Investigating Software Reconnaissance as a Technique to Support Feature Location and Program Analysis Tasks using Sequence Diagrams

by

Sean Stevenson
B.Eng., University of Pretoria, 2008
B.Eng., University of Pretoria, 2010

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Investigating Software Reconnaissance as a Technique to Support Feature Location and Program Analysis Tasks using Sequence Diagrams

by

Sean Stevenson
B.Eng., University of Pretoria, 2008
B.Eng., University of Pretoria, 2010

Supervisory Committee

Dr. Margaret-Anne Storey, Co-Supervisor
(Department of Computer Science)

Dr. Daniel German, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Margaret-Anne Storey, Co-Supervisor
(Department of Computer Science)

Dr. Daniel German, Co-Supervisor
(Department of Computer Science)

ABSTRACT

Software reconnaissance is a very useful technique for locating features in software systems that are unfamiliar to a developer. The technique was, however, limited by the need to execute multiple test cases and record the components used in each one. Tools that recorded the execution traces of a program made it more practical to use the software reconnaissance technique. Diver was developed as an execution trace visualization tool using sequence diagrams to display the dynamic behaviour of a program. The addition of software reconnaissance to Diver and its *trace-focused user interface* feature improved the filtering of the Eclipse environment based on the contents of execution traces and led to a very powerful program comprehension tool. Myers' work on Diver was grounded in cognitive support theory research into how to build tools. He conducted a user study to validate the work done on Diver, but the study's findings were limited due to a number of issues. In this thesis, we expand on the study run by Myers, improve on its design, and investigate if software reconnaissance improves Diver's effectiveness and efficiency for program comprehension tasks. We also analyze the influence of software reconnaissance on the interactions of Diver's users, which allows us to identify successful usage patterns for completing program comprehension and feature location tasks. We research the connection between cognitive support theory and the design of Diver and use the study to attempt to validate the cognitive support offered by Diver. Finally, we present the results of a survey of the study participants to determine the usefulness, ease of use, and ease of learning of the tool.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

The research and work that went into this thesis was thanks to the support provided by so many people who wre involved in various ways throughout my time in Victoria. I would especially like to thank the following:

Cassandra Petrachenko for all the editorial help on this thesis; her input was invaluable.

Dr. Peggy Storey for the wonderful opportunity you gave me, the guidance, and the support.

Dr. Daniel German for the advice and knowledge he conveyed.

Brendan Cleary, Leif Singer, Christoph Treude, and Bo Fu and other members of CHISEL for their input and advice.

Martin Salois, David Ouellet, and Phillipe Charland, and the DRDC for their support of this ongoing research.

Del Myers for all the hard work that went into Diver and laying the foundation for this research.

My parents, Linda and Steve, and sister, Emma, for the support offered from so far away.

DEDICATION

Frances McGregor (1914 - 2013)

For instilling a love of traveling in me that brought me on this adventure.

—

Nelson Rolihlahla Mandela (1918 - 2013)

For serving as an inspiration and role model for myself and so many others.

Chapter 1

Introduction

New software engineering tools are being developed constantly to support developers' efforts in the design, development, and maintenance of software systems. Of all the phases of development, maintenance takes up the most time and resources [10]. Tasks associated with maintenance include adapting, perfecting, or correcting software systems [33]. Program comprehension forms a major part of the software maintenance phase, and comprehension tools often use visualisations to convey information about the system [29]. These visualisations can be manually created by developers, although automatically-generated diagrams are much more powerful.

Reverse engineering is one method of generating these diagrams. Reverse engineering extracts information from a software system that can be presented visually to help developers understand the program. Reverse engineering tools provide avenues to gain a better understanding of programs through information exploration, an activity which Tilley *et al.* [32] argue "holds the key to program understanding." One visualisation that can display program behaviour is a sequence diagram.

1.1 Are Sequence Diagram Tools Helpful During Program Comprehension?

Software architects use sequence diagrams to document system design. Kruchten's 4+1 architecture model demonstrates the value of using sequence diagrams to create a behavioural view of a system that brings together the other four views: Logical, Development, Process, and Physical [16]. These documents are often out of date by the time development reaches the maintenance phase. That is if they even exist,

as their usage is neither consistent nor popular according to Petre [25]. Generated sequence diagrams sidestep this problem and provide developers with an up-to-date and abstracted view of how a system behaves. Static or dynamic analysis can generate these diagrams. Static analysis uses source code and other software artifacts to gather information on a system. On the other hand, dynamic analysis gathers information by monitoring a system's execution.

Dynamic analysis of program execution traces can generate more accurate sequence diagrams than static methods. The drawback of dynamic analysis is that diagrams can become extremely large. Therefore, the problems of understanding large software programs were reduced, but the problem of understanding large sequence diagrams was introduced. Bennett *et al.* researched ways to solve the scalability problems affecting extremely large sequence diagrams [4]. The main difficulty of these diagrams was the cognitive overload experienced by users.

The subsequent research focused on solving cognitive overload with better tool design and improved feature understanding. Bennett *et al.* surveyed many sequence diagram tools to determine the features that supported better program comprehension [5]. OASIS Sequence Explorer—developed based on earlier research by Bennett *et al.* [4]—helped evaluate the research. This research bridged the areas of cognitive support theory with sequence diagram-based tool design [3]. As a result, Bennett identified theoretical foundations for key features that support the use of sequence diagrams.

Myers continued this line of research by focusing on improving the scalability of large sequence diagrams [24]. He implemented a set of features based on those features found to offer better cognitive support [3]. OASIS Sequence Explorer was developed into a fully-fledged tool called Dynamic Interactive Views for Reverse Engineering (Diver) that Myers used to support his research.

The tool requirements from Bennett [3] showed that developers require better cognitive support for program understanding tasks such as feature location and program analysis. Diver's aim was to provide improved navigation of sequence diagrams, and therefore, provide improved cognitive support for developers. Diver was developed as a plugin for the Eclipse Integrated Development Environment (IDE) based on requirements established in the study [5], and Figure 1.1 shows Diver's user interface.

Diver allows developers to monitor Java applications in Eclipse and then generate sequence diagrams of the execution traces. These sequence diagrams allow easy navigation of execution calls and links these calls to the underlying code. Myers implemented compaction algorithms that collapsed *for* and *while* loops to simplify

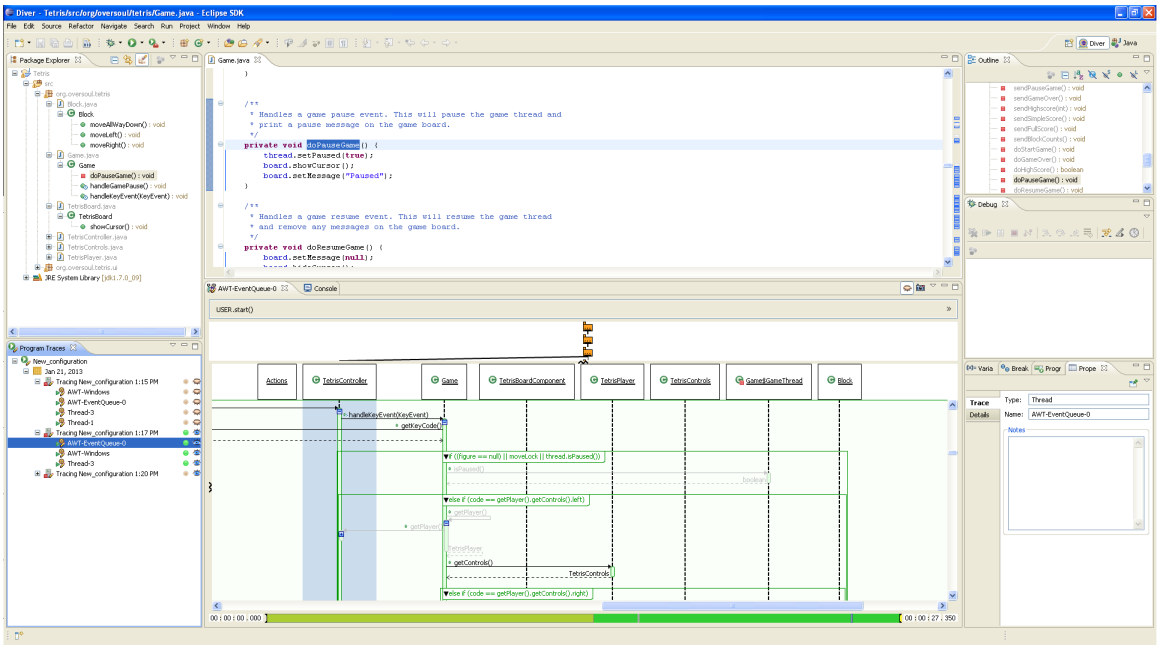


Figure 1.1: Diver's user interface in Eclipse

the diagrams [23]. The Mylyn *task-focused user interface* developed by Kersten and Murphy [14] provided the inspiration for Diver's *trace-focused user interface* feature. This feature filtered the Eclipse IDE to reflect the contents of traces [22].

The *trace-focused user interface* feature implements the software reconnaissance technique. This technique was developed by Wilde and Scully to map features to their locations in code [39]. In Diver, the technique's implementation filters unwanted methods from a trace by using a second trace. This technique vastly improved the original filtering that was only based on one trace. Myers conducted a study to evaluate the effectiveness of software reconnaissance in Diver which he presented in his thesis [24]. His findings appeared to confirm the usefulness of software reconnaissance in Diver.

Unfortunately, the evaluation of software reconnaissance was undermined by the study design. The problems with the study design were a number of weaknesses which affected the results of the experiment. These problems were identified by the reviewers of a conference-submitted paper based on the study [2], as well as our own analysis. These weaknesses included: a difference in difficulty between the two tasks given to participants, and some participants failing to use software reconnaissance during the experiment. There was only useful data from eight participants who actually used software reconnaissance and many of them failed to complete the more difficult task.

1.2 Solution: Evaluating software reconnaissance's support for feature location and program analysis

Diver is an important and popular academic tool that attracted over 3500 downloads between July 2012 and June 2013 [27]. Therefore, investigating software reconnaissance's support for feature location and program analysis would evaluate the work done to improve the usefulness of the tool.

Our hypothesis is that the addition of software reconnaissance to Diver is beneficial and should improve the efficiency of programmers performing feature location and program comprehension tasks. We also hypothesize that software reconnaissance should improve the cognitive support offered by the tool to programmers during these tasks. A study of the feature is necessary to test these hypotheses.

In this thesis, we extend the evaluation of the software reconnaissance feature implemented in Diver. We design a new user study to investigate if software reconnaissance is of value to trace analysis tools and the tools' users. This new study will draw from the experience of the previous study by Myers [24] and learn from the weaknesses in the original design. LaToza *et al.* argue that "the usefulness of a tool depends both on its success in solving a problem and supporting work" as well as the scale of the problem it addresses [17]. Therefore, it is not enough to study how quickly and successfully users complete tasks using the feature — changes in how they complete the tasks must also be studied. The experiment documented in this thesis seeks to answer questions about software reconnaissance's effect on user navigation in Diver, changes to user effectiveness and efficiency at completing tasks, and the tool's ability to provide cognitive support. We developed hypotheses to help us determine whether Diver implemented a number of cognitive support theories.

This thesis is divided into five chapters. Chapter 1 introduced the problem to be addressed by this thesis, and the proposed solution. Chapter 2 describes the background research and related work that this research builds upon. This chapter highlights cognitive support theories and their implementation in software engineering tools. We introduce Diver, its development, and features to support comprehension. Chapter 3 describes how we investigated the links between cognitive support theories and the features of Diver, and explains how we plan to study the cognitive support offered by the features in the new study. This chapter presents the purpose of the

research on Diver and software reconnaissance, the information being sought from the new user study, and an analysis of Myers's previous study and its shortcomings. Chapter 4 reports on the empirical study conducted using Diver, including the study design, results, and discussion. Finally, Chapter 5 explains the contributions of this thesis, future work and research opportunities, and a conclusion to the thesis as a whole.

Chapter 2

Background

This chapter reviews the research of various fields, such as program comprehension and software reconnaissance, that influenced the development of Diver. It also describes how Diver works and how a user would use software reconnaissance for a feature location task.

2.1 Program Comprehension

Due to the increasing complexity of software, it is becoming more and more difficult for programmers to understand unfamiliar programs. Given the importance of program comprehension, extensive research has been done into discovering how programmers understand programs, especially through the use of tools [31]. Storey groups program comprehension research into two main areas: programmers' cognitive processes to understand programs, and technological improvements through tool support [28].

The programmer's cognitive processes are responsible for building *mental models* to represent their understanding of a program. A variety of sources, such as conversations with team members or testing the program, are used to build these *mental models* [18]. *Cognitive models* describe the creation and storage of these *mental models* by programmers. Various *cognitive models* have been developed to explain the different program comprehension strategies that programmers follow. These strategies have been identified through observing how programmers approach program comprehension tasks and they are explained in more detail in the next section.

2.1.1 Program Comprehension Strategies

Storey *et al.* presented a number of program comprehension strategies and talked about the cognitive models that were associated with them [31]. The main strategies are listed below.

Bottom-up. The programmer analyzes the source code and, possibly, control flow information to gain a better understanding of different parts of the program. Higher-level abstractions are built by combining the knowledge of different areas of the program and this is repeated until a broad overview of the software is achieved.

Top-down. This strategy sees programmers starting with a global hypothesis of the program. Application domain knowledge is used to make additional hypotheses, which are tested as the programmer explores the system. The global hypothesis is adjusted as the underlying code and behaviour is explored. Experienced programmers use cues (hints that provide information about the type of architecture or possible implementation) to speed up this process.

Knowledge-based. This strategy introduces a more practical view of program analysis. It views programmers as being more opportunistic and less rigid in their search for understanding a system. A programmer combines application knowledge with programming knowledge to take cues from both bottom-up and top-down perspectives. These are assimilated into the *mental model* to improve program comprehension. Similar to the hypotheses/exploration process in top-down, *inquiry episodes* occur when a programmers ask a question, then forms a hypothesis using previous knowledge for the answer, and finally uses software exploration to determine if the hypothesis is correct or not.

Systematic and as-needed. A systematic approach can be likened to a student reading an entire textbook before completing exercises from just one chapter. The programmer reads the source code and follows the program flow to understand the entire system with little reliance on hypotheses and *inquiry episodes*. This results in a complete understanding of the entire program, but is time intensive, especially for large programs. The corollary to the systematic strategy is an as-needed approach, where a programmer only focuses on the code directly related to the functionality or

system behaviour being investigated. This results in a narrow understanding of the program, which saves time but risks missing important dependencies.

Integrated approaches. Von Mayrhausser and Vans observed that, in practice, programmers switched between the first three strategies, discussed above, as needed [33]. A *mental model* of the program is built at all levels of abstraction simultaneously. This leads to the development of a model combining the three strategies and their associated *cognitive models*.

Additionally, Storey *et al.*[31] highlights the main factors that affect a programmer's choice of strategies. They include the differences in programs, tasks, and programmers.

The program comprehension strategies, mentioned above, fall under the research area dealing with programmers' cognitive processes. The other research area involves finding technological advances to improving program understanding tool support.

2.1.2 Program Comprehension Tools

To facilitate the successful execution of the strategies above, program comprehension tools are constantly being developed and improved. Cornelissen *et al.* listed the five areas that cover most of program comprehension research: feature location and analysis, execution trace analysis, program behavioural analysis, information visualization, and design and architecture reverse engineering [7]. This research forms the basis for new features to improve comprehension tool design.

Storey reviewed the body of research on program comprehension and explored the relationship between that research and tools [28]. She presented a number of areas where tools could support the comprehension strategies discussed above.

- Navigation support could be used to help users with both top-down or bottom-up strategies. The top-down approach requires users to navigate from higher level views to more detailed views and the bottom-up approach requires users to navigate through method calls and class hierarchies. A tool that provides an array of options to navigate between multiple views could cater for both strategies.

- Searching and querying supports users who use *inquiry episodes* to explore the program. These features can be used to create program slices or provide users with a list of possible methods related to a keyword.

- Multiple views give the user the flexibility to use different strategies depending on the type of task or program. Tool designers can increase the versatility of their tools by catering for many different scenarios and strategies with the use of multiple views.
- Context-driven views provide adaptive interfaces that display data relevant to a user's current task.
- Cognitive support helps to ease the *cognitive load* associated with performing tasks using the tool.

Storey categorized program comprehension features into three broad groups: extraction, analysis and presentation [28]. Tools implement one or more of these types of features to help programmers complete program comprehension tasks.

Extraction features involve acquiring information about the program through various sources. These sources could be static, such as source code or configuration files, or dynamic, such as program execution traces. Both static and dynamic approaches have positive and negative implications for tools and their users. Static analysis is much easier to automate and analyzing source code requires little input from users. One drawback of static approaches is the limited view of the system because no runtime behaviour is collected. On the other hand, dynamic analysis approaches usually require more input from users, such as using the program while tracing, but collect important information about the execution of the program. Storey mentions that program traces can grow very large, which introduces more potential problems for the user.

The other two feature types (analysis and presentation) are closely related. Analysis features support programmers' activities to understand the information extracted from the program. For example, these activities could be feature location or concept identification and allocation. Depending on the type of analysis, more programmer input is require than the automated static extraction processes. As a result, programmers require cognitive support from tools to complete these activities. Presentation features convey information about the program to the users and examples of these features include code editors and visualizations.

Most tools combine these three types of features to create a platform to complete program understanding tasks, such as an IDE. Static and dynamic analysis are also combined to create versatile tools. To effectively combine all these different features,

designers need to create their tools with cognitive support in mind, which is crucial for effective tool use.

2.1.3 Cognitive Support in Program Comprehension Tools

Software comprehension tools such as Diver are built to support the problem solving and critical thinking required to understand how a program works. Tool builders implement best practices that are thought to rely on cognitive support theories, either consciously or subconsciously. However, according to Walenstein [36], the best practices being implemented may not be grounded in those theories. In his paper, Walenstein describes three cognitive support theories: *redistribution*, *perceptual substitution*, and *ends-means reification*.

Redistribution: This theory deals with cognitive resources or processing being externalized and stored outside the mind of the individual to support with organization, remembering results, or performing complex calculations.

Perceptual Substitution: The cognitive ability to process data depends on how the data is presented. This forms the basis of the perceptual substitution theory and is what motivates the study of information visualization.

Ends-Means Reification: This theory relies on reifying (making concrete) the possible states of the problem space and actions that change the states into a more concrete structure. This strategy involves analyzing a certain state repeatedly to find possible actions that help move towards the solution of the problem. A tool would support this by mapping the ends to the means. The example given by Walenstein [36] is a compiler error list. The ends are the list of errors and the means are the interactions available to move towards the goal of solving the compiler errors. There are various ways to apply this theory.

Bennett [3] applied this theory to sequence diagrams. He argued that mapping the ends to the means supports the navigation of sequence diagrams because it gives users cues on where to navigate to in a visualization.

2.1.4 Program Comprehension Through Dynamic Analysis

Dynamic analysis is the analysis of data acquired from the execution of a system. Cornelissen *et al.* conducted a systematic survey [7] on this "common" program comprehension technique and chronicled the research conducted over the years. Importance was placed on information visualization techniques which improve the communication of program information to programmers. From their survey, Cornelissen *et al.* identified Unified Modelling Language (UML) diagrams as the most popular choice for visualizing traces, however, they did mention the scalability issues that came with using these diagrams. Previously, Cornelissen *et al.* conducted research into circular views as an effective visualization for traces [8]. They point out research that was focused on solving these issues, such as Zaidman [41] and Hamou-Lhadj *et al.* [12, 11, 13]. Myers also worked on the problem of scalability [24]. A common theme of the research in scalability by Hamou-Lhadj and Myers was trace compression and elevating the visualizations of traces to a higher abstraction level through pattern identification. In addition to these approaches, new techniques were being sought to help solve the scalability problems of large sequence diagrams to improve dynamic analysis tools. One such technique was software reconnaissance.

2.2 Software Reconnaissance

Wilde and Scully created the term *software reconnaissance* to describe the technique they developed to map program features to source code [39]. Software reconnaissance can help developers locate functionality of interest by identifying the unique software elements involved in a feature's execution. Test cases are formulated by the developer and used to record traces. The execution traces, based on those test cases, record all of the software elements involved. The software elements that are unique to the feature are found by comparing the traces of test cases that include the functionality with those test cases that exclude the functionality.

2.2.1 From Theory to Practice

Software reconnaissance can best be described using set theory. The technique is used to identify different sets of software elements that may be helpful for a program comprehension task. The theory is briefly explained within the context of Diver's

use of the technique for feature location tasks. The sets and subsets were defined by Wilde and Scully [39], and Myers [24]. The main sets are as follows:

- **T** represents all the execution traces recorded for the program.
- **F** represents all the features of the program
- **E** represents all the software elements or methods of the program.

A feature location task requires finding methods related to feature $f \in \mathbf{F}$. A list of feature subsets are:

- **COMMON**: The elements that are common to all traces, such as utility methods.
- **UNIQUE**(f): The elements in **E** that are unique to feature *f*.
- **SHARED**(f): The elements that are indispensable to f but are not found in UNIQUE or COMMON.
- **INDISPENSIBLE**(f): The elements that occur in all traces involving *f*.
- **POTENTIAL**(f): The elements that occur in some traces involving *f*.
- **RELEVANT**(f): The elements that are indispensable to f but not in COMMON.

To begin locating a feature, a trace t_1 is recorded based on a test case involving the functionality of interest. Trace t_1 is defined as a subset of elements from **E**. These elements fall into one or more of the feature subsets listed above. The classification of these elements cannot be determined given only one trace. The set COMMON requires all possible test cases to be recorded to determine its contents. However, this set would be useful for determining utility methods and not methods related to the feature *f*. The set INDISPENSIBLE requires all possible traces involving feature f to be recorded to determine its contents. The sets SHARED and RELEVANT rely on COMMON for their definition and therefore, require all possible test cases to determine their contents. The set POTENTIAL contains elements that have nothing to do with the feature f and is therefore, not useful. This leaves the set UNIQUE which Wilde and Scully argued could be used to gain a "foothold" into the program to help locate the functionality of interest.

Theoretically, software reconnaissance can involve any number of traces and some subsets can only be identified when all possible traces of the functionality of interest are used. Myers improved on Wilde and Scully's formula to calculate the subset UNIQUE so that fewer traces were needed to determine the subset. The original practical way to determine the UNIQUE set was use a large number of test cases to identify components only occurring in test cases involving the feature of interest. In Wilde and Scully's case study [39], anywhere from 11 to 18 test cases were needed to identify a feature's UNIQUE elements. Myers' simplified formula definition meant that two traces could effectively determine a fair representation of the subset UNIQUE. We record a second trace, t_2, based on a test case that does not use the functionality of interest. In this case the formula is:

$$UNIQUE(f) = t_1 - t_2$$

Myers's changes allowed software reconnaissance to be used together with the *degree-of-interest* model used by Kersten and Murphy [14] for the Mylyn *task-focused user interface*. Diver is not the only tool to have implemented software reconnaissance and other research projects have produced tools using the technique. However, Diver differs from the other tools due to the *degree-of-interest* model which allows custom filtering of the IDE based on a trace. The other software reconnaissance projects are discussed in the next section.

