

# **Improving The Scalability of Tools Incorporating Sequence Diagram Visualizations of Large Execution Traces**

by

Del Myers

B.Sc., University of Victoria, 2005

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

MASTER OF SCIENCE

in the Department of Computer Science

© Del Myers, 2011

University of Victoria

*All rights reserved. This thesis may not be reproduced in whole or in part, by photocopy  
or other means, without the permission of the author.*

Improving The Scalability of Tools Incorporating Sequence Diagram Visualizations of  
Large Execution Traces

by

Del Myers

B.Sc., University of Victoria, 2005

Supervisory Committee

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

---

Dr. Daniel German, Departmental Member  
(Department of Computer Science)

---

## Supervisory Committee

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

---

Dr. Daniel German, Departmental Member  
(Department of Computer Science)

---

## ABSTRACT

Sequence diagrams are a popular way to visualize dynamic software execution traces. However, they tend to be extremely large, causing significant scalability problems. Not only is it difficult from a technical perspective to build interactive sequence diagram tools that are able to display large traces, it is also difficult for people to understand them. While cognitive support theory exists to help cope with the latter problem, no work to date has described how to implement the cognitive support theory in sequence diagram tools. In this thesis, we tackle both the technical and cognitive support problems. First, we use previous research about cognitive support feature requirements to design and engineer an interactive, widget-based sequence diagram visualization. After implementing the visualization, we use benchmarks to test its scalability and ensure that it is efficient enough to be used in realistic applications. Then, we present two novel approaches for reducing the cognitive overhead required to understand large sequence diagrams. The first approach is to compact sequence diagrams using loops found in source code. We present an algorithm that is able to compact diagrams by up to 80%. The second approach is called the *trace-focused user interface* which uses software reconnaissance to create a degree-of-interest model to help users focus on particular software features and navigate to portions of the sequence diagram that are related to those features. We present a small user study that indicates the viability of the trace-focused user interface. Finally, we present the results of a small survey that indicates that users of the software find the loop compaction and the trace-focused user interface both useful.

---

## Table of Contents

---

<b>Supervisory Committee</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Table of Contents</b>	<b>iv</b>
<b>List of Listings</b>	<b>x</b>
<b>List of Tables</b>	<b>xi</b>
<b>List of Figures</b>	<b>xii</b>
<b>Acknowledgements</b>	<b>xiv</b>
 <b>I Introduction</b>	 <b>1</b>
<b>1 Introduction</b>	<b>2</b>
1.1 The Problem: Sequence Diagram Scalability . . . . .	2
1.2 Solution: A Cognitive Approach to Sequence Diagrams . . . . .	6

<b>2</b>	<b>Dynamic Interactive Views for Reverse Engineering (Diver)</b>	<b>10</b>
2.1	The Diver Views . . . . .	11
2.2	Capturing Traces in Diver . . . . .	12
<b>II</b>	<b>Building Scalable Sequence Diagrams</b>	<b>16</b>
<b>3</b>	<b>Engineering a Scalable Sequence Diagram Viewer</b>	<b>17</b>
3.1	Requirements . . . . .	18
3.1.1	General Requirements . . . . .	19
3.1.2	The Components of a Sequence Diagram . . . . .	20
3.1.3	Usability Through Cognitive Support Features . . . . .	21
3.2	A Widget-Based Sequence Diagram . . . . .	25
<b>4</b>	<b>Evaluating the Scalability of Sequence Diagrams</b>	<b>29</b>
4.1	Sequence Diagram Effectiveness . . . . .	29
4.2	Sequence Diagram Efficiency . . . . .	31
4.3	Discussion . . . . .	35
<b>III</b>	<b>Reducing the Size of Sequence Diagrams</b>	<b>37</b>
<b>5</b>	<b>Using Loop Detection to Compact Large Sequence Diagrams</b>	<b>38</b>
5.1	A Short Survey of Trace Compaction Techniques . . . . .	40
5.1.1	Trace Compression as Related to Compaction . . . . .	40
5.1.2	Using Trace Compression to Support Analysis . . . . .	42
5.1.3	Using Repetition for Sequence Diagram Compaction . . . . .	43
5.2	Using Source Code to Compact Execution Traces . . . . .	45
5.3	Applying Source Code Compaction to Sequence Diagram Visualizations . .	48
<b>6</b>	<b>An Algorithm to Compact Loops in Sequence Diagrams</b>	<b>52</b>
6.1	Data Structures . . . . .	53

6.2	Algorithm Details . . . . .	56
6.3	Caveats . . . . .	60
6.4	Extensions . . . . .	62
<b>7</b>	<b>Experiment: Measuring Sequence Diagram Compaction Using Loops</b>	<b>63</b>
7.1	Experimental Design . . . . .	63
7.2	Results . . . . .	67
7.3	Time Analysis . . . . .	70
7.4	Threats To Validity . . . . .	71
7.5	Conclusions of the Experiment . . . . .	71
<b>IV</b>	<b>Navigating Sequence Diagrams</b>	<b>73</b>
<b>8</b>	<b>Focusing on Traces: Using Software Reconnaissance as a Degree-of-Interest Model for IDEs</b>	<b>74</b>
8.1	Degree-of-Interest Models and the Task-Focused User Interface . . . . .	75
8.2	The Trace-Focused User Interface . . . . .	77
8.2.1	Defining Software Reconnaissance . . . . .	78
8.2.2	Creating a DOI Using Software Reconnaissance . . . . .	81
8.3	Navigation Execution Traces using Software Reconnaissance in Diver . . . .	83
<b>9</b>	<b>The Trace-Focused User Interface: A User Study</b>	<b>88</b>
9.1	User Study . . . . .	88
9.1.1	Methodology . . . . .	89
9.1.2	Participants and Apparatus . . . . .	89
9.1.3	Tasks . . . . .	90
9.1.4	Procedure . . . . .	92
9.1.5	Data Collection and Analysis . . . . .	93
9.2	Findings . . . . .	94

## TABLE OF CONTENTS

vii

9.2.1	Time to First Foothold . . . . .	94
9.2.2	Frustration Utterances . . . . .	95
9.2.3	User Interaction Patterns . . . . .	97
9.2.4	Interview Data . . . . .	98
9.3	Discussion . . . . .	99
9.3.1	Interpretation of Findings . . . . .	99
9.3.2	User Study Limitations . . . . .	101
9.4	User Study Conclusions . . . . .	102
<b>V</b>	<b>Synthesis</b>	<b>104</b>
<b>10</b>	<b>Dynamic Interactive Views for Reverse Engineering: User Survey</b>	<b>105</b>
10.1	Survey Design . . . . .	106
10.1.1	Survey Results . . . . .	106
10.2	Survey Discussion . . . . .	109
10.3	Threats to the Validity of the Survey . . . . .	109
<b>11</b>	<b>Conclusions</b>	<b>111</b>
11.1	Questions Answered . . . . .	111
11.2	Contributions . . . . .	113
11.3	Future Directions . . . . .	114
11.3.1	Further Validation . . . . .	115
11.3.2	Extending the DOI Model . . . . .	115
11.3.3	Further Applications of the Techniques . . . . .	117
11.4	Conclusion . . . . .	117
	<b>Bibliography</b>	<b>118</b>

<b>VI</b>	<b>Appendices</b>	<b>130</b>
<b>A</b>	<b>The Diver Resources</b>	<b>131</b>
<b>B</b>	<b>Sequence Diagram Implementation Details</b>	<b>132</b>
B.1	The SWT Widget Framework . . . . .	133
B.2	The Draw2D Framework . . . . .	135
B.3	Creating Widgets Using Draw2D . . . . .	136
B.4	Building the Layered Architecture . . . . .	138
<b>C</b>	<b>JFace and the Model-View-Controller Pattern</b>	<b>141</b>
<b>D</b>	<b>A <math>O(n)</math> Layout Algorithm for Sequence Diagrams</b>	<b>145</b>
D.1	Layout Requirements . . . . .	145
D.2	Implementation . . . . .	147
D.3	Analyzing the Layout Algorithm . . . . .	151
<b>E</b>	<b>User Study Documents</b>	<b>153</b>
E.1	Dynamic Interactive Views For Reverse Engineering (Diver) User Study Consent Form . . . . .	153
E.2	General Information . . . . .	155
E.3	Available Diver Features . . . . .	156
E.4	Tasks . . . . .	157
E.4.1	Task 1 – Linking to Source Code . . . . .	157
E.4.2	Task 2 – Exchanging Repetitions . . . . .	158
E.5	Task Questions . . . . .	159
E.6	Interview Questions . . . . .	160
<b>F</b>	<b>User Stories</b>	<b>161</b>
F.1	Participant P1 . . . . .	161
F.2	Participant P2 . . . . .	162



F.3	Participant P3 . . . . .	163
F.4	Participant P4 . . . . .	164
F.5	Participant P5 . . . . .	164
F.6	Participant P6 . . . . .	165
F.7	Participant P7 . . . . .	166
F.8	Participant P8 . . . . .	166
F.9	Participant P9 . . . . .	167
F.10	Participant P10 . . . . .	168
<b>G</b>	<b>Diver: Dynamic Interactive Views for Reverse Engineering (User Survey)</b>	<b>169</b>

---

## List of Listings

---

1.1	The essential components of a sequence diagram . . . . .	5
3.1	List of presentation features . . . . .	22
3.2	List of interaction features . . . . .	23
4.1	Mapping of presentation features to software design . . . . .	30
4.2	Mapping of interaction features to software design . . . . .	31
6.1	Grouping invocations into loops . . . . .	57
7.1	An example of nested loops . . . . .	66
8.1	Wilde and Scully's list of software reconnaissance sets . . . . .	79
10.1	The list of Diver features ranked in our survey . . . . .	107
C.1	The sequence diagram content provider interface . . . . .	144
D.1	$O(n)$ Layout algorithm . . . . .	147
D.2	Setting the spacing for lifelines and activations . . . . .	148
D.3	Applying the layout constraints to the x coordinates of the activations . . .	149

---

## List of Tables

---

4.1	Sequence diagram efficiency benchmarks . . . . .	34
7.1	The results of the three experimental use cases . . . . .	67
7.2	The average algorithm run-time . . . . .	70
9.1	Program understanding tasks given to the participants for each session . . .	91
9.2	Time to first foothold (minutes) and feature investigated for each session . .	95
9.3	Frustration utterances for each participant per session . . . . .	96

---

## List of Figures

---

1.1	A simple sequence diagram . . . . .	4
1.2	The outline of this thesis . . . . .	9
2.1	The Diver views . . . . .	11
2.2	The Diver launch dialog . . . . .	13
2.3	Pausing and resuming a trace in Diver . . . . .	15
3.1	Analogy between SWT's delegation to the operating system and our solution	26
4.1	Sequence diagram time results . . . . .	35
4.2	Sequence diagram memory results . . . . .	36
5.1	Illustration of a solution to the common subexpression problem . . . . .	41
5.2	(A) A sequence diagram zoomed to fit; (B) the same sequence diagram compacted using source code; (C) hiding details by collapsing the com- bined fragment . . . . .	49
5.3	Selecting different iterations in the sequence diagram . . . . .	49
5.4	Selecting invocations of a method using a time line . . . . .	50

5.5	Conditional and error handling blocks discovered by an extension to the algorithm . . . . .	51
6.1	The data structures: (a) is the input data, and (b) is the output data . . . . .	54
6.2	An example transformation from the input data to the output data . . . . .	55
6.3	An example of a program for which the compaction algorithm gives inconsistent results . . . . .	61
7.1	Normal probability plots for Eclipse and Jetty . . . . .	69
8.1	Using the Diver filters . . . . .	83
8.2	The Tetris game. The <i>Resume</i> button is the feature of interest . . . . .	84
8.3	Interacting with traces using the Program Traces View . . . . .	84
8.4	The <i>Reveal In</i> action for the Sequence Diagram View . . . . .	86
9.1	The total transitions made between Diver's major views. . . . .	98
10.1	Results related to loop compaction . . . . .	108
10.2	Results related to navigation . . . . .	108
B.1	The classes of SWT . . . . .	133
B.2	The classes of Draw2D . . . . .	136
B.3	The sequence diagram widgets . . . . .	137
B.4	The classes involved in the layered architecture . . . . .	139
C.1	Using the same data model for two viewers . . . . .	143

## ACKNOWLEDGEMENTS

This work could not have been completed without the support of many people. As with all research, I am indebted to those who came before me and laid the groundwork for this project. I am also indebted to all my research collaborators who helped build and inspire this work. I would like to thank these people directly:

**My parents, Paul and Gloria Myers** and the rest of my family for raising me in an environment that made me believe that I am skilled and smart enough to pull something like this off.

**My supervisor, Dr. Margaret-Anne Storey** for gently nudging me with echoes of, “You should do a masters on this”. Without her encouragement before and during this project, I would have never completed it.

**Martin Salois, David Oulette, Philippe Charland and the Department of National Defence** for their research and financial support and for thinking that this work is important enough to keep going.

**Chris Bennett** for the research that inspired this work, and for his collaboration in the study described in Chapter 9.

**Dr. Daniel German** for his collaboration in our previous work that helped lay a foundation for this thesis.

**Dr. Jim Buckley** for his research collaboration and contribution to the user study found in Chapter 9.

**Cassandra Petrachenko** for her editorial help.

*Is not all true virtue the companion of Wisdom? – Socrates*  
*Wisdom exalts her sons and gives help to those who seek her. Whoever loves her loves life,*  
*and those who seek her early will be filled with joy. – Sirach 4:11-12*

# **Part I**

## **Introduction**

# CHAPTER 1

---

## Introduction

---

### 1.1 The Problem: Sequence Diagram Scalability

Software can be complex. The complexity of software has such a large impact that an entire field of research has been dedicated to its measurement (eg., [98, 104]). Correspondingly, software can be difficult to understand and maintain. Some research indicates that at least 50% of software maintenance effort is spent reverse engineering and comprehending programs [26]. Users need support for comprehending software.

Von Mayrhauser and Vans give an integrated theory for the processes that people use to comprehend software [93]. Individuals understand software using top-down and bottom-up approaches to build mental models. In the top-down approach, individuals generate a mental *domain* model by using pre-existing domain knowledge to form and test hypotheses about software. In the bottom-up approach, individuals build *program* models that describes small chunks of information concerning details about program flow and execution. A third mental model, called the *situational* model, contains abstractions about data



flow and functionality (e.g., “this code sorts a list”), and it is used to map understanding about program behaviour to understanding about the domain through varying levels of abstraction.

Program behaviour is important in both the program and situational mental models. Source code is a static representation of software and analytical processing performed on it is called *static analysis*. While source code is the primary representation of software, it is difficult to model program behaviour using source code alone. Modern integrated development environments (IDEs) such as Eclipse [83] support developers by offering hypertext links in source code that can be used to trace between method calls and build mental models of the program behaviour. Unfortunately, modern programming language features, such as polymorphism and dynamic type binding, often make it impossible to build consistent mental models of the dynamic behaviour of software using static analysis alone.

Another option that is well supported by IDEs is the interactive debugger. Debuggers allow programmers to trace the run-time behaviour of software by stepping through lines of source code as they are executed. Since the source code shown is resolved during the debug session, there are no longer any issues regarding dynamic typing. However, users must supply the debugger with “breakpoints” in source code that trigger suspension of the program’s execution. This means that the programmer must know what part of the software he or she needs to analyze before the debugging session can begin. Unfortunately, locating the source code that needs to be analyzed is a difficult task in itself. Programmers may resort to lexical searches in code, but they often have mixed results [77].

Both of these common approaches have the additional disadvantage that they require that programmers read large amounts of source code. Two independent studies by Fagan [22] and Weller [97] indicate that individuals can read no more than two hundred lines of code per hour before comprehension begins to degrade making it difficult to scale browsing or debugging of source code up to larger systems.

An alternative approach to building mental models of program behaviour is to perform post-mortem analysis on an execution trace [103]. Execution traces are gathered by

monitoring a target piece of software during its execution and logging information about important events such as method calls. Execution traces may be represented using the Unified Modeling Language (UML) version 2<sup>1</sup> sequence diagrams or similar visualizations. This is a popular approach (eg., [17, 37, 40, 70, 82, 86]) and it is the one that we will pursue in this thesis. Figure 1.1 shows a simple example of a sequence diagram and Listing 1.1 describes its essential components.

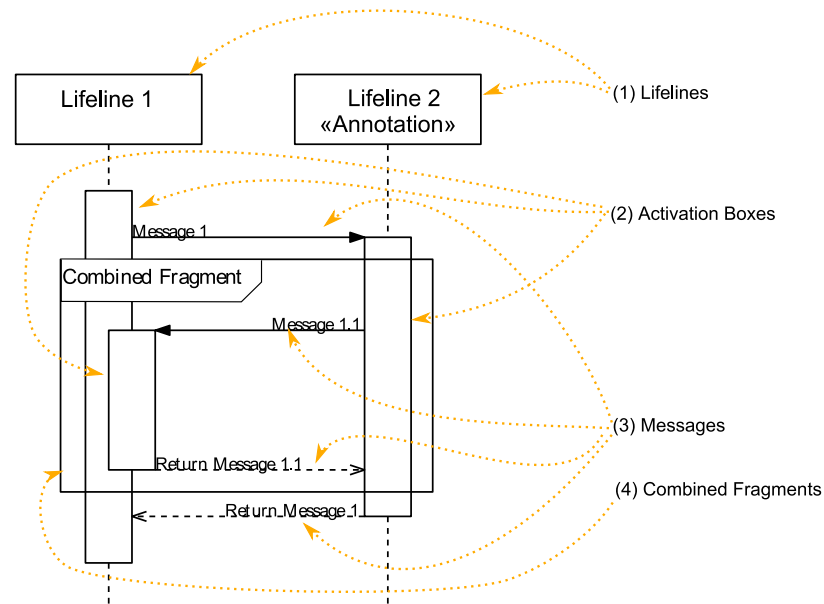


Figure 1.1: A simple sequence diagram

### Sequence Diagram Components

1. **Lifelines** The most basic component of a sequence diagram, lifelines represent the objects in the software system. At the top of a lifeline is a figure with a label, sometimes referred to as a classifier. In this example, the classifiers are boxes, which represent software objects. Other figures may be used as well. Some standard figures are “stick-men” representing human “actors” who interact with the system, or cylinders representing data stores. The label may have an optional annotation enclosed in *guillemets* («, »). Below the classifier is a long, vertical dashed line that indicates the time in the system during which an object is “alive”; the time during which the object exists in the system.
2. **Activation Boxes** Also called *Execution Specifications*, or simply “activations,” these components are not essential to standard UML sequence diagrams, but they

<sup>1</sup>We will assume UML version 2 or higher throughout this thesis unless otherwise stated.

are convenient for notation. Notated by long, vertical boxes, they indicate the time during which an object is “active” or “performing work”. A typical use of an activation box is the time during which a method is executing. In cases of recursion or of coupling between objects, activation boxes may become stacked. This indicates that the object is performing some work initiated by itself. Activation boxes are typically initiated by a message from another object in the system and end by sending a returning message to that same object. However, depending on context, return messages are not always necessary.

3. **Messages** Indicated by horizontal arrows in the diagram, messages show a flow of communication between objects. Typical examples of messages in software are method calls, returns, and exception raising. Calls initiate a new activation of a life-line and are indicated by a solid line. Messages that indicate the end of an activation (eg., returns and exceptions) are indicated by a dashed line. Although activations are always initiated by a single message, it is not necessary that they end with a single message. If applied to design of a system, or to static analysis of source code, multiple return paths (through conditional blocks, for example) can be indicated by multiple dashed lines. It is not required that a “return” message end at the activation that originated the first call either. For example, an exception message may have its end-point on the activation that catches the exception, which might not be the same as the calling activation.
4. **Combined Fragments** Combined fragments are simple blocks that surround messages. They separate messages and their resulting activations into logical groups such as loops or conditional blocks.

---

Listing 1.1: The essential components of a sequence diagram

Sequence diagrams that visualize dynamic execution traces are useful for program understanding because they display particular execution scenarios of a system. In Kruchten’s 4+1 view model of software, scenarios are important because they tie together the four main architectural views (logical, process, development, and physical) [49]. The usefulness of sequence diagrams and similar visualizations are also backed by several small studies [40, 81].

However, we risk replacing one hard problem with another: understanding source code with understanding large sequence diagrams. Dynamic execution traces can contain millions of software interactions and their sequence diagram representations will be extremely large. In 2008, we created a tool called the Oasis Sequence Explorer, which was designed to allow users to explore the dynamic execution behaviour of software using sequence di-

agrams [8]. Using the tool, we performed a study that indicated sequence diagrams are useful but that their sheer size causes cognitive overload for users [8].

If computerized tools are to be effective in helping users in their understanding tasks, they must be able to help users wade through large amounts of data to get at the information that interests them. This presents two major problems: managing the size of the presented data and supporting navigation to important data. So, scalability problems with sequence diagrams are due to both technical and human factors. Reverse engineered sequence diagrams of dynamic execution traces must be computationally efficient and effective for human use. This motivates three questions that will be addressed in this thesis:

*TQ1 How can a feature-rich and computationally scalable sequence diagram viewer be designed and built?*

*RQ1 How can the size of sequence diagrams be reduced, given that their size hinders users' understanding?*

*RQ2 How can navigation in sequence diagrams be supported?*

Question TQ1 is technical in nature while RQ1 and RQ2 are research questions dealing with the human factors involved in understanding software using sequence diagrams. In the following section, we introduce the approach used in this thesis to answer the questions.

## **1.2 Solution: A Cognitive Approach to Sequence Diagrams**

Storey suggests that we “design tools to enhance program comprehension” [79]. When individuals are tasked with understanding large information spaces, including software, they must be able to transform data into knowledge. The work involved in this transformation is called *cognitive processing*. When this work is partially offloaded onto automated tools,

it is called *cognitive support*. A tool is able to enhance program comprehension when it offers cognitive support to the user.

Bennett applied cognitive support theory to sequence diagrams using a framework of 16 *cognitive design elements* meant to guide integration of sequence diagrams into tools designed for program comprehension [7]. The design elements are also mapped to a set of cognitive support feature requirements for sequence diagrams. However, he did not give any guide for concretely implementing the design elements. Instead, he suggested that the framework may help one “appreciate the reasons” for a new feature and that it is useful for guiding developers to implement features that “help the user in the construction of mental models.”

In this thesis, with Bennett’s framework as a guide, we design solutions that use a cognitive approach to improving the scalability of tools incorporating sequence diagrams of large execution traces. To address the issue of sequence diagram size, we create a compaction algorithm that abstracts repeated patterns of execution into loops that are defined in source code. To address the issue of navigation, we introduce a novel solution called the *trace-focused user interface*. It supplies filtering mechanisms that help users locate software elements of interest in static structural views, and investigate their dynamic behaviour by supporting cross-navigation to and from a sequence diagram.

The questions that this thesis addresses are unified by the cognitive theory that will be used to answer them. Nonetheless, each question can be answered individually. In this thesis, we answer each question in parallel following a unified research plan as outlined in Figure 1.2. For each question, we first discuss the background information and literature associated with the problem that we are addressing. This leads us to propose a novel approach to solving the problem. A tool called Dynamic Interactive Views for Reverse Engineering (Diver) is used to implement and validate each solution.

This thesis is divided into several parts, each comprising several chapters. Part I contains introductory material. A short overview of the Diver tool is given in Chapter 2 to help orient the reader. Later chapters discuss the details about how Diver is used to implement

and validate the solutions to each of the questions posed in this thesis.

Part II discusses the technical problems that we face in TQ1. In Chapter 3 we state the technical and cognitive support feature requirements for a scalable sequence diagram viewer and present an architecture and design for implementing it. In Chapter 4, we use the Diver implementation of the design to run some benchmarks and test the sequence diagram viewer's scalability.

Part III addresses question RQ1. Chapter 5 includes a survey of previous techniques for reducing the size of execution traces and their corresponding visualizations. Chapter 6 introduces a novel algorithm that uses loops found in source code to compact execution traces. The algorithm is used by Diver to reduce the size of sequence diagrams. The Diver implementation of the algorithm is used in Chapter 7 within an experiment that demonstrates how well it can compact execution traces and their corresponding diagrams.

Question RQ2 is discussed in Part IV. In Chapter 8 we develop a novel method for supporting navigation in sequence diagrams. We pull inspiration from two sources: the Mylyn task-focused user interface [41], and Wilde and Scully's software reconnaissance [101]. We combine the two techniques into a novel approach for navigating sequence diagrams, called the *trace-focused user interface*, which is implemented in the Diver tool. In Chapter 9, we validate the approach with a user study.

Part V synthesizes the findings and concludes the thesis. Chapter 10 discusses a small user survey about the Diver tool that is used to add validity to the findings of Parts III and IV. Chapter 11 concludes with an analysis of the investigation of our three questions, the contributions of this thesis, and some future directions for the research.

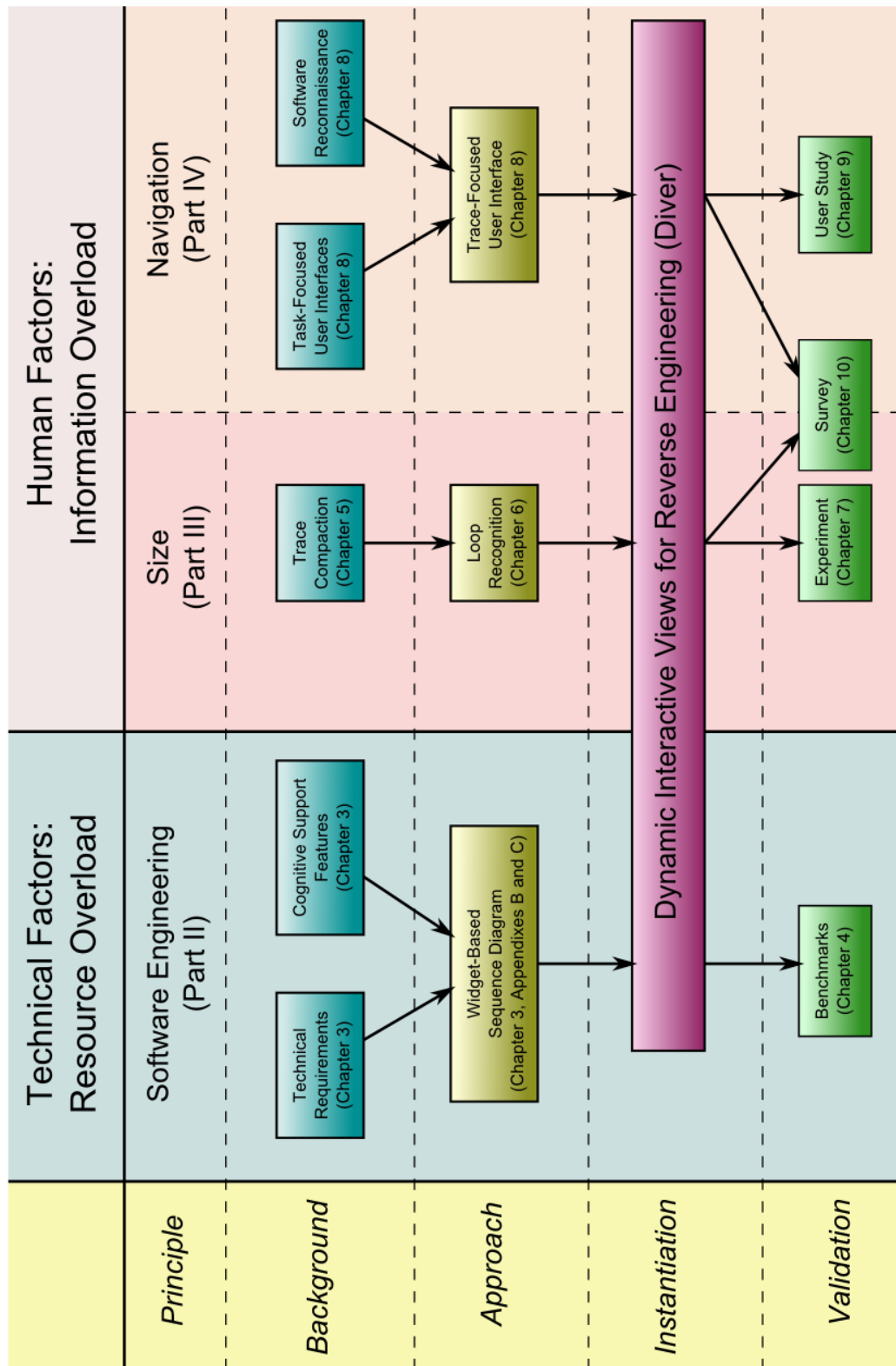


Figure 1.2: The outline of this thesis

## CHAPTER 2

---

# Dynamic Interactive Views for Reverse Engineering (Diver)

---

Throughout this work, we investigate new methods for addressing scalability issues in large sequence diagrams. This cannot be done using purely theoretical approaches. Although frameworks for cognitive support exist, (e.g., Bennett [7] and Storey [79]) and it is possible to analyze algorithms for correctness and complexity, it is nonetheless important to provide empirical evaluations of new approaches. Even proven algorithms benefit from real-world applications and testing. As it was famously quipped by Donald Knuth, “Beware of bugs in the above code; I have only proved it correct, not tried it.” [45]

The tool created to test the approaches in this thesis is called *Dynamic Interactive Views for Reverse Engineering* (Diver). It is built as a set of plug-ins for the Java Eclipse IDE [83] and designed to help programmers locate and analyze the behaviour of Java software. This chapter gives a short overview of the Diver tools in order to orient the reader in preparation for reading the rest of this thesis. Section 2.1 describes the views that Diver contributes to the Eclipse IDE. Section 2.2 describes how Diver captures and stores execution traces



of Java software. Later chapters describe the implementation and validation of the various new approaches for cognitive support that are employed by Diver.

## 2.1 The Diver Views

Figure 2.1 shows a screenshot of the Diver perspective. It shows several linked views that help the user interact with source code and dynamic execution traces. Execution traces are created by the user through the standard Eclipse launch facilities and through specialised actions contributed to the Eclipse *Debug View* (Figure 2.1-A).

Once a trace is collected, it is displayed in the *Program Traces View* (Figure 2.1-B). The traces are organized by launch name and the date and time of the trace. Each thread of execution contained in the trace is also accessible from this view.

