compute the coupling between object classes (CBO) of each major class.
	**T1:** Compute the afferent (Ca) and efferent (Ce) coupling of each major component. **T2:** Identify the components with the highest afferent and lowest efferent coupling. **T3:** Identify the components with the lowest afferent and highest efferent coupling. **T4:** Identify the components with the lowest afferent and efferent coupling.

Table 40. *Source Code Visualization Tasks*

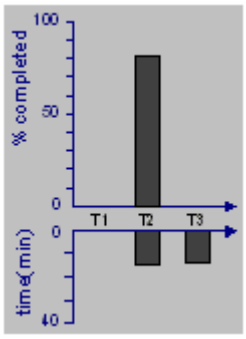
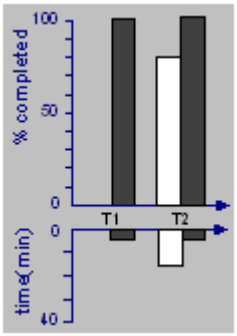
CHART	DESCRIPTION
	**T1:** Show the overall structure of the application at the component level as well as the interaction dependencies between them. **T2:** Isolate a large group of classes (at least four) involved in an inheritance dependency and show the corresponding inheritance cluster of classes for each group. **T3:** Isolate a large group of classes (at least four) involved in an aggregation dependency and show the corresponding aggregation cluster of classes for each group. **Note:** T2 and T3 were not performed using the IDE. They require the analysis of a large quantity of source code and it would not have been possible to perform them given the time constraints. T1 was skipped by mistake.
	**T1:** Show a top-down component dependency hierarchy of the application. **T2:** Compute the layer of dependency of each component. **Note:** T1 was not performed using the IDE. It requires the analysis of a large quantity of source code and it would not have been possible to perform them given the time constraints.

Table 41. *Execution Trace Visualization Tasks*

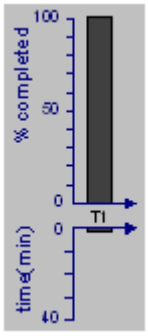
CHART	DESCRIPTION
	**T1:** Perform a representative run of the application and identify the creation/deletion of processes/threads. **Note:** T1 was not performed using the IDE. It requires dynamic analysis features that an IDE cannot provide.

Table 42. *Execution Trace Visualization Tasks (Continued)*

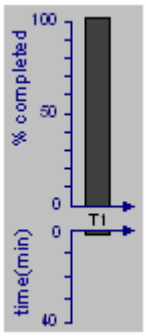
CHART	DESCRIPTION
	**T1:** Describe the interactions between the different processes/threads. **Note:** T1 was not performed using the IDE. It requires dynamic analysis features that an IDE cannot provide.

Table 43. *Execution Trace Analysis Tasks*

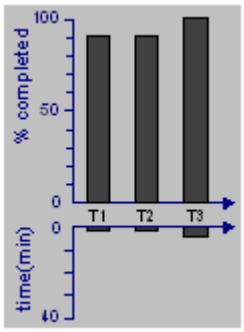
CHART	DESCRIPTION
	**T1:** Execute a set of representative runs and identify the covered and non-covered areas of the application. **T2:** Execute a set of representative runs. Identify the most solicited areas of the application. **T3:** Identify the initialization hierarchy of the components. **Note:** T1, T2, and T3 were not performed using the IDE. They require dynamic analysis features that an IDE cannot provide.

Table 44. *Data Exchange Format Task*

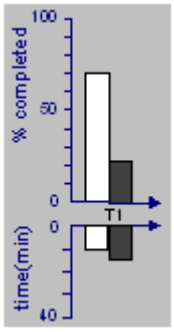
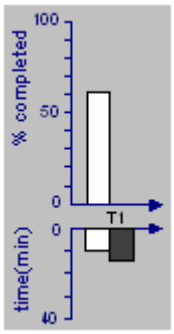
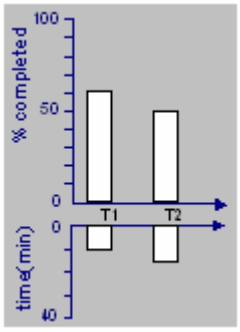
CHART	DESCRIPTION
	**T1:** Analyze the data exchange format (e.g., binaries, serializable objects, and XML).

Table 45. *Reduction/Simplification Tasks*

CHART	DESCRIPTION
	**T1:** Extract a subset of information that is of interest for the user.
	**T1:** Identify the deepest inheritance tree in the application. **T2:** Identify the deepest composition/aggregation tree in the application. **Note:** T1 and T2 were not performed using the analysis tools due to time constraints.