2.2.2 Software Reconnaissance In Practice

Several researchers have continued to research the software reconnaissance technique and its practical uses. Wilde was involved with much of the subsequent research into the technique. He and Scully conducted a limited protocol case study to understand the ability of programmers to use their technique [39]. The two participants' task was to identify the code related to functionality that required changing. They were asked to identify test cases which were used to develop a functionality-based view of the tool. The participants were able to successfully apply the software reconnaissance technique to complete the tasks. After the study, they agreed that the UNIQUE subset was an "accurate and appropriate starting point" for the feature location tasks as it contained software elements that were unique to the functionality.

The technique was verified further by Wilde and Casey [38], focusing on the C programming language. They used an academic tool that they had developed called

RECON for their research. The goal of their work was to develop software reconnaissance into a "usable industrial technique." Software reconnaissance was used by a knowledgeable programmer on two systems and the researchers gained new insight into the technique. They discovered that software reconnaissance did not always identify all the software elements that the programmer thought was related to the feature. They found that software reconnaissance could be very useful at determining all the features that use a specific piece of code. This is important for a programmer who is editing a feature and needs to know the ramifications of the changes. The choice of test cases was highlighted as an important factor to using the technique successfully. The RECON tool was a very basic set of command-line tools and the authors noted that the user interface was "awkward." This underlines the fact that RECON was purely an academic research prototype, unlike Diver which was developed for widespread use.

Wilde *et al.* updated RECON and used RECON2 to conduct additional research on the technique's usefulness with legacy Fortran code [40, 37]. The tool features three components: an instrumentor, trace manager, and analysis program. Their case study [40] compared the software reconnaissance technique with the dependency graph search method. The researchers commented that software reconnaissance could only locate features and not "concepts" which the dependency graph method was able to locate. *Concepts* were described as "human-oriented expressions of intent." Analysis of the results showed that the two techniques had different strengths. Software reconnaissance was deemed better for larger less frequently changed programs. The opposite was true for the dependency graph search method.

In another paper, Rajlich and Wilde combined software reconnaissance and software tests to research the role of *concepts* in program understanding [26]. Software reconnaissance was used to analyze a program and uncover all the features that required software tests. The authors found a number of features that were not mentioned in the user guide which was used develop the test set. These newly discovered features allow for new tests to be written and the test coverage to be improved, largely due to software reconnaissance. This showed another useful feature of the technique: finding undocumented features.

Aside from Wilde's research with RECON, researchers have investigated a number of other tools using software reconnaissance. Agrawal *et al.* [1] conducted a software reconnaissance study for a C project, using a commercial software suite called χSuds. χVue, which is one of the tools of the suite, was used to locate features using the con-

trol graph, program execution trace, and the user's prior knowledge of the program. Agrawal *et al.* concluded that feature location techniques, such as the one used in χVue, are helpful at understanding features, writing unit tests efficiently and quickly, and discovering subtle bugs [1]. The product no longer exists and neither does the company who made it, Bellcore, who were bought by another company.

Cleary *et al.* developed an Eclipse plugin called CHIVE that combined software reconnaissance and static analysis to visualise reuse in systems [6]. The same research group that made CHIVE also developed a technique called *software reconn-exion* which combined software reflection with software reconnaissance [19]. CHIVE and *software reconn-exion* both used software reconnaissance to identify software element reuse instead of feature location like Diver and RECON.

Myers identified software reconnaissance as a useful technique to filter the potentially huge execution traces of Diver and the subsequently large sequence diagrams created from the traces.

2.3 Introduction to Sequence Diagrams

The +1 in Kruchten's 4+1 architecture model represents UML sequence diagrams and these diagrams provide a dynamic view of the system [16]. A sequence diagram documents the behaviour of software as functionality is executed.

The sequence diagram shows how software elements such as classes and methods are connected and call each other, see the example in Figure 2.1. The classes or objects of a system are represented by the lifelines that appear along the top of the diagram. The chronological order of events goes from top to bottom, and the lifelines are represented through the diagram by vertical dashed lines.

If a class becomes *active*, i.e. it is called by another class method, then the dashed line is replaced with a thicker activation bar. This denotes the length of time where an instance of the class is doing work. Messages or calls from one lifeline to another are represented by a horizontal dashed arrow. The arrow represents a method call if it is pointing right, and a method return if the arrow is pointing left. A method return would coincide with the activation box ending.

The last component of a sequence diagram is a combined fragment: a box that logically groups messages and activation bars to improve readability. For example, conditional statements, such as if and else, are easier to identify when grouped in a box.

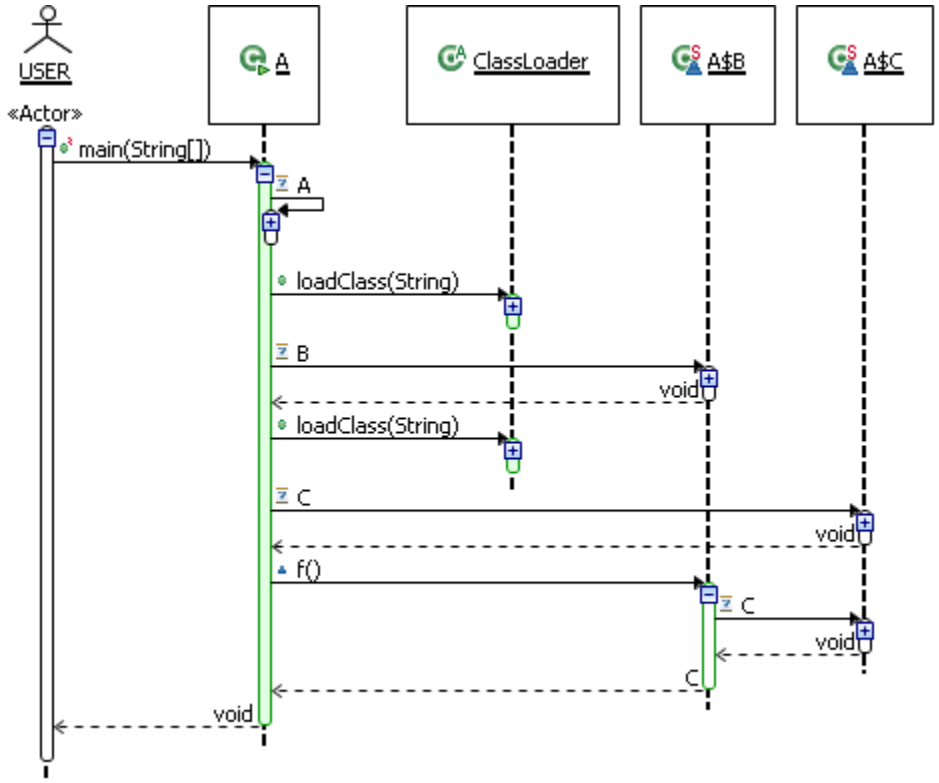


Figure 2.1: An example showing the various components of a sequence diagram

2.4 Sequence Diagrams in Diver

The Diver plugin is a suite of Eclipse plugins that work together and depend on execution traces to analyze programs and their features. The execution of a Java program through Eclipse can be recorded to create a trace. Diver records the method calls that are executed when the user interacts with the program and stores them in a trace file. These could be methods related to the features that were used or methods that do background work and are not related to any one feature. Each program thread has a sequence diagram, so one trace can generate multiple diagrams. Diver's functionality revolves around sequence diagrams and features to help make it easier to use the diagrams, which can get very large.

The *trace-focused user interface* feature filters the IDE's various views to only display the software elements that appear in the trace. The feature is enabled by activating a trace, which can be done in the Program Traces view (where traces are managed). Only one trace can be activated at a time. When a trace is activated, the Package Explorer only shows the classes and methods from the trace. This is

beneficial for feature location tasks where completely unrelated methods in the project are filtered out. To locate specific functionality in a program, an activated trace should only contain the relevant software elements. However, it is very difficult to record a trace with only these elements. Typically, an activated trace will also contain unrelated methods, which we will refer to as noise.

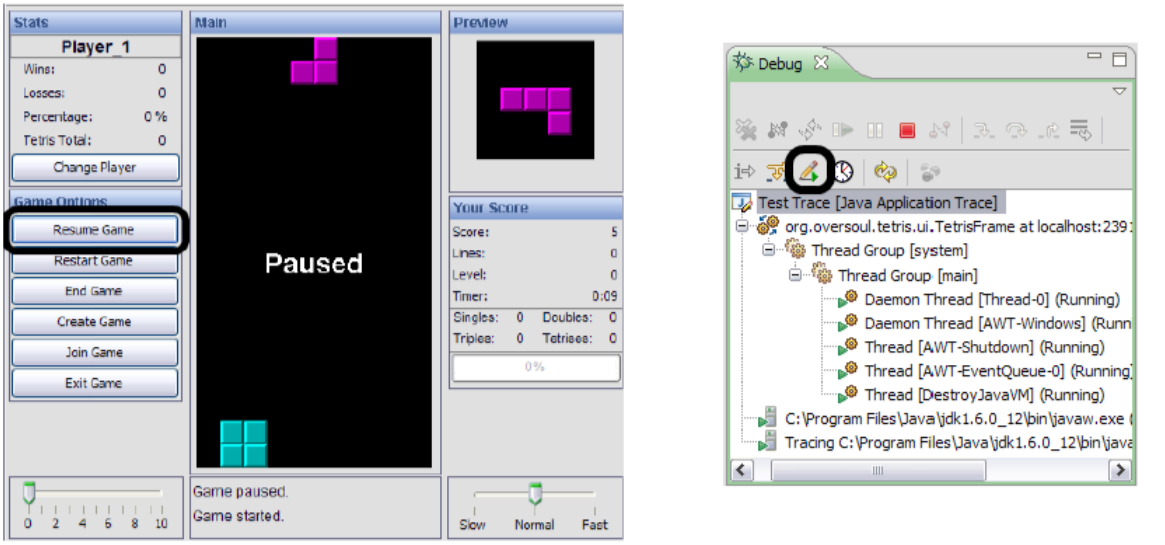
The software reconnaissance technique improves the *trace-focused user interface* filtering by helping remove noise. The technique uses a second trace to identify and remove the common method calls between the activated and filter traces. This trace is called a *filter trace*. The efficiency of the software reconnaissance technique filtering increases when the filter trace contains as many of the unrelated methods as possible. If the filter trace contains all of the unrelated methods present in the activated trace, software reconnaissance can filter all the noise. It is difficult to record all the unrelated methods in a filter trace while, at the same time, avoiding any methods related to the functionality of interest. For example, the functionality of interest may involve a button click and any unrelated methods called as a result of clicking the button that cannot be recorded separately.

Software reconnaissance filters the sequence diagram, in addition to the Package Explorer, by colouring the common method calls light grey. The filtering is shown in Figure 2.6. This adds emphasis to the unique calls of the activated trace without affecting the diagram's structure. The improved filtering by software reconnaissance reduces the number of possible methods one has to investigate during feature location tasks.

2.4.1 Diver Usage Example

To show how software reconnaissance is used in Diver and what participants were required to do during the study, we will go through the steps needed to apply the technique using a Tetris project that consists of eight packages and 56 high-level classes. The packages and classes are shown in Figure 2.3a. In this example, we want to identify the methods called when the Resume Game button (highlighted in Figure 2.2a) is clicked. To start with, we must record a trace of the button click.

A trace is created using Diver by executing the program in Eclipse. Recording is paused by default when a program starts through Eclipse and can be resumed with the Trace Record button – this button is located on the Diver control panel in Eclipse and is circled in Figure 2.2b. Once the button is clicked, Diver starts recording the



(a) The example program Tetris with the Resume Game button

(b) The Diver control panel with the Record button

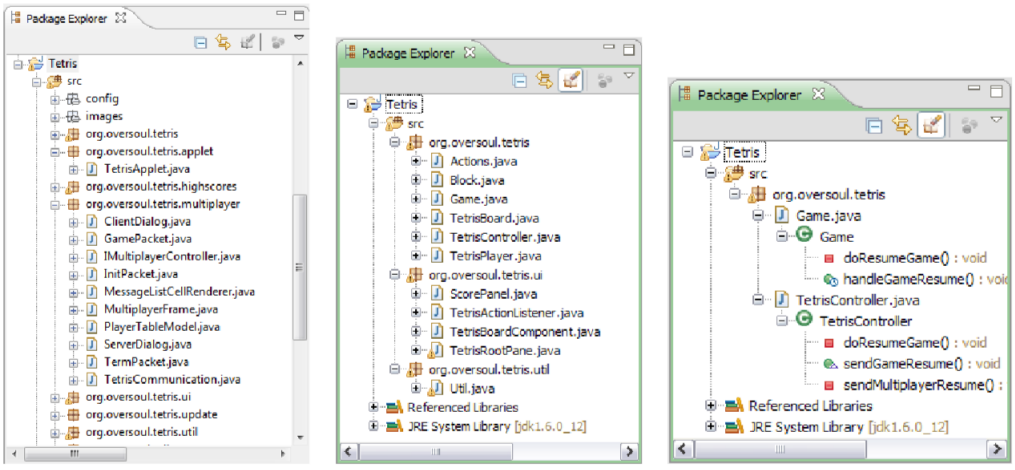
Figure 2.2: Diver records traces in the background while the Tetris program runs

program's threads. Recording can be paused and resumed at any time, and stops when the user closes Tetris. Once the program is closed, the trace data is saved to a database. Depending on the size of the recorded trace, this can take a few minutes. Once the trace is saved, it is ready for use in the Program Traces view.

Saved traces are displayed as a tree, with their threads as children, in the Program Traces view (see Figure 2.4). They are organized by date to make it easier to locate the correct trace. Double clicking one of the threads of a trace will open its sequence diagram. Right clicking on a trace will bring up a menu that includes options for activating/deactivating, filtering, and deleting the trace (see Figure 2.4a). From this view, we can start using the trace to locate the functionality of interest.

To locate the code that implements the Resume Game button, we can either open a sequence diagram or look at the contents of the Package Explorer. If we look in the Package Explorer, as seen in Figure 2.3a, all eight packages and 56 high-level classes are currently visible. This presents a large number of items to search through, but Diver's *trace-focused user interface* can help with this problem.

The *trace-focused user interface* links the recorded trace to the source code displayed in the IDE. The feature can be enabled by activating the trace through the menu shown in Figure 2.4a. An activated trace can be identified by a green dot and open eye icon next to it in the Program Traces view. After activating the trace,



(a) unfiltered Tetris program (b) Tetris filtered with one active trace (c) Tetris filtered by excluding a trace

Figure 2.3: The Package Explorer for the example Tetris program showing the three different states of filtering

the Package Explorer view is filtered to only display the elements recorded in the activated trace. The filtered view is shown in Figure 2.3b and can be compared to the unfiltered version in Figure 2.3a. As a result, the Package Explorer is easier to navigate and search for interesting methods after activating the trace.

As a result of activating the trace, the Package Explorer only displays the contents of the trace. However, this probably includes methods unrelated to the Resume Game button functionality. This is where software reconnaissance can be used to improve the filtering of the IDE. A new filter trace must first be recorded to collect as many of the unrelated methods as possible. Once the trace is saved, software reconnaissance is applied by clicking on the open eye icon next to the filter trace. This causes the icon to change to a closed eye and indicates that software reconnaissance has been applied (see Figure 2.4b).

With software reconnaissance applied successfully, a portion of the unrelated methods are filtered from the Package Explorer and sequence diagram. Figure 2.3c shows that the Package Explorer has few remaining classes and methods compared to before software reconnaissance was applied (shown in Figure 2.3b). In addition to this, the sequence diagram is filtered, as shown in Figure 2.6. Therefore, the *trace-focused user interface* and software reconnaissance have transformed the IDE into a customized, feature-focused state, and have greatly reduced the search space for the feature location task at hand.

From here, we can search the remaining methods for involvement in the Resume

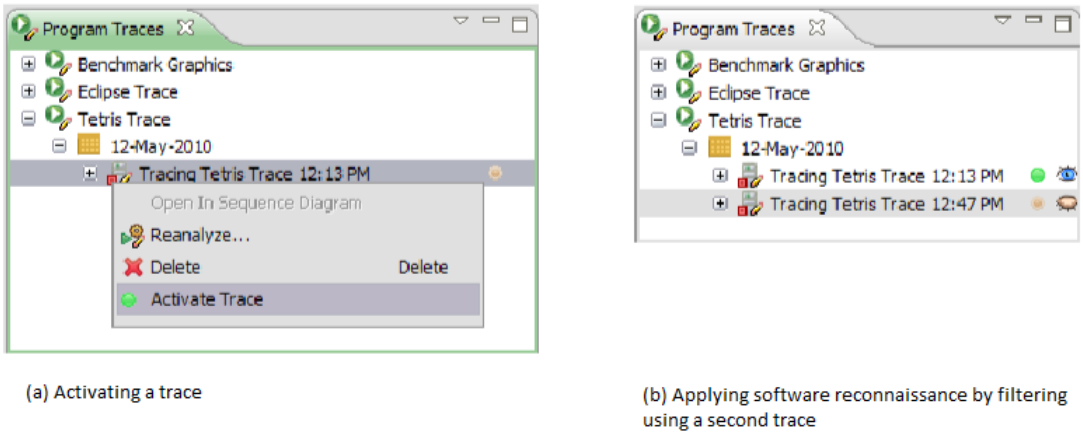


Figure 2.4: The Program Traces window displaying the recorded traces.

Game button functionality. The Package Explorer is an attractive place to start because so few methods remain active. When an interesting method is identified, the user can jump from the Package Explorer to that method in the sequence diagram. The *Reveal In* feature lets the user perform this navigation and can be accessed from the method's right click menu (see Figure 2.5). The user has the option of selecting the thread they want to jump to, such as the *AWT-EventQueue-0* thread shown in Figure 2.5. After using the Package Explorer to find an interesting method, the sequence diagram can be used to find out more about the method.

The sequence diagram contains information about the methods that were called before and after the interesting method. This information helps the user decide if the method was involved with the clicking of the *Resume Game* button. If not, the user can repeat the process of searching the Package Explorer and then navigating to the sequence diagram for more information. The addition of software reconnaissance means that there are fewer options to search through, and locating the calls related to the functionality of interest should be easier.

This example shows how software reconnaissance is implemented and used in Diver for completing tasks, such as those done in Myers' user study.

2.5 Overview

In conclusion, Diver provides a platform for the implementation of various cognitive support theories and program comprehension models. This background work formed the basis for the knowledge about the field and previous research done. It provided

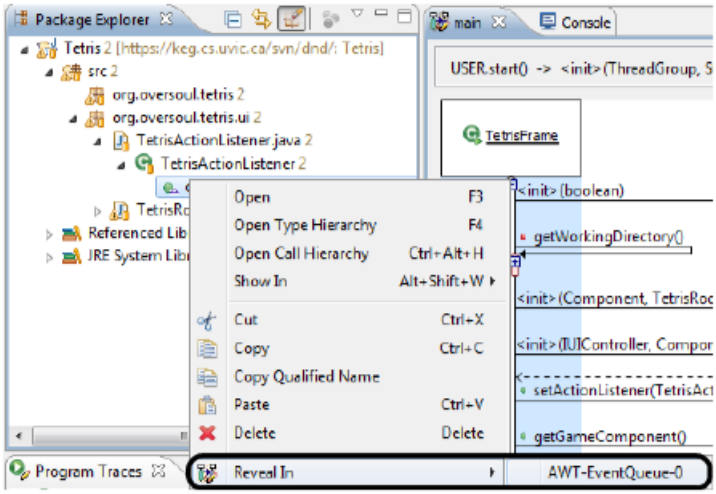


Figure 2.5: The *Reveal In* option to navigate to the sequence diagram from a class in the Package Explorer

a foundation for the detailed research conducted into sequence diagram tools and cognitive support theory which is presented next.

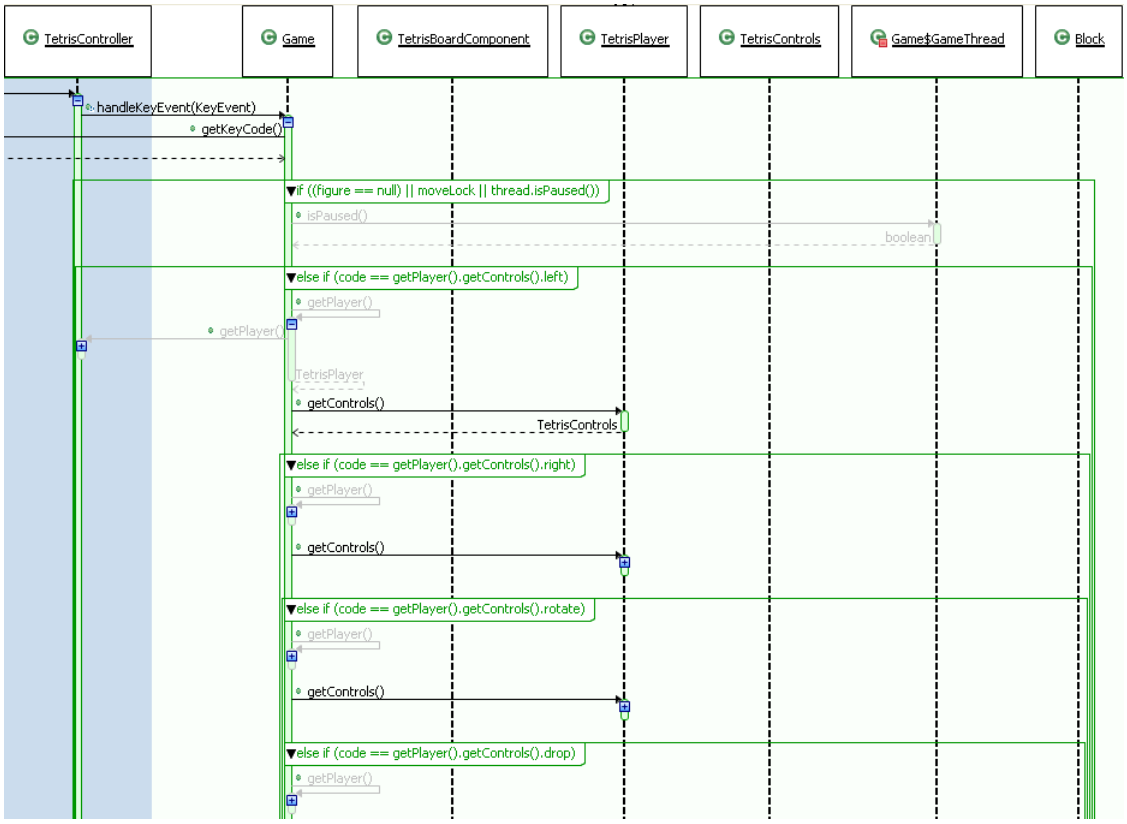


Figure 2.6: The filtering of the sequence diagram

Chapter 3

Cognitive Support Mapping

This chapter explores the links between Diver's features and cognitive support theory. These links provide the basis for studying the effectiveness of the cognitive support of Diver. This is followed by an analysis of Myers' 2010 user study [24] that evaluated software reconnaissance in Diver. The study contained a number of weaknesses and these are highlighted and discussed. From there, avenues for further research, through an improved study, are investigated.