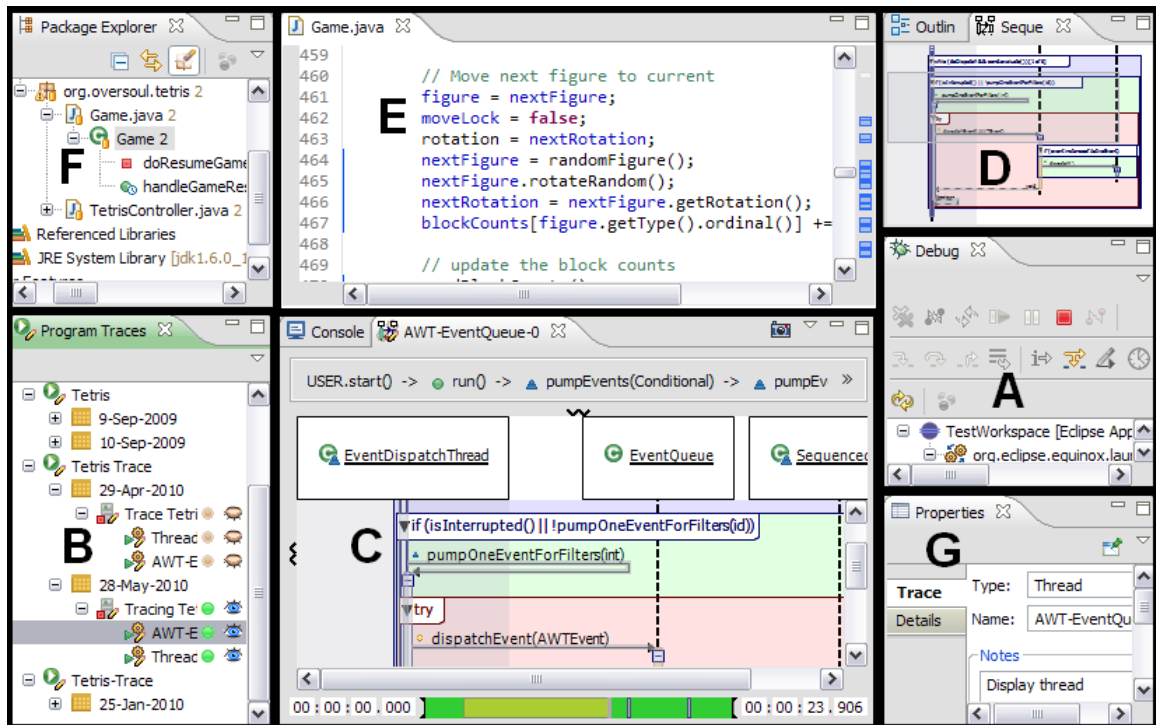


Figure 2.1: The Diver views

The user can use the *Program Traces View* to open a visualization of a thread in Diver's custom *Sequence Diagram View* (Figure 2.1-C). This view is unique in the way it visualizes

run-time behaviour as well as source code information. The design and implementation of the Sequence Diagram View is described in Chapter 3 with additional details in Appendices B and D. Chapter 6 describes the design and implementation of a new algorithm that allows us to compact the traces visualized in the diagram and supply additional cognitive support by uniting the diagram with source code. The sequence diagram is fully interactive and can be navigated using a linked outline view (Figure 2.1-D).

The Diver perspective also includes several views that are a standard part of the Eclipse IDE. Figure 2.1-E shows the Java Editor and 2.1-F shows the Package Explorer. Diver extends these views with new features that supply additional cognitive support to the user. The Java Editor may be annotated with code-coverage information pertaining to a particular execution trace (a feature not discussed further in this thesis). The Package Explorer may be filtered in order to show only the software elements unique to a particular feature of interest. The filters in the Package Explorer are a major component of the trace-focused user interface that we have created to support users in their software understanding tasks. The trace-focused user interface is the topic of Chapter 8 and the subject of a user study in Chapter 9.

Finally, every element in a trace may be annotated using the standard Eclipse *Properties View* (Figure 2.1-G). Annotations can later be searched using the Diver Trace Search. Wild-card searches for classes and methods are also supported. This feature is not covered further in this thesis.

## 2.2 Capturing Traces in Diver

The data contained within the traces that Diver stores is essential to Diver's functionality. It is therefore appropriate to discuss how Diver creates traces and makes them available for use within the Eclipse IDE.

Diver currently works only with Java software developed using version 1.6 or higher of the Java programming language and run-time. It uses Oracle's Java Virtual Machine

Tooling Interface (JVMTI) [66] to log method executions of Java programs in real time. The use of the JVMTI allows Diver to hide the technical details of how the tool generates trace data. The Diver user can simply run his or her Java application in the same way as any other Java application in Eclipse. Diver extends the standard Eclipse launch facilities to allow users to capture traces (Figure 2.2).

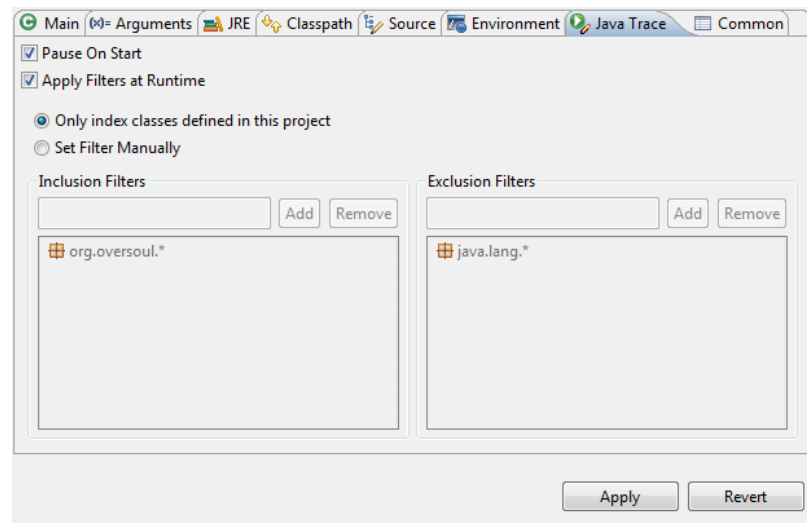


Figure 2.2: The Diver launch dialog

Typically, dynamic analysis tools that rely on execution traces suffer from two major shortcomings: data explosion and slowed execution speed due to large input/output overhead. These two problems occur because software programs can execute thousands, or millions, of instructions every second. These two shortcomings lead to further problems with analysis of the logged data. One is that the very fact of observing the run-time behaviour of the program affects its run-time behaviour. Timing information that is stored will not reflect a “natural” run of the software. So, much care must be taken to ensure that the tracer impacts the traced program as little as possible.

One way to improve efficiency is to limit the number of collisions that occur as the software writes to files from different threads. Multiple threads writing to a single file effectively transforms a multi-threaded application into a single-threaded one. Diver handles

this problem by using a different file for each thread that is being traced. This approach is also used by Reiss and Renieris [69]. However, the effect that Diver has on the actual run-time behaviour of the program is dependent on factors outside of Diver's control such as how time-dependent a program is and how the operating system writes data to disk.

The large amounts of data that is stored for a trace can also affect the responsiveness of the tool that uses that data. If there is too much data to search through, or if it is poorly organized, then it will take too long to find useful information or perform any useful analysis. One way that Diver handles this problem is to record only the calls and returns to and from methods and object constructors. Users may also apply filters so that only classes and methods that have names matching the filters will be traced. This is a common approach and variations of it can be found in Liu *et al.* [54] and Heuzeroth *et al.* [36]. Users can choose to apply these filters during the tracing process, or during an indexing step that Diver performs after the trace is completed. The traces are ultimately stored in a local database that is indexed based on things such as package, class, or method name, and the time that messages occur. Filtering reduces the size of the database, increasing query speed and user interface responsiveness. If the filters are applied only after trace completion, users may choose to change the filtering criteria during their analysis in order to get a richer understanding of the system. However, there is a trade-off between this richness of information and the amount of impact Diver has on the target application during the trace. Applying filters during the trace increases the responsiveness of the target application.

Diver also keeps trace size to a minimum by giving the user control over what gets traced in real time. By default, Diver does not store trace data unless the user explicitly instructs the tool to do so. This is done using a *Pause/Resume Trace* action that Diver contributes to Eclipse's Debug View (Figure 2.3). When Diver is in the *paused* state, it will not log program behaviour. This allows users to trace their programs only when the program is performing functionality that they are interested in, reducing both the size of the traces and Diver's impact on the execution of the program.

Diver does not offer any data compression other than that supplied by the database

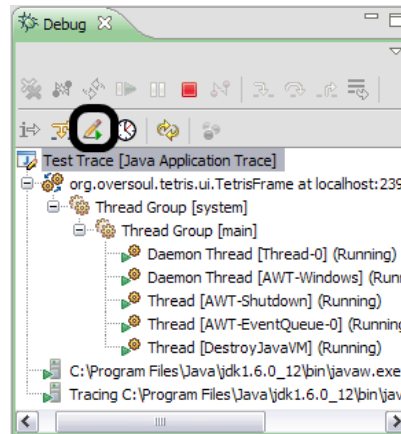


Figure 2.3: Pausing and resuming a trace in Diver

back-end. Many forms of compression exist (eg., [13, 31, 43]), but there are many trade-offs to such compression formats. They often result in some loss of information (such as timing information) and they can make it difficult to index the data for efficient information retrieval. Diver's method means that its stored trace data may take up several gigabytes of disk space. However, it is now common for disk drives to have a capacity of several hundred gigabytes, or even multiple terabytes, of available storage. The size-on-disk of a trace is becoming less of a concern.

In this chapter, we introduced the Diver tool, which we used to implement the proposed solutions to our research questions. The following chapters will investigate those questions in more depth, explaining our implementations, and testing them using experiments and user studies.

## **Part II**

# **Building Scalable Sequence Diagrams**

## CHAPTER 3

---

### Engineering a Scalable Sequence Diagram Viewer

---

The first question that we approach in this thesis is technical in nature. TQ1 is, *How can a feature-rich and computationally scalable sequence diagram view be designed and built?*

One of the first challenges in improving scalability for sequence diagram visualizations is building a sequence diagram viewer that is able to display large amounts of data. It must be able to do so within a reasonable amount of time, and without exceeding the memory available in the machine that is running it.

However, it is not enough that the viewer display information quickly and efficiently. It must also be effective in that display. That is, it must be able to display information in such a way that it helps users understand the information that they are viewing. In this sense, scalability has a broader meaning than the typical software engineering definition. Sequence diagram visualizations display a lot of information, so they must scale in the sense of supporting users while minimizing information overload. This chapter discusses the requirements for such a sequence diagram viewer, how we have instantiated them in Diver and other tools, and how the sequence diagram viewer was engineered.

The design described in this chapter has already been implemented and used in three different projects. It was first used as a part of the Oasis Sequence Explorer tool that was used in a previous study that helped inspire this work (see Chapter 1) [8]. It was also a part of a tool for tracing debug sessions of assembly language programs [6], and it is a major part of the Diver tool presented in this work. Due to the technical nature of the topic discussed in this chapter, and the one following it, readers interested in the human factors involved in the scalability of sequence diagrams may wish to proceed to Chapter 5.

## 3.1 Requirements

In software engineering, requirements are generally separated into two categories; *functional* and *non-functional*. Non-functional requirements will impose constraints on design and impact decisions about the implementation of the system. Standards such as ISO/IEC 9126 [39] and ISO 9241 [38] codify non-functional requirements in terms of software quality. The quality characteristic of usability is paramount [2]. The usability of a sequence diagram can be measured by how effectively it supports users in their understanding of software. To understand the effectiveness of our sequence diagram tools, it is important that they be built with cognitive support theory in mind.

A number of other researchers and tool suppliers have produced sequence diagram or sequence diagram-like visualizations (e.g., Systä [81], Jerding *et al.* [40], Lange and Nakamura [50], Koskimies and Mösenböck [47], Borland [16], The Eclipse Foundation [86], and McGavin *et al.* [58]). The diagrams provided by the Fujaba tool are designed to be aesthetically pleasing [65, 68]. Wong and Sun [102] used theories of perception to evaluate the sequence diagrams given by Borland Together [16] and IBM's Rational Rose [56]. However, none of these were built with specific reference to cognitive support and most have only been tested on relatively small visualizations.

We cover the different functional and non-functional requirements, and the rationale behind them in the following order. Section 3.1.1 discusses the general non-functional re-



quirements of our viewer. These requirements are not specific to sequence diagram visualizations, but are important to the overall design. Section 3.1.2 covers the basic components of a sequence diagram and discuss which ones will be supported by our viewer. Finally, Section 3.1.3 outlines the cognitive support requirements for the sequence diagram viewer that have high impact on the usability of the visualization.

### 3.1.1 General Requirements

Before we describe the specific functional and cognitive support requirements of the sequence diagram viewer, there are some general non-functional requirements that will guide our design in terms of the technologies that we will use. These requirements are as follows:

**Portability** One of our goals is to support programmers in their software understanding tasks. Programmers often work in multiple operating system environments, so we should support as many as possible. We therefore choose Java as our programming language and run-time environment since it is available on all common platforms.

**Pluggability** This viewer is part of the Diver system, but is also a research tool that may be used for other purposes and therefore should be able to be plugged into tools for different projects. We choose the Eclipse [83] platform and its underlying OSGi plug-in framework to support pluggability.

**Data-Independence** Related to pluggability, the data model underlying the view should be replaceable to support different projects and to support fast prototyping. Eclipse offers a convenient approach to supplying data independence through its JFace [90] technology for building custom viewers. We will be using this technology to build our sequence viewer. This also implies that we will be using the Standard Widget Toolkit (SWT) [91]. These technologies will be further explained in Appendices B and C.

### 3.1.2 The Components of a Sequence Diagram

The most basic functional requirement that has to be fulfilled is that our visualization must be in the form of a sequence diagram. The Object Management Group (OMG) has a detailed specification for the components of a UML sequence diagram [1]. We introduced sequenced diagrams in Chapter 1. Figure 1.1 and Listing 1.1 described the most basic components of sequence diagrams. Our implementation supports the layout and components previously described. UML sequence diagrams support other standard components that our implementation does not for the following reasons.

UML sequence diagrams may include two different kinds of *events*; **Creation** and **Destruction**. These are not supported because they can be treated the same way as normal messages. If one object calls for the creation or destruction of another object, then a simple *init* or *delete* message can be sent, respectively. This has the advantage of being able to model languages such as Java and C++, which may invoke further processing as a result of their creation (constructor calls, for example). The UML concept of an **Interaction** is left out as well. Interactions are notated as boxes with labels that surround all the elements in a sequence diagram (much like a combined fragment surrounds messages and activations). This construct is extraneous because the target platform we have chosen (Eclipse) supports the concept of a *view* that acts as a container for the diagram. Views have a label, which can serve the same purpose as the interaction label.

The UML has much more detailed specifications for messages and combined fragments as well. Eleven different combined fragments are defined in the UML and are indicated by the fragment's label. Rather than formally including these different types in our sequence diagram definition, we can make the fragment label variable, which is simpler and adds more flexibility. Some of the combined fragments defined in the UML may be composed using dashed lines; we will support nested fragments instead because it is technically simpler. The UML also formally defines concepts for the endpoints of messages. These are notated by different symbols at the endpoints (circles versus arrows, for example). Once again, rather than including the formal definition in the model for our sequence diagram,

a simpler and more flexible approach is to allow the endpoints to be styled and drawn according to user specifications.

UML also supports several constructs for time-sensitive information. Slanted message lines may be used to indicate the passage of time. However, mapping vertical space to the passage of time can waste space for information that may just as easily be shown in other ways in an interactive application (using hovers, for example). UML sequence diagrams are also able to indicate parallelism using a specialized `par` combined fragment. However, in real systems, parallelism is extremely difficult to detect and display in this manner because of its complexities. Parallelism is always synchronized using constructs such as processes, threads, or separate physical machines, and never truly occurs within a single interaction, so our implementation does not support it.

### 3.1.3 Usability Through Cognitive Support Features

If the goal of our sequence diagram viewer is to aid users in understanding complex software, the usability of the viewer will be defined by how well it supports a user's cognition. Fortunately, a lot of the work has already been done in gathering the cognitive support requirements for effective sequence diagram viewers. In a previous work, we gathered the requirements necessary through a survey of previous tools, as well as a focus group session, and validated them in a small user study [8]. Bennett further validated the features by mapping the features to a framework of cognitive design elements [7]. A full discussion of cognitive support theory and how to design tools for cognitive support is beyond the scope of this thesis. Bennett gives a good overview of cognitive support theory. What Bennett has not provided is information about how to implement the features that he describes. In this work, we provide an implementation.

Bennett's features are separated into two categories; *presentation features* and *interaction features*. Presentation features are those features that affect how the diagram is displayed to the user, while interaction features describe how the user may manipulate what is presented. The two classes of features are listed in Listings 3.1 and 3.2, respectively.

---

### Presentation Features

---

**Layout** The setting of the size, shape, and location of visual elements for viewing, specifically pertaining to sequence diagrams.

**Multiple Linked Views** The mapping of visual elements in one view to the visual elements in another view, where the elements in each view represent the same or related concepts. Linking the views coordinates them such that interactions in one view affect the others according to the mapping. For example, linking a lifeline in a sequence diagram to the source code definition of the class.

**Hiding** The ability to remove visual elements from the view. For example, filtering a set of calls from the view or collapsing a series of sub-calls so that only a single parent is visible.

**Visual Attributes** The attributes of an element that define what is drawn on-screen. For example, colour, texture, and shape.

**Labels** Display of a textual name or identifier for the element on-screen. For example, method names, return values, and class names.

**Animation** The ability to seamlessly change states in the view through the illusion of motion. This can be done through methods such as the linear interpolation of the location of elements in the view or smooth scrolling.

---

#### Listing 3.1: List of presentation features

The various feature requirements for cognitive support introduce their own challenges that influence how they should best be implemented. We categorize them into drawing, control, application framework, and data model challenges.

### Drawing Challenges

The cognitive support features of *layout*, *visual attributes*, *labels*, *animation*, and *zooming and scrolling* introduce challenges relating to how elements of the view can be drawn. Layout involves defining shapes to be drawn on the screen as well as their size and location; visual attributes such as colour and texture must be defined to draw those shapes; labels require that we be able to draw text to the screen; animation requires that the drawing may

---

**Interaction Features**

---

**Selection** A prerequisite for many other interactions. Elements must be selectable so that they can be manipulated.

**Component Navigation** Simple ordered movement between elements, such as traversal along a call tree.

**Focusing** The ability to narrow the view on a specific portion of the diagram so that it can hold the user's attention.

**Zooming and Scrolling** Standard techniques for displaying more information than can be legibly shown in a single window. Zooming scales images so that a desired level of detail may be seen. Scrolling moves a "view-port" onto the diagram so that only a portion of it is seen at once.

**Queries and Slicing** The ability for a user to identify elements in the diagram. This can be done by textual or semantic queries or searches. Slicing is a specific form of query that selects only the elements related to a single selected component.

**Grouping** A method of gathering together related elements in the diagram to collapse them into a single abstraction. Visual attributes are used to indicate when a set of elements may be grouped or ungrouped.

**Annotating** The ability to add additional user-defined, textual metadata to elements in the diagram.

**Saving Views** The ability to keep the state of the diagram so that it can be reset at a later time.

---

**Listing 3.2: List of interaction features**

---

be fluidly changed over time; and zooming requires that we be able to draw the shapes at different scales.

There are various technologies that can help overcome these challenges. Since we target the Eclipse platform, a drawing framework designed for Eclipse is preferable. We selected Draw2D [87], which is built on top of Eclipse's Standard Widget Toolkit (SWT) [91]. It uses native operating system widgets where available and supports all of the drawing requirements named here. Pertinent details of Draw2D are discussed in Appendix B.2.

### Control Challenges

In user interface terminology, a *control* is a recognizable visual element with which a user can interact. Such controls are commonly called *widgets*. Some common widgets are buttons, sliders, check boxes, and lists.

Since controls are user interface units of interaction, many of the interaction requirements from Section 3.1.3 introduce problems in this category. *Hiding, selection, component navigation, focusing, slicing* and *grouping* all require that the visual components drawn on screen can also be directly manipulated by the user. Therefore, our approach is to implement each component of the sequence diagram, as indicated by Figure 1.1, as an individual widget control. There are widgets representing activations, lifelines, messages, and combined fragments. Since we developed the tool for the Eclipse platform, we created these widgets to conform to the SWT standards. Appendix B.1 discusses the details of SWT necessary for our implementation.

### Application Framework Challenges

The *multiple linked views* feature is a complex one. It requires that our solution can interact with multiple views of data represented within the sequence diagram. This means that it must be interoperable with views that we did not build ourselves and that we may not be able to anticipate.

The capability of orienting, structuring, and providing communication between various views is part of the job of application framework technology. The application framework supported by Eclipse is called the Rich Client Platform (RCP). RCP supplies a standard Model-View-Controller (MVC) [48] pattern through a technology called JFace [90] that supplies a standard method of adapting between user interface widgets and the underlying data model. It also supplies facilities for adapting between different data models that contain related objects. Pertinent information about JFace is included in Appendix C.

### Data Model Challenges

Other cognitive support requirements are more related to the underlying data model than to the viewer itself. *Annotation* requires that metadata is stored with the data model that is being viewed. Annotations may be viewed directly in the sequence diagram itself, but that may cause excessive clutter. Other options are to use linked views or hovers to show annotations. *Saving views* is also a data model problem because it requires that a mapping be saved between the underlying data model and the state of the viewer. Finally, *querying* requires that the data model be designed and implemented in such a way that efficient queries can be made on the data.

Since one of our other requirements is that the viewer be implemented in a data-independent manner, we cannot offer a concrete implementation of each of these feature requirements in the viewer design itself. What is required is that we supply hooks by which a tool may be able to affect the viewer state based on stored metadata. For example, the viewer must expose an application programming interface (API) that allows a tool designer to affect which elements in the view are grouped based on a previously saved state. There must also be an API that enables tool designers to highlight elements or scroll the viewer based on the results of a query. The Diver tool makes use of such hooks in its implementation of these features.

## 3.2 A Widget-Based Sequence Diagram

To overcome the challenges introduced by supporting the cognitive support features in Section 3.1.3, we have chosen to implement our sequence diagram using a widget-based design. In this design, each component of the sequence diagram (lifelines, messages, activations, and combined fragments) is implemented as an interactive widget that can be manipulated by the user. Since we are using Eclipse as our platform, we will use the Standard Widget Toolkit (SWT) to build our sequence diagram.

SWT is designed to work in conjunction with the operating system that the application

is running on. Unlike other toolkits such as Swing [67], SWT does not manage widgets directly. Instead, each widget adapts to a concrete implementation created by the underlying operating system. In fact, what is meant by “operating system” in SWT can be a little ambiguous. SWT only requires the operating system to supply concrete implementations of user interface widgets, allowing for the implementation to be replaced. For example, the Eclipse Rich Ajax Platform (RAP) treats HTML, Javascript, and CSS as an “operating system” that can supply widgets for web-based applications in a browser [85].

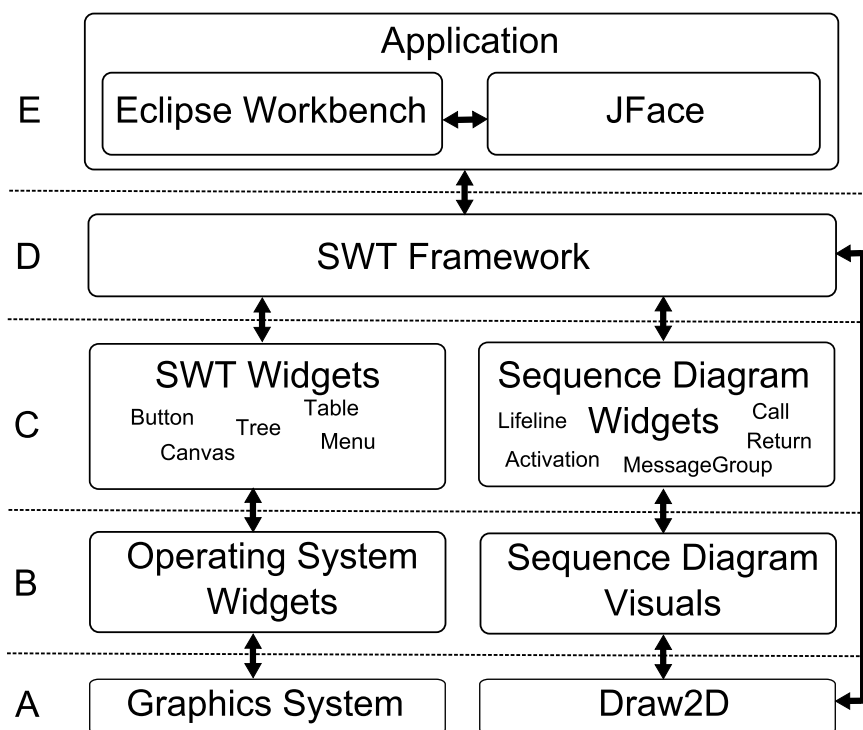


Figure 3.1: Analogy between SWT’s delegation to the operating system and our solution

Figure 3.1 shows a layered architecture in which each layer is responsible for different parts of the process. The lowest layer (A) contains the graphics drawing capabilities. On the operating system, this would include low-level graphics libraries. In our solution, it is the Draw2D drawing framework. Above this, in layer B, are the “operating system” representations of our widgets. This layer is responsible for creating handles to widgets and generating events, such as button pushes, key presses, etc., to be consumed by the appli-



cation. To emulate this functionality, we create a “visual layer” that manages a “visual” for each sequence diagram widget in the chart. We call it a visual layer because it is responsible for managing all the elements that are actually visible in the diagram. Layer C contains the widget classes that are accessible to the Java application. In both approaches, layer D is the SWT framework itself, which includes the display and the event processing logic. Finally, layer E contains the application user interface built on the Eclipse Platform Workbench and the JFace framework. The workbench is responsible for organizing the SWT widgets into contained views, menus, toolbars, editors, etc., to create a full-featured graphical user interface. JFace is Eclipse’s implementation of the Model-View-Controller pattern and it helps us to build our sequence diagram in a model-independent way.

The arrows in Figure 3.1 indicate channels of communication between the different layers. Notice that in the traditional SWT model, communication can be isolated between layers. Our approach is confounded by the fact that Draw2D is based on SWT itself. This means that what we implement in Draw2D will generate user interface events that would not be present in the traditional architecture. These events must be intercepted and translated so that they do not cause difficulty when plugging the sequence diagram into a host application. Appendix B.4 discusses the details of how each layer is built.

This design resembles Ian Bull’s approach to building widget-based graph viewers in the Zest framework [11, 12]. However, the Zest graph viewer does not use a layered architecture nor does it attempt to hide the details of Draw2D from the client. This was deliberate design decision because it allows clients to plug-in custom figures to represent nodes in the graph. Also, the design of Zest does not address how to tackle the difficulties involved with translating events that come from the Draw2D canvas. With our sequence diagram viewer, it is desirable to hide the details of the Draw2D implementation because we need to have tight control over the way that the graph is drawn. If clients could access the Draw2D layer, then they could change layout and look of the sequence diagram in such a way that it becomes inconsistent. In addition, the layered approach allows us to use a graphics drawing library other than Draw2D if at some stage Draw2D no longer fulfills our

needs.

This chapter gave an overview of the cognitive support feature requirements for an effective sequence diagram viewer and provided a brief description of a software design to implement those features. The details of the implementation can be found in Appendices B and C, as well as in the Diver source code. In the following chapter, we will evaluate the design in terms of cognitive support and the efficiency of the implementation.

## CHAPTER 4

---

# Evaluating the Scalability of Sequence Diagrams

---

In the previous chapter, we set out to engineer a scalable sequence diagram. We pointed out that the term “scalable” for us extends beyond typical performance measurements of software. There are two measures for scalability that we are concerned with: efficiency and effectiveness. Effectiveness describes how well the sequence diagram viewer can support users’ cognitive processes so that they can manage large amounts of data. Efficiency describes how well the viewer can manage and display large amounts of data. In this chapter, we will evaluate the overall design of the sequence diagram according to these criteria.

## 4.1 Sequence Diagram Effectiveness: Mapping Design to Cognitive Support Features

In the requirements outlined in Section 3.1, the effectiveness of the sequence diagram was of highest importance. We used a list of feature requirements grounded in cognitive support theory to drive the design. In this section, we validate the design by mapping its

various components to the cognitive support features. Listings 4.1 and 4.2 map provide the mapping.

---

### Presentation Features

---

**Layout** Achieved by incorporating the components of a sequence diagram into the layout as described in Section 3.1.2 and Appendix D. The diagram is laid out and drawn using lightweight widgets and the Draw2D framework (Appendices B.1, B.2, and B.3).

**Multiple Linked Views** The Eclipse Rich Client Platform (RCP) was used for this purpose. This platform allows us to generate widget-based views that can communicate with each other using view listeners provided by JFace. This supports communication using model objects rather than user interface objects (Appendices B.1 and C).

**Hiding** This is provided by the widget controls that we created and drawn using the support of Draw2D (Appendix B.3).

**Visual Attributes** The widgets we created support multiple visual attributes such as colour, outlines and icons. These can be adjusted directly through the widget interfaces, or through the label/style providers created for our JFace viewer (Appendices B.3 and C).

**Labels** Labels can be created for any widget in the diagram directly or according to the underlying data model by using the JFace viewer (Appendices B.1 and C).

**Animation** Animation is provided by the Draw2D framework (Appendix B.2).

---

Listing 4.1: Mapping of presentation features to software design

We must note that although our implementation supplies all of the cognitive support feature requirements, we have not yet discussed how to *apply* the features. For example, hiding elements at random will likely hinder cognition where hiding elements in order to remove redundancies and support abstraction will support cognition. In Chapters 5 and 6, we discuss how the hiding feature is applied to abstract repetitive patterns in traces using loops found in source code.

---

### Interaction Features

---

**Selection** Selection is a standard interaction provided by the widget design, and drawn using Draw2D. Selections can be propagated to the application using data model objects through the JFace implementation (Appendices B.1, B.2, and C).

**Component Navigation** Keyboard interactions are captured by Draw2D and translated into widget events by our implementation (Appendix B.3).

**Focusing** The widget design allows visual elements to be added, removed or highlighted according to user interaction so that particular portions of the diagram can be focused on (Section 3.2 and Appendix B.3).

**Zooming and Scrolling** This is provided directly by the SWT widget framework, without any additional design or implementation on our part (Appendix B.1).

**Queries and Slicing** Queries are not directly supported by the viewer since they must be performed on the underlying data. However, the JFace design offers a mapping between the data model and the view so that results of queries can be shown using a combination of labels, visual attributes, and scrolling (Appendix C).