10. List of Acronyms

ARMIN	Architecture Reconstruction and MINing
ATS	Athene Tactical System
C2	Command and Control
C2IS	Command and Control Information System
C4ISR	Command, Control, Communications, Computers, Intelligence Surveillance and Reconnaissance
Ca	Afferent Coupling
CASE_ATTI	Concept Analysis and Simulation Environment for Automatic Target Tracking and Identification
CBO	Coupling between Object
CAD	Canadian Dollar
Ce	Efferent Coupling
CF	Canadian Forces
COPlanS	Collaborative Operations Planning System
DSL	Data Service Layer
DND	Department of National Defence
EDI	Environnement de Développement Intégré
FC	Forces canadiennes
HCI_CASE_ ATTI	Human Computer Interface Concept Analysis and Simulation Environment for Automatic Target Tracking and Identification
IDE	Integrated Development Environment
OASIS	Opening up Architecture of Software Intensive Systems
RSF	Rigi Standard Format

SHriMP	Simple Hierarchical Multi-Perspective
SoS	System of Systems
USD	United States Dollar

11. Distribution List

INTERNAL DISTRIBUTION

1 - Director General

3 - Document Library

1 - Head, System of Systems

1 - Philippe Charland (author)

1 - Dany Dessureault (author)

1 - Michel Lizotte (author)

1 - David Ouellet (author)

1 - Christophe Nécaille (author)

1 - Head, Information and Knowledge Management

1 - François Lemieux

1 - Martin Salois

EXTERNAL DISTRIBUTION

1 – DRDKIM (PDF file)

DOCUMENT CONTROL DATA

1. ORIGINATOR (name and address)
Defence Research and Development Canada Valcartier
2459, Pie-XI Blvd North
Québec, Quebec
G3J 1X5 Canada

2. SECURITY CLASSIFICATION
(Including special warning terms if applicable)
Unclassified

3. TITLE (Its classification should be indicated by the appropriate abbreviation (S, C, R or U)
Using software analysis tools to understand military applications: A qualitative study (U)

4. AUTHORS (Last name, first name, middle initial. If military, show rank, e.g. Doe, Maj. John E.)
Charland, Philippe; Dessureault, Dany; Lizotte, Michel; Ouellet, David; Nécaille, Christophe

5. DATE OF PUBLICATION (month and year)
August 2006

6a. NO. OF PAGES
73

6b .NO. OF REFERENCES
65

7. DESCRIPTIVE NOTES (the category of the document, e.g. technical report, technical note or memorandum. Give the inclusive dates when a specific reporting period is covered.)
Technical Memorandum

8. SPONSORING ACTIVITY (name and address)
Defence Research and Development Canada Valcartier
2459, Pie-XI Blvd North
Val-Bélair, Québec
G3J 1X5 Canada

9a. PROJECT OR GRANT NO. (Please specify whether project or grant)
15ak40

9b. CONTRACT NO.

10a. ORIGINATOR'S DOCUMENT NUMBER
DRDC Valcartier TM 2005-425

10b. OTHER DOCUMENT NOS
N/A

11. DOCUMENT AVAILABILITY (any limitations on further dissemination of the document, other than those imposed by security classification)

- ☒ Unlimited distribution
- ☐ Restricted to contractors in approved countries (specify)
- ☐ Restricted to Canadian contractors (with need-to-know)
- ☐ Restricted to Government (with need-to-know)
- ☐ Restricted to Defense departments
- ☐ Others

12. DOCUMENT ANNOUNCEMENT (any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in 11) is possible, a wider announcement audience may be selected.)

13. ABSTRACT (a brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual).

Although some studies have already been conducted to evaluate the effect of reverse engineering and visualization tools on programmers' understanding, most of them were conducted under conditions which do not prevail in the industry. They involved undergraduate and graduate students performing comprehension tasks on relatively small scale programs. Also, they either focused exclusively on the static or dynamic aspect of the software under examination. This technical memorandum describes the design and reports the observations of a qualitative study conducted to assess the value added by one reverse engineering and two dynamic analysis tools. The software examined were three large scale military applications written in C++ and Java. In this study, five participants had to perform 31 comprehension tasks, taking into consideration both the static and dynamic aspects of the applications under examination. The tasks were intended to be as close as possible to the ones performed during an understanding effort at the architectural level on large scale software. Although it was observed that the tools aided the participants to understand the applications under examination, some deficiencies were observed. These stem from the fact that the tools do not always provide the appropriate viewpoints, abstraction levels, and filters needed to understand the architecture of applications of considerable size. This is especially true in the case of the dynamic tools.

14. KEYWORDS, DESCRIPTORS or IDENTIFIERS (technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus-identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.)

Software architecture recovery, program comprehension, program understanding tools, reverse engineering, user study.

Defence R&D Canada

Canada's Leader in Defence
and National Security
Science and Technology

R & D pour la défense Canada

Chef de file au Canada en matière
de science et de technologie pour
la défense et la sécurité nationale

WWW.drdc-rddc.gc.ca