3.1 Cognitive Support and Diver's Features

Diver was based on the OASIS sequence explorer, which was used by Bennett for evaluation of his research into tool features for sequence diagrams [3]. Myers used the results of this research to implement features that provided better cognitive support to programmers. Myers also conducted his own research into ways to reduce cognitive overload of large sequence diagrams [24]. The research of Bennett, Myers, and many others was used to design a program comprehension tool based on cognitive support theory.

Bennett's research identified features offering cognitive support for sequence diagram tools [3], which helped solve problems such as cognitive overload from huge sequence diagrams. He linked cognitive support theory to these sequence diagram tool features by mapping the features to design elements put forward by Storey *et al.* [30]. Storey *et al.* developed these design elements, that were backed by cognitive support theory, as building blocks for software exploration tools. The research done by Walenstein [34] provided a number of cognitive support theories that described

the support provided by the design elements of Storey *et al.*

The connection between Walenstein's cognitive support theories and Diver's features can be made as follows. Walenstein's cognitive support theories [36] support the cognitive models and comprehension strategies of Storey *et al.* [31]. These models and strategies in turn support the design elements that Storey *et al.* [30] developed for software exploration tools. These design elements map to the sequence diagram features drawn up by Bennett *et al.* [5]. These features were implemented in Diver by Myers. This connection is explained in more detail below.

Walenstein's work [35] sought to develop "high-level cognition support terms" instead of the complex assortment of narrowly defined task or technology dependent design elements. He identified a gap between cognitive models and tools that seek to implement them. Walenstein looked into the problem of gaps between cognitive theories and their implementations in software comprehension tools [36]. He refined three cognitive support theories and applied them to cognitive models such as the top-down model. Walenstein's theories of redistribution, perceptual substitution, and ends-means reification formed the basis for adding cognitive support to tools, which he explained in great detail in his thesis [34] and demonstrated with a tool [36].

Walenstein's theories needed developers to design their tools effectively to successfully implement the cognitive support theories described. Storey *et al.* introduced 16 cognitive design elements, seen in Figure 3.1, that tool designers could use "to support the construction of a mental model to facilitate program understanding" [30]. These elements either improve program understanding or reduce cognitive overhead. There were seven program comprehension elements that enhanced and supported the use of two different comprehension strategies (bottom-up and top-down), which were among the list of cognitive models Storey *et al.* presented in [31]. The remaining elements reduced cognitive overload by increasing orientation cues/reducing disorientation and improving navigation. Storey *et al.* [30] explained the links between these elements and their cognitive models [31] that allowed developers to provide users with cognitive support during program comprehension tasks. Even so, the design elements had to be implemented correctly to be effective.

Bennett identified features used in sequence diagram-based tools that would support the development of better mental models by developers [5]. He took Storey *et al.*'s cognitive framework for the design of software exploration tools and mapped their design elements to the features he had identified [3]. He argued that their framework was at the best level of abstraction compared to the other frameworks he

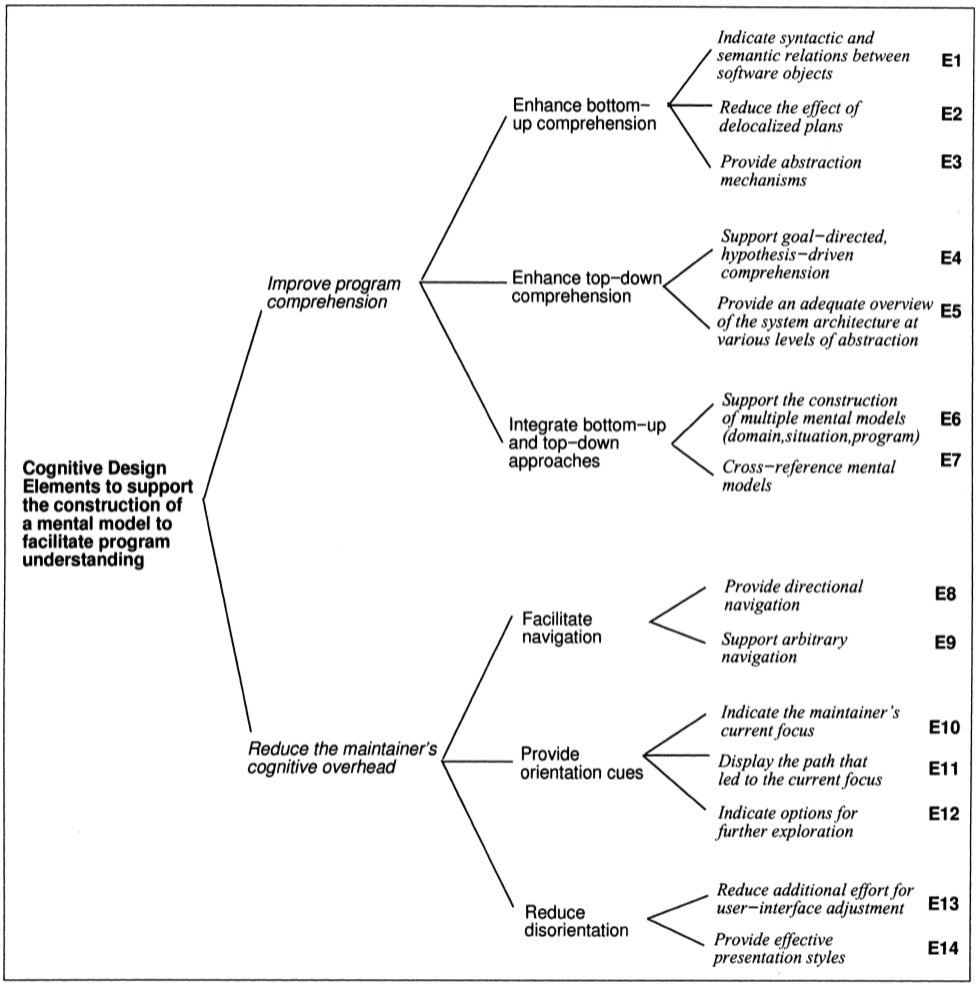


Figure 3.1: Cognitive Design Elements for Software Exploration from Storey *et al.* [30]

investigated. Bennett relied on the work Storey *et al.* had done to show that the design elements were grounded in program comprehension theories. The result was a matrix of sequence diagram features versus cognitive design elements, with most of the features mapped to more than one design element. Myers implemented Bennett's features into Diver with the goal of building a tool that provided better cognitive support.

Bennett's sequence diagram tool features are divided into Presentation Features and Interaction Features and matched to their related Diver features below. This list completes the connection between Diver's features and cognitive support theories behind their inclusion, and also, identifies theories that can be investigated using certain features of Diver.

3.1.1 Presentation Features

Bennett's presentation features deal with how the diagrams are displayed, how different views are presented, and how the user interface conveys information efficiently.

Layout: The sequence diagram follows a UML format with some extensions to improve readability, such as coloured code blocks for loops and conditional statements.

Multiple Linked Views: The Package Explorer is linked to sequence diagrams through the *Reveal In* feature. This feature allows the user to navigate from a method in the Package Explorer to its location in the diagram. The sequence diagram is linked to the source code through the *Jump To Code* feature. This feature allows the user to navigate from a method call in the diagram to the line of code where the call was made.

Highlighting: The elements of the sequence diagram are highlighted when the user hovers the mouse cursor over them or clicks them. Code coverage is indicated by highlighting relevant lines of code with a blue line.

Hiding: Different parts of the sequence diagram can be hidden from view using various features. The Focus On feature hides all method calls that are not associated with an activation bar. Unwanted Java class life lines can be collapsed. Sections of the diagram can also be collapsed so that sub-message calls are not displayed. Lastly, filtering the Package Explorer hides uninteresting methods.

Visual Attributes: Colour is used to differentiate between static and dynamic activation boxes. The code blocks are also colour-coded with blue highlighting for loops, red showing exception handling, and green indicating conditional statements.

Labels: The method call arrows are labelled with the method's name on the sequence diagram, as are the return types.

Animation: The only animation in Diver is performed when sections are collapsed or expanded.

3.1.2 Interaction Features

Interaction features include ways that the diagrams can be navigated, configured and customized, and automatically searched using queries.

Selection: Sequence diagram elements, such as method calls and activation bars, are selectable and turn bold when clicked.

Component Navigation: When the Program Traces view is in focus, the arrow key changes the selection of one thread to another.

Focusing: The Focus On and Focus Up features implement focusing on a section of a large trace. The Focus On method focuses on a method call and removes all parts of the diagram not involved with the call. The Focus Up feature involves navigating from focusing on the current method call to focusing on the preceding method call.

Zooming and Scrolling: Scrolling is implemented for the sequence diagram. Zooming is only partially implemented in Diver's Outline view of the sequence diagram, which provides a zoomed out perspective.

Queries and Slicing: Diver provides search functionality in traces. Slicing is implemented using the software reconnaissance technique to isolate a certain part of the program by filtering one trace with another.

Grouping: The life line filtering feature implements grouping by collapsing the classes of a Java package and grouping their life lines into a single life line.

Annotating: Although not a major feature, Diver allows users to make notes that are associated with objects, such as traces. The notes are saved and viewable later.

Saving Views: Traces are saved and can be opened at any time. The sequence diagram view is saved on close and restored when the trace is reopened.

Saving State: This feature relies on storing the history of actions performed on the diagram so that a previous state can be accessible. Diver does not include this functionality and neither did any of the other tools Bennett surveyed. This feature can be complex to implement and require large amounts of storage space. This made it unattractive to implement. A screen capture feature was added to Diver that saved an image of the current state of the sequence diagram.

3.1.3 Cognitive Support Theories In Diver

Based on the discussion of theories in Section 3.1, the cognitive support theories of redistribution, perceptual substitution, and ends-means reification can be identified in Diver's features.

The redistribution support theory covers all features in Diver that store information for the users, such as method calls from program executions. Without this, a developer would have to remember these events while stepping through the code line by line. Additionally, Diver provides filtering that turns the Package Explorer into a list of methods of interest for the user to reference while completing the task; focusing and hiding work the same as filtering. The cognitive support provided by redistribution can, therefore, be measured by the users' perceptions of the sequence diagram and filtering features.

Perceptual substitution deals with displaying data in a way that takes advantage of the user's ability to process visual information better and quicker than text-based data. This is why a sequence diagram is used in Diver instead of a list of method calls. It shows the flow of the program, including class and method names, with the aim of providing better understanding of the program's behaviour. During feature location and program comprehension tasks, a user would be expected to rely on the sequence diagram more than the source code if it provides better cognitive support for completing the tasks.

Ends-means reification is a more complex theory involving reifying (making concrete) certain states in a tool that a user can move between while solving a task, such as feature location. These states give the user a better idea of the problem space as well as providing options or steps to solve the problem. One set of states would be the filtered Package Explorer and its methods, and the sequence diagram and its information about the methods. Thanks to filtering, the list of methods from the trace are potential solutions to the feature location task. The user can move from the Package Explorer to the sequence diagram using the *Reveal In* feature to determine if a selected method is the one being sought. Moving from one state (the Package Explorer) to the other (the sequence diagram) and back again, repeating the steps until an interesting method is found, constitutes the behaviour expected when the ends-means reification theory is put into practice. The use of the *Reveal In* feature is key to this type of cognitive support in Diver.

3.2 Myers' Software Reconnaissance User Study

To evaluate the software reconnaissance feature implemented in Diver, Myers ran a study featuring an experimental simulation [21] using professional programmer participants to solve tasks using Diver [24]. Myers explained that an experimental simulation was used so that a balance was achieved between realism and control of the experiment. The professional programmers and realistic tasks were used to make the context as close to a real feature location and program understanding task as possible. Myers reported that ten professional developers were recruited to participate in the study. The participants were required to complete two large feature location tasks that were coupled with five program comprehension questions based on those developed by Ko *et al.* [15].

The three research questions that the initial study tried to answer were:

1. Does the use of software reconnaissance improve the efficiency of feature location in Diver?

2. Does the use of software reconnaissance reduce frustration during feature location tasks?

3. How does the use of software reconnaissance influence navigation patterns through the various views in Diver?

It was decided that Diver's own source code would be used in the study as it was the required size, had sufficient complexity, and was unknown to all the users. Myers' knowledge of the code also helped with the selection of the two tasks.

To account for differences in experience between users, the study used a *within subjects* approach; one task was performed using software reconnaissance and one without. The performance of each participant using software reconnaissance was compared with them not using it. A drawback of a *within subjects* approach is learning bias which could result in the second task being easier due to the experience gained completing the first one. To ensure software reconnaissance was not favoured due to this phenomenon, the first task used the software reconnaissance scenario and the non-software reconnaissance scenario was applied for the second task. The order of the two tasks was alternated over the course of the study. After a training session, the users had 40 minutes to complete each task and the accompanying questions, and the entire study took two hours per user.

An interview was conducted afterwards to gain insight into the users' experiences. In addition to the interview, data was collected through screen and video recordings, and observations were written down by the experimenter. The users were asked to think out loud to aid the understanding of their thought processes and stumbling blocks. Together with the time measurements, the quantitative and qualitative data collected made it possible to use a mixed-methods methodology to analyze the results.

The first research question was answered by timing the participants. The initial metric of how long it took to complete an entire task was replaced with a *time to first foothold* (TTFF). This adjustment was made because the experimenter observed that after finding the feature, participants leisurely answered the questions, using all of the remaining time available. The TTFF started once the participant had completed tracing and ended when a method was found that resulted in gaining a foothold.

The second research question was answered by analyzing the participants' utterances recorded on video and by the experimenter. The frustration utterances were coded and then sorted into three categories. The number of occurrences during the software reconnaissance scenario was compared to the occurrences during the non-software reconnaissance scenario, and then results were discussed.

The third research question on participant interactions was answered by analysing the Diver usage data that was recorded. The use of different features is compared between scenarios. Myers used the differences in Package Explorer usage to show that software reconnaissance provided better ends-means reification support.

3.2.1 Shortcomings in the Initial User Study to Evaluate Diver

The initial study had a number of problems identified by reviewers of a submitted (but not published) paper on the study and through our own analysis of the design and results.

- **Problem 1:** Not all the participants used software reconnaissance during the session even though they were instructed to do so. As the study relied on comparing the data from tasks being completed using the feature versus not using the feature, their data was not useful and excluded for analysis. Of the ten participants, two did not follow the instructions to use software reconnaissance so the data from eight participants could only be used.

- **Problem 2:** After the study, a mismatch was discovered in terms of difficulty between the two tasks. One task was much more difficult to complete than the other task, especially without the use of software reconnaissance. With only two tasks, this had a noticeable affect on the results because many participants were unable to finish the problem task, especially without the help of software reconnaissance.

- **Problem 3:** There was confusion with Diver being used to trace itself. As two instances of Eclipse were open during tracing, participants were often unsure which instance of Eclipse was recording the other instance. A study participant said that the confusing multiple instances of Eclipse led to waste time during the tasks.

- **Problem 4:** The think-aloud method did not result in some participants voicing many frustrations. The results showed that half the participants made less than 10 frustrating comments each in an 80 minute period. It could be seen that some participants were more inclined to utter frustrations than others. One reviewer pointed out that it was hard to separate frustration caused by the tool and frustration caused by the process. Because of this flaw, the second research question could not be adequately answered.

- **Problem 5:** The study tried to evaluate the efficiency of software reconnaissance using the TTFF metric. Measuring of the TTFF was started once the user completed tracing. This is a problem because software reconnaissance takes longer to record the necessary traces. The software reconnaissance technique requires at least two traces to be recorded, but only one trace is necessary when the technique is not used. The time taken to record traces is, therefore, relevant to the study. By ignoring the tracing time, there was a bias towards software reconnaissance.

- **Problem 6:** Two reviewers pointed to the confounding effect of having participants use software reconnaissance in the first session and then taking it away in the second session. This potentially biased the participant frustration that Myers was measuring.

3.2.2 Improving the Study Design

Based on these identified problems and lessons learnt from the initial study, we designed a new study with a number of significant changes to try to improve the experiment. These changes are listed below.

The methodology was changed from an experimental simulation to a laboratory experiment. A laboratory experiment takes place in an artificial laboratory setting but this methodology involves more precise measurements being taken for a narrow scope. This change allowed a number of other changes be made that were not compatible with an experimental simulation.

Problem 1, dealing with participants not using software reconnaissance, was addressed by reminding participants to use software reconnaissance during the SR scenario if they initially failed to do so.

To address Problem 2, which involved differences in the task difficulties affecting analysis, a *within subject* approach was not used in favour of comparing subjects' performances with a mixed model design. To this end, a larger number of participants was needed and, given the difficulty of recruiting professionals, student participants were used instead. This allowed for more quantitative analysis of the data, but introduced the potential problem of using inexperienced students.

One of the causes of Problem 2 was the complexity of the tasks; hence simpler tasks that required less time were selected. Simpler tasks meant that participants with less programming experience could be used, thus reducing the risk of using inexperienced students. Following the decision to use simpler tasks, the number of tasks was increased to eight. This change mitigated the risk of a task being biased towards one of the scenarios and provided more data for analysis.

Due to the potential of inexperienced students and differences in experience between participants, a pre-study questionnaire was introduced to get more information about their backgrounds. This information could be used in the analysis of the data to determine if experience played a significant part in the results. The questionnaire asked for information on academic background, general and specific programming experience, and exposure to Tetris. The complete pre-study questionnaire form can be found in Appendix B.

Problem 3, related to the confusion around tracing Diver with Diver, was addressed by changing the software project under analysis from Diver to a Tetris game. Also, a simple Sudoku game was introduced for training. This avoided the confusion

created by tracing Diver using Diver.

Problem 4 and 6, about measuring the participants' frustrations, was addressed by moving away from investigating participants' frustration utterances during the sessions and the research question was dropped.

To solve Problem 5, where the TTFF was only measured after tracing had been completed, the TTFF start time was changed to when the participant started the task, and not when the tracing was finished. The end time remained when a foothold related to the feature was established in the code. The drawback is that participants who take a long time tracing will take longer to complete the tasks than those who find it easy to trace. However, the change leads to an improved reflection of the time it takes to complete a task.

For the initial study, five program comprehension questions were included as part of each task. For this study, the questions were reduced to three to simplify the tasks and save time. Each task involved finding where certain functionality occurs in the Tetris code and then answering three program comprehension questions explaining how and why that functionality was executed.

3.2.3 Rephrasing Myers' Research Questions based on the Problems Uncovered

Myers' research questions were reconsidered due to the problems and subsequent changes to the study. For example, question two on frustration was not included due to the difficulties associated with accurately measuring participant frustration. With all the experience gained from the initial study, analysis of the participants' navigation trends resulted in some important observations. The cognitive support of Diver and software reconnaissance was only briefly touched on in Myers' study. We believe this topic was important enough to have its own research question. The question on the efficiency of software reconnaissance is still important and the TTFF measurements can be used to answer it. We decided to also investigate the effectiveness of completing tasks using software reconnaissance. The statistics on how many tasks were completed using software reconnaissance can be used to measure effectiveness. The efficiency and effectiveness concepts were combined into one broader research question. These questions are presented in detail in the next chapter.

3.2.4 Summary

This chapter described the relationship between cognitive support theory and Diver's features. It then presented the problems with Myers' 2010 study and proposed changes that would hopefully result in a new, improved study. The research questions were reconsidered and set the foundation for the new study design, which is presented in the next chapter.

Chapter 4

Empirical Study

This chapter describes the design, results, and observations of our empirical study investigating if the use of Diver's software reconnaissance feature helps improve the user's ability to locate features and perform program comprehension tasks. The research questions are presented and discussed, after which the study design is explained in detail. The results are presented, with observations provided in the discussion sections after each research question.

4.1 Research Questions

The research questions for this new study were adjusted based on problems with the results from the initial Diver study. The second research question was removed because no suitable changes to the study design could be found to acquire useful data regarding user frustration. A new research question based on Diver's cognitive support was added, which queried the ability of software reconnaissance to improve the support provided by Diver during program understanding tasks. The new research questions are:

1. How does the use of software reconnaissance influence navigation and interaction strategies in Diver? (**RQ1**)

2. Does the use of software reconnaissance improve the cognitive support of Diver during program understanding tasks? (**RQ2**)

3. Does the use of software reconnaissance in Diver improve a user's efficiency at feature location tasks? (**RQ3**)

The data collected from the study will help answer the three questions. **RQ1** will be answered by investigating the user navigation data and searching for recurring patterns and trends. Successfully completed program comprehension tasks will be compared to incomplete tasks across the two scenarios. We will present the Diver usage differences that resulted in the feature location tasks being completed successfully.

RQ2 involves considering the three main cognitive support theories mentioned by Walenstein [34] and identifying whether the support offered by their implementation in Diver is improved with software reconnaissance. Analysis of user perceptions of the features of Diver as well as their interactions with the tool will help us to answer this research question.

RQ3 questions the performance of the participants and how successfully completed tasks differ depending on whether or not software reconnaissance is used. This question will be answered by measuring the time taken by users to complete program comprehension tasks in Diver.

In the Discussion section, the results are analyzed to find answers to the research questions above.

4.2 Methodology

This section describes the study design and how participants and tasks were selected. The processes followed during the study are outlined and the pilot study that was used to refine the process is discussed. In preparation for conducting the study, ethics approval was sought from and granted by the University of Victoria Ethics Committee and the approval certificate can be found in Appendix G.

4.2.1 Tasks

The tasks required participants to conduct feature location and program comprehension activities using Diver. These activities simulated program comprehension tasks performed in the real world and supplied data to answer the three research questions. Participants were asked to complete eight tasks, which were based on the different features of Tetris. The following criteria were used to select the features for the tasks:

- Each task should be simple enough to be completed in approximately 10 minutes.

- The features being located should be visible and easy to trace.
- The features should be spread throughout the software project.
- The features should involve different execution flows such as setter methods or GUI interactions.

The objective for each task was to locate a method related to the feature of interest and then investigate how the functionality was executed. Based on their investigations, participants had to answer three program comprehension questions which were based on questions developed by Ko *et al.* [15]. The questions were:

- In which thread is the functionality primarily executed?
- Please describe the program flow that preceded the execution of the functionality.
- What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Appendix C includes the task description forms that explained the eight features that were used for the study. Participants completed these forms for each task.

4.2.2 Participants

We recruited participants using posters and email advertisements. 16 participants volunteered for the study. The participants varied from undergraduates to post-doctoral researchers. They were required to have Java programming experience and all 16 met this criteria although the amount of experience differed. This is discussed in Section 4.3.8. At the end of the experiment, participants were given a $20 stipend for their participation.

4.2.3 Experimenter's Process

The experimenter (and author of this thesis) referenced a handbook that listed the steps needed to run the experiment. After the design was finalised, the handbook was created to outline the process followed for each participant. The handbook was used as a check-list to make sure that all equipment and forms were ready beforehand and all instructions about Diver and the study were explained to the participant. A

training script was followed to explain Diver's features. The handbook can be found in Appendix F.