**Grouping** Grouping is supported as an interaction on the widgets that we have defined (Appendix B.1 and B.3).

**Annotating** Annotation is not directly supported since data model elements, not view elements, are normally annotated. However, like Queries and Slicing, the presence of an annotation can be revealed in the view using a combination of labels and visual attributes through the JFace design (Appendix C).

**Saving Views** The widgets design exposes an interface that allows application developers to reset any view to a previous saved state, though the metadata saved and loaded must be defined by the developer (Section 3.2 and Appendix B.3).

---

Listing 4.2: Mapping of interaction features to software design

## 4.2 Sequence Diagram Efficiency: Running Benchmarks

An important part of the analysis of any new software is measurement of its efficiency in terms of memory usage and speed. Proving memory usage is straight-forward. Assuming that textual and graphical labels take a constant amount of memory, Figure B.3 indicates

that the widgets stored in the control can be linked using a simple graph. The graph may be stored using an edge list that is  $O(|\mathbf{E}| + |\mathbf{N}|)$  in size (where  $\mathbf{E}$  and  $\mathbf{N}$  are the edge and node sets for the graph). Since there will be a constant number of links between each type of widget in the graph, the size of the graph will be  $O(n)$  where  $n$  is the number of widgets.

Poranen *et al.* performed time analysis for laying out sequence diagrams based on a large set of aesthetic criteria [68]. They treat a sequence diagram as a general graph and find that for many of the aesthetic criteria, the problem of finding an optimal layout for a sequence diagram is NP-complete. For large sequence diagrams, this result is unacceptable. Fortunately, sequence diagrams are not general graphs. It is possible to place a number of constraints on the layout in order to achieve an  $O(n)$  layout algorithm. We have created such an algorithm and implemented it in our sequence diagram control. The full algorithm is available in the Diver source code. Appendix D gives an outline and analysis of the algorithm.

However, it is not enough to perform theoretical analysis of algorithms in order to gain insight about the performance of our visualization. The reason is that our implementation interacts with 3rd-party libraries (namely SWT and Draw2D) and their performance can impact our results. Therefore, we ran an empirical evaluation to benchmark how much time and memory the sequence diagram control consumes.

The benchmarks were run on a 2.8 GHz, 8-core system with 8 GB of RAM running Windows 7. However, SWT display processing is single threaded (see Appendix B.1), meaning that our tests ran, at best, with 2.8 GHz of power on a single core. Also, we placed limits on the amount of heap space that the Java Virtual Machine was allowed to consume for our tests, giving it a maximum of 1 GB of memory.

For our benchmarks, we created seven independent sequence diagrams ranging in size from 100 to 100,000 messages. We built each sequence diagram using a random process. First, we created a single activation as the root of the diagram and set it as the “current” activation. Then, we started creating messages. If the current activation was set as the root, then the message would be created as a call to a new activation that would be set as the

new current activation. If the current activation was not the root, then the next message would have a 50% chance of being a new call and a 50% chance of being a return to the previous activation. There were 1/10th as many lifelines created as messages. When a new activation was created, it would have an even chance of being placed on any of those lifelines. We followed this process to ensure that there would be no bias in the way that the sequence diagram was laid out.

This procedure created approximately 1.6 times as many widgets as messages (1 activation for each call/return pair and 1 lifeline for every 10 messages). So, the diagram containing 100,000 messages will actually contain 160,000 widgets. The number of messages will always dominate all other widgets in the diagram. This is typical of most sequence diagrams. In any given diagram, the only element that may dominate the number of messages is the number of nested combined fragments. This will only happen if there is an extremely large number of nested blocks in the source code. Such deep nesting is considered extremely bad practice, and as such is very rare [57]. The benchmarks would not be affected even if there were more combined fragments than messages. In our implementation, combined fragments are widgets just like any other visual element, and are subject to the same set of rules that govern the rest of the diagram.

For sequence diagrams from between 1,000 and 50,000 messages, we measured the amount of memory that was used by the sequence diagram control. We did this using the Eclipse Memory Analyzer Tool to analyze a heap dump of the Java Virtual Machine [88]. We measured the amount of heap space retained and used by the Draw2D classes and the classes defined by our sequence diagram. *Used* heap is the memory that is consumed by the objects on their own, whereas *retained* heap is the memory taken by those objects and all of the objects that they reference (textual labels, colours, etc.).

We took four speed measurements for each of the seven diagrams we created:

- **Creation:** The process of allocating new widgets and linking them together in a graph structure.
- **Update:** The process of creating new Draw2D figures for the newly-created widgets.

- **Layout:** The process of laying out the individual widgets to define where they will be drawn.
- **Animate:** The process by which Draw2D draws and animates the figures that have been created.

Creation and Layout involve mainly data structures and algorithms created specifically for the sequence diagram visualization. Update and Animate rely on libraries supplied by Draw2D. The raw data that we collected is in Table 4.1 and is graphically represented in Figure 4.2.

Number of Messages	Time (ms)				Memory (MB)	
	Creation	Update	Layout	Animate	Retained	Used
100	317	10	11	285	–	–
1,000	533	73	95	466	3.65	1.13
10,000	2,212	1,310	842	3,098	35.2	11.2
25,000	5,525	4,790	2,596	11,775	89.5	28.1
50,000	10,670	16,054	4,064	45,207	176	56.1
75,000	14,741	31,828	6,334	78,101	–	–
100,000	19,267	54,439	7,821	131,116	–	–

Table 4.1: Sequence diagram efficiency benchmarks

As can be seen by the data presented, the memory consumption of the sequence diagram is linear in the number of messages that are in the diagram. Approximately 3 to 4 KB of retained heap is allocated per message in total. This is relatively high, but not surprising considering the number of links that must be created to maintain the graph, and the fact that each widget in the chart must retain information about its position, size, colour, textual label, etc. We can see by comparing the used heap and the retained heap that about 2/3rd of the memory is retained for this kind of information.

The time analysis is a little more complicated. As can be seen, the two processes that do not rely on the Draw2D toolkit (Creation and Layout) have a growth linear in the number of messages created on the sequence diagram. However, the two that rely heavily on Draw2D (Update and Animate) grow by some higher order polynomial. We ran regression tests



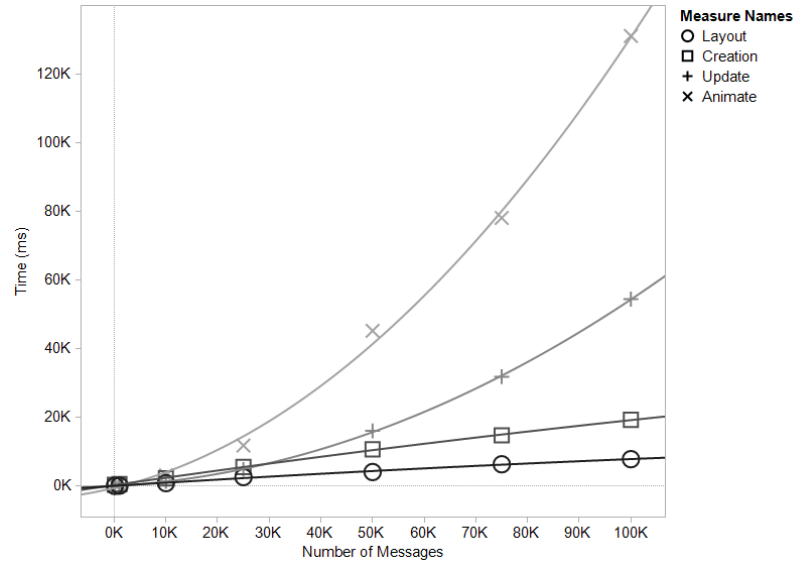


Figure 4.1: Sequence diagram time results

on the results and found that both the Update and Animate processes had  $O(n^2)$  growth rates ( $p = 1.63 \times 10^{-6}$  and  $8.55 \times 10^{-3}$ , respectively). Upon investigation, we found that Draw2D uses simple arrays to store figures with little optimization. Adding to an array is an  $O(n^2)$  operation. Large sequence diagrams require that a lot of these operations be performed by Draw2D, explaining the high growth rate for those processes interacting with Draw2D. The time efficiency of the sequence diagram could probably be increased greatly if Draw2D used a more efficient data structure to store figures.

## 4.3 Discussion

In this chapter, we analyzed the scalability our sequence diagram viewer both in terms of cognitive support and efficiency. Our viewer is a widget-based, pluggable, and data-model independent viewer, conforming to the cognitive support requirements as described by Bennett [7]. This should enable it to be an effective component in tools integrating sequence diagrams.

The sequence diagram viewer is also efficient. We were able to create sequence dia-

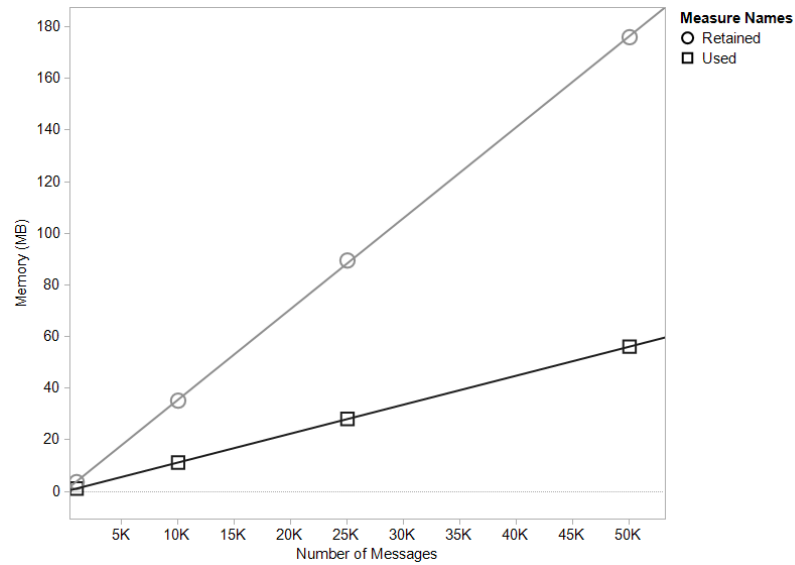


Figure 4.2: Sequence diagram memory results

grams with up to 100,000 messages, though performance began to degrade significantly both in terms of memory and processing time at around 25,000 messages. Some of this degradation is due to the data structures used by Draw2D.

However, the performance degradation at about 25,000 messages should not concern us too much. In our previous work [8], we used the sequence diagram described in this chapter to perform a user study and found that participants began to experience difficulty understanding sequence diagrams that were much smaller than 25,000 messages. The performance of the viewer is not the limiting factor.

According to our previous research, the limiting factor for using sequence diagrams is that users have difficulty comprehending the large amounts of data that they display. Although our sequence diagram design supports all of the cognitive support features described by Bennett, we have not yet discussed how they should be applied. In the following chapters, we will discuss some applications that will offer more cognitive support.

## **Part III**

# **Reducing the Size of Sequence Diagrams**

## CHAPTER 5

---

# Using Loop Detection to Compact Large Sequence Diagrams

---

One of the big challenges with visualizing large execution traces is the sheer volume of data contained in a trace. Typically, large volumes of data translate into large amounts of screen real estate, which then translates into high memory usage and lowered comprehension on the part of the user. This fact leads to our research question RQ1: *How can the size of sequence diagrams be reduced, given that their size hinders users' understanding?* We will address this question in the following chapters.

One way of addressing this problem is to scale the sequence diagram so that more information is displayed in a smaller space. Another is to create a virtual viewport onto the data, which can be scrolled. These are popular techniques used by tools such as Interaction Scenario Visualizer (ISVis) [40], Jinsight [17], Program Explorer [50], Software Exploration and Analysis Tool (SEAT) [35], the Eclipse Testing and Profiling Tools Platform (TPTP) [86] and our own Oasis Sequence Explorer (OSE) [8]. Cornelissen *et al.* [14] created their Massive Sequence View using a compressed visualization of a sequence diagram

to display an overview of the execution of software. However, zooming results in a large loss of detail, which can make it difficult to infer the real behaviour of the software. Our own experience with the OSE indicates that users prefer to scroll rather than to zoom [8].

An alternative to scrolling a viewport or scaling diagrams to visualize more or less information (geometric zooming) is *semantic zooming*. In this approach, a user zooms on the “meaning” of a visualized object, rather than on its geometric shape. For example, in common calendar visualizations one may start with a visualization of an entire month and “zoom in” to see a particular week, day, or appointment. As the user zooms in, the visualization may change completely, but it is easy to understand that it is displaying a higher level of detail than the last zoom level. Some researchers have used this method to visualize program behaviour. For example, Stasko and Muthukumarasamy used semantic zooming to show the effect of simple sorting and graph algorithms on large data sets [78]. However, their visualizations had to be tailored to the particular algorithms. DeLine *et al.* support semantic zooming of the static structure in their of software *Code Canvas* tool using levels of detail such as modules, types, and methods [18]. They provide facilities to overlay execution traces onto the visualization of the software using arrows that indicate program flow. However, we focused on the use of sequence diagrams for program understanding. Since semantic zoom often involves drastic changes to visualizations at varying levels of detail, it is out of the scope of this thesis.

In Section 4.1, we alluded to the idea that the cognitive support feature called *hiding* may be employed to deal with the large amounts of data that could potentially be drawn. Rather than trying to display all of the information contained in a trace, we can hide elements from view according to some selection criteria, and create a more compact view of the diagram. This chapter will explore how such compaction may be achieved.

Section 5.1 gives a short survey of previous work done in the area of trace compaction. Section 5.2 extends this work to introduce a new way of compacting execution traces by using source code to detect loops in execution traces. Section 5.3 discusses how the new method may be applied to sequence diagram visualizations.

## 5.1 A Short Survey of Trace Compaction Techniques

In general, the term *compaction* means to reduce the size of a thing by increasing its density. In this respect, it is related to the concept of compression. In visualization terms, it may be confused with geometric zooming. However, for this work, *compaction* means the ability to *reduce the size of a visualization by hiding data from the display*. In order for compaction to be useful, it must preserve the important semantics of the uncompacted visualization. This section surveys some of the state-of-the-art literature related to this problem.

### 5.1.1 Trace Compression as Related to Compaction

In the context of computing technology, the term *compression* refers to a method of reducing the amount of memory that is needed to represent information contained in a data set. Various methods can be used to perform compression. One of the common methods is to search data for repeated patterns and remove redundancies. So, though data compression is not our final goal, we can gain insight into how it may be possible to remove redundancies from visualizations of traces by studying trace compression methods.

There are numerous algorithms that take advantage of the repetitious nature of execution traces. Hamou-Lhadj *et al.* offer a two-stage algorithm for compressing execution traces [29]. In the first step, they pre-process an execution trace to find simple local repetitions and recursions that can be removed from the trace file because they are redundant. They present a linear time algorithms for removing such redundancies. The algorithm fails in a number of common cases, but they trade off precision for speed due to the size of the traces that they are trying to compress. This step can be repeated to locate nested repetitions. The second step is to treat the trace as a tree graph and transform it into a directed acyclic graph (DAG) by locating common subtrees that can be factored and represented only once (see Figure 5.1 for an example). Locating such common subtrees is called the common subexpression problem and Flajolet *et al.* [23] supply the solution used by Hamou-Lhadj *et al.*. In their experimentation, they were able to achieve more than 90%

reduction in trace file size after the first step (median approximately 80%). After the second step, they were able to achieve more than 97% compression (median approximately 94%). These techniques were used to create a proposed execution trace exchange format called the *Compact Trace Format* [32].

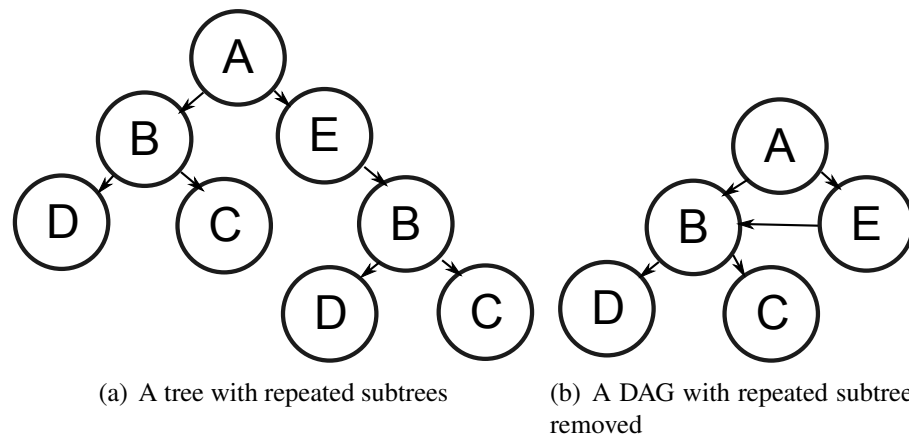


Figure 5.1: Illustration of a solution to the common subexpression problem

Reiss and Renieris use related approaches to compress execution traces [69]. They treat execution traces as long character strings in which methods are assigned their own “characters”. When a method is called, its character is recorded in a string representation of the execution trace. A special character is used to represent method returns in order to remove ambiguities. One string is recorded for each thread of execution. They discuss two approaches for compressing the traces. The first is to use the Sequitur algorithm for extracting grammars from strings [64]. Second, they offer their own approach for inferring finite state automata (FSA) from the string input. They ran experiments over execution traces of several pieces of software in order to find the compression ratios. They also tried the two methods of compression on traces that were pre-processed in order to produce DAGs similar to the ones used by Hamou-Lhadj *et al.* [29]. For all of the four combinations of compression techniques (DAG grammar, DAG FSA, string grammar, and string FSA) they achieved more than 95% compression in almost all experiments. The best performers were traces stored as strings and compressed using FSAs.

The experimentation done in execution trace compression shows conclusively that re-

moving repetitions from execution traces makes it possible to achieve high levels of compression. While the authors mentioned above recognize that such repetition is often due to loops, none of them make explicit use of this fact. Ketterlin and Claus give a method of using nested loops found in dynamic execution traces for compression [43]. Whereas the above authors consider behavioural execution traces (mostly traces of method or function calls), Ketterlin and Claus analyze traces of data access and manipulation. Their process watches updates to memory locations and looks for incremental changes that indicate that loops may be occurring. They also ran numerous experiments and compared their results to the compression that could be achieved using the standard bzip [73] compression algorithm. They found that their process was more than 8,000 times better than bzip according to the arithmetic mean and slightly more than 200 times better according to the geometric mean. The results found in the trace compression literature demonstrates the validity of the Pareto Principle<sup>1</sup> in software. It also indicates that repetition in execution traces is a very good candidate for compaction of visualizations.

### 5.1.2 Using Trace Compression to Support Analysis

Section 5.1.1 shows that the repetitive nature of execution traces can be exploited to enable information reduction. There are a number of tools and techniques that use repetition to aid comprehension and analysis. A popular example is the GNU *gprof* tool [27], which does profiling analysis by aggregating and summing function calls in C/C++ programs in order to enable the location of inefficient code. The Eclipse TPTP tools use similar methods [86]. There is research into more advanced techniques of utilizing repetition to enable analysis of software systems. We highlight some of that work here.

Hamou-Lhadj and Lethbridge extended the compression system discussed in section 5.1.1 to define what they call a *comprehension unit* [29, 33]. Comprehension units are distinct subtrees of a trace. They can be used to discover non-contiguous repetitions in the software that indicate patterns of execution. They have also been applied to extract

---

<sup>1</sup>Also known as the 80/20 rule: 80% of observed effects are produced by 20% of the possible causes.



summaries of execution traces analogous to automatic extraction of abstracts for text documents [34]. This is done using an automatic method of removing utility methods and implementation details from the trace (discovered using fan-in and fan-out analysis). Comprehension units are used as the primary components that describe the behaviour of the system. These methods of enabling comprehension and analysis have been realized in SEAT [35].

Bohnet *et al.* use similar methods to define what they term *call fingerprints* [9]. A call fingerprint is a bit vector containing many dimensions, one of which is a similarity metric used to abstract the behaviour of a running system. The similarity metric is calculated in a similar way to how comprehension units are defined, but it is more “relaxed”. Comprehension units are only considered the same if they represent the exact same subtree of a trace. Bohnet *et al.* allow two nodes in a call tree to be considered “similar” (on a scale of 0-1) according to the number of equivalent subcalls that the two nodes share. Hamou-Lhadj and Lethbridge [30] offer a categorization of properties that may be used to create heuristics for the same kind of “fuzzy” matching, but they do not offer any metrics. These kinds of matching criteria allow for less rigid summaries of the program execution which, in turn, enables greater compaction.

Locating repetitions in execution traces can also help in feature location. Safyallah and Sartipi use a technique called *execution pattern mining* [72] which analyzes multiple execution traces, each exercising a particular feature of interest. It adapts sequential pattern mining [4] to expose behaviour that implements the feature. Although this process is not specifically associated with compaction, it uses some of the same ideas in that it makes use of the fact that execution traces tend to be highly repetitive.

### 5.1.3 Using Repetition for Sequence Diagram Compaction

The work surveyed so far has discussed how repetitive patterns can be used to compact execution traces and extract information from them for analysis, but none of this work has been applied to sequence diagrams. While there has been a general indication

that such repetition is often due to loops in the software under investigation, most of the techniques do not explicitly take advantage of this fact. This subsection will review some of the literature and tools that discuss how to infer loops from execution traces and apply them to sequence diagram visualizations.

Sharp and Rountev mention interactive visualizations of loop fragments in their “wish-list” for sequence diagram visualizations [74]. However, support for such interactions tends to be rudimentary in state-of-the-art tools. MaintainJ [37] offers the ability to remove contiguous repetitions of a single method call from its sequence diagram visualization in order to remove clutter. Our own OSE tool uses an unpublished nested loop recognition algorithm to abstract repetitions and hide iterations inside a collapsible combined fragment. Taniguchi *et al.* [82] offer several “rules” for locating repetitions and compacting them for view in sequence diagram visualizations, but no implementation details are given.

Lui *et al.* treat sequence diagrams as long character strings. They perform analysis on the strings to extract suffix trees that are used to infer loops which can then be visualized in a sequence diagram [55]. Their process is reminiscent of the compression technique used by Reiss and Renieris as discussed in Section 5.1 [69]. The technique employed by Lui *et al.* is applied directly to a sequence diagram representation, but it could also be applied to execution traces before they are visualized.

Jerding *et al.* [40] also use DAGs to produce compact visual representations of execution traces. Their DAG extraction process walks the execution call tree from the leaves to the root and uses hashing to organize similar subtrees into equivalence classes. They extend the process using substring matching heuristics to ensure that contiguous repeated patterns are recognized as looped regions. Their work predates the UML specification of sequence diagrams. However, they give a visualization called the *Global Execution Mural*, which is similar to the Massive Sequence View described previously [14]. Repeated patterns are visualized in a view called the *Pattern Mural* that uses the same drawing technique, but only shows the repeated patterns that have been detected in the order that they were encountered.

Briand *et al.* [10] treat the issue of visualizing loops as a modelling problem. They

use the Object Constraint Language (OCL) to unify extensive models of Java execution traces and UML sequence diagrams. They compact loops in sequence diagrams by using combined fragments. However, the use of OCL requires that they strictly comply with the UML and its limited vocabulary. They do not offer an explicit algorithm or set of procedures for applying their process.

In this section, we have reviewed literature concerned with trace compression. We have seen that high levels of compression can be achieved due to the repetitive nature of execution traces. The repetition within traces has also been used to support trace analysis and to help compact visualizations traces. The approaches reviewed so far (with the exception of Lui *et al.* [55]) all use execution traces as their primary input. They do not use other sources of information to aid in compaction or support comprehension. When concerned with trace analysis and visualization, authors appear to make the (safe) assumption that “less is more”, but none of them make explicit reference to any cognitive support theory. In the next section, we will investigate how additional information available in source code can be used to compact sequence diagrams and improve cognitive support.

## 5.2 Using Source Code to Compact Execution Traces

Typically, repetition in execution traces results from one of three programming practices, all of which originate in source code:

1. **Recursion:** the ability for functions and methods to make self-calls. Often indicates a divide-and-conquer technique of solving a problem by breaking it down into smaller sub-problems of the same kind.
2. **Cross-Cutting Dependencies:** methods or functions that are called from numerous locations in the source code. These often indicate utility methods or cross-cutting concerns that may be encapsulated as aspects.
3. **Loops:** a typical programming language feature that allows the software to iterate

over the same segment of code in order to repeat some calculations.

Appendix B.3 discusses the implementation of our sequence diagram visualization and it offers a natural way to deal with recursion: an activation box may be collapsed to hide recursive calls. Cross-cutting dependencies may be filtered from sequence diagrams by detecting utility methods by using the process given by Hamou-Lhadj and Lethbridge [34].

Loops are the final source of repetition in execution traces. No work to date has considered the fact that loops in execution traces occur because loops are defined in source code. If source code is available for a piece of software, it may be useful in compacting sequence diagram visualizations of execution traces.

Using source code to compact execution traces may seem counter-intuitive because it introduces the need for additional data. However, our goal is not data compression, but compaction of large sequence diagrams to support software comprehension. Utilizing source code can help us achieve compaction while introducing semantic meaning. If a loop can be detected in an execution trace by mapping the loop to its source code representation, then the sequence diagram can be drawn in such a way as to expose the source code that caused the loop to occur. Other methods of compaction are only able to indicate *that* a loop occurred, they are not able to show *what caused* the loop.

According to Bennett, indicating the semantic relationships between software objects in sequence diagrams is important because it helps to enhance bottom-up comprehension of software [7]. Using loops in source code further supports cognition because it provides a mechanism for abstraction. Loops allow repetitive operations to be abstracted into concepts like *scan for item x* or *sort this list*. According to Von Mayrhauser and Vans, such loops may serve as beacons that help programmers build mental models of their software from the bottom-up [93]. Abstraction mechanisms such as these are also an important part of Bennett's cognitive design framework [7].

Using source code to compact sequence diagram visualizations of execution traces requires that we use static and dynamic analysis approaches. We must combine static source code with dynamic execution traces. Although no other other work has combined source

code and execution traces to compact sequence diagrams, there has been other research that joins static and dynamic analysis. For the sake of completeness, it is important to discuss some of that research.

Kimelman *et al.* [44] offers an application called Program Visualization (PV), which has an *active loop view*. This view acts as a postmortem debugger, highlighting source code as it is reached in a play-back of an execution trace. PV is tied to the IBM AIX system, relying on the trace data that it supplies. The process given does not, in fact, attempt to compact traces for visualization, but it does help users locate source code from within an execution trace.

A popular form of analysis is that of the *program slice*. Slices are defined in terms of behaviour. A slice is a small, valid program that produces a specified behaviour and is extracted from a larger program that also produces that behaviour (among others). Slices can be of static source code [95, 96] or dynamic execution traces [3]. Several authors have suggested combining the two kinds of slices in various ways (e.g., Ashida *et al.* [5] and Rilling *et al.* [71]).

A number of researchers also combine static and dynamic analysis for visualization. The Alborz project [72] is built on Eclipse and offers several coordinated views of reverse-engineered software, some of which are of dynamic traces and others are of static source code. Systä [80] augments static visualizations of Java software in Rigi [61] by annotating them with code coverage information from execution traces.

The background in combined static and dynamic analysis of software does not give us any direct insights into how to compact loops using source code. A novel approach is necessary. In the next section, we will present the solution provided in the Diver tool. Chapter 6 will discuss its implementation.

## 5.3 Applying Source Code Compaction to Sequence Diagram Visualizations

We modeled the user interface that Diver presents for loop compaction on our previous work [8] and Sharp and Rountev’s “wish list” [74]. It applies the cognitive support feature called *grouping* (see Section 3.1.3) by using combined fragments to group sequences of messages contained in loops and conditional blocks. Standard UML has a limited vocabulary for combined fragments, including labels such as “loop” and “alt”. Diver extends this vocabulary by using the text of the statement that defines the block. This is an application of the *labels* cognitive support feature that, according to Bennett, helps to indicate the semantic relationships between software objects.

Figure 5.2 gives a simple example of how much our approach can potentially compact a sequence diagram. We present a small sample program that uses a `for` loop in Java to call two methods 100 times. Figure 5.2 (A) shows the resulting trace without loop compaction, and Figure 5.2 (B) shows it with compaction. Considering that approximately 15-20 method invocations can be shown on a typical modern display using Diver, an unzoomed version of 5.2 (A) would occupy 10-12 screens. Compaction is achieved in 5.2 (B) by applying the *hiding* cognitive support feature to display only one iteration of a loop at a time. It can be seen that for even very simple programs, the compacted view is much simpler and easier to read. The label for the combined fragment also displays semantic information about the loop that it represents. Since Diver was created using the widget design described in Section 3.2, the sequence diagram viewer also supports information hiding. Combined fragments can be collapsed, abstracting away the information contained in the fragment if the user deems it uninteresting (Figure 5.2 (C)).

When interacting with execution traces, it is not enough to hide iterations of loops. Different iterations can have different side effects, so it must be possible to inspect each of them. Diver offers the ability to interactively select which iteration to view via a pop-up menu (Figure 5.3). This allows users to check individual loop iterations to investigate

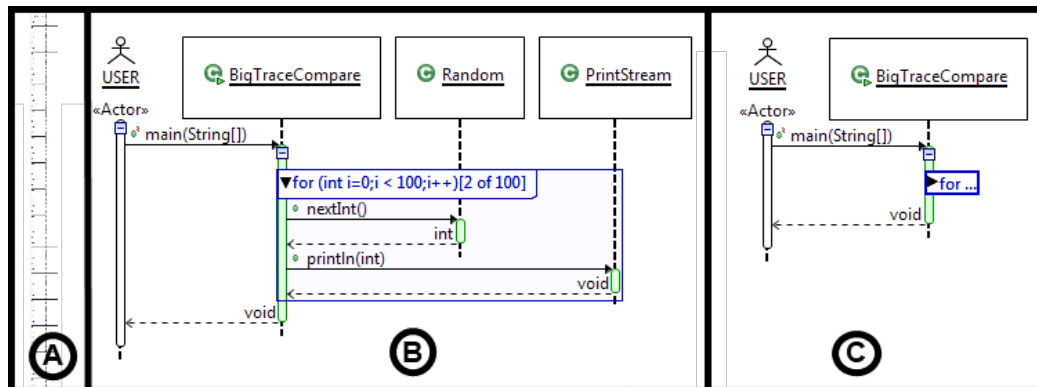


Figure 5.2: (A) A sequence diagram zoomed to fit; (B) the same sequence diagram compacted using source code; (C) hiding details by collapsing the combined fragment

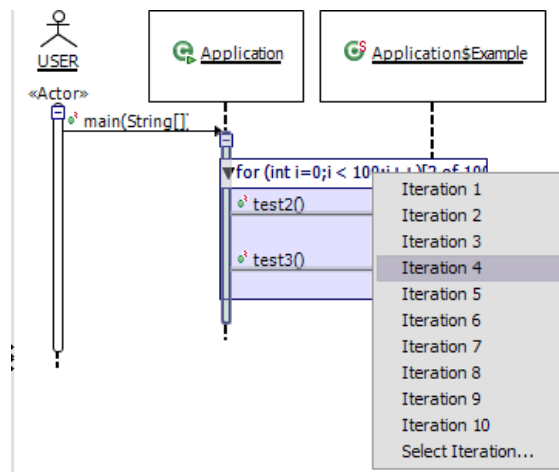


Figure 5.3: Selecting different iterations in the sequence diagram

whether various iterations cause different invocations.