All the equipment was prepared in the laboratory. Participants were provided with an installation of Eclipse with Diver installed and the Tetris project open. The participants' activities were recorded by a video camera, a screen recorder, and on paper based on the observations of the experimenter.

The study was two hours in length and consisted of a number of sections involving different duties for the experimenter to perform.

1. In the **Orientation** section, after giving an overview of the study and explaining the data usage and privacy policies, the experimenter provided a consent form for the participant to sign. Next, a pre-study questionnaire was given to the participant to fill out. (5 minutes)

2. In the **Training** section, the experimenter introduced Diver and trained the participant to conduct feature location and program comprehension activities. The scripted training took the form of a guided tour around Diver as the participant completed a sample task, called Task 0, involving an example Sudoku project. (25 minutes)

3. In the **Experimentation** section, the experimenter provided the participants with instructions for the tasks and then assigned them one task at a time to complete. This section was divided into two 40 minute sessions with a five minute break in-between. The experimenter instructed the participants to use software reconnaissance in the first session and not to use it in the second session. The participants were asked to verbalize their thoughts, problem solving ideas, and processes using a *think-aloud* technique [9]. The experimenter took note of these verbalized thoughts as well as other observations while the participants completed the tasks. (85 minutes)

4. In the **Conclusion** section, the experimenter handed out the post-study questionnaire to complete, and thereafter, gave the stipend to the participant. (5 minutes)

4.2.3.1 Task Order Selection

Eight tasks were performed under two different scenarios (four tasks each). The software reconnaissance (SR) scenario had no restrictions: participants could use all of

Diver's features including the sequence diagram, trace activation, navigation between views, and trace filtering to implement the software reconnaissance technique. The non-software reconnaissance (NSR) scenario had only one restriction: participants could use all of Diver's features except for trace filtering, which was the key feature used to implement software reconnaissance (and had no other uses). The participants were told that software reconnaissance was the focus of the study and they were encouraged and reminded to use the technique for the SR scenario. These two scenarios were arranged so that the SR scenario was applied to the first four tasks attempted, and the NSR scenario to the last four tasks attempted.

The order in which a participant executed the eight tasks was determined as follows:

1. A unique order for the eight tasks was selected for each participant.
2. The tasks were divided into two groups of four tasks each: task group A and task group B.
3. For tasks in task group A, participants performed the SR scenario; for task group B, the NSR scenario.

Figure 4.1 displays how the order of the tasks was determined. For each odd numbered participant, the experimenter selected a sequence that was unique. Some combinations of tasks may have had unknown dependencies or links between them, and selecting different orders mitigated this risk. The eight tasks were then divided into the two task groups and assigned the SR or NSR scenario. For each even numbered participant, the two task groups were swapped so that they were assigned the other scenario. Ideally, an even number of participants was required so that each task group was attempted for each scenario. A task group could contain unknown dependencies just like the overall order, and these dependencies could affect how easy the tasks were under one scenario versus the other. To check that the assignment of the tasks was evenly distributed, the frequency of the tasks in each of the eight positions was monitored.

The participants' task orders used in the study are shown in Table 4.1

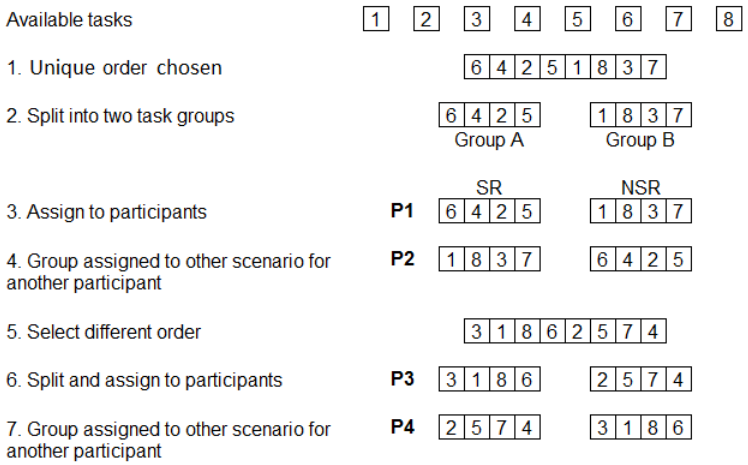


Figure 4.1: Depiction of task order assignment process followed to determine unique orders for all participants.

Participant	Tasks							
	SR				Non-SR			
P1	1	2	3	4	5	6	7	8
P2	5	6	7	8	1	2	3	4
P3	6	1	8	3	2	5	4	7
P4	2	5	4	7	6	1	8	3
P5	3	8	1	5	7	4	6	2
P6	7	4	6	2	3	8	1	5
P7	5	2	1	6	8	3	7	4
P8	8	3	7	4	5	2	1	6
P9	6	7	2	1	4	8	5	3
P10	4	8	5	3	6	7	2	1
P11	2	4	8	7	1	3	6	5
P12	1	3	6	5	2	4	8	7
P13	3	5	2	6	8	1	4	7
P14	8	1	4	7	3	5	2	6
P15	4	6	3	1	7	2	5	8
P16	7	2	5	8	4	6	3	1

Table 4.1: Task Allocation

4.2.4 Participant Feedback

The participants were required to fill in two questionnaire forms, which can be found in Appendix B and D. A pre-study questionnaire was administered to gain a better understanding of the study participants. The questions inquired about the participants' educational and technical experience. It also ascertained if they had knowledge of the game Tetris, which was being used in the study.

A post-study questionnaire was administered to collect feedback on the participants' experiences with Diver and software reconnaissance. The questions were divided into free-form and multiple choice answers. The questions with free-form answers asked participants about the helpful and unhelpful aspects of the tool and their impressions of the study. The multiple choice questions were based on a USE questionnaire [20] to gauge the usefulness, ease of use, ease of learning, and satisfaction of Diver and software reconnaissance.

4.2.5 Pilot Study

A pilot study was conducted to test the experiment and see if any problems occurred. It also allowed the experimenter to practice conducting the experiment. Three participants were recruited to take part in the pilot study. One participant was an undergraduate, one was a Masters student, and one was a PhD student. They were all familiar with Tetris and had played it before. In addition, they had all programmed in Java, and while only one had never used the Eclipse IDE, this participant had previously used a similar IDE called Netbeans. The first participant was only able to do one hour of the study, but he provided valuable advice and insights into the process. The pilot participants completed the study and provided input on the process.

All the tasks were finished successfully and no major problems were discovered with the limited pilot data. The NSR scenario had the quickest completion time for five of the eight tasks. For most tasks, the pilot participants' times were closely grouped together. This indicated that no tasks were likely biased towards one scenario or the other. The pilot participants' completion times also varied over the course of the two scenarios and did not show a decreasing trend, which would have indicated a learning curve.

Nonetheless, we identified a number of slight improvements for the study:

1. The training script needed improvement as some pilot participants requested

better explanations of features and their uses. The training script was changed to better highlight Diver's functionality and explain some use cases.

2. Based on the suggestion of the first pilot participant, a sample task (T0) was introduced for participant training. The goal of the tasks was to help participants understand the process involved with solving the tasks. This change was introduced for the final pilot participant.

3. A few of the participants were unsure of the level of detail required for answering the program understanding questions. Therefore, an explanation of the questions was added to the tutorial script along with the T0 sample task.

4. The participant instructions were updated to inform the participants that Diver is in beta and crashes on occasion. They were instructed to be patient while the traces were saving as Diver sometimes crashed when the user tried to interact with a trace before it had finished saving.

The pilot successfully highlighted ways to improve the study. The data from the three pilot participants was not used in the final study as the subsequent changes may have affected the results.

4.2.6 Data Analysis

The study followed a mixed-methods methodology with both quantitative and qualitative data being collected and analyzed by the researchers. The data collected from the pre-study questionnaire was used to compare experience with performance and discard outliers with too little or too much experience if necessary. This is discussed in Section 4.3.8.

For the first research question on navigation and interaction strategies (RQ1), the participants' activities were analyzed and their interactions with Diver were recorded. From this data, navigation patterns could be identified for both the successful and unsuccessful completion of the program understanding tasks. The differences between the patterns have lead to usage pattern recommendations for users.

The second research question on cognitive support (RQ2) was answered by using data from a number of sources. To analyse redistribution support, we looked at the participants' positive and negative feature feedback to determine their perceptions of the sequence diagram. For perceptual substitution theory support, we hypothesize

that participants will navigate to and read the code less often. We can compare differences in the *Jump To Code* feature usage between the two scenarios to determine if software reconnaissance improves support for the theory. Software reconnaissance filters the sequence diagram to only display unique method calls related to the functionality being sought. If the sequence diagram did not provide enough information or cues, a participant would need to jump to the code to read it in order to gain a better understanding of the program.

Ends-means reification can be studied by observing if the participants use the Package Explorer's filtered elements as the ends, and the navigation options to the sequence diagram or source code as the means. If Diver successfully maps the ends to the means by implementing a concrete structure, we hypothesize that the Package Explorer will be used more during the Software Reconnaissance scenario. The Package Explorer should have had more items filtered out, making it a good starting point for finding interesting methods. Participants would search through the Package Explorer looking for interesting methods, and then use the *Reveal In* functionality to jump to the sequence diagram. Information about related methods could be discovered in the sequence diagram and the method related to the functionality found. If the method was not found, the participant would go back to the Package Explorer and continue searching.

The final research question on efficiency and effectiveness (RQ3) was answered in two parts. The first part answered the efficiency question by using data from the *time-to-first-foothold* (TTFF) metric. TTFF measures how long it took the participants to discover one of the main software elements related to the functionality sought (see Section 3.2 for more detail). The time was recorded by the experimenter and then confirmed from the recordings. The second part of the answer addressed the effectiveness part of the research question. Effectiveness was evaluated by examining the number of tasks completed successfully for both scenarios. In addition to the performance data, the data from the post-study questionnaire about the participants' experience with Diver was used.

4.3 Results and Discussion

In this section, we present the results of the user study and the data collected from the sessions. We present the data on the cognitive support offered, navigation and interaction strategies used, and the task times and success rates of the participants.

This data was collected from a number of sources: video recordings, application logs, experimenter's observations, and the participants' questionnaire responses. After explaining the results, we interpret the results and discuss our findings.

The 16 participants attempted the tasks assigned them with varying levels of success. Success was evaluated on the basis of them finding a method related to the functionality identified by the task. Although at least one participant struggled due to lack of experience, all participant data was used during the analysis of the study and no outliers were discounted. Tasks were either completed correctly, completed incorrectly, or not completed. All participants followed the instructions and used software reconnaissance during the SR scenario.

4.3.1 RQ1: Interaction and Navigation Patterns

The first research question, which deals with the interaction and navigation patterns in participants' Diver usage, was answered by analyzing Diver's data logs. This analysis involved extracting the relevant entries from the logs, categorizing and compiling the data, and then comparing the usage patterns.

Four events related to Diver's various features were selected to analyze participants' navigation and interaction with the tool. The four events were: navigating to the Program Traces window, navigating to the Sequence Diagram window, using the *Jump To Code* feature, and using the *Reveal In* feature. Entries related to these four events were extracted from the log files. The extracted entries were categorized into four groups according to the scenario associated with the task attempt and the outcome of the attempt. The rate per minute for these events was determined by dividing the number of entries by the total time taken to complete the tasks.

Task attempts were classified as successful (*SUC*), incorrect (*INC*), did not finish (*DNF*), and did not attempt (*DNA*). *DNF* was for all incomplete tasks attempts, including when a participant ran out of time at the end of a session or chose to move on to another task. *INC* represented incorrectly completed tasks, which occurred when a participant mistakenly thought they had found a method related to the functionality being sought. Participants may have been on the right track and just selected an unrelated method or they could have been completely on the wrong track. As a result, the data from the few task attempts classified as *INC* were not included in the navigation analysis of this study.

Table 4.2 show the data for the four events divided into the four groups, which

are determined by task outcome and scenario type.

The Program Traces section of Table 4.2 shows how many times per minute the participants navigated to the Program Traces window. The Program Traces features, such as switching threads or activating traces, were accessed slightly more during the SR scenario for both *SUC* and *DNF* tasks. Incomplete task attempts saw an increase in navigation to the window irrespective of the scenario.

Event		SR	NSR
Program Traces	SUC	6.59	5.39
	DNF	9.67	6.17
Sequence Diagram	SUC	10.36	9.13
	DNF	8.42	12.00
Jump To Code	SUC	10.93	12.24
	DNF	6.33	13.50
Reveal In	SUC	1.41	1.63
	DNF	1.67	1.50

Table 4.2: Diver Usage Statistics

The navigation data for the Sequence Diagram window is also presented in Table 4.2. The data shows the average number of times per minute that participants navigated to the window to exploring the sequence diagram. The participants navigated from the Package Explorer, source code, and Program Traces window 10.36 times per minute for successfully completed tasks under the SR scenario. The average was slightly less under the NSR scenario, measuring 9.13 for successful tasks. There was a decrease under the SR scenario for incomplete tasks compared to those completed successfully. However, under the NSR scenario, there was a slight increase in navigation to the Sequence Diagram window during tasks that were not completed.

The *Jump To Code* section of Table 4.2 shows the usage data for the *Jump To Code* feature. The feature was used less under the SR scenario compared to the the NSR scenario for both successful and incomplete tasks. Successful tasks had higher usage rates under the SR scenario, however, the incomplete tasks had a slightly higher rate under the NSR scenario.

The *Reveal In* section of Table 4.2 shows how often the *Reveal In* feature was used on average. The feature usage was similar throughout the four groups, however, successful task attempts under the SR scenario had the lowest rate.

4.3.2 RQ1: Interpretation of the Results

The participants' interactions highlighted what navigation strategies were used to successfully complete the tasks. Once classified, *INC* task attempts were not included, as explained in Section 4.3.1. By combining the navigation data with observations made by the experimenter, a more complete view of participants' interactions with Diver can be created.

4.3.2.1 Program Traces window

The results in Table 4.2 show that the program traces window was used more during the SR scenario than the NSR scenario for both *SUC* and *DNF* tasks. This was expected because the software reconnaissance technique required participants to record and manage two traces. The TTFF measurement includes the time taken to record the traces, as this differentiated the SR scenario. The data confirms that the addition of another trace under the SR scenario results in an increase in Program Traces window usage.

The data also showed an increase in the use of the program traces window during unsuccessful task attempts, especially a 47% increase under the SR scenario. Two possible reasons for the increase, which the experimenter observed, were tracing problems and trouble identifying the correct thread. The main causes of the tracing problems were recording a trace that was either too large or did not contain the correct methods. For instance, **P4** recorded an extremely large trace for Task 4 after a lengthy attempt to complete one line in Tetris. As mentioned in the training, such a large trace would be difficult to use in Diver. **P4** had to eventually record another, smaller trace, and therefore used the Program Traces window again. The mistake tracing lead to increased use of the Program Traces window and contributed to an unsuccessful task attempt.

These observations identify navigation patterns that highlight areas where users can negatively affect their program comprehension tasks and lose their way with Diver.

4.3.2.2 Sequence Diagram window

We hypothesized that software reconnaissance was better at supporting ends-means reification and that participants would use the sequence diagram as the means to identify a method related to the functionality being sought. The participant behaviour

we expected was a *hypothesis and confirmation* approach where participants would look for possible interesting methods in the Package Explorer and then try to confirm their relevance in the sequence diagram and code.

The data found in Table 4.2 in the Sequence Diagram section shows increased navigation to and from the sequence diagram for successful and SR scenario tasks. This increase can be explained by various possible navigation patterns. One pattern, related to the ends-means reification hypothesis mentioned above, has participants navigating to and from the Package Explorer more for successfully completed tasks using software reconnaissance. The *Reveal In* feature data found in Table 4.2 would need to match the navigation increase seen by the sequence diagram window to support this navigation pattern.

4.3.2.3 *Reveal In* feature

Taking into account the *Reveal In* feature data, the pattern of frequent navigation between the Package Explorer and sequence diagram can be ruled out as a possible navigation pattern. For successfully completed tasks under the SR scenario, the feature was never used more than twice per minute on average, well below the average of around ten times per minute for the sequence diagram.

This statement is backed up by observations that saw participants find an interesting method with the help of software reconnaissance and searching the Package Explorer. The most successful participants, like **P1**, easily found relevant methods in the Package Explorer and then only used the *Reveal In* feature once to find where the method was called in the sequence diagram. This is discussed further in Section 4.3.4.

4.3.2.4 *Jump To Code* feature

The data from the *Jump To Code* feature section in Table 4.2 completes the picture of the participants' interactions with Diver. The data shows that there was an increase in the use of the *Jump To Code* feature for the tasks successfully completed under SR scenario tasks, similar to the *Reveal In* usage. This indicates that participants were moving to and from the sequence diagram and source code to gain an understanding of the program and methods of interest. This proved successful for the SR scenario but not for the NSR scenario.

The usage under the NSR scenario shows that participants were less successful

when they frequently navigated between the sequence diagram and source code. The experimenter observed that participants who struggled to gain a foothold in the sequence diagram through the Package Explorer were more likely to get lost in the sequence diagram. This lead to them switching between exploring the diagram and the source code, often with little success. Some of the successful tasks attempts were solved by participants reading through the code, but mostly they included use of the sequence diagram as well.

4.3.2.5 Summary

Software reconnaissance was found to influence navigation and interaction patterns in a number of different ways. The Program Traces window was used more frequently for tasks involving software reconnaissance. The *Reveal In* feature was used less, contrary to what was expected, as a result of software reconnaissance effectively filtering the Package Explorer and allowing footholds to be gained in the sequence diagrams. The *Reveal In* feature was used the least for successfully completed tasks using software reconnaissance — a fact, that together with observations of the participants success, indicates that the *Reveal In* method was more effective with the software reconnaissance technique.

More frequent navigation between the sequence diagram and source code was seen by participants using software reconnaissance. This was attributed to short enquiry episodes to confirm the operation of methods, as opposed to participants without software reconnaissance who lingered on a specific window for a longer period of time. In response to the research question, these results put forward navigation and interaction patterns that were linked to a higher chance of success completing tasks. The data and observations also found navigation patterns that led to slower, less successful task attempts. All of these patterns can be used to train future users of the tool and help improve the design of Diver.

4.3.3 RQ2: Cognitive Support

The second research question on whether improved cognitive support is offered by software reconnaissance can be ascertained by exploring participants' interaction behaviour and feedback about the tool. The logging of the interactions showed the use of various pieces of functionality within the Eclipse IDE. The observations of the participants and their responses to the questionnaire were used to create a better picture of the experience of using Diver with and without the aid of software reconnaissance. We investigate the three main cognitive support theories (redistribution support, perceptual substitution, and ends-means reification) and how software reconnaissance influences their implementation in Diver.

4.3.3.1 Redistribution Support

As mentioned in Chapter 2, the redistribution support theory involves when a person stores data externally from the mind to solve a problem. For example, the user can use the sequence diagram to store the method calls of a trace. The trace-focused user interface also stores information about a trace's unique method calls. The aim of these two features was to solve cognition overload, especially with large traces [22]. Diver was designed to provide redistribution support for users so that they could complete feature location and program comprehension tasks. Software reconnaissance was introduced to improve this support. By analyzing the perceptions of the participants, we can determine if software reconnaissance does improve the support. Our analysis of the participants' opinions of Diver's sequence diagrams and related features revealed three main themes related to the problems they experienced:

1. Size and complexity
2. Navigation
3. Awareness

There were both positive and negative comments from the participants' experience using Diver with and without software reconnaissance. Many participants complained about the size and complexity of the diagrams that they had to work with during the study. These complaints were mentioned without talking about the presence of software reconnaissance; however, the positive comments below do make specific mention of the software reconnaissance features. **P8** was "overwhelmed" by the amount

of data, and **P7** found the diagrams intimidating and very "dense" with information. For navigation, **P2**, **P4**, and **P14** had problems navigating and scrolling around the diagram view. **P10** admitted to getting "lost" in the large diagrams. Awareness was not mentioned as much, but **P7** complained about not being able to zoom to get a better perspective. **P5** had a similar problem with the long horizontal arrows that connected to artifacts that were not being displayed. **P13** made the point that the tool was "rough around the edges which takes additional cognitive capacity away from the already complicated task."

Software reconnaissance did feature in much of the positive participant feedback, which focused on the features involving the technique. Most participants mentioned that using software reconnaissance to filter the Package Explorer and sequence diagram was the most helpful technique for completing the assigned tasks. This filtering of the IDE based on the contents of the trace is, in part, based on the redistribution support theory. Of the 16 participants, 13 pointed to the filtering of the IDE being the most useful feature, especially with the help of software reconnaissance. For instance, **P8** was only "overwhelmed" by the sequence diagram when software reconnaissance filtering was not applied. **P3**, **P6**, and **P15** all agreed that the filtering provided by software reconnaissance made the sequence diagram usable and helpful when completing the feature location and program understanding tasks. The remaining three participants, who did not list filtering as the most useful feature, mentioned the *Reveal In* and/or *Jump To Code* features. The *Reveal In* feature is the basis for our hypothesis on perceptual substitution and the *Jump To Code* feature is the basis for our hypothesis on ends-means reification — both of which are discussed below.

4.3.3.2 Perceptual Substitution

The perceptual substitution theory was defined in Chapter 2 as a way of transforming how data is displayed to support better cognitive processing and, therefore, understanding. A simple example is replacing a data table in favour of a chart when displaying data. Similarly, Diver was built around the idea of visualising an execution trace as a sequence diagram. The sequence diagram provides a visual substitute for the textual information recorded by tracing a program's method calls. As discussed previously, Diver's sequence diagram enables users to visually process data instead of having to analyze the source code.

In this study, we investigated the effect that software reconnaissance has on using

Diver. To measure the perceptual substitution support offered, we looked at the *Jump To Code* feature usage data. We hypothesized that if software reconnaissance resulted in participants reading less code and using the sequence diagram more, it supported perceptual substitution better than Diver without software reconnaissance.

Every participant used the *Jump To Code* feature at least once for both the SR and NSR scenarios. Table 4.3 shows the usage of the *Jump To Code* feature by participants. The mean number of times the *Jump To Code* feature was used per task is displayed for both SR and NSR scenarios. Alongside the averages is the percentage difference between SR and NSR. The mean number of times *Jump To Code* was used for SR was 9.82 times compared to 11.86 for NSR. The difference represents a 20.78% increase in use with a standard deviation of 8.64.

A hypothesis t-test was performed and the p-value was calculated to be 0.195, meaning that the chance of getting an mean of 11.86 or more was 19.5%. The null hypothesis, which stated that the NSR scenario has no effect on the *Jump To Code* feature usage, could not be rejected as the high p-value indicated the mean was statistically insignificant.

Participant	SR	NSR	Diff (%)
P1	1.5	2	33.33
P2	21.5	8.5	-60.47
P3	2	0.67	-66.67
P4	5.33	18.5	246.88
P5	13.33	33	147.50
P6	19	17.33	-8.77
P7	1.33	0.5	-62.50
P8	5.33	3.33	-37.50
P9	12.5	24.67	97.33
P10	16	7	-56.25
P11	4.67	19	307.14
P12	5.33	12	125.00
P13	1.33	4.33	225.00
P14	28	16.67	-40.48
P15	6	11	83.33
P16	14	11.33	-19.05
Ave	9.82	11.86	20.78

Table 4.3: The *Jump To Code* feature usage for all participants

4.3.3.3 Ends-means Reification

Chapter 2 explains the complex ends-means reification theory in detail. In Diver, ends-mean reification is applied when the filtered Package Explorer is used as a navigation start point when searching for a method in a sequence diagram. It is also applied when using a compiler list to jump to places in the code where errors occurred in order to fix them. The *Reveal In* feature in Diver constitutes the participant applying the ends-means reification theory. We can compare the usage of the feature from one scenario to the other to determine which scenario provides better support according to the theory.

The feature usage was analyzed to find out which scenario favoured the *Reveal In* feature. Table 4.4 displays the *Reveal In* usage statistics for each participant. The mean number of times *Reveal In* was used per task is shown for the SR and NSR scenarios. The percentage differences of NSR versus SR are included as well. **P1** and **P2** showed no difference in their usage between scenarios. They both used *Reveal In* once per task (on average) regardless of the scenario. **P4** never used the feature at all, and **P7** and **P11** only used it for one of the scenarios. The NSR scenario mean was 1.73 times per task and the SR scenario mean was 1.42 per task. The difference of -18.32% had a standard deviation of 1.17.

A hypothesis t-test was performed and the p-value was calculated to be 0.105, meaning that the chance of getting an mean of 1.42 or less was 10.5%. The null hypothesis, which stated that the SR scenario has no effect on the *Reveal In* feature usage, could not be rejected as the high p-value indicated the mean was statistically insignificant.

4.3.4 RQ2: Interpretation of the Results

The three cognitive support theories, introduced in Chapter 2, and their support in Diver were investigated by analyzing the data presented above.

4.3.4.1 Redistribution Support

The participants' positive perceptions of the tool offered overwhelming support for the software reconnaissance filtering of the Package Explorer and sequence diagram. Participants used software reconnaissance to filter out the noisy methods from the two windows and more easily complete the feature location and program understanding

Participant	NSR	SR	Diff (%)
P1	1	1	0
P2	1	1	0
P3	3.67	1.75	-52.27
P4	0	0	0
P5	3.25	2.67	-17.95
P6	0.33	2	500
P7	1	0	N/A
P8	0.67	0.33	-50
P9	2.33	0.75	-67.86
P10	2	1	-50
P11	0	0.67	N/A
P12	3	2.67	-11.11
P13	0.67	1.67	150
P14	2.33	2.5	7.14
P15	4.5	2	-55.56
P16	2	2.67	33.33
Ave	1.73	1.42	-18.32

Table 4.4: The *Reveal In* feature usage data for all participants

tasks. The participants' negative perceptions included many complaints about the sequence diagrams containing too much information and being difficult to navigate. Based on these perceptions, the filtered Package Explorer and sequence diagram were a welcome improvement as a useful storage mechanism for the unique method calls in the main trace. Participants would otherwise have had to keep track of these calls in their memory. The observations of the participants showed that they quickly navigated the filtered method calls without having to identify or analyse them. This allowed the participants to focus their attention on the remaining unfiltered method calls, which represented a smaller, more manageable search space. However, the support voiced for software reconnaissance filtering was not conclusive. The ordering of the SR and NSR scenarios could have affected the feedback given by the participants. Taking away software reconnaissance could have frustrated them, and therefore, the sequence diagram and lack of filtering could have been negatively biased.

4.3.4.2 Perceptual Substitution

We hypothesized that participants would use the *Jump To Code* feature less (clicking on a method in the sequence diagram and opening the related source code) when

there was an increase in the cognitive support provided by the sequence diagram. The mean average increase in difference (20.78%) between SR and NSR scenarios supports our hypothesis. Some participants were observed failing to gain a useful foothold in the sequence diagram due to the lack of filtering provided by software reconnaissance. This resulted in them unsuccessfully navigating around the sequence diagram without a clear search strategy. As the data shows, they were more likely to click on methods and navigate to the code to try and understand what the method did. However, the results are statistically insignificant according to the calculated p-value.

We discovered some themes behind the usage of the various features, such as sticking with tried and tested methods of solving the tasks. In the questionnaires, **P2** and **P4** mentioned finding it hard to overcome old habits, such as reading code snippets, to understand the program better. **P2** did say that after a while she overcame the sequence diagram learning curve and started using it more. Other users could also have reverted to old habits by using *Jump To Code* more than users who were able to use the sequence diagram effectively. **P6** highlighted that the sequence diagram learning curve was a hindrance, and so the large differences in *Jump To Code* use could indicate how participants overcame the learning curve at different rates. Another theme that was observed was forgetting about features: some participants forgot about certain available features either in the beginning or at some time during the experiment. This problem lead to the sequence diagram being more difficult to use and reading the code being a more attractive option. In almost all cases, the sequence diagram was used to identify interesting methods to navigate to in the code and was never completely circumvented.

4.3.4.3 Ends-means Reification

Our hypothesis on the support for ends-means reification is based on the *Reveal In* feature which provides navigation between the Package Explorer and sequence diagram windows in Eclipse. Our hypothesis is that the *Reveal In* functionality would be used more for the SR scenario. Use of the *Reveal In* feature varied from one participant to the next, which can be explained by participants using different strategies to complete the tasks. The difference in the mean values of the SR and NSR scenario usage does not support the hypothesis stated above. However, the p-value was 0.105 from a t-test, and therefore, the results are deemed insignificant.

This shows that it is possible that the hypothesis might have been incorrect to begin with.

Given that Tetris was not a large or complex program, using *Reveal In* was still a viable strategy without software reconnaissance. The experience gained by completing tasks for the SR scenario first could have given the participants knowledge about the various classes that encouraged them to use the Package Explorer, and accordingly, they used the *Reveal In* feature more. With that in mind, having *Reveal In* used less during the SR scenario could point to the effectiveness of the filtering of the Package Explorer. The participants had a smaller list of possible options; their searches would typically take less time and the chance of finding the correct method using *Reveal In* would be much higher. **P1** and **P2** were the most successful participants as they finished all eight tasks. They both averaged one use of *Reveal In* per task for both SR and NSR scenarios. Again, this points to the fact that using *Reveal In* less was responsible for a more effective performance.

4.3.4.4 Summary

The investigation into the support for the three cognitive support theories produced results in response to the second research question. Redistribution support was shown by the participants' positive perceptions of Diver with software reconnaissance and the support offered with the improved filtering that the technique provided. Perceptual substitution support was shown for software reconnaissance through the increased use of the sequence diagram instead of reading the source code. The results, however, were not statistically significant. Ends-means reification support was not supported by the results. The results were the opposite of the expected outcome, as the *Reveal In* feature was used less with software reconnaissance. Again, this result was not statistically significant due to a high p-value.

4.3.5 RQ3: Effectiveness and Efficiency of Software Reconnaissance

The effectiveness and efficiency of the software reconnaissance technique in Diver was investigated by collecting task completion metrics as well as timing data for all the tasks attempted. The completion metrics are tabulated for each participant in Table 4.5. The timing results per task can be seen below in Tables 4.6 - 4.13.

4.3.5.1 Effectiveness

Table 4.5 contains each participant's task completion results from the study. The data is divided into the two scenarios — SR and NSR — and shows the number of successes and failures. The *Tried* column contains the number of tasks attempted. The tasks were classified as either successful (*SUC*), incorrect (*INC*), or did not finish (*DNF*).

Table 4.5 shows that an almost identical number of tasks were attempted under each scenario (50 for SR versus 49 for NSR). For SR, 39 out of the 50 (78%) were completed successfully, and for NSR, 34 out of 49 (69.4%) were completed successfully.

Participant	SR Tasks					NSR Tasks				Total
	Tried	SUC	DNF	INC		Tried	SUC	DNF	INC	SUC
P1	4	4	0	0		4	4	0	0	8
P2	4	4	0	0		4	4	0	0	8
P7	4	3	1	0		4	4	0	0	7
P13	3	3	0	0		3	3	0	0	6
P9	4	3	1	0		3	3	0	0	6
P12	3	3	0	0		2	2	0	0	5
P15	3	3	0	0		2	1	1	0	4
P16	3	1	2	0		3	3	0	0	4
P5	3	2	1	0		4	2	2	0	4
P3	4	3	0	1		3	1	1	1	4
P10	2	2	0	0		2	1	1	0	3
P6	2	2	0	0		3	1	2	0	3
P11	3	2	1	0		2	1	1	0	3
P8	3	2	1	0		3	1	1	1	3
P14	2	2	0	0		3	1	1	1	3
P4	3	0	1	2		4	2	2	0	2
Tot	50	39	8	3		49	34	12	3	73

Table 4.5: The task completion statistics ordered by the total number of tasks completed successfully

4.3.5.2 Efficiency

We measured task completion efficiency using the TTFF results by comparing the times recorded from the two scenarios. Tables 4.6 - 4.13 contain each task's TTFF results. Each table entry contains a participant code, scenario code, order number of task for the participant, and the *time-to-first-foothold*. The results are ordered by the TTFF times from shortest to longest.

Task 1: Task 1 was completed successfully by 11 out of the 13 participants that attempted it. The best time for this task was 02:10 by **P2** using the NSR scenario. The average time for task 1 using the NSR scenario was 02:45, and for the SR scenario, it was 04:35.

P. Code	Scenario	Attempt	TTFF
P2	NSR	5th	02:10
P1	SR	1st	02:42
P13	NSR	6th	02:45
P5	SR	3rd	03:00
P4	NSR	6th	03:01
P8	NSR	7th	03:03
P9	SR	4th	03:30
P14	SR	2nd	03:35
P7	SR	3rd	05:38
P12	SR	1st	06:10
P3	SR	2nd	07:27
P6	NSR	7th	DNF
P11	NSR	5th	DNF
P10	NSR	8th	DNA
P15	SR	4th	DNA
P16	NSR	8th	DNA

Table 4.6: Task 1 results

Task 2: Out of the 11 participants that completed the task, only **P4** completed it incorrectly. The best time of 01:40 was recorded by both **P12** and **P3** during the NSR session. The average time for task 2 using the NSR scenario was 03:34, and for the SR scenario was 05:53.

Task 3: Only 1 of the 14 participants who attempted the task did not complete it successfully. This task produced the quickest TTFF of the entire study; 01:02 by **P2**

P. Code	Scenario	Attempt	TTFF
P12	NSR	5th	01:40
P3	NSR	5th	01:40
P2	NSR	6th	01:45
P9	SR	3rd	02:25
P15	NSR	6th	02:45
P1	SR	2nd	04:35
P13	SR	3rd	05:30
P11	SR	1st	08:13
P7	SR	2nd	08:40
P5	NSR	8th	10:00
P4	SR	1st	INC
P8	NSR	6th	DNF
P16	SR	2nd	DNF
P14	NSR	7th	DNF
P6	SR	4th	DNA
P10	NSR	7th	DNA

Table 4.7: Task 2 results

under the NSR scenario. The average time for the NSR scenario was 02:28, and for the SR scenario was 05:44 (more than double the length).

Task 4: Four participants did not finish task 4. The remaining 10 of the 14 participants were successful as no one completed the task incorrectly. The best time was 03:00 by **P2** under the NSR scenario. The average times were high: 07:19 for NSR and 10:44 for SR.

Task 5: Task 5 was attempted by 10 participants and only five participants finished it successfully. A further four completed the task unsuccessfully, the most for any task in the study. The remaining person was unable to complete the task. The lowest TTFF recorded was 02:25 under the NSR scenario by **P1**. The average time for the two successful tasks under the NSR scenario was 05:47, and the three tasks under the SR scenario had an average of 05:13. This was the only task where the average for the SR scenario was lower than the average for the NSR scenario.

Task 6: Out of the 11 participants that attempted the task, 9 completed it successfully and 2 did not finish. The best time for this task was 02:00 by **P1** under the NSR scenario. The average time for the NSR scenario was 04:27, and for the SR

P. Code	Scenario	Attempt	TTFF
P2	NSR	7th	01:02
P7	NSR	6th	01:05
P16	NSR	7th	01:19
P12	SR	2nd	01:50
P1	SR	3rd	01:50
P14	NSR	5th	02:05
P3	SR	4th	02:50
P11	NSR	6th	03:25
P13	SR	1st	03:40
P8	SR	2nd	04:28
P6	NSR	5th	05:50
P15	SR	3rd	11:54
P5	SR	1st	13:35
P4	NSR	8th	DNF
P9	NSR	8th	DNA
P10	SR	4th	DNA

Table 4.8: Task 3 results

scenario was 05:32.

Task 7: Task 7 had 6 successful participants, only slightly better than task 5. A further four participants did not complete the task. The best time for this task was 02:35 by **P1** under the NSR scenario (almost half the next closest time of 05:00). The average time for the NSR scenario was 04:28, and for the SR scenario was 09:03.

Task 8: Task 8 was completed by 9 out of the 13 participants that attempted it. An additional participant completed the task but did so incorrectly. The remaining 3 participants were unable to finish the task. The best time for the task was 03:10 by **P1** under the NSR scenario. The average time for the NSR scenario was 04:32, and for the SR scenario was 08:21.

P. Code	Scenario	Attempt	TTFF
P2	NSR	8th	03:00
P7	NSR	8th	03:55
P1	SR	4th	05:45
P9	NSR	5th	07:37
P16	NSR	5th	08:25
P15	SR	1st	08:52
P12	NSR	6th	09:40
P10	SR	1st	10:38
P13	NSR	7th	11:15
P6	SR	2nd	17:40
P3	NSR	7th	DNF
P4	SR	3rd	DNF
P5	NSR	6th	DNF
P11	SR	2nd	DNF
P8	SR	4th	DNA
P14	SR	3rd	DNA

Table 4.9: Task 4 results

P. Code	Scenario	Attempt	TTFF
P1	NSR	5th	02:25
P2	SR	1st	03:20
P7	SR	1st	04:53
P13	SR	2nd	07:25
P9	NSR	7th	09:10
P3	NSR	6th	INC
P4	SR	2nd	INC
P8	NSR	5th	INC
P14	NSR	6th	INC
P16	SR	3rd	DNF
P5	SR	4th	DNA
P6	NSR	8th	DNA
P10	SR	3rd	DNA
P11	NSR	8th	DNA
P12	SR	4th	DNA
P15	NSR	7th	DNA

Table 4.10: Task 5 results

P. Code	Scenario	Attempt	TTFF
P1	NSR	6th	02:00
P12	SR	3rd	02:45
P16	NSR	6th	03:00
P2	SR	2nd	03:05
P5	NSR	7th	03:30
P9	SR	1st	05:05
P15	SR	2nd	08:20
P3	SR	1st	08:27
P4	NSR	5th	09:17
P7	SR	4th	DNF
P10	NSR	5th	DNF
P6	SR	3rd	DNA
P8	NSR	8th	DNA
P11	NSR	7th	DNA
P13	SR	4th	DNA
P14	NSR	8th	DNA

Table 4.11: Task 6 results

P. Code	Scenario	Attempt	TTFF
P1	NSR	7th	02:35
P7	NSR	7th	05:00
P10	NSR	6th	05:50
P2	SR	3rd	06:15
P16	SR	1st	08:06
P6	SR	1st	12:48
P5	NSR	5th	DNF
P8	SR	3rd	DNF
P9	SR	2nd	DNF
P15	NSR	5th	DNF
P4	SR	4th	DNA
P11	SR	4th	DNA
P12	NSR	8th	DNA
P13	NSR	8th	DNA
P14	SR	4th	DNA
P3	NSR	8th	DNA

Table 4.12: Task 7 results

P. Code	Scenario	Attempt	TTFF
P1	NSR	8th	03:10
P9	NSR	6th	03:20
P2	SR	4th	04:10
P11	SR	3rd	04:40
P7	NSR	5th	05:35
P13	NSR	5th	06:05
P8	SR	1st	06:51
P10	SR	2nd	12:35
P14	SR	1st	13:30
P3	SR	3rd	INC
P4	NSR	7th	DNF
P5	SR	2nd	DNF
P6	NSR	6th	DNF
P12	NSR	7th	DNA
P15	NSR	8th	DNA
P16	SR	4th	DNA

Table 4.13: Task 8 results

4.3.5.3 Time-To-First-Foothold Trends and Aggregates

Figure 4.2 shows the TTFF results of the participants with the most successfully-completed tasks. The best fit trend lines are included to show the trends of the most successful participants' times over the course of the study. The graph has the TTFF times plotted in the order they were attempted. The first four TTFF times were accomplished under SR and the last four under NSR. The results are not normalized and differences in the difficulties of the tasks could easily affect the trend lines, but the graph serves to show that there was no extreme bias showing a learning effect that affected the later tasks.

Figures 4.3 - 4.10 plots the TTFF results as boxplots for each task, with the averages for both scenarios, side by side. This chart shows the range of TTFF times for both scenarios and makes it easier to see which tasks favoured one scenario over the other.

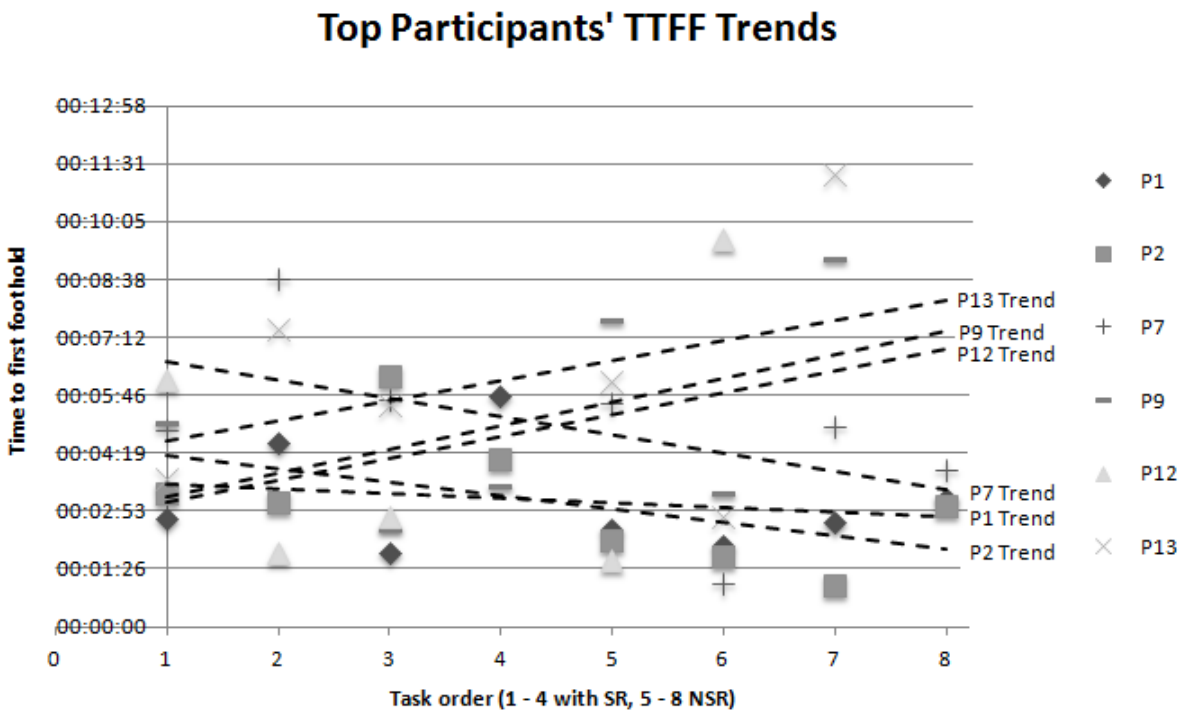


Figure 4.2: The TTFF results and their performance trends for the top six participants.

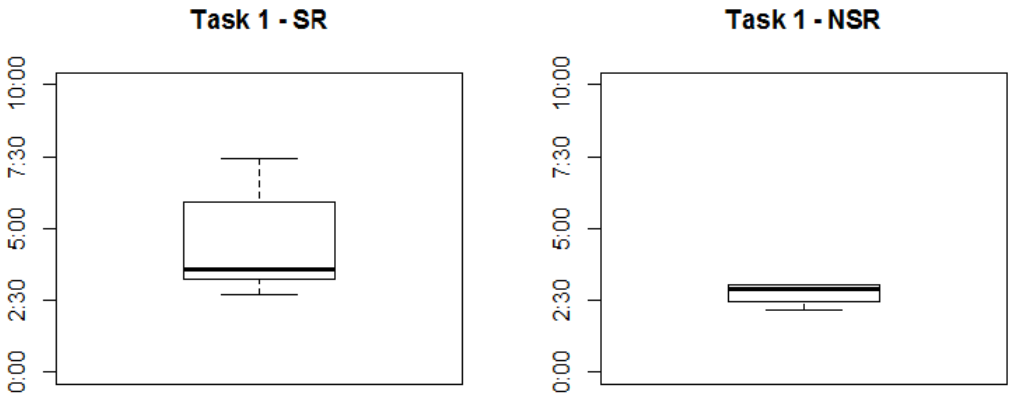


Figure 4.3: The Task 1 boxplots for TTFF times under both scenarios

4.3.6 RQ3: Interpretation of the Results

4.3.6.1 Effectiveness

The participants' effectiveness at completing tasks while using software reconnaissance can be measured by comparing the successfully completed tasks under the SR scenario to those completed under the NSR scenario. Table 4.5 shows the completion statistics for each participant as well as the totals. Based on tasks successfully completed, the top six participants were **P1**, **P2**, **P7**, **P13**, **P9**, and **P12**. **P1** and **P2** completed all the tasks in the given time, and between the two of them, shared seven out of the eight fastest TTFF times for the tasks. The least successful participant was **P4** who completed only two tasks successfully. The averages show 78% of tasks were completed successfully under SR compared to 69.4% under NSR. This shows an 8.6% improvement in the effectiveness of completing the tasks successfully while using software reconnaissance. The standard deviation was 29.5% for all the participants and the p-value was 0.122 from a t-test, which makes the improvement insignificant. This can be attributed to there being only 16 participants and a sample size of 99 attempted tasks. From the totals, it can also be noted that only three tasks were completed incorrectly in each of the scenarios.