The sequence diagram is tightly integrated into the Eclipse user interface because it uses the JFace patterns as described in Appendix C. This makes it possible for the sequence diagram to respond to interactions with other views using the *multiple linked views* cognitive support feature. One view that the sequence diagram responds to is the Eclipse Package Explorer. Methods are likely invoked many times within the execution of a program, but methods are only represented once in the Package Explorer. To help solve this problem, Diver offers a time line that shows the various invocations of a selected method over the span of the execution trace. Invocations are shown as small vertical strips in the time line.

Users can right-click on a strip and either reveal the invocation (viewed as an activation box) in the sequence diagram, or focus on it (Figure 5.4). Revealing expands and scrolls the view to show the invocation. Focusing sets the root of the diagram to the selected invocation, creating a slice of the call tree. In both cases, loop iterations are swapped automatically to ensure that the invocation can be shown in context. These represent applications of both the *queries and slicing* and *hiding* cognitive support features.

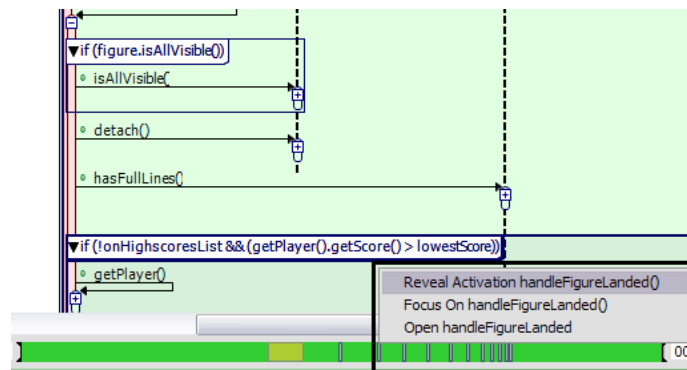


Figure 5.4: Selecting invocations of a method using a time line

One final pertinent feature of Diver is that it extends the concepts outlined in this chapter so that combined fragments for other Java code blocks can be visualized as well. Diver supports conditional blocks (`if` and `switch`) and error handling blocks (`try` and `catch`). Figure 5.5 illustrates combined fragments for Java code blocks. It shows a `try` block, a `for` loop, an `if` block, and a `catch` block. Beyond the fact that the algorithm is able to compact the loops in the diagram (only 1 of 31 loops is shown), this example also demonstrates the expressive power of using source code to visualize loops and other code blocks in sequence diagrams. By inspecting the figure we can see not only what happened, but we are also able to see some information about why it happened. In this example, it can be seen that on the 31st iteration of a `for` loop, which is counting down, the method `checkRange` is called on the `TestUtilities` class. Inside an `if` block, the condition `number <= 0` evaluates as `true`, so a new `IllegalArgumentException` is created, and the method immediately returns (indicating that the exception was thrown). Finally, we can see that the exception is caught in the `catch` block following the `for` loop. This can



all be seen in the sequence diagram in a compact way without the need to navigate through source code that would typically require inspecting several source files.

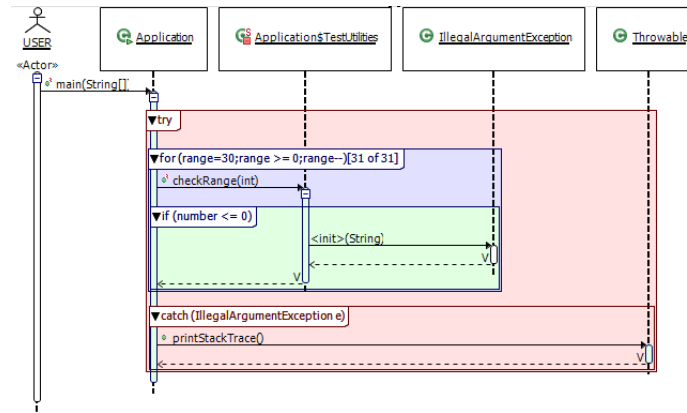


Figure 5.5: Conditional and error handling blocks discovered by an extension to the algorithm

Having given an overview of the user interface for our solution, we describe the algorithms and data structures that allow us to create it in the next chapter.

## CHAPTER 6

---

### An Algorithm to Compact Loops in Sequence Diagrams

---

Chapter 5 discussed methods for compacting execution traces and gave the motivation for a technique that uses source code to compact sequence diagram visualizations of such traces. The Diver tool was mentioned as a reference implementation of the technique. This chapter outlines the algorithm that is used by Diver to achieve compaction.

The basic idea of the algorithm is relatively simple. We consider one invocation of a program statement at a time and try to compact all of the statements that it invokes. What constitutes an invocation is language and application dependent. Method or function calls are some of the most commonly-traced invocations, but other statements, such as variable assignments or expressions, could also be considered invocations.

We use the source code representation of the program to compact the trace. In order to do this, it must be possible to match a traced invocation to its original representation in source code. For the algorithm to be usable within a reverse engineering tool, we would like there to be few adoption barriers for users, so we make use of readily-available debug data as input for the algorithm. A standard debugging compiler will typically record the

source code line number of an invocation. However, a line number is not sufficient to make a match in source code, since many statements may exist on a single line. The trace data must also contain a representation of the invocation being made so it can be matched to the source code. Most compilers will store the signature for method or function invocations in symbol tables or within the debug information in the executable. This makes method or function calls particularly good candidates as input to our algorithm. Some debugging compilers may also store the column value for the originating source code for an executed instruction. In such cases, the column value may be used instead of a signature.

With these three sources of information (source code, debug data, and trace data), the remaining problem is to match invocations to the loops in the originating source code. Once this is done, we can prune loops by hiding the uninteresting iterations. Section 6.1 describes the data structures that we need for our compaction algorithm, which is described in Section 6.2. Section 6.3 discusses some situations in which the algorithm may produce unexpected output and how to deal with them. Finally, Section 6.4 describes extensions to the algorithm that can provide further cognitive support in sequence diagrams.

## 6.1 Data Structures

We use several simple tree data structures as input to the algorithm (see Figure 6.1 (a)). The first data structure is a call tree stored for the trace. Two data items must be stored with each invocation in the call tree: a line number and a textual representation of the invocation that we call a *signature*. Both can be retrieved from the debug information or symbol tables stored with the program.

The second data structure is the abstract syntax tree (AST) of the source code and standard ASTs for many programming languages will work. The most important part of the AST is the statement. We define statements as any program construct, and a statement may contain any number of other statements. We list two specializations of statements – invocations and loops – because they are important concepts in the algorithm. Invocation state-

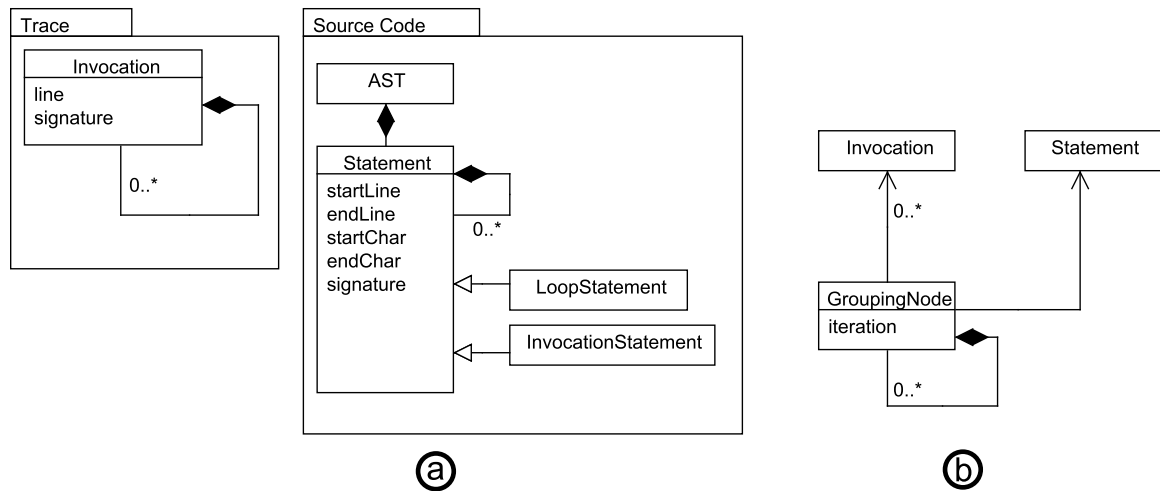


Figure 6.1: The data structures: (a) is the input data, and (b) is the output data

ments require the program to do some work such as invoke a function or assign a variable. A class definition is an example of a statement that is not an invocation. Loops are statements associated with standard keywords such as `for`, `do`, `while`, etc. They may contain other statements including invocations, as well as other loops. All statements must be associated with the region of text that they occupy in the originating source (`startLine`, `endLine`, `startChar`, `endChar`), and a signature. These regions will be used to match statements to the invocations in the trace.

The objective of the algorithm is to group repeated calls found within loops. The GroupingNode tree structure of Figure 6.1(b) is used for this purpose. A GroupingNode has one statement that defines a group of invocations. This statement could be the block that defines a loop or a method. The GroupingNode also contains a list of invocations that occur within the group. In these elements, the GroupingNode mirrors the abstract syntax tree. Loops may repeat (i.e., iterate) during the execution of a program. A GroupingNode is created for each iteration of the loop. Each GroupingNode is given an iteration count that represents the order in which the iteration occurs in the trace.

An example of the transformation from the input data to the output data is shown in Figure 6.2. In this example, we have the AST for some code that consists of a single loop that is executed twice and invokes two methods each time. The corresponding trace is

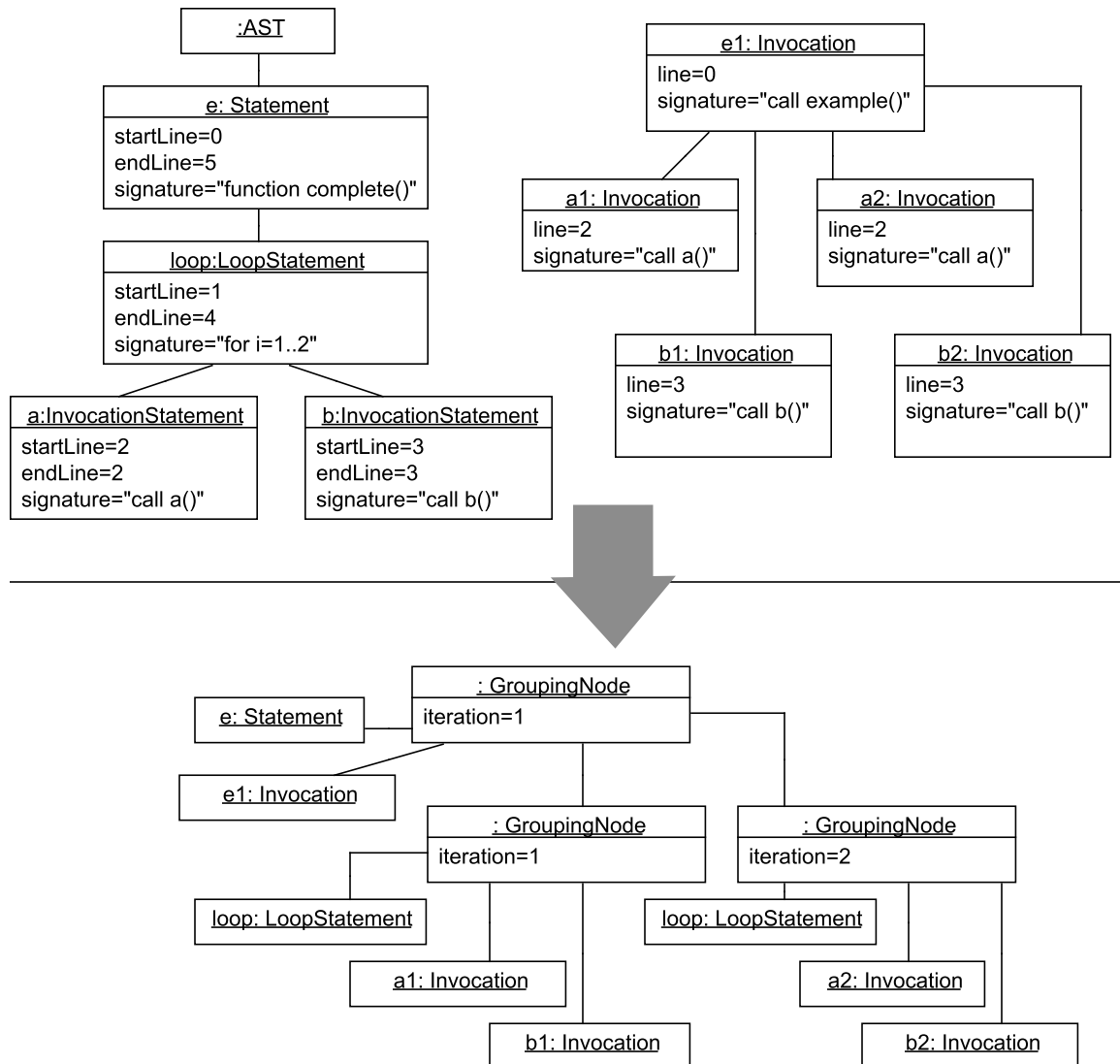


Figure 6.2: An example transformation from the input data to the output data

shown beside it. It contains five method invocations (one start invocation and four children). On the far right, the data is combined into one grouping tree.

The grouping tree is used to reduce the amount of data that must be visualized. Each GroupingNode has an iteration associated with it. In order to reduce the size of the visualized trace, we display only one iteration at a time. A selection criteria must be used to decide which iterations to show. Section 5.3 gives an example of how user input via a context menu on the visualization can act as selection criteria. The experiment described

in Chapter 7 uses another automatic criterion, which is also the default view of sequence diagrams in Diver.

## 6.2 Algorithm Details

The main part of the algorithm is shown in Listing 6.1. It is called Group-Invocations and it performs the transformation explained in Section 6.1. For the sake of brevity, Group-Invocations is the only part of the algorithm listed in full detail. Other details will be explained within the text. A full implementation can be found in the Diver source code (see Appendix A). Our implementation supports the Java programming language, but the following algorithm should generalize, with a few modifications, to many object-oriented or procedural programming languages.

The algorithm takes an invocation,  $i$ , and its associated AST statement,  $a$ , as input. The invocation must be one that is able to have child invocations, such as a method call. However, the associated statement for this invocation is the method definition and not the originating method call.

Since the algorithm associates invocations to corresponding statements, the first five lines set up associative structures. Line 3 creates a new GroupingNode tree called *root* that is the output of the algorithm. *m* (line 4) is an initially-empty associative array used to look up the statement associated with an invocation. Similarly, *gs* is used as a quick look-up into the GroupingNode tree. When looking up a GroupingNode for the statement representing the loop, the algorithm is only concerned with its current iteration, so *gs* contains references to the GroupingNode that represents the latest iteration of a loop for a given statement. The references held in *gs* are used by the sub-algorithms Get-Grouping and Next-Iteration to update the tree structure. Lines 4 and 5 set the initial associations for the inputs  $i$  and  $a$  in *m* and *gs*.

Lines 6 through 29 contain the main loop of the algorithm in which each sub-invocation is processed. The first step (line 7) locates the source code that caused the invocation of

**Algorithm** Group-Invocations**Require:** A traced Invocation  $i$ **Require:** A Statement  $a$  that defines  $i$ **Ensure:** A GroupingNode tree structure as output

---

```

1: Let  $m$  be an (initially empty) associative array of Invocations to Statements
2: Let  $gs$  be an (initially empty) associative array of Statements to GroupingNodes
3: Create a new, empty GroupingNode,  $root$  to represent the root of the node tree
4:  $m[i] \leftarrow a$ 
5:  $gs[a] \leftarrow root$ 
6: for all  $c$  of  $i.children$  do
7:    $m[c] \leftarrow \text{Associated-Statement}(a, c)$ 
8:    $b \leftarrow \text{Parent-Block}(m[c])$ 
9:    $gs[b] \leftarrow \text{Get-Grouping}(b, gs)$ 
10:  if  $gs[b].\text{Empty}()$  then
11:     $gs[b].\text{Add}(c)$ 
12:  else
13:     $l \leftarrow \text{Last-Invocation}(gs[b])$ 
14:    if  $l.line \leq c.line$  then
15:      if  $m[l].\text{Is-Child-Of}(m[c])$  then
16:         $m[l]$  is expected to be later than  $m[c]$  according to syntax... continue
17:      else if  $m[c].\text{Is-Child-Of}(m[l])$  then
18:        if  $m[l].startChar \leq m[c].startChar$  then
19:           $gs[b] \leftarrow \text{Next-Iteration}(b, gs)$ 
20:        end if
21:      else if  $m[l].startChar \geq m[c].startChar$  then
22:         $gs[b] \leftarrow \text{Next-Iteration}(b, gs)$ 
23:      end if
24:    else
25:       $gs[b] \leftarrow \text{Next-Iteration}(b, gs)$ 
26:    end if
27:     $gs[b].\text{Add}(c)$ 
28:  end if
29: end for
30: return  $root$ 

```

---

Listing 6.1: Grouping invocations into loops

the current sub-invocation  $c$ . This is done by utilizing the debug information that has been stored with the trace. The sub-algorithm Associated-Statement traverses the AST starting at statement  $a$  to find the line on which the invocation  $c$  is executed. Associated-Statement

then iterates through the statements on that line, and returns the first invocation statement that matches the signature for  $c$ . Parent-Block (line 8) traverses up the AST, starting at  $m[c]$  to retrieve the block statement (i.e., loop)  $b$  in which the previously-located statement is contained. If no loop can be found, then  $a$  is returned.

Get-Grouping creates the necessary GroupingNodes needed to associate the child invocation  $c$  with the loop statement  $b$ . First, the associative array  $gs$  is checked to see if  $b$  already has a GroupingNode associated with it. If it does, then  $gs[b]$  is returned. If it does not, then a new GroupingNode associated with  $b$  is created and put in  $gs[b]$ . Get-Grouping must also ensure that GroupingNodes exist for all of the containing blocks for  $b$ , so it recurses on  $\text{Get-Grouping}(\text{Parent-Block}(b), gs)$  until  $gs[b] = \text{root}$ . This way, Get-Grouping ensures a consistent tree structure.

What is left is to ensure that the current GroupingNode,  $gs[b]$ , represents the correct iteration of the loop represented by  $b$ , and then to add the invocation  $c$  to the group. First, if the GroupingNode  $gs[b]$  has no invocations in it (line 10), then it must be on the first iteration, and  $c$  can be added to it (line 11). Otherwise, we compare line numbers and associated AST positions to check if the trace is looping. The basic idea is this: if the current invocation  $c$  precedes the last invocation  $l$  (lines 13 and 14) in the syntactic structure of the source code, and both are in the group  $gs[b]$  (line 9), then the trace has begun a new iteration of a loop, and the groups must be updated. To make this process possible, sub-algorithm Last-Invocation (line 13) simply returns the last invocation statement added to the GroupingNode found in  $gs[b]$ .

Normally, one would assume that if the line for  $l$  is less than the line for  $c$ , then the program is progressing without looping. This normal case of loop iteration is dealt with on line 25. However, there are exceptions to this rule. Lines 15 through 23 take care of the exceptions. If the statement associated with the last invocation is a child of the statement for the current invocation, then the relationship is reversed. For example, `foo(bar())` will call `bar`, then `foo`, but `foo` comes before `bar` in the source code. So, line 15 checks for this condition and allows the algorithm to proceed without creating a new iteration for



the loop. Note that since Last-Invocation returns the last invocation added to a particular GroupingNode, and the main loop on line 6 iterates through all sub-invocations of the input  $i$  in order, the association  $m[l]$  found on lines 15 through 21 must have been previously set by line 7.

The algorithm also has to deal with complications that occur when it encounters loops that only span a single line. Lines 21 through 23 deal with the normal case by comparing the location of the starting character for the statements associated with the last and current invocations. For example, given a loop that contains only the single line `foo(); bar();`, each iteration of the loop would begin with `foo`. It is on the same line as `bar` but with a lower character index.

Once again, if there is a parent-child relationship, then the character ordering in source code is reversed. Lines 18 through 20 take care of such instances of single-line loops. Lines 18 through 20 also handle an exception that can occur in some object-oriented languages in which a method may be called on the return value of another method. For example in `foo().bar()`, `foo` gets called before `bar`, but in the AST, `foo` is a child of `foo().bar()`, and they both have the same start position. So, when a loop occurs on a single line, the relationship is similar to that of `foo(bar())`.

The above discussion accounts for parent-child relationships for invocation statements that occur on a single line. Similar checks can be added to line 25 of the algorithm to handle cases that are split across multiple lines.

The last detail to explain is the functioning of the Next-Iteration sub-algorithm. Next-Iteration updates the GroupingNode tree structure when a loop occurs. First, the current iteration for the GroupingNode in  $gs[b]$  must be temporarily stored: call this iteration  $it$ . The associative array  $gs$  is then cleared for the AST statement  $b$  and all of its children. Next, Get-Grouping is called for  $b$  and  $gs$ . Since the associations in  $gs$  for  $b$  and its children have been cleared, this will force Get-Grouping to create a new tree structure for the AST of  $b$  that will hold all of the invocations for the next iteration of the loop.  $gs[b]$  is set to this new grouping, and its iteration number is set to  $it + 1$ . The new  $gs[b]$  is returned, which updates

the tree for the next loop. The algorithm runs on one invocation at a time and can be run iteratively over each invocation as required.

## 6.3 Caveats

There are a number of conditions that will cause the algorithm to fail. First, the algorithm assumes that invocations in the trace can always be matched with statements in source code (line 7). This is not the case for some programming languages. For example, Java will insert method invocations that load classes from disk the first time that they are referenced in the program. Such invocations are typically side effects of invocations that are present in the source code, and the problem can be solved by associating invocations with their side effects. These cases are language dependent and must be dealt with on a language-to-language basis.

Second, the sub-algorithm Associated-Statement cannot guarantee that it returns the correct statement in the case that multiple invocations with the same signature occur on the same line of code. It is not possible to distinguish the two invocations because the only data given in the input for the invocation is the signature and the line number. This can be a problem in examples such as single-line loops. In such a case, the algorithm may count more iterations than there actually are, though invocations will still be associated with the correct code block. To solve this problem, the sub-algorithm Associated-Statement may use the source code column associated with an invocation statement, rather than the statement's signature if the debugging facilities support it. Also, standard coding conventions will normally eliminate the problem.

Finally, the algorithm assumes that invocations occur in the same order in the trace as they appear in the source code. This will not always be the case with optimizing compilers, which may reorder invocations in order to speed up processing. Compiler optimizations should be turned off to avoid this problem. Conditional blocks may also cause invocations to occur in an order unexpected by the algorithm, which may cause inconsistent results.

Figure 6.3 gives a simple example. The program in Figure 6.3 contains a single loop that increments the variable `i` 100 times. On each iteration, if `i` is even, then the method `SampleSupport.a()` is called; otherwise `SampleSupport.b()` is called. The diagram does not show the expected output. It indicates that both methods `a()` and `b()` are called within the same loop iteration even though such a situation is impossible. That it is impossible can be seen in the diagram itself, which shows that both the `if` block *and* its associated `else` block are being executed. This would require a condition to be both true and false at the same time.

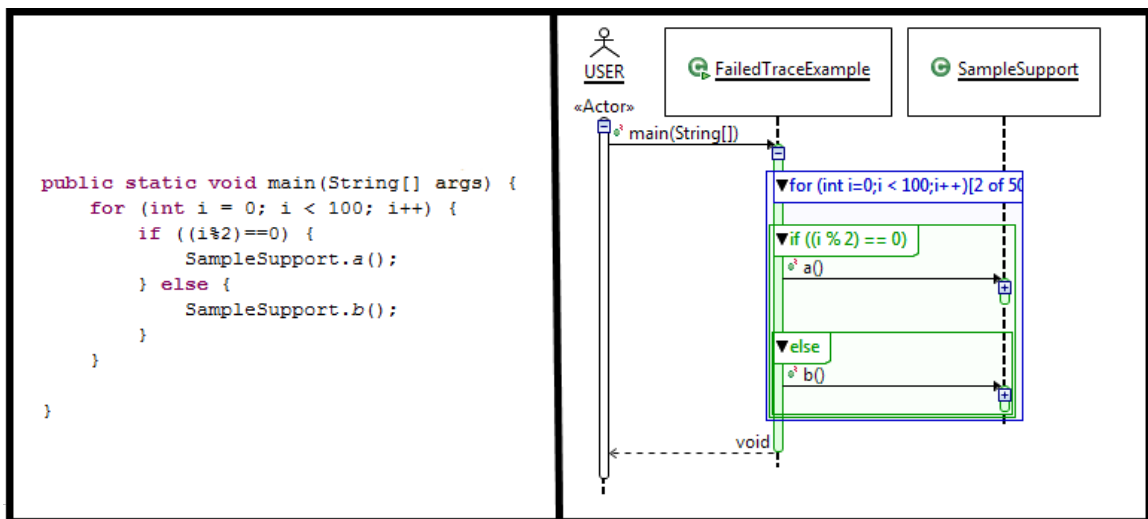


Figure 6.3: An example of a program for which the compaction algorithm gives inconsistent results

The reason that this error occurs is that the particular condition `(i % 2) == 0` (i.e. `i` divides evenly by 2) causes the loop to skip the invocation of `a()` on every odd iteration, so that only `b()` is called. Since `b()` comes syntactically later than `a()`, the algorithm does not increase its iteration count until the next call of `a()`, which accounts for the mistake. This kind of error requires a very specific kind of control flow that causes contiguous loop iterations to appear as a single iteration. The same error would also occur in the compression and compaction techniques discussed in Section 5.1.

There may be other conditions that cause the algorithm to create wrong associations

between execution traces and source code. It would be interesting to find out how often these conditions cause a problem. We hypothesize that most source code will not cause any problems, and our experience indicates that such problems are rare, but we do not have empirical evidence to support this claim. Nonetheless, the approach does represent a significant step forward in compaction for sequence diagram visualizations. Even if the semantic information is sometimes misleading, previous approaches do not allow such information to be displayed in sequence diagrams at all.

## 6.4 Extensions

The algorithm presented is not limited to detecting loops. As discussed in Section 5.3 and seen in Figure 6.3, it can also be used to detect other block statements such as conditional or error handling blocks. In these cases, the blocks are treated like loops that execute only once. They can then be presented in a user interface to enrich the visualizations with more semantic information related to the source code.

## CHAPTER 7

---

# Experiment: Measuring Sequence Diagram Compaction Using Loops

---

Chapter 6 discusses an algorithm for using source code to compact sequence diagram visualizations of execution traces. The goal of this algorithm is to locate and compact portions of execution traces that are repetitious due to loops. It is impossible to formally prove the effectiveness of the algorithm because the amount that the algorithm can compact a trace visualization depends on the source code and traces that are given as input. Source code varies widely between projects. Therefore, we chose to perform an experiment that demonstrates the level of compaction that this approach can achieve. This chapter describes the experimental design and results.

## 7.1 Experimental Design

We used the Diver implementation of the algorithm in Chapter 6 to evaluate its ability to compact execution traces. The Diver implementation of the algorithm works on an AST

for the Java programming language. It associates method calls and constructor invocations with their containing loops. All Java version 1.6 loop types are recognized, including the `for`, `while`, `do...while`, and the extended `for` (otherwise known as the `foreach` loop).

We tested Diver's implementation of the algorithm against three industrial Open Source Java applications and utilities: the Eclipse IDE 3.5R1 [83], the HSQLDB database engine 1.8.1 [28], and the Jetty web server platform 6.1.21 [84]. These three applications were chosen because they are high-quality software products that are popular within the software industry, and their source code is readily available. They also support a wide variety of different computer usage scenarios.

Diver was used to generate traces for each of the programs. Since each program is designed to supply very different functionality, a use case was chosen for each.

**Eclipse.** Eclipse was chosen because it is an interactive graphical application that supplies functionality for many typical computer usage scenarios (graphical interaction, document editing, file manipulation, etc.). The use case for this experiment was that of opening a project in the Eclipse workspace. This use case was chosen because it is a simple operation that is typical of normal Eclipse usage, and it exercises many portions of Eclipse. Diver allows users to pause and restart traces at any time. In order to trace information related to opening the project, a breakpoint was set within the `open` method of the `Project` class of the Eclipse Resources plug-in. When the breakpoint was hit, the Diver tracer was started, and Eclipse was allowed to run until the user interface indicated that the project had been opened, at which point the Diver trace was paused and Eclipse was closed down. Note that Diver captures data from all threads so long as the trace is running. Therefore, any operation that occurred while the project was being opened was recorded regardless of whether it was directly related to the opening of the project. This is not considered a defect in the experiment, as it is part of the normal operation of Eclipse.

**HSQLDB.** HSQLDB supplies a good use case because it exercises low-level data and disk manipulation. The use case we chose was an SQL select. To gather data for HSQLDB,

the database was configured to run in embedded mode within the Java virtual machine. A program was created that uses the Java Database Connectivity drivers for HSQLDB to perform a simple select on a pre-populated table. After the selection, the program iterates through all of the results of the query and caches them into a local data structure. This program exercises the functionality that is typically used in database systems by loading tables, retrieving data, and storing that data for manipulation in the program. Once again, Diver was used for the trace, and was paused until a breakpoint was hit immediately before the execution of the query. The trace was then started, and the program was allowed to run until all of the results had been cached, at which point the trace was paused and the program was exited.

**Jetty.** Jetty was chosen because it represents another very common instance of computer usage. It is a multi-threaded HTTP server that hosts web pages for communication over network connections. The Jetty use case included the start-up of a server and a simple page request. Diver was configured to trace from the beginning of the Jetty program in order to record the server start-up process. One page request was then made for the Jetty default test page using a web browser on the local host. Once the page was loaded into the browser, Jetty was shut down and the trace ended. This usage was chosen to exercise the common functionality of the software as it loaded and processed communication over the network.