While the statistics could not confirm the effectiveness of software reconnaissance, the SR scenario did see more tasks being completed successfully in spite of it being scheduled first. As the study was investigating software reconnaissance, placing the SR scenario first put it at a disadvantage due to the learning effect. For example, **P4** did not complete any tasks during SR but completed two during NSR. She specifically

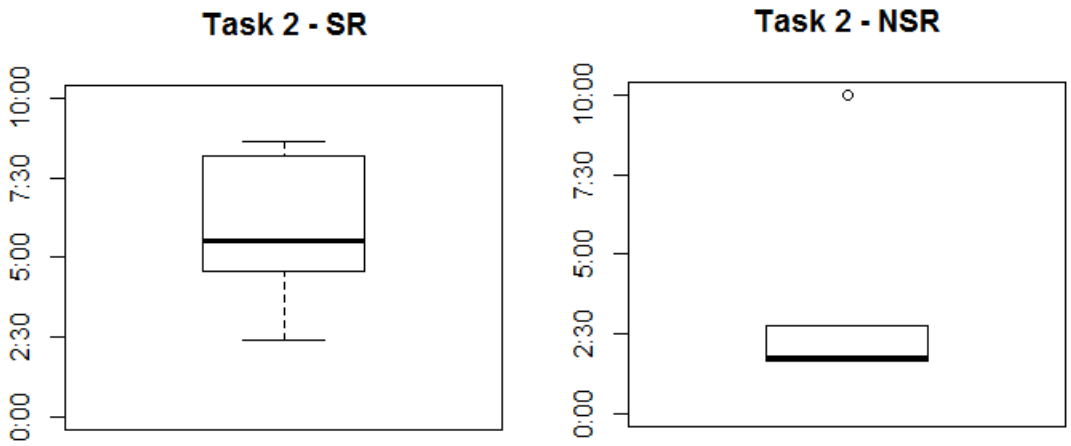


Figure 4.4: The Task 2 boxplots for TTFF times under both scenarios

mentioned the new tool and old habits as affecting her performance. She first relied heavily on reading the code and only started to use the sequence diagram later on. While the order of the scenarios put software reconnaissance at a disadvantage, it did not help NSR dominate the effectiveness statistics; many of the top participants were able to effectively complete tasks under SR. This could point to some participants being slower to acclimatise to the new tool than others. Having a set time for training and practice could have left some participants without enough time to get to know the tool. In some cases, participants forgot about certain features such as collapsing the Java classes' life lines in the sequence diagrams.

The counterargument to the learning effect is that, by putting the SR scenario first, the participants were taught how to use Diver with software reconnaissance in a specific way. Once software reconnaissance was removed, they continued trying to use Diver the same way, to their detriment. This is not the case for Diver and software reconnaissance because the improvements added by the technique were incremental. With the exception of sequence diagram filtering, software reconnaissance improves the filtering of features already in Diver; no new features were added to change the possible ways to use Diver. Software reconnaissance did, however, make some options more appealing to use. It improved the filtering of the Package Explorer so that fewer classes and methods were displayed. It also greyed out those filtered methods in the sequence diagram to improve the visibility of the unique method calls. With these enhancements removed, the Package Explorer was still filtered by activating a trace

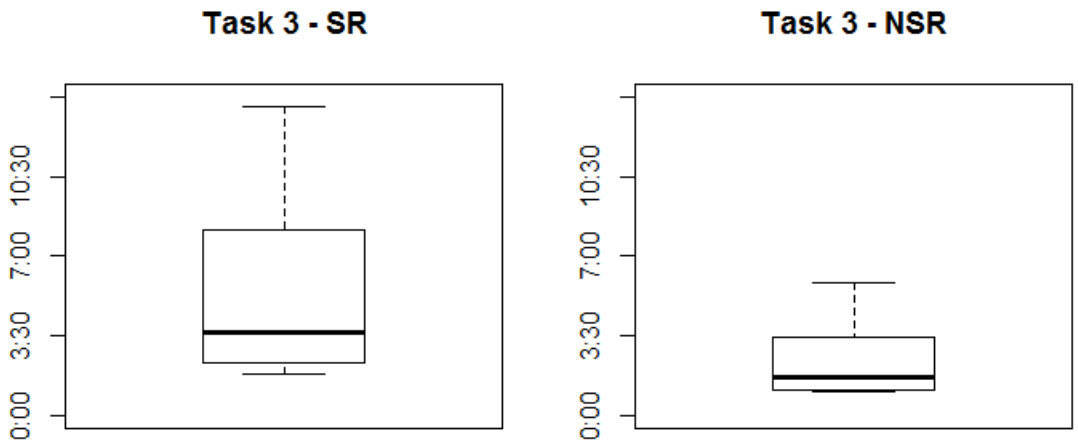


Figure 4.5: The Task 3 boxplots for TTFF times under both scenarios

and was a viable option for searching for interesting methods. Likewise, the sequence diagram was still useful for finding out what happened during the trace, even if it was less readable due to a lack of greying out of noisy methods.

4.3.6.2 Efficiency

The time taken to locate the functionality being sought in each task represents the metric for measuring efficiency of completing the tasks. The *time-to-first-foothold* was recorded to measure how long it took participants to locate a key method related to the feature under investigation. Overall, Tables 4.6 - 4.13 show that footholds were gained the fastest under NSR for all eight tasks. NSR tasks also had lower TTFF averages for all tasks except one (T5). Despite this, some tasks saw smaller differences than others between NSR and SR. T2, T3, and T7 saw the three fastest TTFFs come from NSR. T2 and T3 were easier tasks with the features being easy to isolate and having well-named methods. The NSR average for T3 was half that of SR, which is understandable given that the feature — setting the game speed through the menu — involved few noisy methods, making software reconnaissance less useful. T7 was more difficult, as the averages of 04:28 for NSR and 09:03 for SR show. This task required the participants to search for where the Weighted Score was calculated. This score represented how close the game was to ending. The score was calculated every time a Tetris block would land, but the score did not necessarily change for the first few blocks that landed. This caused a problem for some participants using software

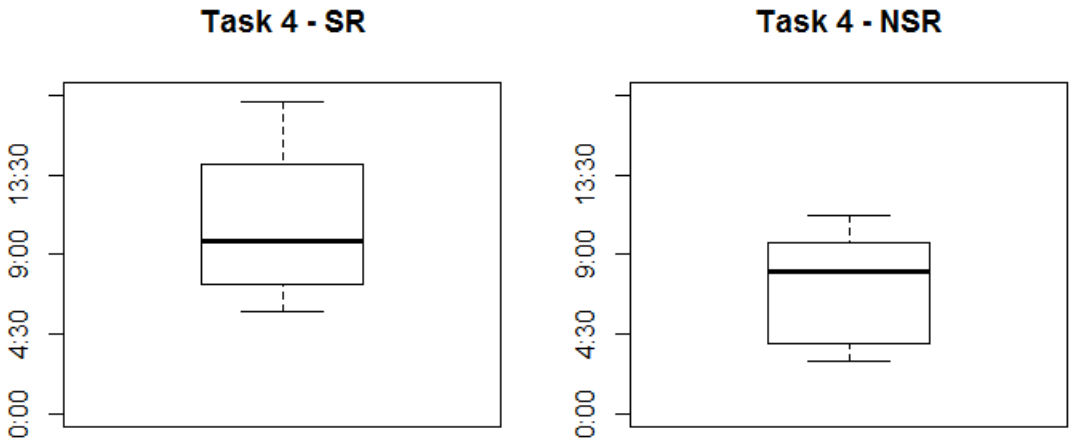


Figure 4.6: The Task 4 boxplots for TTFF times under both scenarios

reconnaissance who mistakenly recorded the method in their filter trace, thus filtering all of the interesting methods out of the IDE. For T5, SR had a slightly lower average than NSR (05:13 versus 05:47), but as with all the results plotted in Figures 4.3 - 4.10, there are overlaps with the times and NSR is not clearly superior to SR.

We observed that discovering an interesting method could happen just as easily through thorough searching as random discoveries. The nature of the software reconnaissance technique requires the recording of two traces, which puts it at a time disadvantage from the beginning. In addition to the time taken running Tetris and saving a second trace, the participants had to strategize how to isolate the functionality and record an effective filtering trace. From the results, this could have contributed to NSR being responsible for quicker TTFF times. Software reconnaissance emphasized the usefulness of the Package Explorer for uncovering interesting methods very quickly and navigating from there to the sequence diagram. Participants used this same technique for NSR and were able to discover well-named methods quickly. Two factors could have helped with this.

The learning effect mentioned above could have played a part in the lower NSR TTFF times achieved in the second session after participants had gained knowledge of the project and its classes. This knowledge was seen to direct participants to potentially interesting classes that they had seen before, such as the Game or Player

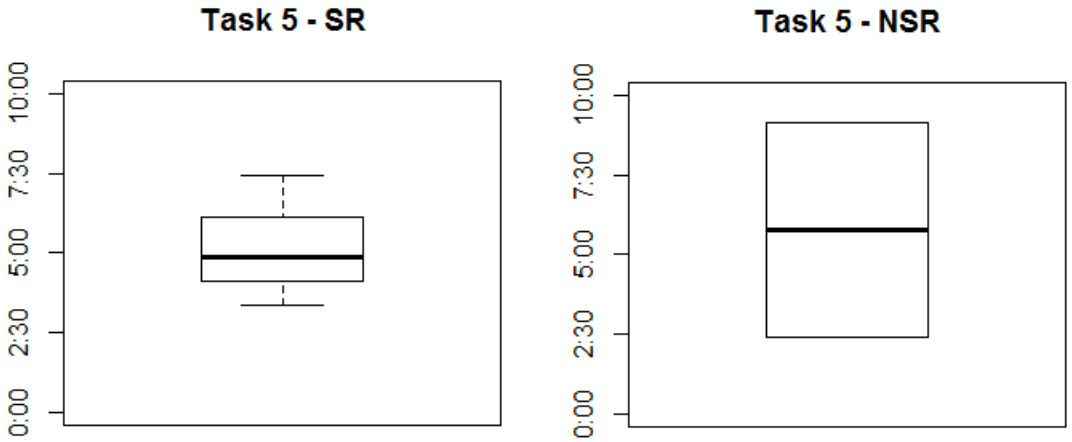


Figure 4.7: The Task 5 boxplots for TTFF times under both scenarios

classes.

Another factor for the lower overall NSR times was the size and complexity of the Tetris project. The study contained eight tasks to be completed with a Java implementation of Tetris. The expected result was that SR was more efficient and effective than NSR as it represents an improvement on the features offered by Diver. These results could point to the tasks being too simple and the project too small to allow the benefits of software reconnaissance to be seen. The time taken recording the second trace plays a much bigger role when tasks are short, as was the case here. This highlights an important point that the benefits of software reconnaissance may be realised only when you have a project of sufficient size and complexity.

Mistakes with tracing were a common occurrence. For instance, **P16** forgot to start recording a trace at the beginning of the program execution. As the button for starting and stopping recording is the same, he clicked to stop recording and instead started recording for a short time until he exited Tetris. This was his main trace for software reconnaissance and so when he applied the filtering to the blank trace, his Package Explorer and the sequence diagram were empty. The added complexity of recording two traces instead of one could have been a contributing factor to the poor SR results.

As mentioned in the Effectiveness section, some participants were just slower than others due to a variety of reasons. Some were hindered by old habits that were at odds with how Diver is used. Some participants used a *Bottom-up* program compre-

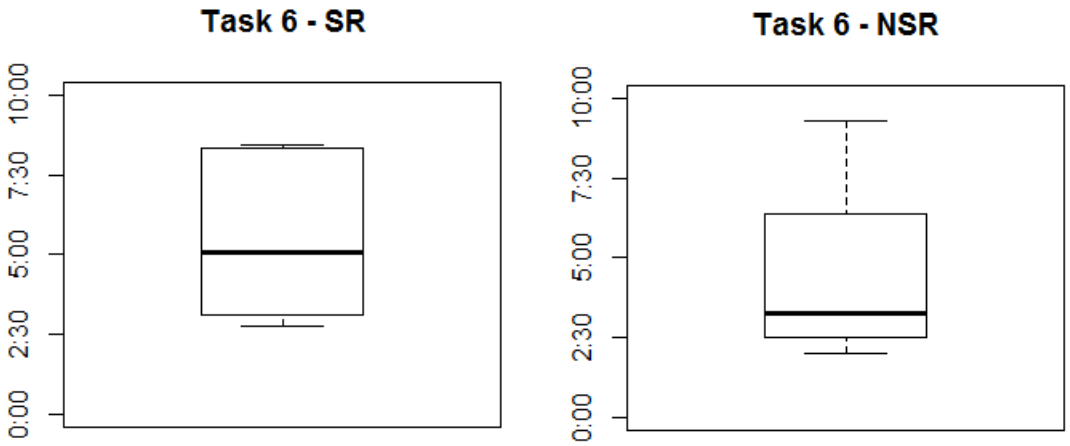


Figure 4.8: The Task 6 boxplots for TTFF times under both scenarios

hension strategy and tried to read code and look through the sequence diagram to build an understanding of the program. This resulted in some participants getting 'lost' in the diagrams as **P10** described. It should be noted that **P10** was very methodical while attempting to complete the tasks and used a *Systematic* approach to understanding the program. He used a *Top-down* strategy to learn about the different classes which other participants also followed, using the Package Explorer to understand the components of the program and then navigating to the sequence diagram to understand how it worked. He did not complete the program understanding tasks as quickly as the other participants, and in the end, was only able to attempt four tasks given the limited time; he successfully completed three of those. From his thinking out loud, however, he very clearly gained an understanding of the system that the other participants did not. Most other participants used an *As-needed* program comprehension strategy to complete the questions as they quickly searched for the required functionality without stopping to understand the program as a whole. Some of them made wrong assumptions about aspects of the program, such as which threads were responsible for what functionality, causing them not to complete some tasks. The use of Diver and the Trace-focused UI effectively means that users automatically follow an *As-needed* strategy by only looking at components related to the task at hand. *Top-down*, *Bottom-up*, *Systematic*, and *As-needed* strategies were used by participants while still following an overall *As-needed* approach.

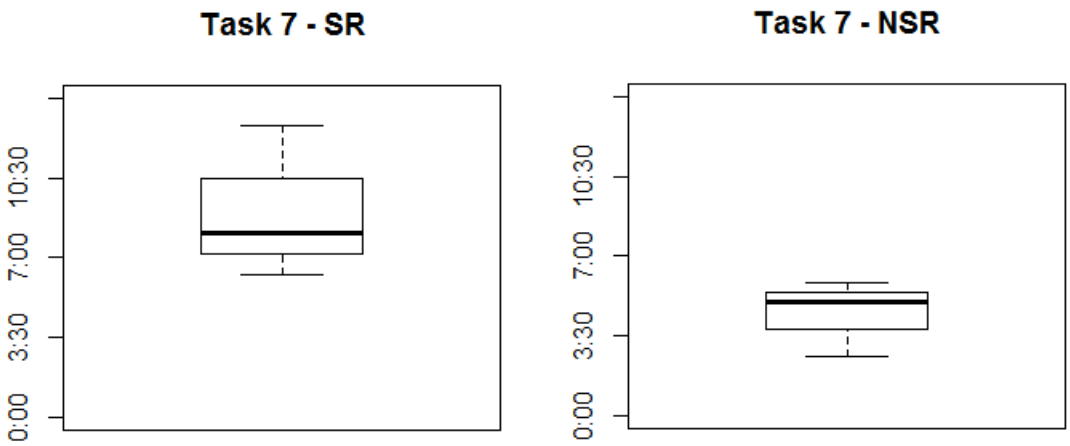


Figure 4.9: The Task 7 boxplots for TTFF times under both scenarios

4.3.6.3 Summary

The results of the effectiveness and efficiency analysis was mixed. For efficiency, the non-software reconnaissance scenario produced the lowest TTFF times for both single quickest and average quickest times, with one exception. There were various reasons why this was the case and went against the hypothesis that software reconnaissance does improve the efficiency of task completion. Software reconnaissance did fair better for effectiveness, however, the 9% improvement in task completion was not statistically significant.

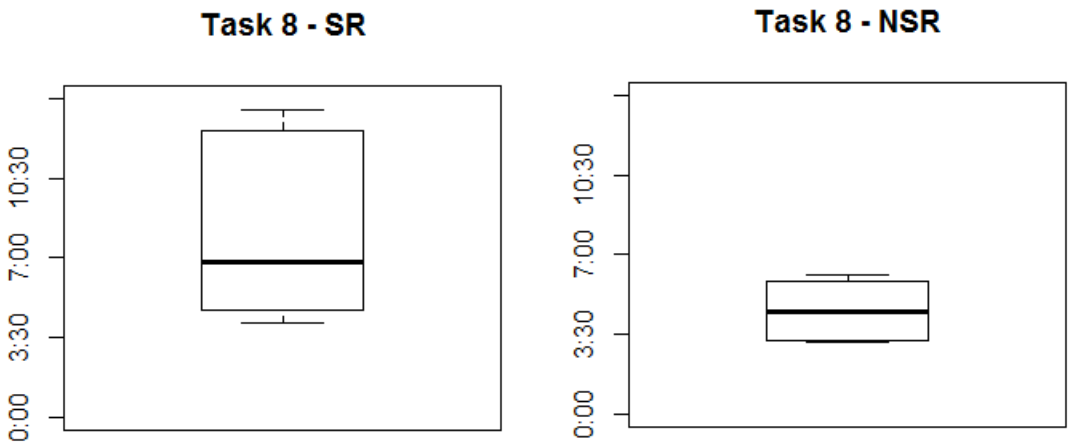


Figure 4.10: The Task 8 boxplots for TTFF times under both scenarios

4.3.7 Participant Responses

4.3.7.1 Pre-study Questionnaire

The pre-study questionnaire was administered to allow us to understand the backgrounds and technical experience of the participants of the study. Participants were recruited using various communication channels at the University of Victoria. Figure 4.11 shows the clustering of the participants based on their educational backgrounds. Seven were current undergraduates, six were current or recently graduated postgraduates, two were on post-doctoral fellowships, and the remaining participant had a Computer Engineering diploma.

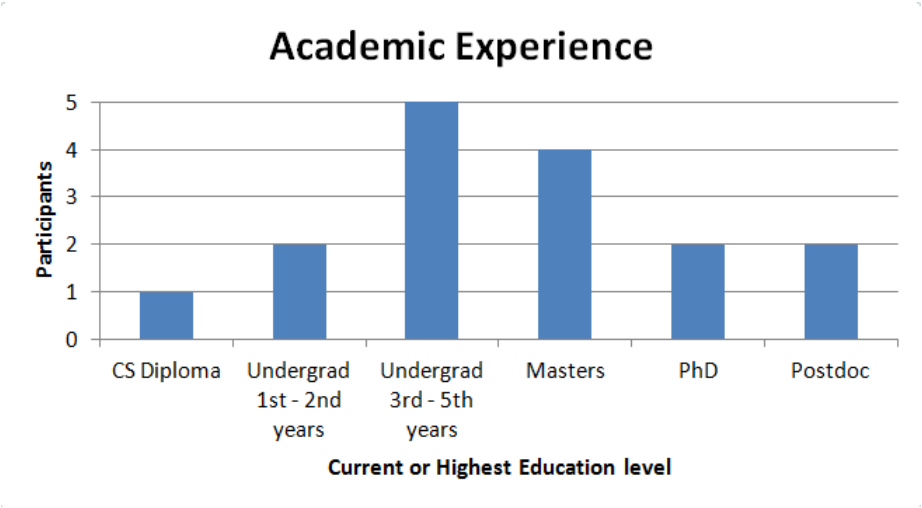


Figure 4.11: The breakdown of academic experience

Figure 4.12 shows the boxplots for the four types of technical experience asked about in the questionnaire. One of the requirements for the study was that the participants must have experience with Java; shows the distribution of Java experience. **P1** was the most experienced by far, with 15+ years of Java programming experience. The mean general programming experience was just under five and a half years and the median is five years. For Java experience, the mean was around two years and eight months and the median was two and a half years.

To understand what experience participants had with program comprehension tasks, we asked how much experience they had maintaining someone else's code. The mean was just over two years with a median of two years, indicating that participants had much less experience maintaining someone else's code compared to programming.

Participants' experience with Eclipse and UML was determined as both of these skills were helpful during the experiment, which involved using Eclipse and UML sequence diagrams during the tasks. The responses are plotted in Figure 4.12. Experience was low for both and many participants answered in months rather than years. The mean for participants' Eclipse experience was 22 months and the median was 12 months. The mean for UML experience was 15 months and the median was 6 months. The standard deviation for the UML experience was extremely large at over 24 months. **P6**, **P8**, **P11**, and **P12** did not have any experience creating or using UML diagrams. **P5** and **P6** did not have any Eclipse experience. **P5** also stated that he did not have any experience with other IDEs, unlike **P6** who had.

Finally, participants were asked to specify the size of the largest project they had worked on. They could answer in lines of code (LOC), number of files, or time. In terms of LOC, **P2** had experience with the largest project (1M LOC), and **P1**'s largest project was second (100K LOC). **P13** worked on a project that spanned one year of development time. The participant that worked on the smallest project was **P8** who had only worked on 300 lines of code.

The experience of the participants was correlated to their performance, and results showed mostly weak correlations between the different types of experience and participants' average TTFF times or number of tasks completed. Table 4.14 shows the results of the correlated data.

For average TTFF times, all the correlation coefficients indicated a weak, negative correlation. Experience with Eclipse had the strongest correlation with a coefficient of -0.473. For tasks completed, there was a wider range of correlation values, but programming, maintenance, and UML experience were all considered weakly correlated. While programming experience garnered a coefficient of 0.463, experience with Java had the highest correlation coefficient of 0.617. However, the result was still not high enough to be considered strongly correlated.

Performance	Experience	Correlation
Average TTFF	Programming	-0.221
	Java	-0.363
	Maintenance	-0.098
	Eclipse	-0.473
	UML	-0.337
Tasks Completed	Programming	0.463
	Java	0.617
	Maintenance	0.299
	Eclipse	0.512
	UML	0.335

Table 4.14: Participants' performance correlated with their experience

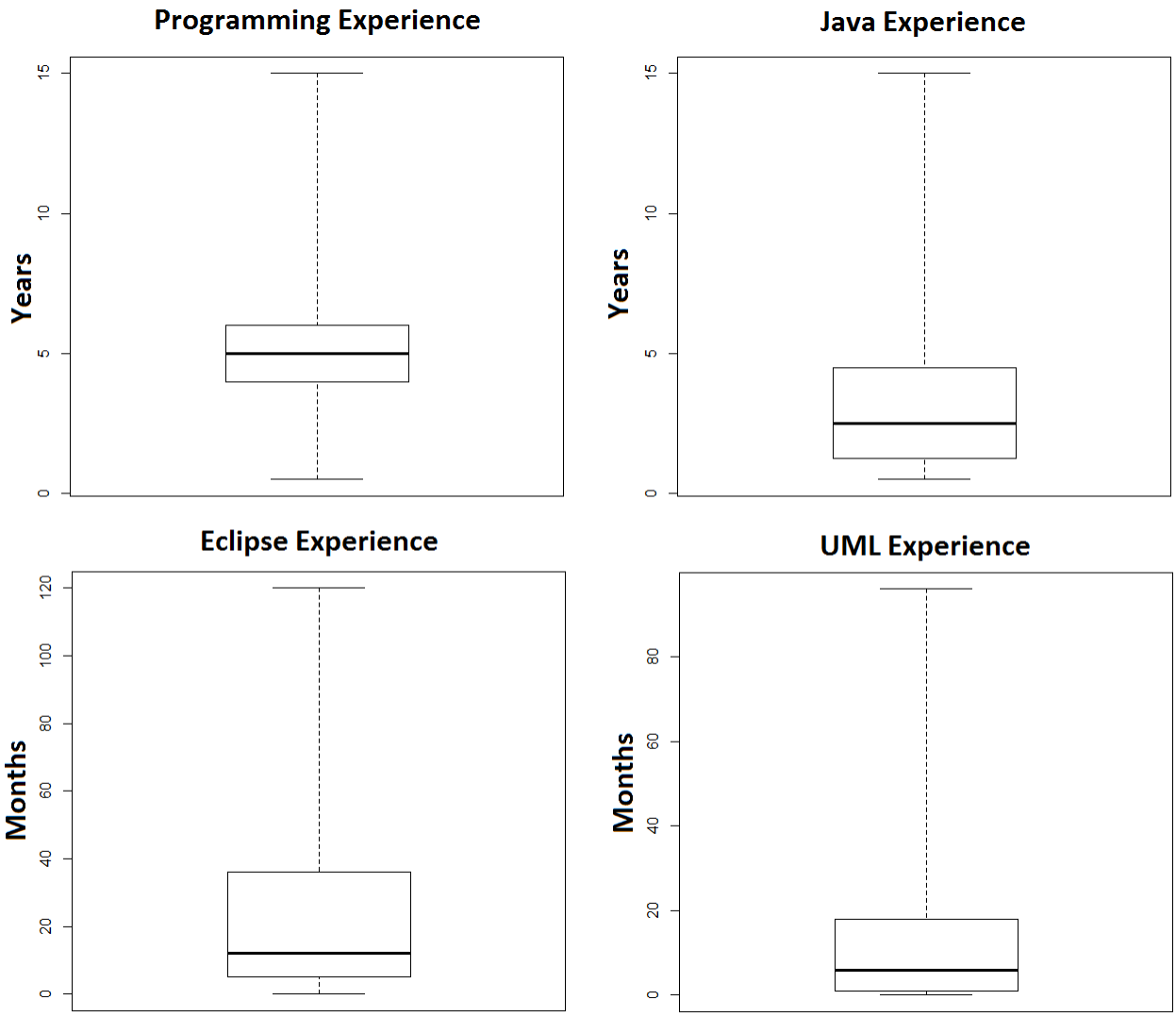


Figure 4.12: Boxplots of the participants' experience

4.3.7.2 Post-study Questionnaire

Regarding the post-study questionnaire, some of the responses were mentioned in Section 4.3.3 for the Cognitive Support research question. Various positive remarks were mentioned such as "Easy to use", "Great tool", "Useful visual aid", "Cool", and "Mostly simple and straight forward to use" by **P1**, **P3**, **P5**, **P10**, and **P11**. **P15** said "I liked it, very helpful in finding methods even when I had zero knowledge of the source code." Overall, only **P4** and **P13** did not have positive general experiences with the tool. **P4** response focused on Diver's lack of search functionality. **P13** mentioned being frustrated by the experience, but admitted that completing the same tasks without Diver would have been even more frustrating.

When asked whether they had enough time to complete all the tasks, participants provided a range of answers. The answers are summarised in Table 4.15. **P1**, **P5**, and **P12** thought they had enough time for all the tasks, however, most other participants mentioned reasons why their progress had been slowed down.

Time Comments	Participants
Had enough time	P1, P5, P12
Needed more practice	P2, P8, P16
Practice was enough to the tasks	P7
Old habit of reading code slowed progress	P2, P4
Single task slowed progress	P7, P13
Methodical planning habit slowed progress	P10
Slow due to lack of software reconnaissance under NSR scenario	P6, P15

Table 4.15: Summary of comments on whether the participants had enough time or not

Participants were asked "What helped you complete the tasks?" and "What tool features did you find most useful?" A combined summary of the responses can be seen in Table 4.16. Filtering of the Package Explorer and sequence diagram using software reconnaissance was the most commonly mentioned set of features. 10 participants made mention of these features and another three mentioned software reconnaissance by name. The sequence diagram was the second most cited feature that participants felt helped them to complete the tasks. There were some participants who disagreed with this, as we discuss below.

In contrast, two questions were asked to find out what hindered the participants and which features were found to be the least helpful. Table 4.17 contains the summarised results of these questions. 10 participants thought that the sequence diagrams

Positives	Participant
Filtering of Package Explorer	P1, P2, P3, P4, P6, P7, P8
and sequence diagram with SR	P9, P10, P15
Sequence Diagram	P3, P4, P7, P8, P10, P14
Jumping to code and reading it	P2, P3, P5, P11, P13
Navigating to sequence diagram with *Reveal In*	P11, P13, P14, P15
Software Reconnaissance	P12, P14, P15, P16
Searching the Package Explorer	P12, P14
Filtering sequence diagram classes	P8
Timeline	P15

Table 4.16: Summary of positive comments on what helped complete the tasks

were too large from lengthy traces and they were difficult to use and understand. Lengthy traces also contributed to the second most cited hindrance: slow save times for traces. This frustrated some participants and led to some time being wasted.

Negatives	Participant
Large, complex sequence diagrams	P2, P3, P5, P6, P7, P8, P11, P13, P14, P16
Slow trace save times	P1, P7, P8, P9
Lack of programming experience	P3, P7, P16
Lack of tool experience	P10, P12, P15
Expanding and collapse diagram	P3, P4
Time/frustration	P9, P12
Focus On/Up features complicated	P15, P16
Multiple threads	P1
SD class filtering not automatic	P2
No breakpoints	P5
No zooming in sequence diagram	P7
Only one SD open at a time	P11

Table 4.17: Summary of negative comments

When asked about the eye tracking, most participants did not see it as a distraction. **P2** said it was no more distracting than the video camera and thinking aloud process. **P12** was a little nervous about it, but soon forgot it was there.

The post-study questionnaire was useful, not just to solicit participants' perceptions and experience with the tool, but also to gather their opinions on the tool's future development. Given their limited experience with the tool, the participants were asked about what improvements to Diver they would like to see. Some were feasible and others not. Many were related to automating manual activities, such as

automatically collapsing the Java classes' lifelines by default. The suggestions can be seen in Table 4.18. One participant mentioned wanting to view traces in real-time. This would be an extremely useful feature but it technically not possible given the design and limitations of the system. Two people wanted the slow save times for traces improved, which was mentioned as a hindrance in Table 4.17.

Suggestions	Participants
Automatically collapsing the lifelines	P1
View traces in real-time	P2
Automatically close program when pausing trace	P2
Search sequence diagrams	P4
Faster trace saving times	P6, P8
Link code to sequence diagram	P6
Rename traces in the Program Traces view	P9
Better highlighting in sequence diagrams	P10
Multiple sequence diagrams open at once	P11

Table 4.18: Summary of suggestions to improve Diver

The second section of the post-study questionnaire asked the participants to use a Likert scale to rate how much they agreed with certain statements. The responses ranged from 1 (representing *I strongly disagree*) to 7 (representing *I strongly agree*). The first five statements were specific to Diver and software reconnaissance and how it helped them to complete the tasks. The rest of the questionnaire measured general usability with a USE questionnaire [20]. USE questionnaires are used to rate products based on four categories: usefulness, ease of use, ease of learning, and satisfaction. Participants were required to rate how much they agreed with statements based on Diver's usefulness, ease of use, ease of learning, and satisfaction. The first five statements are listed below.

1. Diver was very helpful for completing the feature location and program understanding tasks and questions.

2. Software reconnaissance was very helpful for completing the program understanding tasks.

3. Diver with software reconnaissance made it much easier to know the steps required to locate a feature and understand the functionality.

4. Diver with all its features helps to inform the user about what needs to be done next to move towards completing the program understanding tasks.

5. Diver makes it much easier to know when you are close to completing the task.

The answers to the non-USE questionnaire statements were plotted on boxplots in Figure 4.13

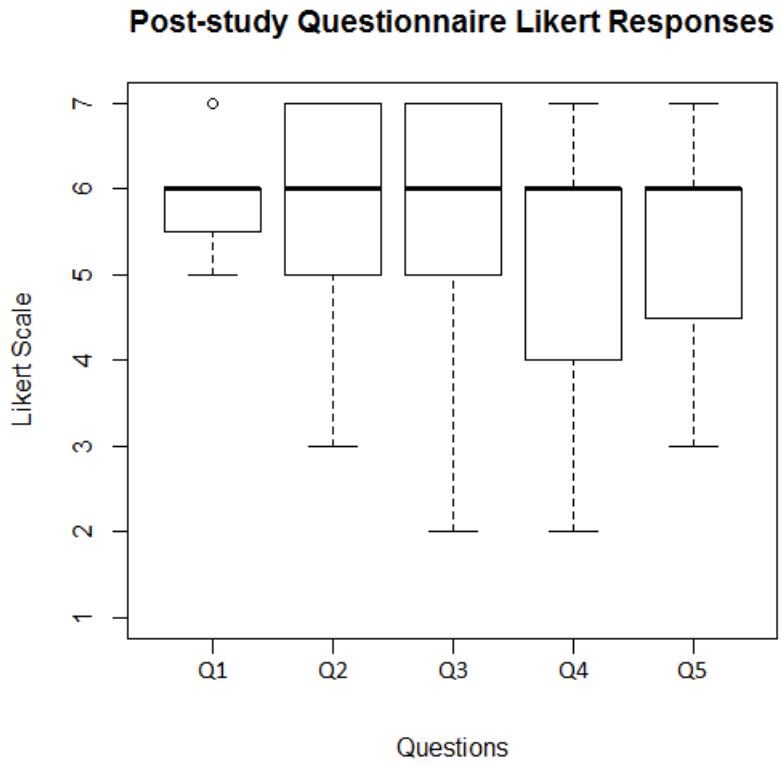


Figure 4.13: Boxplot of the Likert-based Questionnaire Results

The USE questionnaire results are represented by a radar chart in Figure 4.15 and boxplots in Figure 4.14. The four areas can be grouped into two by their similar ratings. Usefulness and Satisfaction scored highest. Usefulness scored a mean of 5.91 and a median of 6. Satisfaction had a mean of 5.86 and a median of 6. The group with the two lower averages included Ease of Use and Ease of Learning. Ease of Use had a mean of 4.90 and median of 5. Ease of Learning was close behind with 4.89 for the mean and 5 for the median.

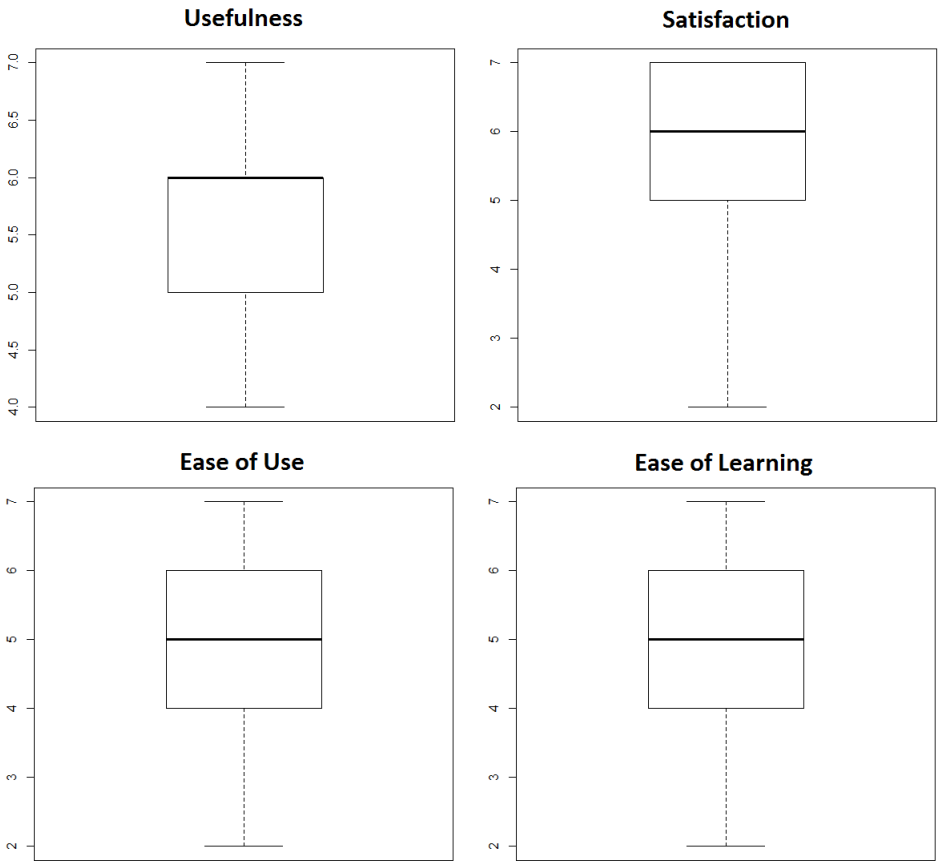


Figure 4.14: USE Questionnaire Results boxplots

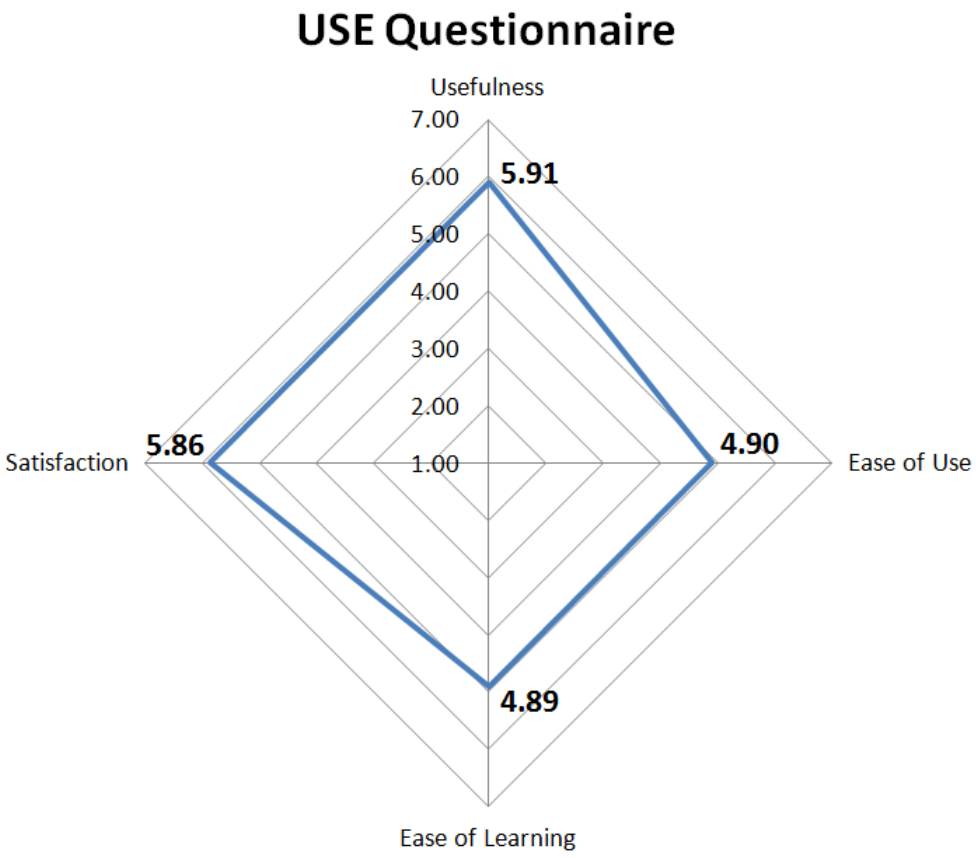


Figure 4.15: USE Questionnaire Results

4.3.8 Interpretation of the Participant Responses

The participants' responses give a very clear picture that while they thought Diver was very useful, they did struggle with the new tool. The pre-study and post-study questionnaires were useful, showing the experience of the participants, or lack thereof, as well as their opinions of Diver after the study.

The most successful participant was **P2** as he completed all eight tasks and did so with lowest average of anyone: 03:06. This is understandable given that **P2** had plenty of experience: over 15 years of Java programming, ten years of Eclipse, and eight years of UML diagrams. Besides that example, there were no noticeable trends between experience and performance. The correlation of the results with experience showed no strong linear relationships between the various types of experience and performance metrics. For instance, second place **P1** had less programming and UML diagram experience than the least successful participant, **P4**. The least experienced participant was **P8**, a first year student whose only programming experience was one Java programming course. Despite that, he was able to complete three tasks and had the third lowest TTFF average for those results. A lack of UML experience was a potential problem as Diver's sequence diagram is a fundamental part of the tool. There were four participants who had no UML diagram experience. Three of them ended up in the bottom six for number of tasks completed, however, two were ranked third and fourth for their TTFF averages. The lack of any strong correlation between tasks completed or average TTFF times and experience discounts the argument that the participant's performances were primarily dependent on experience and not task difficulty and effective use of Diver.

Despite participants performing better under NSR, the general perception from the participants was that the filtering of the Package Explorer and sequence diagram using software reconnaissance was useful and very effective for feature location tasks. The perception was so strong that **P6** commented that the tasks with "software reconnaissance went faster," even though he took much longer to complete his tasks using the technique.

4.3.9 Limitations and Threats to Validity

This section discusses the limitations of the study and the results presented above. Some of the limitations of Myers' study were mitigated in this study, but some limitations were shared and others were introduced with the new study design.

As LaToza *et al.* [17] and Myers [24] mention, conducting empirical research into software tools and their features can be difficult, especially program comprehension tools where understanding is difficult to measure effectively. We conducted a laboratory experiment and compared the 16 participants' results with each other.

One limitation was the low number of participants, given the relatively small margins associated with the short tasks. All the results that were checked for statistical significance failed to have p-values below the threshold of 0.05. As a result, the quantitative analysis did not lead to definitive support for software reconnaissance while answering **RQ2** and **RQ3**. The 8 tasks, which each participant had to complete, did increase the amount of data and observations available for analysis, and thus allowed for richer user stories that supplemented the quantitative analysis. The increased data also increased the reliability of the study.

Using simpler tasks and a small project like Tetris could have lessened the usefulness of software reconnaissance. As the data did not definitively show the effectiveness and efficiency of the technique, this could explain the results especially since it was observed that participants were able to use the Package Explorer effectively without software reconnaissance.

Placing the SR scenario first, followed by the NSR scenario, was brought up as a problem by the reviewers of the paper submitted on Myers' study. The reviewer in question argued that it was problematic to allow participants the use of software reconnaissance in the first session and then take it away in the second. Myers' was measuring the frustration exhibited by participants and the scenario order could have biased participants' perceptions. The new study did not measure frustration to mitigate this problem.

We decided to keep the same order in this study for a number of reasons. One of the reasons was that the research question on participant frustration was removed for this study. Also, participants were new to Diver and the Tetris project, so learning effects were expected to influence participants' performances. As participants completed the tasks and learnt more about the project, the knowledge they gained helped them in the later tasks without software reconnaissance. Consequently, the scenario order acted against the hypothesis that software reconnaissance was an effective and efficient technique for program comprehension using Diver.

An alternative approach of splitting the participants into two groups to study both scenario orders would have weakened the quantitative analysis, which would have been restricted to the two groups of eight participants instead of one group of 16. The focus

on recruiting more participants and gathering more data with additional tasks would have been undermined if this approach had been followed.

The transferability of the results is limited by using student participants. Simpler tasks were used to account for the lack of real world programming experience. However, the use of students did provide insight into the experiences of new and inexperienced Diver users, so the results of the study could be used to help novices learn to use Diver more effectively.

The evidence to support the hypotheses on activities that reflected cognitive support relies on cognitive support theory. The research that was done on this topic was presented in Chapters 2 and 3. The observations did show that in most cases these hypotheses may be valid. The one observation that definitely countered a hypothesis was regarding ends-means reification and the use of the *Reveal In* feature. The feature was so effective at finding a foothold, during the SR scenario, that participants only needed to use *Reveal In* once in many tasks instead of repeatedly using it, as hypothesized.

Credibility was sought though an array of data sources that brought multiple perspectives on the experiences of the participants to the study. Unfortunately, the nature of the study limited the interactions with the participants to the two hours that they conducted the experiment. This was a limited view of using Diver, which would be used regularly by programmers. Also, the Likert questions may have been biased in the post-study questionnaire towards Diver's features.

4.3.10 Summary

In this chapter, we present the research questions and explain the methodology and design that we used for the study. Next, we report on the results, and then discuss them in the context of the research questions. Finally, we discussed the limitations to the study.

The study resulted in the three research questions being answered with varying degrees of certainty. The first research question led to the identification of a number of navigation and interaction patterns that influenced the successful completion of a task. The second research question was answered with data showing that the redistribution and perceptual substitution cognitive support theories were mildly supported by Diver. Support for the ends-means reification theory could not be found, but this may have been due to our hypothesis about the *Reveal In* feature being incorrect.

The last research question addressed the performance of participants using software reconnaissance compared to not using the technique. Software reconnaissance was found to help participants be more effective at completing tasks successfully, but not more efficient at completing them successfully. The limitations were discussed to provide additional context for the results.

Chapter 5

Conclusions

This thesis, and the study it describes, investigated the effectiveness and efficiency of Diver with software reconnaissance, and how the technique influences Diver's use. This chapter concludes the thesis by discussing how the research questions were addressed, the contributions of the research, and possible future work.

5.1 Research Questions Revisited

The core theme of the research questions was software reconnaissance and whether or not its implementation had a positive impact on Diver's usability and usefulness. The study gathered the data necessary to address these questions.

The first research question (RQ1) asked whether or not software reconnaissance influenced participants' interactions with Diver and lead to different navigation patterns. This question was addressed by analyzing Diver's logs and identifying how often various features were used for successful and unsuccessful tasks under both scenarios. We added insights, discovered through observing the participants, to the analysis to determine what navigation and interaction patterns were followed.

In the feature usage statistics for different combinations of task scenario and task success, we found that the *Reveal In* feature was used less for successfully completed tasks using software reconnaissance than tasks completed successfully without it. This was contrary to the hypothesis that software reconnaissance encouraged the use of the *Reveal In* feature. While the data was not statistically conclusive, we did theorize that, given the observation data, participants used the feature less because it was more effective at gaining a foothold with software reconnaissance. This meant

participants had to use the feature more to find a foothold without the help of software reconnaissance.

Another pattern that emerged was that the sequence diagram was used slightly more for successful tasks under the SR scenario than the NSR scenario. Additionally, the *Jump To Code* feature was used more often under the NSR scenario. Again, the differences were not significant, but they did point to participants having to read the code because software reconnaissance was not available to help them gain a foothold.

The second research question (RQ2) was "Does the use of software reconnaissance improve the cognitive support of Diver during program understanding tasks?" We approached this question by considering research by Walenstein [34], Storey *et al.* [30], and Bennett [3]. We connected Walenstein's cognitive support theories to the cognitive support features proposed by Bennett for use with sequence diagrams. We then developed hypotheses for the three cognitive support theories of redistribution, perceptual substitution, and ends-means reification.