For all three cases, Diver was configured to record all method and constructor invocations, and store them in a local database. For the purpose of this discussion, we call the set of all recorded invocations in a use case **I**. For each case, 400 method or constructor invocations were selected using a random procedure. We call this set **R**. Invocations were only considered if they, in turn, caused at least 15 sub-invocations. This was done in order to rule out parts of the call tree that would not cause significant stress on a visualization whether compacted or not. Fifteen method invocations can be easily drawn on a single modern screen as activation boxes in the Diver Sequence Diagram View. Each of the invocations were tested to discover how many method invocations would need to be displayed

if not compacted, and how many would need to be displayed if compacted. The following definitions were used.

**Definition 1** For any  $i \in \mathbf{I}$ , the set  $\mathbf{S}(i) = \{s \in \mathbf{I} \mid s \text{ is invoked by } i\}$  is called the sub-invocations of  $i$ .

**Definition 2** For any  $i \in \mathbf{I}$ , the set  $\mathbf{C}(i) = \{c \in \mathbf{S}(i) \mid c \text{ is (1) not contained in any loop or (2) contained in only the first iteration of any loop}\}$  is called the compaction of  $i$ .

What this means is that when compacting a set of sub-invocations of  $i$ , we make sure that the selected sub-invocations are reachable within the first iteration of any nested loops. In visualization terms, this means, “only display the first iteration of any loop.” This is the default view presented to a Diver user. For example, given an invocation  $i$  of the pseudo code in Listing 7.1,  $\mathbf{S}(i) = \{a_{(1)}, b_{(1,1)}, b_{(1,2)}, c_{(1,2)}, a_{(2)}, b_{(2,1)}, b_{(2,2)}, c_{(2,2)}\}$  and  $\mathbf{C}(i) = \{a_{(1)}, b_{(1,1)}\}$  (given in order of invocation). Note that no invocation of  $c$  occurs in  $\mathbf{C}(i)$  because it never appears in the first iteration of the nested loop.

---

**Algorithm** Loops Example

---

```

for  $l \leftarrow 1$  to 2 do
  invoke  $a_{(l)}$ 
  for  $m \leftarrow 1$  to 2 do
    invoke  $b_{(l,m)}$ 
    if  $m = 2$  then
      invoke  $c_{(l,m)}$ 
    end if
  end for
end for

```

---

Listing 7.1: An example of nested loops

These rules were used to ensure that every invocation was reachable through a simple series of iterations. Slightly different results would have arisen for different iterations of the loops. However, loops are repetitive by nature, so the results would not be expected to be significantly different.



## 7.2 Results

After the data was collected for the three use cases, we took several measurements to help us understand the level of compaction that this algorithm enables. As stated earlier, a random subset,  $\mathbf{R}$ , of 400 invocations was taken from the set of all recorded invocations. The data collected is summarized in Table 7.1. The measurements we took are defined as follows.

Table 7.1: The results of the three experimental use cases

Case	$ \mathbf{I} $	$\sum  \mathbf{S}(i) $	$\sum  \mathbf{C}(i) $	$Compact(\mathbf{R})$	$\overline{Compact}(\mathbf{R})$	$ \mathbf{L} $
<b>Eclipse</b>	2,214,478	15,826	2,948	0.186	0.218	399
<b>Jetty</b>	2,304,848	31,398	4,673	0.149	0.357	318
<b>HSQldb</b>	2,333,381	6,824	6,824	1.0	1.0	0

- **Total Invocations**  $|\mathbf{I}|$ : the total number of calls made during the experiment.
- **Total Sub-Invocations**  $\sum |\mathbf{S}(i)|$ : the total number of sub-invocations measured, defined as  $\sum_{i \in \mathbf{R}} |\mathbf{S}(i)|$ .
- **Total Compaction**  $\sum |\mathbf{C}(i)|$ : the total number of invocations in all sets  $\mathbf{C}(i)$ , defined as  $\sum_{i \in \mathbf{R}} |\mathbf{C}(i)|$ .
- **Overall Compaction Ratio**  $Compact(\mathbf{R})$ : the ratio of compacted invocations to un-compacted invocations over all  $i \in \mathbf{R}$ . Simply the ratio between the previous two measurements.
- **Average Compaction Ratio**  $\overline{Compact}(\mathbf{R})$ : the average of the sum of the compaction ratios for each  $i \in \mathbf{R}$ , defined as  $\frac{\sum_{i \in \mathbf{R}} |\mathbf{C}(i)| / |\mathbf{S}(i)|}{|\mathbf{R}|}$ . This is the amount that the sub-invocations for a single invocation are compacted, on average.
- **Number of Loops**  $|\mathbf{L}|$ : the total number of invocations in  $\mathbf{R}$  that contain loops. In other words the set  $\mathbf{L} = \{i \in \mathbf{R} \mid i \text{ contains at least one loop}\}$ .

As can be seen in Table 7.1, each of the traces recorded more than 2,000,000 invocations. Of the recorded invocations, 400 (approximately 0.02%) were sampled for the experiment in each case. At the time of the experiment, Diver was only capable of storing approximately 2,000,000 invocations in its database due to space restrictions imposed by the software. This has been changed in recent versions of Diver.

Both Jetty and Eclipse yielded similar results. To ensure that the samples were representative of the population, we performed a normal probability plot over the average compaction ratio of the sampled data and found that they both followed a normal distribution (figure 7.1). We can say with confidence that our samples are representative. In terms of overall compaction, Eclipse had a compaction ratio of 0.186 and Jetty had a compaction ratio of 0.149. This means that the compacted sample in the Jetty trace is 6.7 times smaller than the un-compacted one, in terms of sub-invocations. Similarly, the compacted sample in the Eclipse trace is 5.38 times smaller. In other words, if we considered a visualization that displayed all sub invocations of the 400 invocations sampled in Jetty, 85.1% need not be displayed due to looping.

The average compaction ratio indicates how much an individual invocation is compacted on average. Eclipse's average compaction ratio is similar to its overall compaction ratio (0.218 versus 0.186). This indicates that the number and size of the loops in this trace of Eclipse are relatively well distributed and equal. In fact, there was only one invocation that could not be compacted due to a lack of loops in the code. Contrasting this result to Jetty, we see that the difference between the overall compaction ratio and the average compaction ratio is quite large (.149 versus .357). This indicates that there are several "dominator" invocations that allow for a high compaction. These dominators are invocations that contain small loops that iterate many times. The average case, though, is a larger invocation that contains loops without many iterations, or no loops at all. It can be seen that 82 of the 400 invocations (20.5%) did not contain any loops. Nonetheless, the average invocation is compacted by 74.3%.

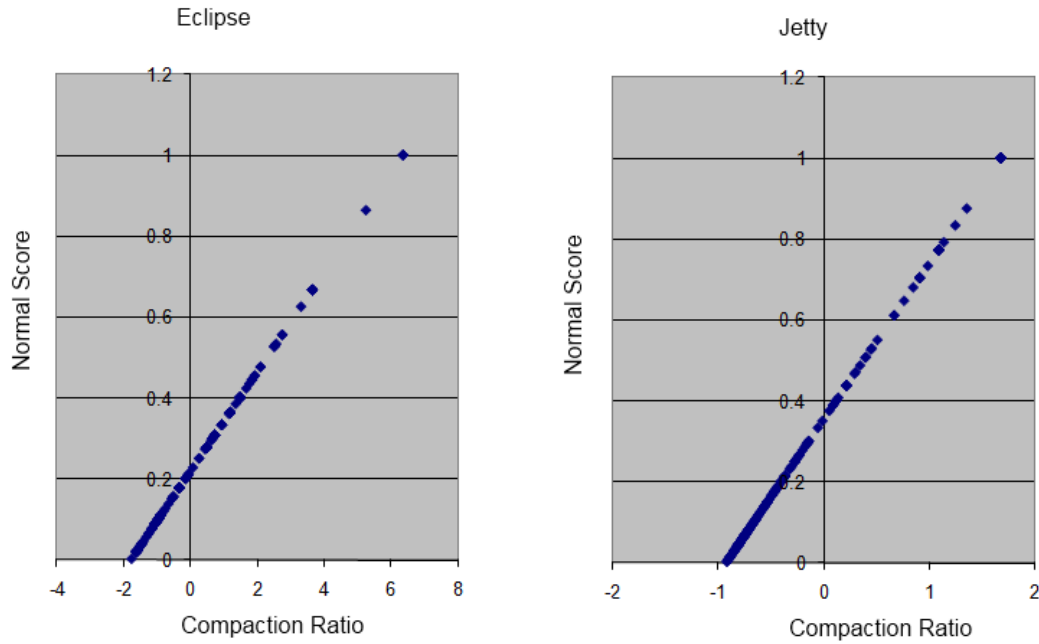


Figure 7.1: Normal probability plots for Eclipse and Jetty

HSQLDB yielded very different results than Eclipse and Jetty. Of the 400 sampled invocations, none of them contained loops, so no compaction was possible. By inspecting the sum of all sub-invocations  $\sum |S(i)|$ , we can gain more insight into why this is. For HSQLDB,  $\sum |S(i)|$  is only 6,824, meaning that the average invocation in the sample set causes only 17.06 sub-invocations. This indicates that most of the calls in HSQLDB are low-level calls that are likely uninteresting. The trace for HSQLDB was further investigated to see what was the cause of these low-level calls. We found that 387 of the 400 calls (96.8%) were in fact different invocations of the same method. The probability of 387 of the same invocations occurring in a sampling of a random set of more than 2,000,000 invocations is nearly 0. Therefore, the invocations that do not contain loops must dominate the few invocations that do. While there may be several invocations that could be compacted to a high degree, they are very unlikely to be selected using a random process. Table 7.1 is representative of the HSQLDB system.

These statistical arguments demonstrate that in spite of the fact that Diver limited the number of recorded invocations to approximately 2,000,000 and we only sampled 400 of those invocations, our samples are still representative of the software that we traced.

## 7.3 Time Analysis

The run-time of the Group-Invocations algorithm used to compact execution traces is also an important consideration. The compaction ratios may be high, but if the algorithm takes too long to run, users will have wait too long for results and the application will have failed as an efficient aid to their reverse engineering tasks.

Theoretical analysis is not a practical approach. The algorithm delegates the work of parsing files and searching for matching code elements (such as the method definition for an invocation) to different processes. In fact, the Diver implementation delegates these tasks to the Eclipse Java Development Tools. These tasks are not trivial, but a time-analysis is not available for them. Therefore, it is very difficult to mathematically prove the run-time of the algorithm.

Instead, we opted for an experimental approach to evaluate the efficiency of the algorithm. We re-ran the same use cases as before except that we measured the average amount of time it took for the algorithm to run. This measure included the times required to parse the file into an abstract syntax tree, to associate invocations with corresponding elements in the abstract syntax tree, and to detect the loops. This experiment was performed on a machine with a 2.21 Ghz Dual Core processor and 2GB of RAM. The results are listed in Table 7.2, rounded to the nearest millisecond.

Table 7.2: The average algorithm run-time

	<b>Eclipse</b>	<b>Jetty</b>	<b>HSQLDB</b>	<b>Mean</b>
<b>Time (ms)</b>	112	167	33	104

The worst performer was Jetty at 167ms per run of the algorithm, which is approximately six executions per second. It is slow enough that it is not instantaneous, but it is

more than fast enough for an interactive application [60]. The speed could be improved significantly by caching the abstract syntax trees and the associations with invocations. However, such a move would also put significant stress on system memory resources when considering large traces. The speed of the algorithm is fast enough as-is, and the memory trade-off is likely not worth the extra milliseconds for most applications.

## 7.4 Threats To Validity

The largest threats to the validity of the experiment are the small sample sizes, both in terms of the range of software tested and the small sample taken from each of the traces.

In regard to the range of the software tested, we note that all three applications are heavily used and of industrial quality. The three applications are representative of a large range of common computing tasks.

To address the small samples taken from the traces (less than 0.02% of each trace), we performed some statistical analysis. It was found that both the Eclipse and Jetty measurements followed a normal distribution, so the sample size does not depend on the population size. The HSQLDB example was not a normal distribution, but the probability of the sampled invocations not being representative of the system was shown to be very low. It is not likely that more insight would have been gained from a larger sample size.

One final threat is that the algorithm was not correctly implemented. This threat has been mitigated through months of testing and inspection, though bugs may still be present.

## 7.5 Conclusions of the Experiment

The results of this experiment confirm that the Pareto principle is applicable to execution traces; 80% of the work is done in approximately 20% of the code. Much of the work done is repeated in loops, and more than 80% compaction could be achieved by removing loop iterations from view. The approach described in the preceding chapters is also efficient.

Despite having to perform computation on a number of different inputs, our algorithm is quick enough to be applied to interactive applications.

It may be noted that this compaction level, though high, is still smaller than what can be achieved by a number of the compression algorithms presented in Section 5.1.1. Flajolet *et al.*'s compression scheme, for example, was able to compress traces by more than 90%. However, the sequence diagram can be compacted in ways other than using loops. The compression schemes given previously also account for recursion. As discussed in Section 5.3, recursion can be handled in interactive sequence diagrams by collapsing activation boxes so that sub-invocations are not displayed until users choose to expand them. It should also be noted that though the compression schemes claim to be lossless, many of them are not. In order for repeated portions of traces to be considered redundant, some information, such as timing or memory usage, must be ignored. Since our approach does not compress any information, none of this data is lost; it is hidden from view, but it can be accessed by adjusting the view.

The approach described in these chapters not only makes it possible to compact sequence diagrams to a high degree by hiding iterations of loops found in source code. It also makes it possible to carry some semantic information from the source code into the sequence diagram in the form of combined fragments, labelled according to code blocks. Our discussion in Section 5.3 indicated that according to Bennett's cognitive design framework, these were appropriate applications of the cognitive support features that we implemented in our sequence diagram visualization. This experiment only tested the approach for its compaction levels, and not its cognitive support capabilities. In Chapter 10, we present a small survey that gives some initial indication that the loop compaction strategy is helpful to users.

This concludes our discussion concerning research question RQ1: *How can the size of sequence diagrams be reduced, given that their size hinders users' understanding?* In Part IV, we will investigate the problem of how to support navigation in large sequence diagrams.

## **Part IV**

# **Navigating Sequence Diagrams**

## CHAPTER 8

---

# Focusing on Traces: Using Software Reconnaissance as a Degree-of-Interest Model for IDEs

---

An important way to aid users in their exploration of large sequence diagrams is to support navigation. This allows them to move through the visualization and locate the items of interest. Navigation is the topic of our research question RQ2: *How can navigation in sequence diagrams be supported?* Our previous study using the Oasis Sequence Explorer showed that users often move back-and-forth between source code and sequence diagrams, indicating a conceptual mapping between the two representations of software [8]. This chapter will discuss some ways that we may be able to leverage this mapping in order to support navigation in sequence diagrams.

Section 8.1 discusses degree-of-interest models (DOIs). DOIs have been previously applied to the problem of navigating through large amounts of source code, but we will attempt to apply them to the problem of navigating large sequence diagrams. Section 8.2 gives a background on the dynamic analysis technique called Software Reconnaissance and builds a degree-of-interest model using it. Finally, section 8.3 shows how a DOI based on



software reconnaissance can support navigation in sequence diagrams.

## 8.1 Degree-of-Interest Models and the Task-Focused User Interface

Modern integrated development environments (IDEs) supply tools that are designed to aid users in navigating large code bases. Some of these tools are also used in sequence diagrams. Lexical search is a common example. IDEs give users the ability to search code for matches to string expressions, but unfortunately, the results are often either too large or too small to be useful [77]. Conversely, IDEs also supply filters that allow users to hide resources that match string expressions or other criteria, such as file type. Many sequence diagram tools support this kind of filtering for traces as well, but our investigation with the Oasis Sequence Explorer tool indicated that it is underutilized because users were unsure whether too much would be removed from the diagram [8].

In spite of the support offered by IDEs, developers still struggle with cluttered workspaces. Murphy *et al.* suggested that advanced filters may be used to reduce clutter and orient them toward the resources that are relevant to their current work [62]. Kersten *et al.* developed this idea into what is now called the *task-focused user interface*, which is implemented in the popular Eclipse plug-in called Mylyn [41]. The technique has proved effective in aiding developers in their day-to-day tasks [42]. However, it has not yet been applied to sequence diagrams. In this section, we give an overview of the task-focused user interface. In Section 8.2, we develop it into a technique for navigating large sequence diagrams.

To decide what elements should be filtered from view, Kersten was inspired by Snowden *et al.*'s [76] extension of Tulving's [92] model of semantic and episodic memory. In this model, *semantic* memory is memory of the relationships between things that allow people to make logical inferences about their meaning. *Episodic* memory is memory of temporal episodes – events and actions that are experienced by a person over time. Seman-

tic and episodic memory work together to allow individuals to understand their perceptions and experiences and to come to conclusions about what they see and experience.

One of the problems that people have with navigating large information spaces is that episodic memory is short. As people work on tasks involving a large amount of information, it is difficult for them to retain actions and experiences in their episodic memory. This, in turn, makes it difficult for them to ascertain the meaning of all of the information they are presented with and how it relates to the task at hand.

Kersten's insight is that computers are able to retain episodic information very effectively. His approach is to first ask the user of the system to define and name the task that he or she is currently performing. Then, he makes the computer monitor and record the user's interactions with the IDE and create a kind of episodic memory of those interactions that decays slower than a human's normal memory. The stored actions include things like resource selections, and file viewing and editing. This stored information is associated with the defined task and constitutes what is called a *task context*.

Kersten further extends the task context to create what is called a *degree-of-interest* (DOI) model that also considers when and how often resources are used. It is assumed that resources that have been used more recently and more frequently are also more interesting to the user. The DOI model is then used as a filter for the IDE. Anything that is not present in the task context (considered uninteresting) is filtered from view. Those things that are considered the most interesting in the model are highlighted. This technique aids users in their navigation by helping them to keep a memory about what has been interesting in the past in order to help them make inferences about the most important resources pertaining to the task at hand. The effectiveness of this approach is demonstrated by its high acceptance among the Eclipse community. As of this writing, Mylyn is among the top 20 used Eclipse plug-ins according to the Eclipse Marketplace [89]. Thousands of people use this tool every day.

## 8.2 The Trace-Focused User Interface: Software Reconnaissance as a Degree-of-Interest Model

From a cognitive support perspective, Kersten's approach may be effective because it is an example of *ends-means reification* [94]. When solving problems related to a particular task, programmers must make repeated searches through a problem space. The problem space for a particular task is a subset of source code artifacts and other resources. The Mylyn degree-of-interest model is used to help users discover the problem space by adding and removing artifacts from the task context. The artifacts displayed to the user are the *ends* (i.e. those artifacts involved in completing the task), and the interactions available in the filtered views (the Package Explorer, source code editor, etc.) are the means to solve the problem. The filters employed by Mylyn reify (make concrete) the mapping between the ends and the means. According to Bennett, this mapping of ends to means is important to sequence diagram navigation because it provides cues that help users navigate through visualizations, progressing toward a goal [7].

The key to Kersten's approach is the episodic information contained in the task context, which is a record of the behaviour of the user. However, when analysts are tasked with understanding large execution traces, their primary interest is in the behaviour of the software (not their own interactions). Fortunately, the behaviour of the software is precisely what is contained in the execution trace. The question that must be resolved is, "what portions of the trace are the most interesting to the user?" To help answer this question, we will employ a technique called *software reconnaissance*, which maps software features to source code. Features are loosely defined as some observable behaviour that is significant to a software analyst. This emphasis on software behaviour indicates that software reconnaissance may help answer our question.

Although software reconnaissance is focused on source code and not execution traces, it may still help aid navigation in large sequence diagrams. Executions of source code elements are recorded in execution traces, so it is possible to map source code to loca-

tions in execution traces (and therefore locations in a sequence diagram). As mentioned in the introduction to this chapter, our experiments with the Oasis Sequence Explorer indicate that users find this mapping important. It is also important according to Bennett's cognitive design framework because it reduces cognitive overhead by helping users navigate between different mental models [7]. Using software reconnaissance, we can define a degree-of-interest model that exposes software elements related to a particular feature. Ends-means reification can be implemented using source code artifacts as a means for navigating through sequence diagrams. In other words, software elements from source code can be used to help users gain a foothold into sequence diagrams.

We call this combination of task-focused user interfaces and software reconnaissance the *trace-focused user interface* because it is designed to help users focus on the information contained within their execution traces. The remainder of this chapter will be dedicated to developing the trace-focused user interface.

### 8.2.1 Defining Software Reconnaissance

Software reconnaissance was first introduced and formally defined by Wilde and Scully in their seminal paper *Software Reconnaissance: Mapping Program Features to Code* [101]. The technique has since been verified for its usefulness in analyzing small to medium sized software systems [51,99,101]. In software reconnaissance, an analyst first decides on a feature of interest and then runs a set of test cases to trace the behaviour of the software. Some of the test cases will exercise the feature in question and some will not. The trace contains references to all the software elements<sup>1</sup> that were exercised during the test cases. Mappings and set operations are performed to locate software elements unique to the feature in question.

To understand how to apply software reconnaissance to execution traces, we must first introduce some terminology. Execution traces contain method calls and returns. For the

---

<sup>1</sup>The original definition of software reconnaissance uses the term *component*. We avoid this term in order to not confuse it with the similar construct used in Component Oriented software design in favour of the term *element*.

purposes of software reconnaissance, it is normally sufficient to use segments of such a trace. We define a scenario  $s$  as a *contiguous and related segment of an execution trace*  $t$ . An example of a scenario would be all the interactions that occur within a thread within an execution trace.

We now define the sets required for software reconnaissance. First, the set  $\mathbf{T}$  is defined as all execution traces gathered for a target piece of software.  $\mathbf{S}$  is a partitioning of  $\mathbf{T}$  into scenarios. Finally,  $\mathbf{E}$  is the set of all software elements (e.g., classes and methods).

Given those initial sets, we define the following subsets:

$\text{EXERCISES}(s) \subseteq \mathbf{E}$ : the software elements  $\{e_1, e_2, \dots, e_x\} \subseteq \mathbf{E}$  that are *exercised* (i.e. recorded in) a given scenario  $s \in \mathbf{S}$  *exercises*

$\text{EXHIBITS}(f) \subseteq \mathbf{S}$ : the scenarios  $\{s_1, s_2, \dots, s_y\} \subseteq \mathbf{S}$  that *exhibit* the feature  $f \in \mathbf{F}$

We would like to use software reconnaissance to build a degree-of-interest model that can be used to navigate sequence diagrams in a way analogous to Kersten's technique for navigating source code. Wilde and Scully offer several candidate subsets of  $\mathbf{E}$ . The subsets are explained in Listing 8.1.

---

**CELEMS**: The *common elements* exercised in in all scenarios. These may represent utility methods.

**IELEMS**( $f$ ): The *potentially involved* elements for the feature  $f$ . These are the elements in  $\mathbf{E}$  that are exercised by at least one scenario exhibiting  $f$ .

**IIELEMS**( $f$ ): The *indispensably involved* elements for  $f$ , defined by the elements in  $\mathbf{E}$  that are exercised by all scenarios exhibiting  $f$ .

**RELEMS**( $f$ ): The *relevant* elements to  $f$  that are all of the indispensably involved elements, minus all of the common elements.

**UNIQUE**( $f$ ): The elements in  $\mathbf{E}$  that are *unique* to the feature  $f$ .

**SHARED**( $f$ ): The elements that are indispensably involved in  $f$  but are neither unique to  $f$  nor are common across all scenarios (i.e. in CELEMS).

---

Listing 8.1: Wilde and Scully's list of software reconnaissance sets

To formulate a degree-of-interest model, we will require a set that is relatively easy to compute so that it can be used effectively in an interactive application (since navigation is an interaction feature). Our choice must also represent a constrained problem space that helps users gain a foothold into sequence diagrams (ends-means reification). We will analyze the candidate sets to decide which is appropriate.

The set CELEMS may be interesting for some use cases. Locating utility methods could be useful for compaction. However, if an analyst is trying to locate a particular feature, it is unlikely to be in this set since most features run only under particular circumstances, not all scenarios. Also, in practice, finding utility methods this way requires one to run very many scenarios (theoretically, *all* possible scenarios need to be run in order to ensure that this set is complete). Other methods, such as the one employed Hamou-Lhadj *et al.* [34] can be used to discover utility methods with the data from only one trace (see Section 5.1.2). Their technique is likely more efficient.

IIELEMS( $f$ ), RELEMS( $f$ ), and SHARED( $f$ ) all have at least one term in their calculation that is defined over *all* possible scenarios as well. In practical terms this means that we likely need a large number of traces in order to calculate them. In certain situations, this may be done. For example, LeGear *et al.* ran an industry case study in which they were able to access all of a company's test cases and run traces of them [52]. They found that the SHARED( $f$ ) set was useful for locating reusable components of software that may reduce duplicated work during software maintenance. However, for an interactive application, it is not practical to run traces for all test cases.

The set IELEMS( $f$ ) may be useful for locating a feature of interest since it contains elements that are involved in the feature  $f$ . However this set there may also contain elements that are not involved in  $f$ . The other option is UNIQUE( $f$ ), which is the set of elements unique to the feature  $f$ . This is the set that Wilde and Scully suggest be used to gain a “foothold” into the software that implements the feature  $f$ . Several studies have indicated that few traces are needed to locate the software implementing  $f$  using this set [100, 101]. Since this set has already been identified as a good starting point for software exploration,

it will be a good fit for our degree-of-interest model.

### 8.2.2 Creating a DOI Using Software Reconnaissance

Wilde and Scully original defined  $\text{UNIQUE}(f)$  as:  $\text{IELEMS}(f) - \{e \in \mathbf{E} \mid \exists s \in \mathbf{S}, s \notin \text{EXHIBITS}(f) \wedge e \in \text{EXERCISES}(f)\}$ . Note that computing  $s \notin \text{EXHIBITS}(f)$  requires enough scenarios to record all elements not exhibiting  $f$ . This is mathematically complete, but not practical for interactive applications. Wilde and Casey in fact report that few scenarios are necessary in most cases [99]. We will define  $\text{UNIQUE}(f)$  in a way that users can generate it with relatively few scenarios.

Recall that  $\text{EXERCISES}(s)$  are the set of software elements in  $\mathbf{E}$  exercised (executed) by the scenario  $s$ . An execution trace contains all the software elements exercised in its scenarios, so it is possible to populate this set automatically. Since the feature  $f$  is defined by a user who is trying to analyze it, the set  $\text{EXHIBITS}(f)$  must be populated by the user as well. In Section 8.3, we will describe the way that Diver allows users to populate the set.

In order to calculate the set  $\text{UNIQUE}(f)$ , we need the set  $\text{EXHIBITS}(f)$ , and a set of scenarios that do not exhibit  $f$ . This can be done by partitioning the set  $\mathbf{S}$ :  $\neg\text{EXHIBITS}(f) = \mathbf{S} - \text{EXHIBITS}(f)$ . This allows us to define  $\text{UNIQUE}(f)$  as follows:

**Definition 3** For  $f \in \mathbf{F}$

$$\text{UNIQUE}(f) = \bigcup_{s \in \text{EXHIBITS}(f)} \text{EXERCISES}(s) - \bigcup_{u \in \neg\text{EXHIBITS}(f)} \text{EXERCISES}(u)$$

Note that the first term of the equation ( $\bigcup_{s \in \text{EXHIBITS}(f)} \text{EXERCISES}(s)$ ) is approximately equivalent to  $\text{IELEMS}(f)$  and as  $\neg\text{EXHIBITS}(f)$  increases in size, our definition of  $\text{UNIQUE}(f)$  more closely approximates Wilde and Sully's original definition. Using  $\text{UNIQUE}(f)$ , we can now define our degree-of-interest model as a function over software elements and features:

**Definition 4** Function  $DOI : \mathbf{E} \times \mathbf{F} \mapsto [0, 1]$

$$DOI(e, f) = \begin{cases} 1 & \text{if } e \in \text{UNIQUE}(f) \\ 1 & \text{if } \exists c \in \text{children}(e) | DOI(c, f) > 0 \\ 0 & \text{otherwise} \end{cases}$$

Where  $\text{children}(e)$  are the structural (contained) children of the software element  $e$  (eg., a class is a child of a package and a method is a child of a class). This function  $DOI$  may be read as the software element  $e$  is “interesting” if it or one of its structural children is in the set  $\text{UNIQUE}(f)$ .

Classical software reconnaissance produces a flat list of software elements that are related to a feature  $f$  or are in one of the sets listed in Listing 8.1. Our approach is different because it assigns a particular interest value to the software elements. This offers a couple of advantages. First, assigning a specific interest value to software elements enables us to apply the value to source code visualizations that are common to IDEs (resource trees, outline views, etc). We may apply the value to affect filters and label decorations on the views in much the same way that Mylyn does. Second, we can extend the degree-of-interest model from a simple 0/1 exclusion/inclusion function to one that represents a more fine-grained and nuanced degree-of-interest. Some possible extensions will be proposed in Section 11.3.2.

For the DOI function to be usable, we need an interactive way for a user to populate the essential  $\text{EXHIBITS}(f)$  and  $\neg\text{EXHIBITS}(f)$  sets. In the following section, we use our Diver tool as an example to show how these sets can be populated by an end user and how software reconnaissance may be used to navigate large sequence diagrams.