For redistribution, we analyzed participants' perceptions of the tool and found the features related to software reconnaissance were most helpful and provided the best support to complete the program comprehension tasks, according to the participants. For perceptual substitution, we investigated how dependent the participants were on reading code, as opposed to understanding the program through the sequence diagram. The 20% increase in *Jump To Code* usage offered some support to our hypothesis, but the results were not statistically significant. For the ends-means reification, we looked at the *Reveal In* usage and found that the -18% difference in usage between the NSR and SR scenarios confounded our hypothesis, but that it too was statistically insignificant.

The third research question (RQ3) was addressed by measuring the effectiveness and efficiency of software reconnaissance using data on task times and task success. In all of the eight tasks, the NSR scenario produced the quickest TTFF time. The NSR scenario also produced the quickest average TTFF times in all but one task. The results pointed to software reconnaissance making participants less efficient at the program comprehension tasks. Responses from the participants and observations during the study did point to the learning effect playing a role in the results. Software reconnaissance was found to be more effective at completing the tasks, as more tasks were completed successfully with it. Again however, the results were not statistically significant.

The responses of the participants were analyzed to help gain insight into how

the participants experienced the experiment and Diver. The general tone was very positive and most were impressed by Diver, even if they were frustrated with issues such as extremely large sequence diagrams. The USE questionnaire showed that Diver was both useful and satisfactory, but not as easy to use or learn as expected.

Overall, the results that came out of the study led to a number of contributions that are discussed in Section 5.2.

5.2 Contributions

The research conducted for this thesis lead to a number of contributions:

- Built a cognitive support map from the three cognitive support theories to Diver's features.
- Hypotheses on the cognitive support theories present in Diver were developed or refined.
- The design of Myers' study was analyzed and its shortcomings were discussed.
- A new study was designed to try to overcome the problems with the previous study.
- The study contributed data on how Diver is used by new users of Diver.
- The responses to the post-study questionnaire contributed user feedback to the Diver development effort.

5.3 Future Work

The study highlighted various places where Diver can be improved. Some of the complaints are effectively feature requests, and some of these are for features present in the system but not used during the study. **P4** wanted to search the sequence diagram and **P5** wanted to use break points, both of which already exist in Diver but were not utilized for the study.

Another feature not used during the study was the sequence diagram outline view. This provides an overview of the sequence diagram with a zoomed-out representation of the diagram and a box showing what area of the diagram is being viewed. This

feature was not working for the study and could have answered the needs of **P7** who asked for zooming capabilities for the sequence diagram. Software reconnaissance is a standard feature of Diver and so the requests for various improvements to the sequence diagram based on the participants' experience using Diver without the technique are somewhat weakened.

Most of the participants were very impressed with the tool and so further development is important. The aim was to improve Diver to produce a stable version that could attract more users who could become future experienced participants in further studies. This is a goal worth working towards and further studies could be performed to analyze how experienced users use Diver. Such studies would further inform the design of Diver.

The evaluation of such studies would have to include both quantitative and qualitative data, as timing and usage metrics only tell part of the story. The time frame of these studies could be extended and the collection of the data could take place while Diver is being used for understanding problems in a real project. The problem with this approach would be that the realism would limit how the data from different tasks could be compared. A more extensive laboratory experiment could follow on from this study to evaluate the findings that this thesis uncovered.

5.4 Conclusion

Diver represents a huge undertaking consisting of solid design and development, and grounding in program comprehension and cognitive science research. This thesis took on the difficult task of trying to measure the effectiveness of such a tool to continue the effort of improving users' experience with program comprehension tools. Diver with software reconnaissance is a very useful tool according to many of the participants who used it during the study. The positive responses support the statement that Diver should be part of every Java developers toolkit. Continued research and development will work to realise this goal and help programmers maintain ever more complex software systems.

Bibliography

[1] Hira Agrawal, James L Alberi, Joseph R Horgan, J Jenny Li, Saul London, W Eric Wong, Sudipto Ghosh, and Norman Wilde. Mining system tests to aid software maintenance. *Computer*, 31(7):64–73, 1998.

[2] Anonymous. Review of "supporting navigation in large sequence diagrams using software reconnaissance", 2011.

[3] C. Bennett. Tool features for understanding large reverse engineered sequence diagrams. Master's thesis, University of Victoria, 2008.

[4] C. Bennett, D. Myers, M.A. Storey, and D. German. Working with monster-traces: Building a scalable, usable sequence viewer. In *In Proceedings of the 3rd International Workshop on Program Comprehension Through Dynamic Analysis (PCODA), Vancouver, Canada*, pages 1–5, 2007.

[5] C. Bennett, D. Myers, M.A. Storey, D.M. German, D. Ouellet, M. Salois, and P. Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):291–315, 2008.

[6] B. Cleary, A. Le Gear, C. Exton, and J. Buckley. A combined software reconnaissance & static analysis eclipse visualisation plug-in. In *Visualizing Software for Understanding and Analysis, 2005. VISSOFT 2005. 3rd IEEE International Workshop on*, pages 1–2. IEEE, 2005.

[7] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.

[8] Bas Cornelissen, Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252–2268, 2008.

[9] K.A. Ericsson and H.A. Simon. Verbal reports as data. *Psychological review*, 87(3):215, 1980.

[10] Virginia R Gibson and James A Senn. System structure and software maintenance performance. *Communications of the ACM*, 32(3):347–358, 1989.

[11] A. Hamou-Lhadj and T.C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 42–55. IBM Press, 2004.

[12] Abdelwahab Hamou-Lhadj and Timothy C Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 159–168. IEEE, 2002.

[13] Abdelwahab Hamou-Lhadj, Timothy C Lethbridge, and Lianjiang Fu. Challenges and requirements for an effective trace exploration tool. In *Program Comprehension, 2004. Proceedings. 12th IEEE International Workshop on*, pages 70–78. IEEE, 2004.

[14] M. Kersten and G.C. Murphy. Mylar: a degree-of-interest model for ides. In *Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168. ACM, 2005.

[15] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353. IEEE Computer Society, 2007.

[16] Philippe B Kruchten. The 4+ 1 view model of architecture. *Software, IEEE*, 12(6):42–50, 1995.

[17] Thomas D. LaToza and Brad A. Myers. Designing useful tools for developers. In *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools*, PLATEAU '11, pages 45–50, New York, NY, USA, 2011. ACM.

[18] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501. ACM, 2006.

[19] A. Le Gear, J. Buckley, JJ Collins, and K. O'Dea. Software reconnexion: understanding software using a variation on software reconnaissance and reflexion modelling. In *Empirical Software Engineering, 2005. 2005 International Symposium on*, pages 10–pp. IEEE, 2005.

[20] Arnold M Lund. Measuring usability with the use questionnaire. *Usability interface*, 8(2):3–6, 2001.

[21] Joseph E McGrath. Methodology matters: doing research in the behavioral and social sciences. In *Human-computer interaction*, pages 152–169. Morgan Kaufmann Publishers Inc., 1995.

[22] D. Myers and M.A. Storey. Using dynamic analysis to create trace-focused user interfaces for ides. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 367–368. ACM, 2010.

[23] D. Myers, M.A. Storey, and M. Salois. Utilizing debug information to compact loops in large program traces. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 41–50. IEEE, 2010.

[24] Del Myers. Improving the scalability of tools incorporating sequence diagram visualizations of large execution traces. Master's thesis, University of Victoria, 2011.

[25] Marian Petre. Uml in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 722–731. IEEE Press, 2013.

[26] V. Rajlich and N. Wilde. The role of concepts in program comprehension. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 271–278. IEEE, 2002.

[27] Sourceforge. Diver's download statistics.

[28] M.A. Storey. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal*, 14(3):187–208, 2006.

[29] M.A.D. Storey. *A cognitive framework for describing and evaluating software exploration tools*. PhD thesis, Simon Fraser University, 1998.

[30] M.A.D. Storey, F.D. Fracchia, and H.A. Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[31] M.A.D. Storey, K. Wong, and H.A. Muller. How do program understanding tools affect how programmers understand programs? In *Reverse Engineering, 1997. Proceedings of the Fourth Working Conference on*, pages 12–21. IEEE, 1997.

[32] Scott R Tilley, Santanu Paul, and Dennis B Smith. Towards a framework for program understanding. In *Program Comprehension, 1996, Proceedings., Fourth Workshop on*, pages 19–28. IEEE, 1996.

[33] A. Von Mayrhauser and A.M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[34] Andrew Walenstein. *Cognitive support in software engineering tools: A distributed cognition framework*. PhD thesis, Simon Fraser University, 2002.

[35] Andrew Walenstein. Foundations of cognitive support: Toward abstract patterns of usefulness. *Interactive Systems: Design, Specification, and Verification*, pages 133–147, 2002.

[36] Andrew Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *Program Comprehension, 2002. Proceedings. 10th International Workshop on*, pages 75–84. IEEE, 2002.

[37] N. Wilde, M. Buckellew, H. Page, V. Rajlich, and L.T. Pounds. A comparison of methods for locating features in legacy software. *Journal of Systems and Software*, 65(2):105–114, 2003.

[38] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 312–318. IEEE, 1996.

[39] N. Wilde and M.C. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.

[40] Norman Wilde, Michelle Buckellew, Henry Page, and Vaclav Rajlich. A case study of feature location in unstructured legacy fortran code. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 68–76. IEEE, 2001.

[41] A. Zaidman. Scalability solutions for program comprehension through dynamic analysis. In *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pages 4–pp. IEEE, 2006.

Appendix A

User Study Consent Form

Introduction

Thank you for offering to participate in our user study. It is our hope that the data collected in this study will help us and others to develop tools that will support developers in their software development, maintenance, and evolution tasks. The study is being conducted by Sean Stevenson of the CHISEL group at the University of Victoria under the supervision of Dr. Margaret-Anne Story. The results of this study will be published in confidential form in scholarly publications including, but not limited to, conference proceedings, journal articles, and in a Masters thesis. If you have any questions, you may contact Dr. Margaret-Anne Storey or Sean Stevenson. For more information about our research group, please see our website: http://www.thechiselgroup.org.

What is Involved

This study involves the performance of 8 feature location and analysis tasks. You will be asked to analyze a piece of software using the tools provided, and to answer some questions about the software under analysis. You will also be asked to complete a pre-study and post-study questionnaire. Participation will involve approximately 2 hours of your time.

Voluntary Participation

Your participation in this study is completely voluntary. Declining participation carries no professional or employment consequences. You may choose to end the

study at any time. If you choose, we can also withdraw you completely from the study so that all data collected during your session will be destroyed and disregarded. If you would like to withdraw from the study after your session has been completed, please contact Sean Stevenson.

Risks

You may experience some frustration while performing the tasks involved in this study. You may choose to withdraw from the study at any time.

Confidentiality

The data collected in this study will be internal to the CHISEL group at the University of Victoria. Your participation in the study will be recorded in audio/visual format using a video camera. Your interactions with the software tools involved in this study will also be automatically recorded by the software, in addition to your eyes being tracked. No identifying information will be shared outside of the individuals involved in conducting this study. All data will be anonymized before being shared or published in order to protect your confidentiality.

Compensation

You have been offered a stipend in the amount of $20 for your participation in this study. This is a free gift and should have no impact on your willingness to perform this study. In other words, if you feel that you would not participate unless you were offered this gift, then you should withdraw from the study.

Benefits

The tool under investigation is free and Open Source. If you like the tool, you may download it from the web site http://eclipsediver.wordpress.com. Your participation may lead to improvements in this tool as well as add to general knowledge about how to develop software tools for software development, maintenance, and reengineering. More Information If you have any questions about the study, you may contact Sean Stevenson or Dr. Margaret-Anne Storey. If you have questions or concerns about the ethical implications of this study, you may contact the University of Victoria Human

Research Ethics Office. This study is part of a larger research investigation titled Reverse Engineered Sequence Diagrams to Support Software Evolution.

| ---------- | ---------- | ---------- |
| Participant Name | Signature | Date |

Appendix B

Pre-study Questionnaire Form

Date:
User Code:

This questionnaire asks you for information concerning your previous programming experience specifically with Eclipse and Java. Previous programming experience and domain knowledge (knowledge about the program you will be examining) will affect the results in this experiment.

You do not have to answer any questions you do not want to.

1. What degree, program and academic year of university are you in?

2. Are you familiar with the game Tetris?

3. How many years/months of programming experience do you have in total?

4. How many years/months have you spent working on code written by someone else?

5. How many years/months of creating and using UML diagram do you have?

6. Which programming languages/scripting languages have you programmed in?

7. How many years/months of programming experience in Java do you have?

8. How many years/months of programming experience using the Eclipse IDE do you have?

9. Have you used any other IDEs before? Which ones?

10. What development tools have you used before (IDE plugins, reflection tools, code coverage, UML generators etc)?

11. What is the largest program you have written or maintained (lines of code, number 0f files, time length of development)?

Appendix C

Tasks

Task 0

Date:
Subject Code:
Task Order:
Software Reconnaissance:

Task Instructions

In the training Sudoku game there is the Solve button. Using Software Reconnaissance, determine what method in the game is called to solve the puzzle. Once found, answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 1

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

To start a new game the button labelled *Start Game* must be clicked. Find the method in the game code that starts a new game. Then answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 2

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

The *Up* arrow key is used to rotate a Tetris piece. Find what method is called to change the rotatation of the piece. Once found, answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 3

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

The speed of the game can be adjusted by the player through the Options menu. Using the menu to select a new speed, find out where the speed of the game is set in the code and answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 4

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

In the Score panel on the right hand side is a number of counters. There is a group of counters that track how much Single, Double or Triple lines are completed. Find the method that increments the count for the single lines completed. Once the method is found, answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 5

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

A new player can be created in the game by selecting Change Player → New. Find out where the new player information is stored and then answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 6

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

If a player wants to know the Tetris Scoring information they can go to *Help* → *Tetris Scoring* and a window will open. Find out what method displays this window and then answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 7

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

The percentage bar below the scores on the right of the screen shows the Weighted Score, i.e. how close a game is to ending. Find out where the Weighted Score is calculated and then answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Task 8

Date:
Participant Code:
Task Order:
Software Reconnaissance:

Task Instructions

The *Game Over* message is displayed when the game ends in the centre of the playing area. Find where this message is displayed and then answer the questions below.

Questions

1. In which thread is the functionality primarily executed?

2. Please describe the program flow that preceded the execution of the functionality.

3. What are the classes and methods involved in the execution of the functionality and describe how they interact to perform the functionality?

Appendix D

Post-study Questionnaire Form

Date:
User Code:

1. Describe your general experience using the tools to carry out the assigned tasks.

2. Did you have enough time to finish the tasks? If no, why not?

3. What helped you to complete the tasks?

4. What hindered you from completing the tasks?

5. What tool features did you find most useful?

6. What tool features did you find to be a hindrance?

7. Did the eye tracker affect your performance?

8. Any suggestions for improvement?

9. Closing remarks?

Appendix E

General Experimenter Instructions

Introduction

Each experiment involves you as the experimenter and one user. This document gives an overview of the experiment procedures and study design.

The handbook explains the various phases of the experiment and what is required from the experimenter. The phases are:

1. Setting up the Usability Lab
2. Setting up the workstation being used
3. Initial instructions to the subject
4. Consent Form instructions and signing
5. Pre-Study Questionnaire completion
6. Study instructions
7. Tool training
8. Task instructions
9. Post-Study Questionnaire completion
10. Final instructions

Experiment Goal

The purpose of this experiement is to study how the use of the Software Reconnaissance feature of Diver helps improve the ability for users to find features and improves analysis and navigation of code. The experimenter is their to aid the user but not directly help achieve the answer. The study will use a Java implementation of Tetris. To get the user familiar with the application and terminology they will be given the opportunity to play the game during the training and practice phase.

Phases

Each experiment is designed to not take more than 2 hours. The time limits in minutes for the phases are in brackets.

1. Setting up (Done in advance)
2. Pre-study Orientation (5)
3. Training and Practice (25)
4. Software Reconnaissance Tasks (40)
5. Break (5)
6. Non-Software Reconnaissance Tasks (40)
7. Post-Study Questionnaire and Experiment Finalising (10)

The time limit for the tasks is generous and although 10 minutes a task is the average, some tasks will take less time and some might take more than 10 minutes.

The Study Design

Research Questions

The tasks assigned to the user shall be divided into 2. One half will be performed with the aid of the software reconnaissance feature in Diver and the other half without. The tasks shall be randomly split in half and used for two subjects to assign them tasks. This will be done so that each half is performed with and without software reconnaissance. The tasks involve finding where certain events happen in the code

and to then answer questions explaining the details about the execution of the code. This is an in person study and the software reconnaissance tasks will be completed first to account for learning bias.

The subjects will fill in a pre-study questionnaire gathering information about their experience. The experiment will be recorded both on the workstation and by a video camera. The experimenter will also note observations while the user is performing the task. The user will be asked to verbalise their thoughts, ideas and processes. There will be a post-study questionnaire to end the experiment.

Experimenter Behaviour

The experimenter will be required to:

1. Go through all forms given to the user and explain them
2. Give the user a demonstration of the tool and allow them to practice what they have learnt.
3. Help the user while completing the tasks with minor problems with using the tools unrelated to the core problem finding the so.
4. Inform the user when a task is taking to long
5. Make observations based on the subjects actions, talking and mood. Take notes of all these observations.
6. Follow the procedures laid down in the handbook and mark them as complete.

Appendix F

Experimenter's Handbook

Date:
Participant Code:

A fresh copy of this handbook should be used in each experiment. Tick off items as they are covered in the experiment. Attach this form to the forms and questionnaires filled in by the participant as well as your observation notes after the experiment is completed.

Set Up

Setting up the Usability Lab:

1. A study pack ready consisting of the:
    - Experimenter's Handbook (this document)
    - User Consent Form x 2
    - Pre-study Questionnaire Form
    - User Instructions
    - Task Forms
    - Post-study Questionnaire Form
    - Paper for observations
    - 2 pens
    - $20 stipend

2. The video camera has a new tape in and is ready to record the session. Another tape ready to change after an hour.

Setting up the workstation:

1. The workstation has Eclipse open with Diver installed and Tetris and the training projects loaded.

2. ClearView is running and ready to record with the participant profile created.

Pre-study Orientation

Time limit: 5 min.

1. Introduce yourself. Help the participant get comfortable.

2. Explain the consent form. Highlight the fact that the session will be video taped and the particpant's eyes will be tracked.

3. Give them the Pre-study Questionnaire Form and briefly explain its aim. Provide guidance where necessary.

4. Inform them of the purpose of the experiment: 'This study is aimed at helping us determine the usefulness of the software reconnaissance feature in Diver.'

5. Explain that the test is about the difference between using the software reconnaissance feature and doing similar tasks without it, not the subjects performance.

6. Detail how the rest of the study will be conducted. Highlight the time limit:

    - Training and Practice (25)
    - Software Reconnaissance Tasks (40)
    - Break (5)
    - Non-Software Reconnaissance Tasks (40)
    - Post-Study Questionnaire and Experiment Finalising (10)

Training and Practice

Time limit: 25 min. Sample Program: Sudoku

- Tell the participant to ask questions at any time.
- Explain that Diver is beta software and that they may come across bugs while using it.
- Introduce the participant to training software which will be used for training and practice.
- Explain the following functionality:
    1. Recording a trace.
    2. Opening the sequence diagram for the trace.
    3. Basics of sequence diagrams.
    4. User object and lifelines.
    5. Expanding and collapsing sections.
    6. Expand All Children.
    7. Code blocks and colour coding: red for error handling blocks, blue for loops and green for conditional blocks.
    8. Activation boxes.
    9. Show grouping of lifelines in package overview.
    10. Show Focus On/Focus Up and breadcrumb functionality
    11. Jumping to the source code.
    12. Code coverage
    13. Activate trace
    14. Reveal In
    15. Explain Timeline.
    16. Software Reconnaissance
- Show participant Task 0 and explain the questions
- Let participant practice using Diver
- Introduce Tetris

Software Reconnaissance Tasks

1. Go through the Instructions with the participant.
2. Explain the eye tracking requirements.
3. Emphasize that the object is to use Diver as much as possible to complete the tasks and not to use searching.
4. Remind the participant that this set of tasks should be done using the Software Reconnaissance feature specifically.
5. Remind the participant that they have 40 minutes to complete the 4 tasks and should be averaging 10 minutes a task.
6. Remind the participant to talk out loud while completing the tasks.
7. While the participant is completing the tasks you should take notes on your observations of them.
8. After completing all the tasks or at the end of the 40 minutes stop the participant and let them have a 5 minute break.

Non-Software Reconnaissance Tasks

1. Remind the participant that for this set of tasks they will be able to use all the features of Diver except for Software Reconnaissance.
2. Once again while the participant is completing the tasks you should take notes on your observations of them.
3. After completing all the tasks or at the end of the 40 minutes stop the participant and then move on the the Post-Study Questionnaire.

Post-Study Questionnaire and Experiment Finalising

1. Go through the questions with the participant and record their responses to questions.

2. After finishing the questionnaire let the participant know that the study is over and thank them for particpating.

3. Give them the stipend and have them initial the consent form by the paragraph talking about the stipend.

Appendix G

Ethics Approval Certificate

University of Victoria

Human Research Ethics Board
Office of Research Services
Administrative Services Building
PO Box 1700 STN CSC
Victoria British Columbia V8W 2Y2 Canada
Tel 250-472-4545, Fax 250-721-8960
Email ethics@uvic.ca Web www.research.uvic.ca

Certificate of Renewed Approval

PRINCIPAL INVESTIGATOR:	**Sean Stevenson**	**ETHICS PROTOCOL NUMBER**	**07-07-349b**
UVic STATUS:	**Master's Student**	ORIGINAL APPROVAL DATE:	27-Nov-07
UVic DEPARTMENT:	**COSI**	RENEWED ON:	15-Nov-12
SUPERVISOR:	**Dr. Margaret-Anne Storey**	APPROVAL EXPIRY DATE:	26-Nov-13

PROJECT TITLE: **Reverse Engineered Sequence Diagrams to Support Software Evolution**

RESEARCH TEAM MEMBERS: Supervisor: Margaret-Anne Storey (UVic)
Investigators: Brendan Cleary (UVic)

DECLARED PROJECT FUNDING: **DRDC Contract (previous)**
DND/NSERC Grant (2009-2012)

CONDITIONS OF APPROVAL

This Certificate of Approval is valid for the above term provided there is no change in the protocol.

Modifications
To make any changes to the approved research procedures in your study, please submit a "Request for Modification" form. You must receive ethics approval before proceeding with your modified protocol.

Renewals
Your ethics approval must be current for the period during which you are recruiting participants or collecting data. To renew your protocol, please submit a "Request for Renewal" form before the expiry date on your certificate. You will be sent an emailed reminder prompting you to renew your protocol about six weeks before your expiry date.

Project Closures
When you have completed all data collection activities and will have no further contact with participants, please notify the Human Research Ethics Board by submitting a "Notice of Project Completion" form.

Certification

This certifies that the UVic Human Research Ethics Board has examined this research protocol and concluded that, in all respects, the proposed research meets the appropriate standards of ethics as outlined by the University of Victoria Research Regulations Involving Human Participants.

Rachael Scarth

Dr. Rachael Scarth
Associate Vice-President, Research

Certificate Issued On: 15-Nov-12

07-07-349b Stevenson, Sean