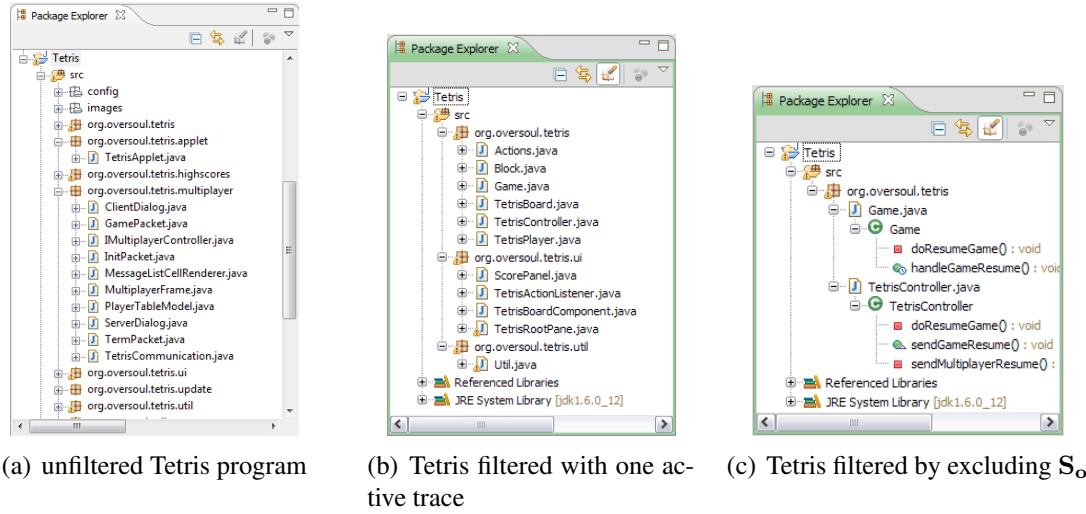


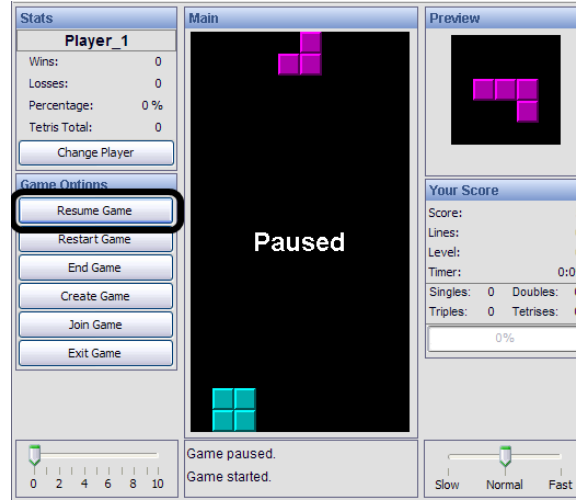
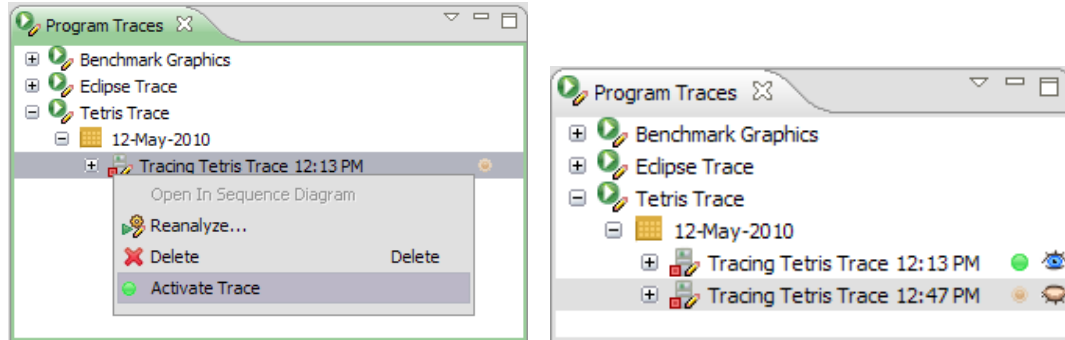
Figure 8.1: Using the Diver filters

## 8.3 Navigation Execution Traces using Software Reconnaissance in Diver

Diver uses the DOI in Definition 4 (Section 8.2) to create filters for Eclipse’s structural Java views (specifically, the Package Explorer). A software element  $e$  is visible in the Package Explorer for a chosen feature  $f$  if and only if  $DOI(e, f) > 0$ . The user is then able to navigate to positions in a sequence diagram by interacting with software elements displayed in the Package Explorer. These are the essential elements of the trace-focused user interface. The sets that are required to calculate the DOI values that affect the user interface are populated through the user’s interactions with his or her recorded traces.

$EXERCISES(s)$  is set through an analysis performed by the Diver tool. A software element  $e$  is said to be exercised by  $s$  if it has been recorded in a trace  $t$  that contains  $s$ . Diver uses Eclipse’s built in Java Search Utilities to reconcile the source code for  $e$  if it is available.

We illustrate Diver’s trace-focused user interface with an example. We apply the techniques in Diver to the Tetris game featured in Figure 8.2. This is a small application consisting of 8 packages and 56 top-level classes. The source code is pictured in Figure 8.1(a).

Figure 8.2: The Tetris game. The *Resume* button is the feature of interest

(a) Activating a trace

(b) “Excluding” a trace using the Diver filters

Figure 8.3: Interacting with traces using the Program Traces View

Suppose that the feature of interest,  $f$ , occurs when the *Resume* button is pressed. The user of Diver can begin a software reconnaissance session by launching a Java Application Trace of the Tetris game and pressing the *Resume* button in the Tetris game. Once the trace is captured, it will be displayed in the Program Traces View, where the user can “activate” it (Figure 8.3(a)). Activating a trace enables the trace-focused user interface.

Naming this activated trace  $t_a$ , we can describe the set of scenarios  $S_a$  in terms of  $t_a$ .  $t_a$  is a list of software interactions, and  $S_a$  contains all of the contiguous and related segments of interactions found in  $t_a$ . Activating  $t_a$  sets  $S = S_a$  and defines the relation  $\text{EXHIBITS}(f)$  such that  $\text{EXHIBITS}(f) = S_a$ . At this point  $t_a$  is the only trace that exists, so

all scenarios exhibit  $f$  ( $\neg\text{EXHIBITS}(f) = \emptyset$ ). Thus, following definition 3,  $\text{UNIQUE}(f) = \bigcup_{s \in S_a} \text{EXERCISES}(s)$ . According to the definition of  $\text{DOI}$ , then,  $\forall s \in S_a, \text{DOI}(e, f) = 1$  if  $e \in \text{EXERCISES}(s)$ . In short, a package, class, or method will pass Diver's filter if it, or one of its structural children, was recorded in  $t_a$ .

This most basic use of software reconnaissance already brings us closer to locating where the Tetris game implements the feature “resume game” (see Figure 8.1(b)). The Diver filters have reduced the number of packages that have to be investigated to three and the number of classes to eleven (an 80% reduction).

Although an 80% reduction in the number of classes to investigate is helpful, the results can be further refined by again applying software reconnaissance techniques. To continue this investigation, the user defines a set  $S_o$  of scenarios that *do not* exhibit  $f$ . This is done by creating a new trace,  $t_o$ , in which the Tetris game will be played using a number of features *excluding*  $f$ . This trace will also now be visible in the Program Traces View, which displays an “eye” icon beside each trace. The user may select and “close” the eye beside the second trace (see Figure 8.3(b)), causing Diver to ignore this trace and hide its software elements.

More formally, we say that “closing the eye” on trace  $t_o$  sets  $S = S_a \cup S_o$ . However, Diver keeps  $\text{EXHIBITS}(f) = S_a$ , meaning that  $\forall s \in S_o, s \notin \text{EXERCISES}(s)$ . By definition,  $\text{UNIQUE}(f)$  includes only those software elements exercised in  $S_o$ , but not in  $S_a$ . So, for all elements  $e$  exercised in  $S_o$ ,  $\text{DOI}(e, f) = 0$  and all such elements are excluded from the Package Explorer by Diver's filter. In our example, the filters have reduced the number of packages to one, the number of classes to two, and the number of methods to six (Figure 8.1(c)). It is possible to execute more traces that do not exercise  $f$  and “hide” them as well. Doing so can increase the number of scenarios in  $S$  and increase the size of  $\neg\text{EXHIBITS}(f)$ , which may further decrease the size of  $\text{UNIQUE}(f)$ .

Diver makes it possible to apply software reconnaissance at an even finer granularity. The Program Traces View displays all of the threads executed during a trace. Suppose an active trace  $t$  has  $n$  threads that can be represented by the scenarios  $\{s_1, s_2, \dots, s_n\} = S_t$ .

If a user selects a thread in the program traces view that represents the scenario  $s_i$ , then Diver sets  $S = \{s_i\} \cup S_o$  and  $\text{EXHIBITS}(f) = \{s_i\}$ . It sets the filters to display elements that can be found in thread  $i$ , but not in any “excluded” trace.

Diver supports navigation into sequence diagrams using the *multiple linked views* cognitive support feature (see Section 3.1.3). Right-clicking on a class or method displays the *Reveal In* action and presents users with a list of threads where the target element was invoked (Figure 8.4). Selecting a thread opens the sequence diagram visualization of the thread and adjusts it so that the first invocation of the element is visible. Other invocations can be displayed using the timeline as discussed in Chapter 5 (see Figure 5.4).

This navigation between the Package Explorer and the sequence diagram is an example of ends-means reification. The filtered Package Explorer offers a guide for navigating to new locations in the sequence diagram. It also supports navigation between different representations and mental models. The Package Explorer presents a top-down view of software, organized by projects, packages, classes, and methods. The sequence diagram presents a behavioural view. These are all important considerations in Bennett’s cognitive design framework [7].

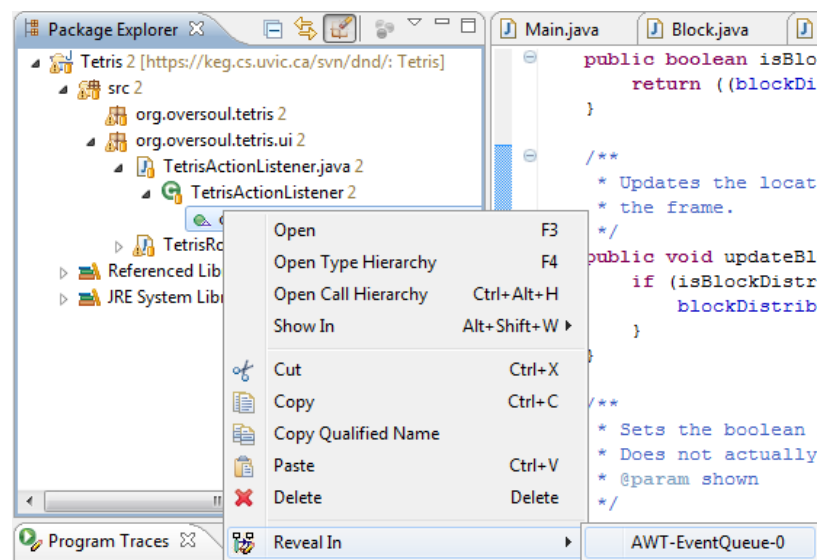


Figure 8.4: The *Reveal In* action for the Sequence Diagram View

This chapter discussed the problem of navigating large software projects, and combined the idea of degree-of-interest models as introduced by Kersten *et al.* [41] with software reconnaissance by Wilde and Scully [101]. Section 8.3 gave an example implementation of this technique as realized in the Diver tool. In the next chapter, we validate the approach through an experimental simulation involving professional software developers.

## CHAPTER 9

---

### The Trace-Focused User Interface: A User Study

---

Throughout this part of this thesis, we have been investigating research question R2: *How can navigation in sequence diagrams be supported?* Our investigation has led us to design and implement the trace-focused user interface in the Diver tool. This chapter will investigate whether or not this approach is useful to software developers. That is, *does software reconnaissance provide cognitive support for feature location and analysis using sequence diagrams?* We carried out a user study to help answer this question. Section 9.1 describes the methodology and design of the study. Section 9.2 presents the data that we gathered during the study and reports on the findings. Section 9.3 discusses those findings and points out limitations to this study. Finally, Section 9.4 draws concludes the study.

#### 9.1 User Study

In this user study, we were interested in whether or not the trace-focused user interface technique helped users locate and navigate to features of interest. So, we posed the following

research questions to help us to answer question RQ2:

- (RQ2.1) *Does the use of software reconnaissance improve the **efficiency of feature location in Diver?***
- (RQ2.2) *Does the use of software reconnaissance reduce **frustration during feature location tasks?***
- (RQ2.3) *How does the use of software reconnaissance influence **navigation patterns through the various views in Diver?***

These three questions allow us to look for specific measures that we can correlate to the effectiveness of Diver. In this section, we will discuss the methodology of the study and its overall design.

### 9.1.1 Methodology

The research strategy used in this study is *experimental simulation* [59], which retains some of the realism of a field study, while maintaining a degree of control more typical of laboratory experiments. This is done by creating a context, as in a laboratory experiment, but making it similar to a real (in-vivo) behaviour setting as much as possible [59].

The main task in our study is feature location, a common software maintenance task that has been widely studied [20, 46, 75]. Since there may be a large performance variability between subjects, we used a *within subjects* design that required participants to perform two controlled sessions (see Section 9.1.4). We compared the results of the sessions for each participant.

### 9.1.2 Participants and Apparatus

We recruited ten professional software developers from three local software companies to participate in the study. Each participant received a small token of appreciation for participating in the study. Except for one developer, all of the participants had over two years of

development experience using Eclipse. We provided the developers with an instrumented version of Diver and directed them to use its software reconnaissance features for one feature location session, but not for the other. We videotaped both conditions and asked the users to use a think-aloud process [21] during the program understanding tasks assigned to them.

### 9.1.3 Tasks

To ensure uniformity in terms of developer familiarity with the test codebase, we asked the participants to analyze Diver itself, as it was not well-known at the time of the study. Diver is a fairly significant project, and the participants were supplied with source code that included 7 plug-in Eclipse projects, 225 classes, 2,602 methods, and 35,563 lines of code. Furthermore, since Diver was written as a set of plug-ins for the Eclipse IDE, understanding Diver's features also involved some understanding of the Eclipse IDE source code. This resulted in a program large enough to be realistic, but small enough to be used in a time-limited experiment.

We controlled the simulation by asking the participants to perform several tasks that would help us to ascertain whether or not they were successful in their analysis. To ensure that our questions were realistic, we based them on previous literature. We considered several different sources, including Erdos and Sneed [20], Sillito *et al.* [75], and Ko *et al.* [46]. Erdos and Sneed used 40 years of personal experience responding to maintenance requests to abstract 7 questions that developers commonly need to answer [20]. However, the questions in their work assume that some previous analysis has already been performed and that developers are given reference to some software elements pertinent to their software task *before* they begin their maintenance. Also, the work is based on personal experience and lacking scientific rigor. Sillito *et al.* performed more rigorous studies to ascertain the information needs of software developers [75]. In their work, they uncovered 4 categories of questions that are asked by developers, but these categories combined are comprised of 44 questions that are too low-level to be of use in our study.



We decided to base our tasks on insights found by Ko *et al.* [46]. They performed a field study, taking notes of the normal work of 45 different developers over the course of 90 minutes each. From their notes, they abstracted 22 different questions in 7 categories. Of particular interest to our study are the questions in the category “understanding execution behaviour” because they were ranked as both highly important and very difficult. However, these questions were abstracted by observing developers using their baseline set of tools. Since the tools that the developers used offered only static analysis of source code (with the exception of the debugger), the questions are biased toward static analysis. We therefore modified the questions to broaden them as follows:

- (Q1) *What code could have caused this behaviour?*
- (Q2) *What is related to the execution of this code?*
- (Q3) *Under what conditions was this code executed?*

Table 9.1: Program understanding tasks given to the participants for each session

<b>Understanding Task</b>	<b>Purpose</b>
<i>In which thread is the functionality primarily executed?</i>	verify feature location
<i>Please describe the program flow that preceded the execution of the functionality.</i>	answer (Q1)
<i>What are the classes involved in the execution of the functionality?</i>	answer (Q2)
<i>Please describe how the classes/methods interact in order to perform the functionality. Please note concrete implementations (no interfaces).</i>	answer (Q2)
<i>Under which conditions do the interactions occur? (i.e. what are some of the conditions that must be true for the functionality to execute?)</i>	answer (Q3)
<i>Please take a screen-shot of the sequence diagram.</i>	verify answers

Participants were asked to perform the program understanding tasks for two different features. The first feature (**F1**), *Link to Source*, allows users to view the source code asso-

ciated with an element selected in the Sequence Diagram View. The second feature (**F2**), *Exchange Repetitions*, allows users to exchange a visible loop iteration with one that has been hidden to reduce sequence size [63]. We chose these features because they were well-defined and it was easy to explain their location in the Diver user interface. Despite this, both features were relatively difficult to locate and analyze. The source code for **F1** was hidden within several layers of anonymous inner-classes and executed by a complex system of call-backs. **F2** exercised significantly more source code, also making it difficult to locate and analyze its functionality.

#### 9.1.4 Procedure

To help control learning effects, we gave a 30-minute training session on how to use the Diver tool. This involved guiding the participants through a small analysis problem using a Tetris game program that demonstrated all the features of Diver applicable to the study. The participants were shown how to use Diver to find and analyze the functionality involved in rotating a Tetris block during game play. Since we were not measuring their ability to learn the tool, we allowed participants to ask questions related to tool functionality.

Each participant performed two 40-minute sessions during which they were asked to perform the six program understanding tasks (Table 9.1) for the features described above. During the first session (**S1**), participants were asked to use Diver's software reconnaissance functionality, as well as any other Diver functionality they desired. For the second session (**S2**), participants were directed not to use the software reconnaissance capabilities of Diver but they were allowed to use other Diver features. Ordering the sessions in this manner should help ensure that increased tool familiarity would not be a factor in any improvements we saw while studying the use of software reconnaissance. To control for any differences there may be between the features, half of the participants investigated **F1** during session **S1** and the others investigated **F2** during session **S1**. Participants were randomly assigned which feature to investigate first.

We followed each session with a 10-minute interview. Participants were asked to de-

scribe the process they used to solve the tasks in each session and to describe which aspects of the tool helped or frustrated them. We also asked them to rate the usefulness of software reconnaissance using a 5-point Likert scale, where 1 meant that software reconnaissance was not useful at all, and 5 meant that it was extremely useful.

We conducted three pilot studies to refine our study design. The first pilot was conducted with a student programmer and the other two with university research programmers.

### 9.1.5 Data Collection and Analysis

We used a mixed-method methodology, combining the analysis of qualitative and quantitative data in a single study.

To answer the first research question (RQ2.1), for each session, we measured the amount of time from when each participant completed their traces to the time that they located the first software element related to the feature in question. We call this the *time to first foothold* (see Table 9.1). This data was obtained from an analysis of videotaped observations. Each session was observed either *post-hoc* or live by at least two researchers. Although we initially anticipated also gathering the times to complete all the tasks, we noted in the pilots that the participants did not focus on completing the assigned task until close to the end of the allotted time. We noted that the task will grow to fill the time given for it.

To answer the second question (RQ2.2), we counted verbal frustration utterances during the think-aloud protocols for both conditions. We later coded these utterances to pull out categories of frustrations.

To answer our third research question (RQ2.3), we analyzed the logs of the users' interactions with Eclipse to see if the filtered Package Explorer view altered their interaction patterns. If the Package Explorer view was successful in providing ends-means reification, we would expect participants to make greater use of it during software reconnaissance. Without software reconnaissance, we would expect them to try alternate avenues of investigation.

Finally, at least two researchers independently analyzed the recorded data from the interview sessions. These data were analyzed, and themes were independently extracted and agreed upon by the researchers. These additional insights were used to augment the timing data, the frustration utterance data, and the log data of user interaction patterns.

## 9.2 Findings

In this section, we present the findings from our study sessions. We present participant times to finding a first foothold, frustration utterances, and interaction patterns with the IDE views. We augment these findings with additional insights from the observation and interview data. Overviews of each participant's sessions are given in Appendix F.

Unfortunately, two of the ten participants did not use software reconnaissance in **S1**. One of the participants was a novice programmer and did not grasp the concept of software reconnaissance. The other participant only used a searching strategy, despite our instructions to use software reconnaissance. Since our goal was to evaluate the use of software reconnaissance, we can only present the results from the eight participants that used it in **S1** below.

### 9.2.1 Time to First Foothold

Table 9.2 details the amount of time each participant used to identify a relevant software element after they had captured the execution traces required for the sessions. A dashed line indicates that the participant did not find a foothold in the 40-minute session. For six of the eight participants, the data suggests that software reconnaissance improved the efficiency of their search. That is, the time to find a source-code foothold was less in **S1**, where programmers had access to the filtering facilities offered by software reconnaissance.

For the two participants that were not successful in **S1**, we analyzed the videotaped protocol and interview data to understand why. P8 used a wide range of unsuccessful navigation strategies, including looking at each trace in the sequence diagram as he initially

expected that the sequence diagram was filtered by reconnaissance (a misconception made explicit in his talk aloud). When he later used the Package Explorer, he only navigated to the file level. He mentioned that his previous experience with writing software led him to expect that the Package Explorer would be “too large.” Though he knew it was filtered, he still tended to avoid browsing it and never used it to navigate to the sequence diagram. However, in S2, he found the relevant code in just over two minutes; a result he attributed to serendipity and the knowledge he had acquired during S1.

P10 attempted software reconnaissance during **S1**, but was never able to find a foothold for the feature. During the session and interviews, he specifically pointed out “learning effects” as being a factor in his difficulty using the tool. Analysis of his traces and his interactions with the Package Explorer indicate that the traces *not exhibiting* the feature of interest did not contain enough information to effectively narrow the  $\text{UNIQUE}(f)$  set. Consequently, this participant spent much of his time exploring software elements that were only marginally involved with the feature. However, he carefully planned his investigation during **S2** and was able to find a first foothold in under eight minutes.

We noted that both P8 and P10 had difficulty with feature **F2** during **S1**, which revealed that there might be an effect between the feature under investigation and the time it took to locate the first foothold. We discuss this further in Section 9.3.

Table 9.2: Time to first foothold (minutes) and feature investigated for each session

		P1	P2	P3	P5	P6	P8	P9	P10
S1	Feature	F2	F1	F1	F2	F1	F2	F1	F2
	Time	9.0	3.5	7.6	5.6	3.5	–	2.5	–
S2	Feature	F1	F2	F2	F1	F2	F1	F2	F1
	Time	10.6	14	–	10.3	37	2.2	–	7.4

### 9.2.2 Frustration Utterances

To compare frustration differences across both conditions (our second research question), at three coders independently analyzed and coded the frustration utterances such that each

participant was reviewed by at least two researchers. In a preliminary coding stage, three distinct categories of frustration utterances emerged: utterances related to scaling issues with the sequence diagram or Package Explorer, utterances related to failed search attempts, and a general category for unspecified frustrations that included sighs and quotes such as “I think I hit a dead end”. After agreeing on the categories, the three coders revisited the data independently. The final codes used were those agreed upon by at least two coders. When there was a lack of agreement, we reviewed the videotapes and discussed the categorization further until agreement was reached.

Table 9.3 details the number of general, size-based, and search-based frustration utterances made by each participant during each session. This data shows a strong tendency towards increased frustration in **S2**, again suggesting that the path to reach a solution was an easier one in **S1**. The following frustration quotations from **S2** illustrate the difficulty that the majority of programmers perceived in using the IDE to determine a path of actions for feature location in **S2**: “Woah ... ‘em ... I’m at a dead-end here ” (P6), “I can’t really think of a better way” (P3), “...expanding the sequence diagram is *not* going to work” (P2), and “I have no clue ... it’s pretty daunting ... I have no idea ...” (P9).

In comparison, there were very few frustration utterances of any category in **S1**. P8 and P10 were the only participants who made more frustration utterances in **S1** than in **S2**, with the possible reasons for that difference being noted in Section 9.2.1. P5 was largely silent during both of his sessions, despite being continually prompted to think aloud.

Table 9.3: Frustration utterances for each participant per session

		P1	P2	P3	P5	P6	P8	P9	P10
S1	Gen.	0	5	0	0	2	1	9	19
	Search	1	0	0	0	0	1	4	2
	Size	1	0	0	0	5	6	3	10
	<b>Total</b>	<b>2</b>	<b>5</b>	<b>0</b>	<b>0</b>	<b>7</b>	<b>8</b>	<b>16</b>	<b>31</b>
S2	Gen.	0	16	0	0	5	0	18	7
	Search	3	2	5	0	0	0	9	0
	Size	2	4	3	0	9	1	11	2
	<b>Total</b>	<b>5</b>	<b>22</b>	<b>8</b>	<b>0</b>	<b>14</b>	<b>1</b>	<b>38</b>	<b>9</b>

### 9.2.3 User Interaction Patterns

To explore our third research question on how software reconnaissance influenced navigation strategies, we logged all user interactions with the tool in both conditions. Figure 9.1 shows, for each session, the aggregate time spent by participants in each Eclipse IDE view (as illustrated by the black vertical bar to the bottom right-hand side of each node), and the number of transitions between each view (as illustrated by the thickness of the edges between each node). Note that edges are only presented in this figure if more than 10 transitions occurred between the nodes over all of the participants' sessions. Finally, each node also shows a timeline by participant, illustrating when these participants were using that view during their sessions.

An obvious difference between the **S1** and **S2** diagrams lies in the usage of the Package Explorer. In **S1**, the view is central to their navigation, with the majority of transitions from the Trace view leading to the Package Explorer, a pattern consistent with end-means reification as proposed in Section 8.2. From there, participants often went back to the Trace view. Video and think-aloud analysis indicate that this was often to re-target their browsing in terms of the thread they were focusing on. Alternatively, Figure 9.1 shows that participants also went from the Package Explorer to browsing the sequence diagram or source code.

The Package Explorer view was used much less in **S2**. The amount of time spent in the view decreased from 14% of overall task time to only 4%. The only view with greater than 10 transitions to or from the Package Explorer was the source code view. These observations suggest that the Package Explorer filtered through software reconnaissance provided more cognitive support for feature location in **S1** than the unfiltered Package Explorer in **S2**.

Additionally, participants in **S2** performed many more trace searches and spent more time navigating through the sequence diagram trying to find the features. Previous research has suggested that browsing through large sequence diagrams unaided is not optimal [8]. Likewise, lexical (trace) searches have been shown to be prone to the vocabulary prob-

lem [24, 77]. That is, search results do not provide end-means reification if the representation does not provide any cues (the ideal lexicons) for the search. The developers frequently acknowledged this problem: “I’m going to revert to searching... no, I don’t know... I could search for just random stuff...”(P2), “I could search, but what to search for?”(P9). The results of their searching also confirmed this lack of support as they usually failed to find relevant software elements.

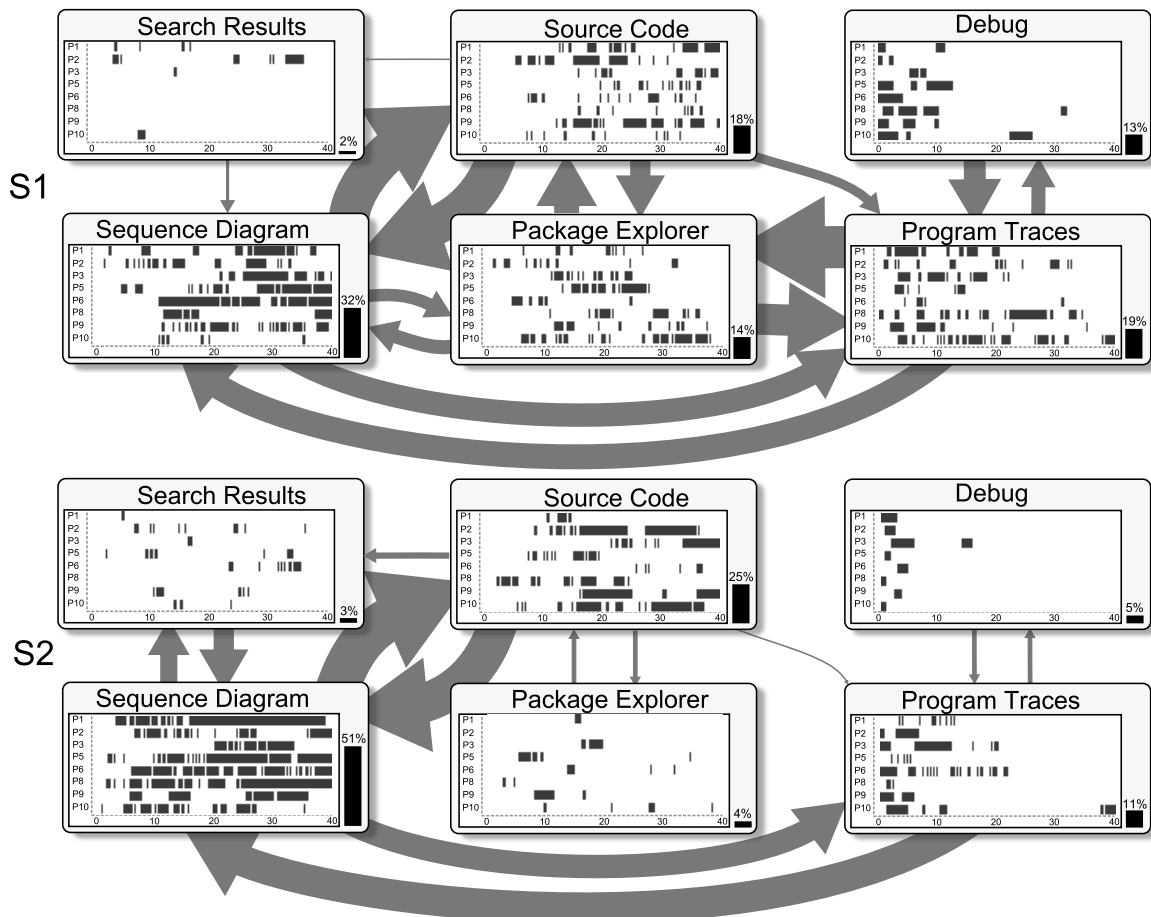


Figure 9.1: The total transitions made between Diver's major views.

### 9.2.4 Interview Data

For six of the eight participants (P1, P2, P3, P5, P6, P9), the interview data suggests a strongly positive attitude towards software reconnaissance in the context of feature loca-



tion. For example, when participants were initially asked to describe their “experience using the tools to carry out the assigned tasks”, five discussed the merits of the software reconnaissance functionality (P1, P3, P5, P6, P9). Likewise, when participants were specifically asked about the tool features they felt most useful, six mentioned reconnaissance. When asked to rank the utility of software reconnaissance out of 5, the participants gave it an average ranking of 4.19.

Several of the participants (P1, P2, P5, and P10) mentioned that learning effects may have hindered their use of software reconnaissance. Despite this, they were positive about the tool and the use of software reconnaissance.

A number of the participants explicitly made reference to the scale of the sequence diagram viewer and the utility of software reconnaissance in alleviating this scale issue. For example, when asked to rank reconnaissance, P3 gave it a 5, adding that the sequence diagram was useless to him without reconnaissance. Likewise, P9 noted that “unless I know the point to look to, the sequence diagram is too daunting... (the) filter helped a lot”, and P5 stated that the “Diff” helped a lot, making it “easy to look at the Package Explorer and get to the sequence diagram”.

## 9.3 Discussion

In this section, we discuss and interpret the findings from the study, and we present the limitations. Because we predominantly collect qualitative data, we use concepts such as dependability, confirmability, and credibility in the limitations, rather than concepts such as internal or external validity that are more frequently used for hypothesis-driven research.

### 9.3.1 Interpretation of Findings

When the participants used software reconnaissance during **S1**, our hypothesis was that a path to software artifacts of interest would be made more apparent through the filtering that software reconnaissance imposed on the Package Explorer view. Hence, for **S1** we ex-

pected to see each of the users run two traces, one with the feature of interest exercised and one without. We then expected them to perform software reconnaissance (by closing the eye icon on the second trace) to identify code that is unique to the exercised feature. This unique code would then be used as a filter in the Package Explorer view. Participants P1, P2, P3, P5, P6, and P9 largely followed this process, turning their attention to the filtered Package Explorer view in **S1** and finding a foothold soon after. This strongly suggests that the filtered Package Explorer provides ends-means reification by directing users toward the goal of locating and analyzing software features.

In **S2**, when the reconnaissance facilities were not available, we anticipated that the representation's (IDE's) ability to direct the programmer to their problem goal would be decreased. For P1, P2, P3, P5, P6, and P9, this seemed to be the case. They tried numerous strategies, such as searching the traces, browsing the source code, searching sections of the sequence diagram, examining the sequence diagram thread by thread, iterating between the sequence diagram and the source code, looking at the breadcrumb trail, and expanding the whole sequence diagram. These strategies are reflected in Figure 9.1, which shows many more searches being performed and much more time being spent in the Sequence Diagram View in **S2**. An interesting finding regarding searches is that some participants tended to search using terms that they had previously found either in source code or in a sequence diagram (see Appendix F). Participants P2, P9, and P10 used this strategy.

The participants' utterances during this session illustrated that these strategies often led to frustration instead of progress towards their navigation goal. This finding is reinforced by the interview data, where the participants expressed that they missed the reconnaissance functionality in **S2**. Indeed P3 and P9 did not solve the feature location problem in the given time in **S2**. While P1, P2, P5, and P6 did find the feature during their **S2** sessions, it took them much longer than it did in **S1**.

The data presented in Tables 9.2 and 9.3 indicate that our results may be correlated to the feature under investigation, as well as to the session. Despite our efforts to specify similarly difficult features, near the end of the study we began to see that feature **F2** seemed

harder to find. However, despite the increased difficulty of **F2**, two of the four participants tasked with identifying **F2** during **S1** still found it more quickly than they did **F1** in **S2** (P1 and P5). This suggests that the tool provided strong support for some programmers, even under more demanding conditions.

Although we only present the data for eight of the ten participants, it is interesting to note that two of the participants did not perform software reconnaissance at all. This may be an indicator that software reconnaissance is a difficult concept for some people, requiring significant training. It is not possible to draw any strong conclusions from our study, however.

Our experimental simulation indicates that software reconnaissance techniques can be usefully applied to reverse engineered sequence diagrams. With the Diver tool, many of the software developers were able to more quickly locate a feature and more effectively analyze it. They also experienced reduced frustration and browsed through fewer elements in the sequence diagram while carrying out their tasks. While Diver did not work well for everyone, the majority of our study participants were able to use the tool to solve realistic feature location problems.

### 9.3.2 User Study Limitations

Designing empirical studies to evaluate program comprehension tools is always a challenging endeavour. There is inevitably a tension between evaluating the tool in an ecologically valid setting and the desire to have significant results from a large number of participants in a controlled setting.

We used an experimental simulation and a within-subjects design to gain rich insights about professional developer performance within a controlled setting. A major limitation of this decision is that the smaller number of subjects makes it difficult to draw strong conclusions from the timing data. However, the results of the study are consistent with the cognitive support theory of ends-means reification, which offsets this limitation. The interview data also further corroborated our findings.

Learning effects may have been an issue in our study. However, given the ordering of the sessions (**S1** incorporated reconnaissance and **S2** did not), any learning effects should only confound the hypothesis that reconnaissance was a useful utility. Despite this, most of the participants ranked it positively and were able to use it effectively. There may be also be a limitation in the dependability of our findings due to bias introduced by the researchers while coding the participants' frustrations. We tried to reduce this problem by having multiple coders for our data.

In terms of confirmability, we need to consider the underlying theory of cognitive support and our assumption that the filtered Package Explorer view provides ends-means reification. However, from our analysis of the think-aloud protocol, as well as statements in the interviews, we are confident that the view did in fact offer this kind of cognitive support. Although we instructed them to run software reconnaissance (by filtering using the unique elements across two traces), we did not instruct them to use the resulting Package Explorer view. Despite this lack of explicit instruction, P8 was the only user that did not navigate to sequence diagrams using the Package Explorer.

In terms of the credibility of our results, we used interviews to bring richer insights to the observations and logging/timing data, as well as to the frustration utterances we observed. However, we recognize that our time of engagement with the users was very short. A future study should consider how the software reconnaissance technique is used to provide cognitive support for the navigation of sequence diagrams over a longer period of use. As Diver is Open Source, we would like to collect longitudinal data from willing users.

## 9.4 User Study Conclusions

Our experimental simulation indicates that software reconnaissance techniques can be usefully applied to reverse engineered sequence diagrams. With the Diver tool, many of the software developers were able to more quickly locate more effectively analyze a feature

when using software reconnaissance and the trace-focused user interface. They also experienced reduced frustration and browsed through fewer elements in the sequence diagram while carrying out their tasks. While Diver did not work well for everyone, the majority of our study participants were able to use the tool to solve realistic feature location problems. Diver's trace-focused user interface is designed to use Bennett's *multiple linked views* cognitive support feature [7] to provide a form of cognitive support called ends-means reification [94]. The results of this study are consistent with the expected results of ends-means reification, indicating that the trace-focused user interface does offer cognitive support.

This concludes our primary investigation of research question RQ2: *How can navigation in sequence diagrams be supported?* In Part V we will verify the results of this investigation and the investigation described in Part III with a small survey that was conducted concerning the features of Diver, and, finally, we will conclude this thesis in Chapter 11.

## **Part V**

# **Synthesis**

## CHAPTER 10

---

### Dynamic Interactive Views for Reverse Engineering: User Survey

---

In Part III, we developed an algorithm that compacts loops in execution traces using source code. The loops are displayed one iteration at a time in our sequence diagram visualization as combined fragments. We extended the algorithm to detect conditional and error handling blocks as well. We designed the algorithm using Bennett's cognitive design elements and worked under the assumption that displaying less information would be useful to users because our previous experiment indicated that users find large sequence diagrams difficult to understand [8]. The technique was not validated using human subjects, however.

We did use human subjects to help validate the navigation technique discussed in Part IV. While the findings were interesting, we were not able to gain statistical significance with our limited number of subjects.

However, we were able to gain further validation of both approaches because they were implemented within the same Diver tool. We released the tool to the public, and solicited users to respond to a survey to help us understand how helpful they found each feature.

This chapter describes that survey. Section 10.1 describes the survey design, Section 10.1.1 reports on the results of the survey, Section 10.2 discusses our findings, and Section 10.3 responds to possible threats to validity.

## 10.1 Survey Design

We collected data in the survey over an 11 month period between January and November of 2010. Participants were solicited through a weblog that is published on the Planet Eclipse<sup>1</sup> syndicated news feed. After one week, users of Diver would be prompted to participate in our on-line survey. The user could choose to participate, or to postpone participation in which case he or she would be prompted again in a week's time.

The survey asked a number of questions regarding the participants' software development experience and presented them with a comprehensive list of Diver features to rate. The respondents were asked to rank the features on a 5-point Likert scale, with 1 meaning the feature in question was not useful, and 5 meaning that it was very useful. Finally, the participants were able to enter free-form text to express any further impressions of the tool.

The survey had a total of 27 questions, but only nine of them were directly related to the questions in this thesis. Five of the questions related to the loop compaction algorithm, and four related to navigating sequence diagrams. These are the features most related to the research questions posed in this thesis, and they are summarized in Listing 10.1. The complete survey can be found in Appendix G.

### 10.1.1 Survey Results

A total of 37 people responded to the survey. Respondents weren't required to fill in the entire survey, however. Many stopped filling in the survey after completing only the initial questions. Ten respondents (R1-R10), however, ranked all of the features listed in Listing 10.1 and their responses may be used to help answer the questions posed in this thesis.

---

<sup>1</sup><http://planetecclipse.org/planet>



---

Surveyed Features
<b>SL1</b> Displaying loops in combined fragments
<b>SL2</b> Displaying conditional blocks in combined fragments
<b>SL3</b> Displaying error handling in combined fragments
<b>SL4</b> Compacting loops into single iterations
<b>SL5</b> Swapping loop iterations

---

<b>SN1</b> Organizing traces in the Program Traces View
<b>SN2</b> Filtering the Package Explorer based on the active trace
<b>SN3</b> Excluding previous traces from the current filter
<b>SN4</b> Revealing activations in the sequence diagram using the Package Explorer

Listing 10.1: The list of Diver features ranked in our survey

Most of the respondents (7 out of 10) were software developers with 10 or more years of experience. Half used Java as their primary programming language. The exceptions were R1 (C#), R6 (C/C++), R7 (Python), and R8 and R10 (JavaScript).

Figures 10.1 and 10.2 show the results of the survey. Figures 10.1(a) and 10.2(a) display the number of times each score (from 1 to 5) was awarded to a feature and is coloured by the respondents. For example, R1 scored every feature with a value of 5 whereas R8 scored every feature with a value of 1. Figure 10.1(b) and 10.2(b) display the mean score for each feature over all participants. Generally, the features ranked between 3 and 4 points on average with an overall mean score of 3.6 in each group of features.

In addition to the Likert scales, respondents could enter free-form text about their impressions of the tool. Generally, the responses were positive. Some examples of positive responses were, “Extremely interesting & innovative tool”, “Nice and impressive work”, and, “Great potential.” Respondent R1 filled out the survey on behalf of a professional organization and stated, “We had used commercial tools prior to using Diver. However

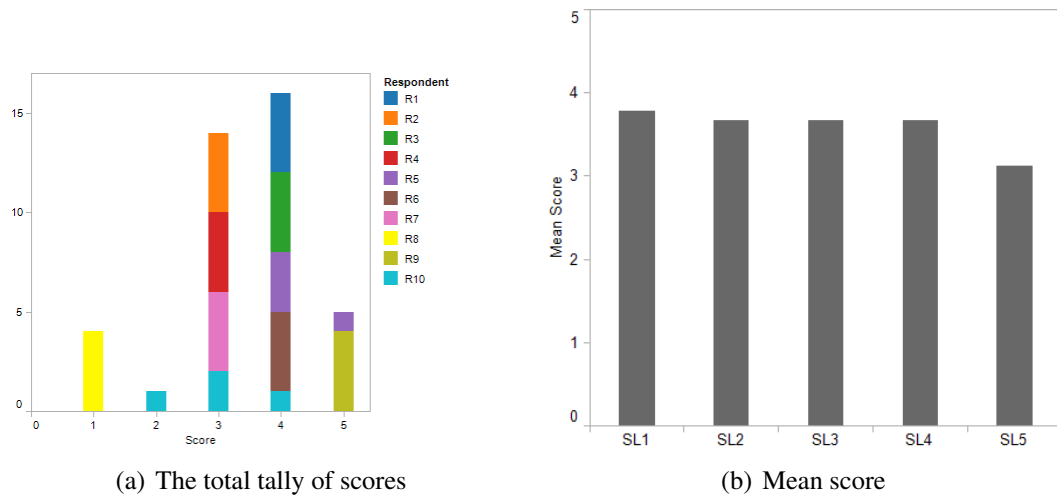


Figure 10.1: Results related to loop compaction

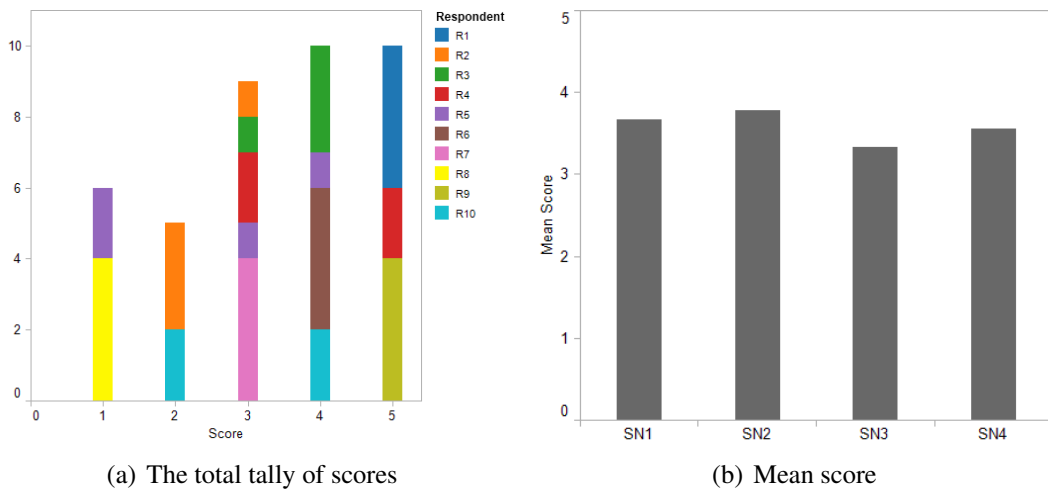


Figure 10.2: Results related to navigation

the level of interactivity provided by Diver was better than most in allowing for a quick navigation between the classes of interest. The ‘Reveal in’ feature helped us narrow down areas of interest very quickly!”

There were some negative comments as well. Most were with respect to the effects of the time overhead introduced by Diver on the application being traced and on the amount of resources that were being consumed by Diver. Some examples are, “The trace capture grealy [sic.] affects program execution speed. Will not be effective in identifying race

conditions”, and “Speed & memory consumption of Diver remain an issue.” Some optimizations have been made to Diver in order to address these issues.

## 10.2 Survey Discussion

The results of this survey are consistent with the user study findings in Chapter 9 and the prediction that Bennett’s cognitive support design elements aid in program comprehension. Most of the responses were positive, indicating that the two approaches described in Parts III and IV are helpful. One interesting result is that the free-form responses to the survey are consistent with the interviews reported on in Chapter 9. Particularly, respondent R1 proactively commented that the *Reveal In* feature of Diver that links the Package Explorer to the sequence diagram when the trace-focused user interface is enabled helped him “narrow down areas of interest very quickly”.

The survey also yielded another interesting result. It is apparent that some respondents found the tool extremely useful while others do not find it useful at all. This may be a function of the primary programming language for each of the respondents. Respondent R8 scored Diver very low and his or her primary programming language is JavaScript/CSS. However, respondent R1’s primary programming language is C# and his or her organization scored Diver very highly, even using it as a replacement for the commercial tools that they had previously purchased. Another possibility is that a user’s ranking of Diver is correlated with that user’s familiarity with sequence diagrams in general. In our survey, R1 said that the people in his or her organization generally had “good” familiarity with sequence diagrams whereas R8 said that he or she only had “limited” familiarity with them.

## 10.3 Threats to the Validity of the Survey

One possible threat to the validity of this survey is that the respondents are self-selected. It may not be representative of software developers as a whole. However, it does show that

there is a selection of users who find Diver useful. This indicates that Diver does offer cognitive support at least to some people, which is an important result.

Another threat is that the respondents may have given positive responses in order to please the researchers. This threat is common to all research of this kind. However, we did receive varied responses with one individual reporting that the tools were not useful at all. For this reason, we believe the respondents to be honest in their answers.

Finally, the number of respondents was quite small so it is difficult to generalize. This survey cannot be used on its own to draw any conclusions about the usefulness of our approaches. Instead, we interpret the results in the context of the rest of the investigation in this thesis. It helps to triangulate our other findings.

# CHAPTER 11

---

## Conclusions

---

This thesis investigated both the technical and human aspects of creating scalable tools incorporating sequence diagrams of large execution traces. In this chapter, we conclude the thesis by taking another look at the questions that we set out to answer, pointing out the contributions that were made during the course of our investigation, and looking forward toward future directions that this research may take.

### 11.1 Questions Answered

The focus of this thesis was to improve the scalability of tools incorporating sequence diagram visualizations of large execution traces. We identified two major research questions: (RQ1) *How can the size of sequence diagrams be reduced, given that their size hinders users' understanding?* and (RQ2) *how can navigation in sequence diagrams be supported?* There are technical challenges in building tools that have sufficient features to solve these problems. So before we build solutions to RQ1 and RQ2, we first had to answer TQ1: *how*

*can a feature-rich and computationally scalable sequence diagram viewer be designed and built?* Since human users experience cognitive overload while viewing large diagrams, we took a cognitive support approach to finding solutions.

We addressed TQ1 in Chapter 3. Bennett's list of cognitive support features [7] was used to guide our design. Supporting all of the feature requirements presented a number of technical challenges, but we were able to implement a widget-based design that could present a fully interactive sequence diagram of more than 10,000 messages. Performance began to degrade after approximately 25,000 messages. Our previous study indicates, however, that users' ability to understand sequence diagrams degrades before it reaches the size of 25,000 messages [8]. More cognitive support must be supplied by tools when diagrams approach this size.

RQ1 was addressed in Part III of this thesis. Our approach was to use loops found in source code to compact execution traces as they are visualized in sequence diagrams. This allowed us to augment sequence diagrams with semantic details from source code about why execution patterns are repeated at run-time. The technique was extended to be able to visualize other blocks of code such as conditional blocks. The approach was effective in two ways. First, we were able to reduce the number of messages that had to be displayed in a sequence diagram by more than 80% in two of the three cases we tried. Second, it takes into account the responses from our previous study, which indicated that users rely on source code as well as diagrams to understand software behaviour [8]. This finding is in line with Bennett's set of cognitive design elements, which recommend that tool designers indicate syntactic and semantic relations between software objects [7]. Loops in source code are also a form of behavioural abstraction so exposing them in sequence diagrams provides an additional abstraction mechanism and helps to enhance bottom-up comprehension according to both Bennett and Storey's cognitive design elements [7, 79].

Finally, RQ2 was addressed in Part IV. In Chapter 8, we combined Kersten's task-focused user interface [41] with software reconnaissance [101] to create the trace-focused user interface, which can be used to navigate large sequence diagrams. Navigation is im-

portant for cognitive support [7, 79]. Filtering the Package Explorer according to the unique software elements related to a particular feature reduces the user's search space and helps with his or her top-down understanding of the software system. The sequence diagram displays behavioural information, which helps users build situational and program mental models [93]. Linking the views provides navigation between mental models as suggested by both Bennett and Storey [7, 79]. According to Walenstein's theory, this linking approach affords a form of cognitive support called ends-means reification [94] because it guides users to their end-goal of understanding program behaviour by highlighting the means (i.e. the software artifacts unique to the behaviour) to achieving that goal. We performed a user study and survey to validate the approach and response of the users was positive overall. In the user study, participants displayed less frustration when they were able to navigate between the sequence diagram and the filtered Package Explorer. They were able to locate the features of interest faster when using software reconnaissance and their usage patterns indicated that the Package Explorer played a more central role to their investigation than when software reconnaissance was not available. It was reported in the survey that one group of respondents appreciated the trace-focused user interface because it helped them to quickly locate features of interest. These results are all consistent with ends-means reification and indicate that Diver does provide cognitive support to users.

Overall, we conclude that the approaches developed in the course of the work of this thesis do help improve the scalability of tools incorporating sequence diagram visualizations of execution traces. The work in this thesis also represents the first application of Bennett's cognitive design elements for sequence diagram tools. The successes that were achieved in this thesis act as an interesting case study for Bennett's theory.

## 11.2 Contributions

The work of this thesis resulted in a number of contributions. They may be summarized as follows.

- A pluggable, extensible, and scalable widget-based sequence diagram visualization that implements a large set of cognitive support features.
- A new algorithm that unites execution traces and source code to compact large sequence diagrams in a way that supports multiple mental models.
- The *trace-focused user interface* that uses a degree-of-interest model based in software reconnaissance to support navigation in large sequence diagrams.
- A small user study and survey demonstrating the usefulness of the trace-focused user interface.
- Dynamic Interactive Views for Reverse Engineering (Diver) an Open Source tool for the Eclipse Integrated Development Environment that incorporates all of the new solutions and technologies discussed in this thesis.

The Diver tool unites all of the solutions into one product. As of June 2010, it is possible to plug Diver directly into an existing Eclipse installation using the Eclipse Marketplace<sup>1</sup>. As of the writing of this thesis, Diver ranks as number 130 of 1,094 projects available from the Marketplace (279 installs), with an average of one new install per day. The binary distribution available for manual installation from SourceForge.net<sup>2</sup> has been downloaded 656 times. It has also been awarded the honour of a finalist ranking for the *Best Developer Tool of 2011* by the Eclipse Foundation.<sup>3</sup>

## 11.3 Future Directions

In this thesis, we set out to answer three questions. Another goal of research should be to open up new frontiers for discovery. In this section, we will focus on three possible avenues for continued study: more in-depth validation, extensions to the degree-of-interest model developed in Part IV, and applications of our techniques to other visualizations and tools.

---

<sup>1</sup><http://marketplace.eclipse.org>

<sup>2</sup><http://sourceforge.net/projects/diver>

<sup>3</sup>[http://www.eclipse.org/org/press-release/20110301\\_awardfinalists.php](http://www.eclipse.org/org/press-release/20110301_awardfinalists.php)



### 11.3.1 Further Validation

Reverse engineering techniques often lack validation through user studies. Cornelissen *et al.* performed a systematic literature review of dynamic analysis techniques for program comprehension and found that only 8 of 176 articles reviewed included studies involving human subjects [15]. The approaches developed throughout this thesis were designed according to cognitive support theory, but there is admittedly little validation involving human subjects. More validation could bring further insights.

Of particular interest for further study is the finding that our new techniques appear to help some individuals more than others. It would be interesting to find out the difference between the two populations in order to better serve them. It may not be important to invent a tool or technique that helps all developers in all situations. If one technique can significantly help a subset of developers or reverse engineers, that is a good result and it would be beneficial to know why the technique helps.

It may also be a good idea to investigate the caveats to the loop detection algorithm as outlined in Section 6.3. It is our experience that these kinds of issues are not common in the code repositories that we have worked with, but it would be nice to have some experimental results to back up our intuition.

### 11.3.2 Extending the DOI Model

One of the advantages of presenting the results of software reconnaissance using a DOI model is that it assigns a numerical “interest” value to different software artifacts. It is possible to manipulate and experiment with this value to possibly find one that is more refined. For example, we have already implemented the following:

**Definition 5** Function  $DOI_x : \mathbf{E} \times \mathbf{F} \mapsto [0, 1]$

$$DOI_x(e, f) = \begin{cases} 1 & \text{if } e \in \text{UNIQUE}(f) \\ .5 & \text{if } e \notin \text{UNIQUE}(f) \wedge (\exists c \in \text{children}(e) | DOI(c, f) > 0) \\ 0 & \text{otherwise} \end{cases}$$

$DOI_x$  is like  $DOI$  except that  $DOI_x(e, f)$  is .5 for elements that have structural children that are in the  $\text{UNIQUE}(f)$  set, but are not unique themselves. We implemented this as a result of our user study in which one participant expressed frustration because the Package Explorer may have displayed some methods that were not in the unique set for the currently selected thread. In his case, the methods that he was seeing declared anonymous inner classes with methods that *were* called in selected thread (see Appendix F.6). The user, however, did not always expand the method to see its structural children. The purpose of  $DOI_x$  is to make it possible to assign a different degree of interest to parent and children elements in cases such as these.  $DOI_x$  has not been formally tested, however.

One other possible extension is to incorporate Hamou-Lhadj *et al.*'s utility method detection algorithm [34] to assign utility methods a lower degree-of-interest. Eisenberg also suggests a number of heuristics in his *Dynamic Feature Trace* technique [19]. His technique does not use software reconnaissance, but some of the measures he proposes might be useful for a degree-of-interest model.

Finally, the DOI model proposed in this thesis only affects structural source code views. It is possible to extend the model so that it can affect the sequence diagram as well. This would allow the diagrams to be compacted even further to aid in software comprehension. However, extending the DOI model so that it can efficiently process all of the information needed to filter a sequence diagram is not an easy task. We have implemented a solution and included it in the latest release of Diver, but a full discussion of the implementation is beyond the scope of this thesis.

### 11.3.3 Further Applications of the Techniques

One final area for further study is the application of the techniques discussed in this thesis to other visualizations and tools. Bennett's cognitive design elements are "tuned" for use with sequence diagrams, but they do not have to be limited to sequence diagrams. Indeed, they are based on Storey's set of cognitive design elements that are meant for use in general program comprehension tools. It may be interesting, for example, to apply the DOI technique to existing tools such as SHriMP or Creole [53], or to other visualizations of software behaviour such as UML 2 collaboration diagrams.

## 11.4 Conclusion

Reverse engineering is a difficult task. Even with the techniques proposed in this thesis, it will remain a difficult task. Nonetheless, we have provided some new tools and techniques designed to aid in program comprehension. They have been shown to be effective and it is our hope that their use will make a real difference in the working lives of software developers, analysts, and reverse engineers.

---

## Bibliography

---

- [1] *OMG Unified Modeling Language<sup>TM</sup>(OMG UML), Superstructure*. Object Management Group, 2003. Standard document URL: <http://omg.org/spec/UML/2.3/Superstructure>.
- [2] A. Abran, A. Khelifi, W. Suryn, and A. Seffah. Usability meanings and interpretations in iso standards. *Software Quality Journal*, 11:325–338, 2003. 10.1023/A:1025869312943.
- [3] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation, PLDI '90*, pages 246–256, New York, NY, USA, 1990. ACM.
- [4] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proc. of the Int'l Conf. on Data Engineering*, pages 3–14. IEEE, March 1995.
- [5] Y. Ashida, F. Ohata, and K. Inoue. Slicing methods using static and dynamic analysis information. In *Proc. of the Asia Pacific Software Engineering Conf.*, pages 344 – 350, 1999.

- [6] J. Baldwin, D. Myers, M.-A. Storey, and Y. Coady. Assembly code visualization and analysis: An old dog can learn new tricks. In *Workshop on Evaluation and Usability of Programming Languages and Tools*, 2009.
- [7] C. Bennett. *Tool features for understanding large reverse engineered sequence diagrams*. University of Victoria, 2008. Masters Thesis.
- [8] C. Bennett, D. Myers, M. Storey, D. 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):290–315, 2008.
- [9] J. Bohnet, M. Koeleman, and J. Doellner. Visualizing massively pruned execution traces to facilitate trace exploration. In *IEEE Workshop on Visualizing Software for Understanding and Analysis*. IEEE, 2009.
- [10] L. Briand, Y. Labiche, and J. Leduc. Toward the reverse engineering of uml sequence diagrams for distributed java software. *IEEE Transactions on Software Engineering*, 32(9):642–663, September 2006.
- [11] R. I. Bull. *Model Driven Visualization: Towards a Model Driven Engineering Approach For Information Visualization*. University of Victoria, 2008. Doctoral Dissertation.
- [12] R. I. Bull, C. Best, and M.-A. Storey. Advanced widgets for eclipse. In *Proc. of the 2004 OOPSLA Workshop on Eclipse technology eXchange*, pages 6–11. ACM, 2004.
- [13] M. Burtscher, I. Ganusov, S. Jackson, J. Ke, P. Ratanaworabhan, and N. Sam. The vpc trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329 – 1344, November 2005.

- [14] B. Cornelissen, D. Holten, A. Zaidman, L. Moonen, J. J. van Wijk, and A. van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Proc. of the IEEE Int'l Conf. on Program Comprehension*, pages 49–58, Washington, DC, USA, 2007. IEEE.
- [15] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.
- [16] B. S. Corporation. Software architecture design, visual uml and business process modelling. web page on-line: <http://www.borland.com/together>, December 2010.
- [17] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In S. Diehl, editor, *Software Visualization*, volume 2269 of *Lecture Notes in Computer Science*, pages 647–650. Springer Berlin / Heidelberg, 2002.
- [18] R. DeLine, G. Venolia, and K. Rowan. Software development with code maps. *ACM Communications*, 53:48–54, August 2010.
- [19] A. D. Eisenberg. *Dynamic feature tracing: finding features in unfamiliar code*. University of British Columbia, 2004. Doctoral Dissertation.
- [20] K. Erdös and H. Sneed. Partial comprehension of complex programs (enough to perform maintenance). *Proc. of the 2nd Int'l Conf. on Program Comprehension*, page 98, 1998.
- [21] K. Ericsson and H. Simon. Verbal reports as data. *Psychological Review*, 87(3):215, 1980.
- [22] M. Fagan. Advances in software inspections. *IEEE Transactions on Software Engineering*, 12(7):744–751, 1986.

- [23] P. Flajolet, P. Sipala, and J.-M. Steyaert. Analytic variations on the common subexpression problem. In M. Paterson, editor, *Automata, Languages and Programming*, volume 443 of *Lecture Notes in Computer Science*, pages 220–234. Springer Berlin / Heidelberg, 1990. 10.1007/BFb0032034.
- [24] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Commun. ACM*, 30:964–971, November 1987.
- [25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [26] V. Gibson and J. Senn. System structure and software maintenance performance. *Communications of the ACM*, 32(3):347–358, 1989.
- [27] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proc. of the SIGPLAN Symp. on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.
- [28] T. H. D. Group. Hsqldb. web page on-line <http://hsqldb.org>. Cited on 13 October 2009.
- [29] A. Hamou-Lhadj and T. C. Lethbridge. Compression techniques to simplify the analysis of large execution traces. In *Proc. of the Int'l Workshop on Program Comprehension*, pages 159 – 168. IEEE, 2002.
- [30] A. Hamou-Lhadj and T. C. Lethbridge. An efficient algorithm for detecting patterns in traces of procedure calls. In *Proc. of the Int'l Conf. on Software Engineering Workshop on Dynamic Analysis*, page 33. IEEE, 2003.
- [31] A. Hamou-Lhadj and T. C. Lethbridge. Techniques for reducing the complexity of object-oriented execution traces. *IEEE Workshop Visualizing Software for Understanding and Analysis*, pages 35–40, 2003.

- [32] A. Hamou-Lhadj and T. C. Lethbridge. A metamodel for dynamic information generated from object-oriented systems. *Electronic Notes in Theoretical Computer Science*, 94:59 – 69, 2004.
- [33] A. Hamou-Lhadj and T. C. Lethbridge. Measuring various properties of execution traces to help build better trace analysis tools. In *Proc. of the Int'l Conf. on Engineering of Complex Computer Systems*, pages 559–568. IEEE, June 2005.
- [34] A. Hamou-Lhadj and T. C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. of the Int'l Conf. on Program Comprehension*, pages 181–190. IEEE, 2006.
- [35] A. Hamou-Lhadj, T. C. Lethbridge, and L. Fu. Seat: a usable trace analysis tool. In *Proc. of the Int'l Workshop on Program Comprehension*, pages 157–160. IEEE, May 2005.
- [36] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *Proc. of the IEEE Int'l Workshop on Program Comprehension*, pages 94 – 103. IEEE, 2003.
- [37] M. Inc. Maintainj - reverse engineer java line never before. web page on-line <http://www.maintainj.com>. Cited on 16 September 2009.
- [38] ISO 9241. *Ergonomics Requirements for Office with Visual Display Terminals (VDTs)*. International Organization for Standardization, 2001.
- [39] ISO/IEC 9126. *Software Product Evaluation – Quality Characteristics and Guidelines for the User*. International Organization for Standardization, 2001.
- [40] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proc. of the Int'l Conf. on Software Engineering, ICSE '97*, pages 360–370, New York, NY, USA, 1997. ACM.



- [41] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *Proc. of the Int'l. Conf. on Aspect-Oriented Software Development*, pages 159–168. ACM, 2005.
- [42] M. Kersten and G. C. Murphy. Using task context to improve programmer productivity. In *Proc. of the 14th ACM SIGSOFT Int'l Symp. on The Foundations of Software Engineering*, pages 1–11, New York, NY, USA, 2006. ACM.
- [43] A. Ketterlin and P. Clauss. Prediction and trace compression of data access addresses through nested loop recognition. In *Proc. of the IEEE/ACM Int'l Symp. on Code Generation and Optimization*, pages 94–103, New York, NY, USA, 2008. ACM.
- [44] D. Kimelman, B. Rosenburg, and T. Roth. Strata-various: multi-layer visualization of dynamics in software system behavior. In *Proc. of the IEEE Conf. on Visualization*, pages 172–178. IEEE, October 1994.
- [45] D. Knuth. Notes on the van emde boas construction of priority dequeues: An instructive use of recursion. personal memo to Peter van Emde Boas, March 1977. cited on 8 September 2010 on-line <http://www-cs-faculty.stanford.edu/~knuth/faq.html>.
- [46] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. of the Int'l Conf. on Software Engineering*, pages 344–353, Washington, DC, USA, 2007. IEEE.
- [47] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Intl'l Conf. on Software Engineering*, pages 366–375. IEEE, 1996.
- [48] G. E. Krasner and S. T. Pope. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

- [49] P. Kruchten. The 4+1 view model of architecture. *Software*, 12(6):42–50, 2002.
- [50] D. B. Lange and Y. Nakamura. Program explorer: a program visualizer for c++. In *Proc. of the USENIX Conf. on Object-Oriented Technologies*. USENIX Association, 1995.
- [51] A. Le Gear, J. Buckley, B. Cleary, J. Collins, and K. O’Dea. Achieving a reuse perspective within a component recovery process: an industrial scale case study. In *Proc. of the Int’l Workshop on Program Comprehension*, pages 279–288. IEEE, 2005.
- [52] A. Le Gear, J. Buckley, B. Cleary, J. J. Collins, and K. O’Dea. Achieving a reuse perspective within a component recovery process: An industrial scale case study. *Int’l Conf. on Program Comprehension*, 0:279–288, 2005.
- [53] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu. Plugging-in visualization: experiences integrating a visualization tool with eclipse. In *Proc. of the ACM Symp. on Software visualization*, pages 47–56. ACM, 2003.
- [54] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proc. of the Int’l Conf. on Automated Software Engineering*, pages 234–243. ACM, 2007.
- [55] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting duplications in sequence diagrams based on suffix trees. In *Proc. of the Asia Pacific Conf. on Software Engineering*, pages 269–276, December 2006.
- [56] I. B. Machines. Ibm developerworks: rational rose. web page on-line: <http://www.ibm.com/developerworks/rational/products/rose>, December 2010.
- [57] S. McConnell. *Code Complete: Second Edition*. Microsoft Press, 2004.

- [58] M. McGavin, T. Wright, and S. Marshall. Visualisations of execution traces (vet): an interactive plugin-based visualisation tool. In *Proc. of the Australasian user interface conference*, AUIC 06, pages 153–160. Australian Computer Society, Inc., 2006.
- [59] J. E. McGrath. Methodology matters: doing research in the behavioral and social sciences. In *Human-computer interaction: toward the year 2000*, pages 152–169. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1995.
- [60] R. Miller. Response time in man-computer conversational transactions. In *Proc. of the fall joint computer conference, part I*, pages 267–277. ACM, December 1968.
- [61] H. Müller, K. Wong, and S. Tilley. Understanding software systems using reverse engineering technology. In S. A. Vangular and R. Missaoui, editors, *Object-oriented technology for database and software systems*, pages 240–252. World Scientific, 1995.
- [62] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Čubranić. The emergent structure of development tasks. In A. P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 33–48. Springer Berlin / Heidelberg, 2005.
- [63] D. Myers, M.-A. Storey, and M. Salois. Utilizing debug information to compact loops in large execution traces. In *Proc. of the European Conf. on Software Maintenance and Re-engineering*, pages 41–50. IEEE, March 2010.
- [64] C. G. Nevel-Manning. *Inferring Sequential Structure*. University of Waikato, May 1996.
- [65] U. Nickel, J. Niere, and A. Zundorf. The fujaba environment. In *Int’l Conf. on Software Engineering*, page 742. IEEE, 2000.

- [66] Oracle. Jdk java virtual machine tool interface (jvmti) – related apis and developer guides. Documentation on-line <http://download.oracle.com/javase/6/docs/technotes/guides/jvmti>. Cited on 15 February 2011.
- [67] Oracle. A swing architecture overview. Documentation on-line <http://java.sun.com/products/jfc/tsc/articles/architecture>. Cited on 15 February 2011.
- [68] T. Poranene, E. Mäkinen, and J. Nummenmaa. How to draw a sequence diagram. In *Proc. of the 8th Symp. on Programming Languages and Software Tools*, pages 91–102, 2003.
- [69] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. of the Int’l Conf. on Software Engineering*, pages 221–230. IEEE, 2001.
- [70] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. of the Int’l Conf. on Software Maintenance*, pages 34–43. IEEE, 2003.
- [71] J. Rilling and B. Karanth. A hybrid program slicing framework. In *Proc. of the Int’l Workshop on Source Code Analysis and Manipulation*, pages 12–23, 2001.
- [72] H. Safyallah and K. Sartipi. Dynamic analysis of software systems using execution pattern mining. In *Proc. of the IEEE Int’l Conf. on Program Comprehension*. IEEE, 2006.
- [73] J. Seward. bzip2: Home. web page on-line <http://www.bzip.org>. Cited on 13 October 2009.
- [74] R. Sharp and A. Rountev. Interactive exploration of uml sequence diagrams. In *IEEE Workshop on Visualizing Software for Understanding and Analysis*, pages 8–13. IEEE, 2005.

- [75] J. Sillito, G. C. Murphy, and K. De Volder. Questions programmers ask during software evolution tasks. In *Proc. of the ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, pages 23–34. ACM, 2006.
- [76] J. Snowden, H. Griffiths, and D. Neary. Semantic-episodic memory interactions in semantic dementia: Implications for retrograde memory function. *Cognitive Neuropsychology*, 13(8):1101–1139, 1996.
- [77] J. Starke. *Finding what is important: understanding and improving code search*. University of Calgary, July 2010. Masters Thesis.
- [78] J. T. Stasko and J. Muthukumarasamy. Visualizing program executions on large data sets. In *Proc. of Symp. on Visual Languages*, pages 166 –173. IEEE, September 1996.
- [79] M.-A. Storey. *A cognitive framework for describing and evaluating software exploration tools*. Simon Fraser University, 1998. Doctoral Dissertation.
- [80] T. Systä. On the relationships between static and dynamic models in reverse engineering java software. In *Proc. of the Working Conf. on Reverse Engineering*, pages 304–313. IEEE, October 1999.
- [81] T. Systä. Understanding the behavior of java programs. In *Proc. of the Working Conf. on Reverse Engineering*, pages 214 –223. IEEE, 2000.
- [82] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue. Extracting sequence diagram from execution trace of java program. In *Proc. of the Int'l Workshop on Principles of Software Evolution*, pages 148 – 151. IEEE, September 2005.
- [83] The Eclipse Foundation. Eclipse.org home. web page on-line <http://www.eclipse.org>. Cited on 13 October 2009.
- [84] The Eclipse Foundation. Jetty. web page on-line <http://www.eclipse.org/jetty>. Cited on 13 October 2009.

- [85] The Eclipse Foundation. Rich ajax platform (rap). Documentation on-line <http://www.eclipse.org/rap>. Cited on 16 February 2011.
- [86] The Eclipse Foundation. Using uml2 trace interaction views. Documentation on-line <http://help.eclipse.org/galileo/index.jsp?topic=/org.eclipse.tptp.platform.doc.user/tasks/tesqanac.xhtml>. Cited on 22 September 2009.
- [87] The Eclipse Foundation. Draw2d. web page on-line: <http://www.eclipse.org/gef/draw2d>, December 2010.
- [88] The Eclipse Foundation. Eclipse memory analyzer tool. web page on-line: <http://www.eclipse.org/mat>, December 2010.
- [89] The Eclipse Foundation. Eclipse plugins, bundles, and produces - eclipse marketplace. web page on-line: <http://marketplace.eclipse.org>, December 2010.
- [90] The Eclipse Foundation. Jface - eclipsepedia. web page on-line: <http://wiki.eclipse.org/index.php/JFace>, December 2010.
- [91] The Eclipse Foundation. Swt: The standard widget toolkit. web page on-line: <http://www.eclipse.org/swt>, December 2010.
- [92] E. Tulving. *Organization of Memory*, chapter Episodic and Semantic Memory. Academic Press, 1972.
- [93] A. Von Mayrhauser and A. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, August 1995.
- [94] A. Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *Proc. of the Int'l Workshop on Program Comprehension*, pages 75–84. IEEE, 2002.

- [95] M. Weiser. Program slicing. In *Proc. of the Int'l Conf. on Software Engineering*, ICSE '81, pages 439–449. IEEE, 1981.
- [96] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.
- [97] E. Weller. Lessons from three years of inspection data. *IEEE Software*, 10:38–38, 1993.
- [98] E. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14:1357–1365, 1988.
- [99] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Proc. of the Working Conf. on Reverse Engineering*, pages 270–276. IEEE, 1996.
- [100] N. Wilde and C. Casey. Early field experience with the software reconnaissance technique for program comprehension. In *Int'l Conf. on Software Maintenance*, pages 312–318, November 1996.
- [101] N. Wilde and M. Scully. Software reconnaissance: mapping program features to code. *Journal of Software Maintenance: Research and Practice*, 7(1):49–62, 1995.
- [102] K. Wong and D. Sun. On evaluating the layout of uml diagrams for program comprehension. *Software Quality Journal*, 14:233–259, 2006.
- [103] I. Zayour. *Reverse Engineering: A Cognitive Approach, a Case Study and a Tool*. University of Ottawa, 2002. Doctoral Dissertation.
- [104] H. Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter & Co., Hawthorne, NJ, USA, 1990.

## **Part VI**

### **Appendices**



## APPENDIX A

---

### The Diver Resources

---

The tools and techniques introduced by this thesis have all been implemented by the author in the Open Source Software called *Dynamic Interactive Views for Reverse Engineering* (Diver). The project is hosted by Sourceforge.net.<sup>1</sup> and all of the resources used in the by it can be retrieved from the following URLs.

- <http://diver.sf.net> The Diver web page, containing general information, news, tutorial videos, and more than 20 pages of documentation for the Diver project.
- <http://sourceforge.net/projects/diver> The development web page for Diver. Users can access help forums and bug reports through this page.
- <http://diver.svn.sourceforge.net/svnroot/diver> Anonymous access to the subversion<sup>2</sup> repository for the Diver project. All source code, videos, and documentation can be retrieved from this location.

---

<sup>1</sup><http://sourceforge.net>

<sup>2</sup><http://svn.tigris.org>

## APPENDIX B

---

### Sequence Diagram Implementation Details

---

The requirements analysis in Section 3.1 has lead us to a design in which each element of the sequence diagram visualization will be represented as a user interface widget that can be controlled and interacted with by the user. This design allows us to represent the elements in the sequence diagram and support the cognitive design requirements discussed in Section 3.1.3. For the reasons of portability, extensibility, and familiarity, we have chosen Java as the target platform and the Eclipse RCP as our target application framework. The technology used as the widget toolkit in RCP is called the Standard Widget Toolkit (SWT). We also need a framework to assist us in drawing the elements of the sequence diagram on-screen. The standard drawing framework for SWT is called Draw2D, so we will use it. Sections B.1 and B.2 will give brief overviews of the two frameworks. Section B.3 will discuss how we combine these frameworks to create custom widgets for the sequence diagram. This information in this appendix is an overview of the design and implementation of the different frameworks, toolkits, and components involved in creating the sequence diagram viewer. It is not intended to be complete design documentation, but it is useful for

explanation purposes.

## B.1 The SWT Widget Framework

SWT stands for *Standard Widget Toolkit*. It is the widget toolkit used by the Eclipse platform. The structure of the toolkit is summarized in Figure B.1.

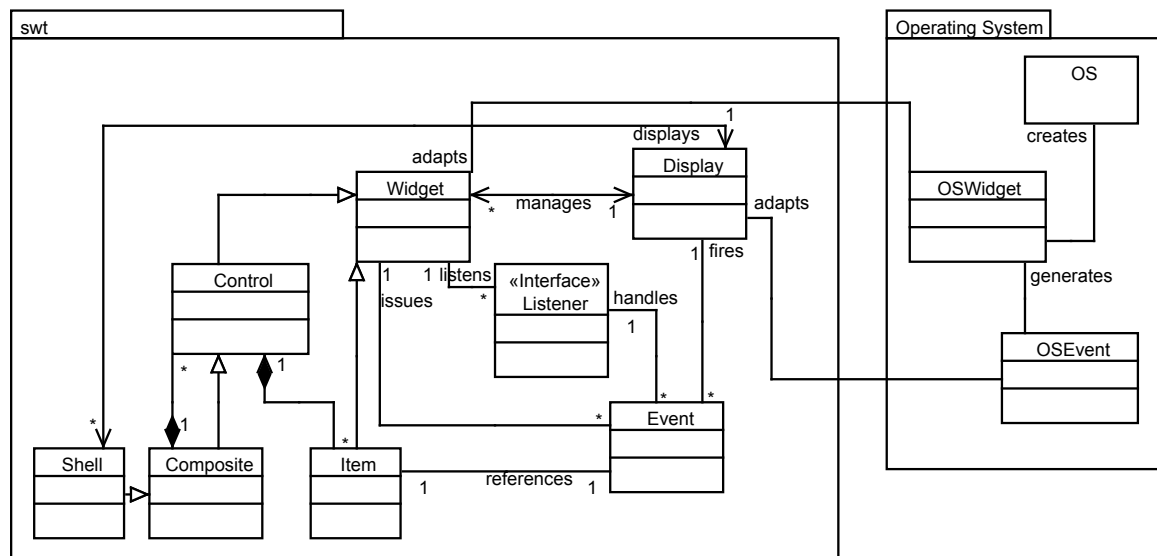


Figure B.1: The classes of SWT

SWT widgets come in two basic forms: *controls* and *items*. Controls are basic user interface elements that support user interaction. Complex controls may contain items. For example, a Tree control may contain many `TreeItems` which are expandable items that represent the elements in the tree using labels and icons. Controls can be laid out in a containment hierarchy inside a composite control. The SWT containment hierarchy must be rooted in a shell composite container. Shells are often represented as “windows” in graphical user interface environments.

At this point it is important to make a quick note about the terminology that we will be using. There is a small difference between the term *control* and the term *viewer*. For this discussion, we follow the Eclipse conventions. In general, a control is a displayer of

widgets. That is, it is purely visual and it is agnostic about any underlying data model. A viewer uses controls to create a visual representation of an underlying data model. So, throughout this portion of our discussion, we will be using the term *sequence diagram control* to mean the pure visual representation of a sequence diagram. The *sequence diagram viewer* is a sequence diagram control connected to an underlying data model. This distinction should become clearer in Section C.

Communication between widgets uses a single-threaded event processing paradigm managed by the `Display` class. The display responds to user interface events generated by the operating system. It also offers facilities to programmatically generate events. Operating system events are adapted by the display to corresponding SWT events. The display finds the SWT widget that corresponds to the operating system widget that generated the event. The SWT widget is made to “issue” a new event through the display. The display then fires the events to listeners that handle the events. Listeners can be registered on an individual widget, in which case they only receive notification about events that are issued by that widget. Listeners can also be registered on the display itself to either respond to events or act as filters on SWT events to prevent them from being handled down stream. Most events are issued by SWT controls, rather than SWT items. If an event occurs on an item, that item’s parent control will typically issue an event that references the item. For example, if a tree node is expanded, the tree control will issue the event and the event will make reference to the tree item that was expanded.

This event system is crucial to the implementation of our sequence diagram control. If the control is to be built to follow the SWT widget system, care must be taken to make sure that our widgets issue the correct events at the correct time. This will enable the control to post events to the user interface system that can then be handled by the platform.

## B.2 The Draw2D Framework

SWT widgets are typically drawn by the operating system. However, no standard operating system supplies widgets corresponding to the ones that we will require to implement a sequence diagram control. Therefore, we are left with the challenge of creating custom widgets.

SWT is an extensible toolkit that allows users to extend the SWT Composite control to create new widgets. The standard approach to creating new widgets is to extend a child of the Composite class and intercept the SWT Paint event that indicates that the composite is about to be drawn. This allows clients to override the composite's default paint capabilities and create new widgets with their own look-and-feel. The `org.eclipse.ui.swt.custom` package, which is part of the standard SWT distribution, uses this technique.

However, any simple implementation following this approach for sequence diagrams would not scale. The simplest approach would be to repaint the entire diagram with every paint event. But this wastes time and resources. It is better to have a method of locating only the elements that need to be painted and paint them. Adding advanced presentation features, such as selection, variation in visual attributes, and animation, complicates matters further.

Fortunately, the Draw2D framework offers solutions to these problems. This standard Eclipse framework supplies APIs that allow clients to use SWT Canvas composites as shape and raster drawing tools. The classes that we are interested in using from the framework are described in Figure B.2

The basic elements of the Draw2D framework are called *figures* and are represented by the `IFigure` interface. Figures are basic shapes or raster images that supply functionality to draw themselves onto an SWT Canvas. The logic of when figures should be drawn or updated is controlled by a class called the `LightweightSystem`. The `LightweightSystem` keeps track of all figures and traps SWT paint events to initiate the drawing of the figures. Figures are only drawn if their visual properties have changed

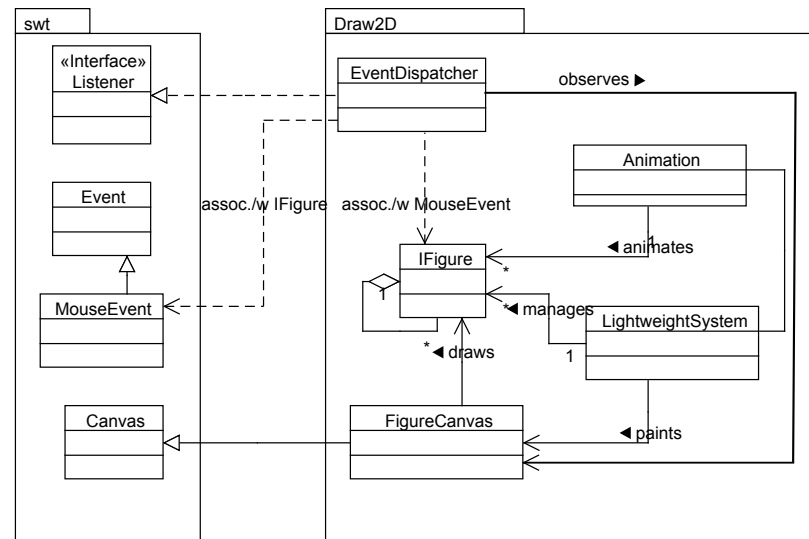


Figure B.2: The classes of Draw2D

and they have been marked as “dirty” in the lightweight system.

Figures can contain other figures and their positions and sizes can be set relative to their parent figure. The location and size of figures is used to create a bounding box around each figure that may be used to compute when figures overlap or to discover the figure that is at a particular location on the screen.

Finally, Draw2D offers an `Animation` class for automatic animating when the size and location of the figures in the lightweight system changes. The details of this class are not important except that we must note that this class is used by our control to animate transitions between sequence diagram states.

## B.3 Creating Widgets Using Draw2D

The basic details about the SWT widget framework and the Draw2D drawing framework have been introduced. What is left to explain is how the two frameworks can be used together to create a sequence diagram. The first thing that we must do is define the widgets that will be used in the sequence diagram. They are shown in Figure B.3.

As can be seen, the sequence diagram control is a composite widget and contains mul-

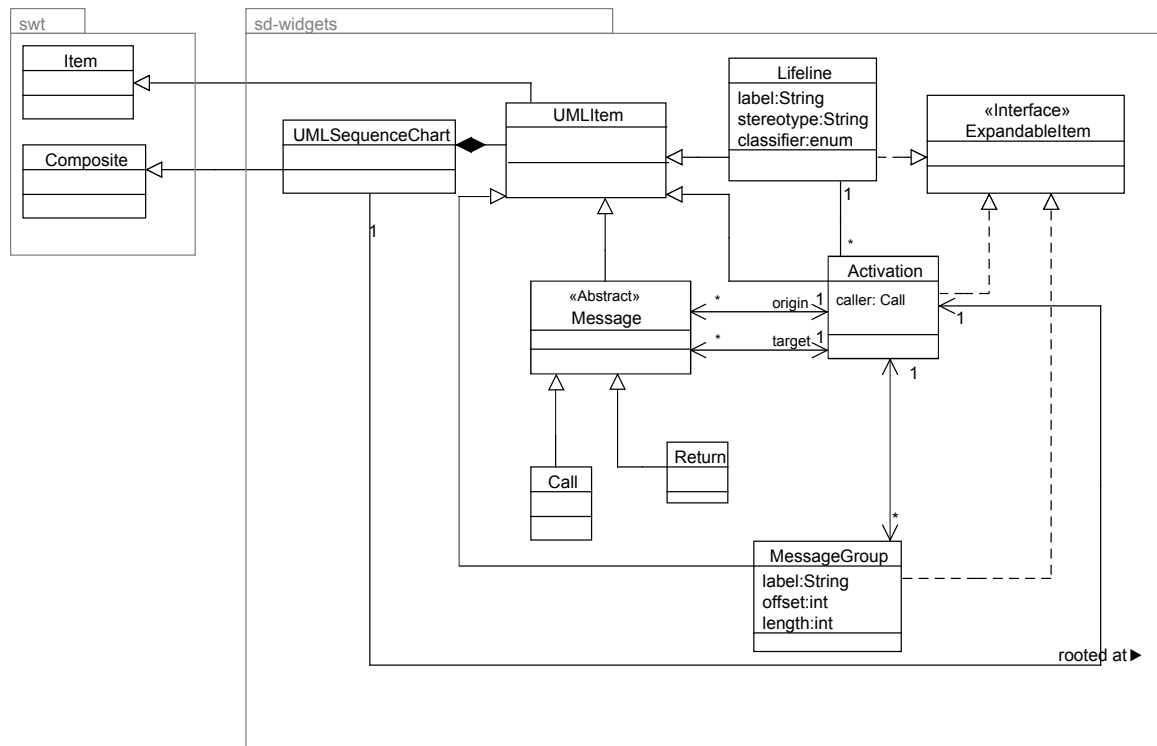


Figure B.3: The sequence diagram widgets

multiple items called UMLItems. There is one UMLItem type for each component in the sequence diagram, as shown in Figure 1.1. The sequence diagram is rooted at one activation that may be changed at any time. This offers a way to display a slice of a sequence diagram at a time. An activation belongs to a single life line and a life line can have many activations. Life lines can be “contained” in other life lines. This is to allow for varying levels of abstraction to be shown in the sequence diagram by defining a hierarchy of life line “types”. For example, an object can be understood to be “contained” within the class that declares the object type and the class can be contained in a package or namespace.

Activations may initiate many messages, and each message references two activations: one source and one target. There are some further constraints on the relationships between messages and activations, which are not captured in the figure for the sake of simplicity. The source and target activations for a message cannot be the same, activations may be the source of many calls but the target of only one, and activations may be the target of many

returns, but only if that activation can be reached via a backward traversal of calls starting from the return's source activation. Finally, messages are ordered in an activation by the "happens before" relationship.

Messages can be grouped within an activation using the `MessageGroup` widget. This widget fulfills the role of combined fragments as described in Section 3.1.2. `MessageGroups` are defined to surround a range of messages and the range is indicated by the offset and length, which relate to the ordering of messages in the activation. Message groups can be nested based on their ranges. If a range of a message group is contained completely within the range of another, then it is nested within that other group. `MessageGroups` are ordered so that if multiple groups have the same ranges then the latter groups are nested in the previous ones. Ranges are not allowed to partially overlap.

Finally, the `Activation`, `MessageGroup`, and `Lifeline` widgets all implement an `IExpandable` interface indicating that they can be expanded or collapsed. Collapsing an activation hides all messages that exist down the call chain originating from that activation. Collapsing a message group hides all messages (and their target activations) contained in that group. These two actions vertically compact the diagram.

Collapsing a life line raises the level of abstraction for that life line such that only a single life line is displayed, which represents all of the contained life lines. For example, a life line may be collapsed such that the level of abstraction is raised to a single package. Now, this life line represents all of the classes in that package, and all of the activations for each class in the package will be displayed on a single life line. This compacts the diagram horizontally.

## B.4 Building the Layered Architecture

The widget classes have been defined, but we still need a process for displaying them. With standard widgets, SWT delegates the creation, display, and interaction of widgets to the operating system. We will use `Draw2D`. Figure 3.1 draws an analogy between our



process and the way that the operating system is used in SWT.

Figure B.4 shows the details of our architecture. Inside the **sd-visuals** layer, we have a class called `SequenceChartVisuals` that is responsible for creating the figure canvas and lightweight system that will be used in the `Draw2D` layer to draw the diagram. The `SequenceChartVisuals` is also responsible for managing instances of the `WidgetVisualPart` class. For each `UMLItem` in the sequence diagram, a `WidgetVisualPart` is created. There is a subclass of `WidgetVisualPart` that corresponds to each subclass of `UMLItem` as described in Figure B.3. The `WidgetVisualPart` maps each `UMLItem` to an `IFigure` that will be drawn on-screen by `Draw2D`. It observes changes that occur on the `UMLItem`, such as label changes, colours, or hidden states, and adjusts updates the `Draw2D` figure to reflect the changes.

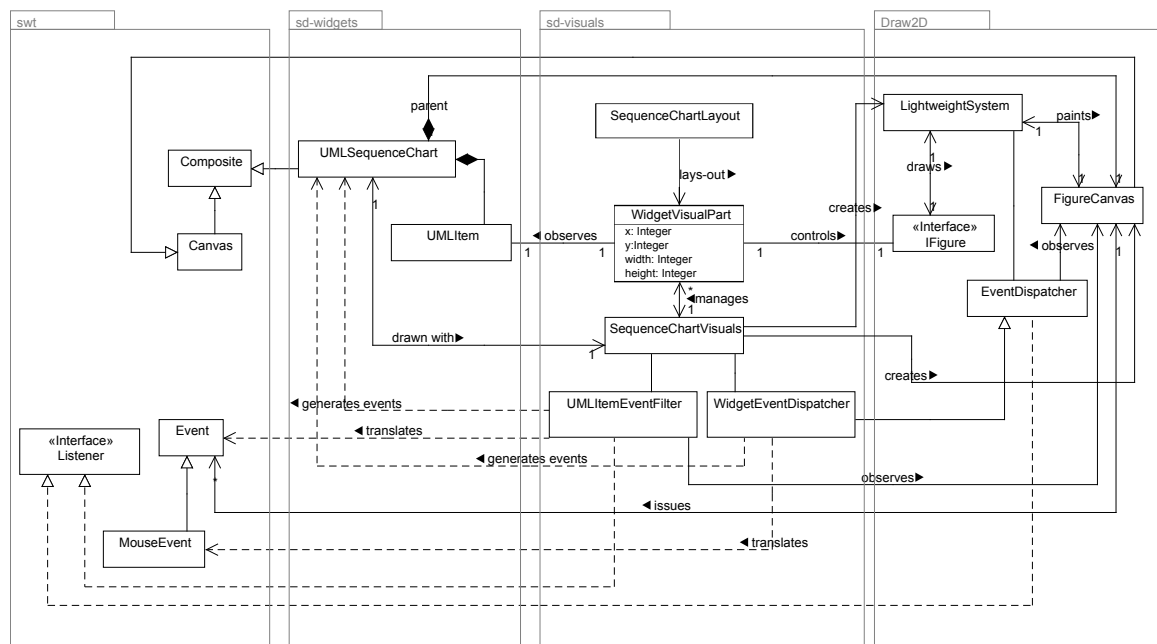


Figure B.4: The classes involved in the layered architecture

In Figure 3.1, we saw that the communication between `Draw2D` and the SWT on a higher layer could cause problems with the events published by the SWT display. Notice that the canvas on which the figures are drawn is a child of the `UMLSequenceChart` composite. This is done to support information hiding. Just as typical clients of SWT do

not have access to the low-level operating system input/output routines that draw widgets on the screen, neither should the clients of the sequence diagram have access to `Draw2D`. However, this causes some difficulty. For example, mouse clicks that occur on the `FigureCanvas` in `Draw2D` should actually be interpreted as selections on the `UMLSequenceChart`. There are two classes that deal with this problem. The first is the `WidgetEventDispatcher` that extends the standard `Draw2D EventDispatcher`. The `EventDispatcher` is the standard `Draw2D` class for handling mouse events that occur within figures drawn on a figure canvas, so it is used to intercept such events and translate them into corresponding events for the `UMLSequenceChart`. The `UMLItemEventFilter` performs similar functionality for all other events, such as key presses and context menu events.

Finally, Figure B.4 indicates that the visual layer is also responsible for the layout of the sequence diagram. It does its layout through the `SequenceDiagramLayout` class which places the `WidgetVisualParts` in their correct positions, which will, in turn, position the figures in the `Draw2D` layer.

The architecture described here is not, in fact, limited to usage with sequence diagrams. This method may be used to create any number of advanced widgets using SWT and a graphics library such as `Draw2D`. However, we have not applied it to any other visualization.

## APPENDIX C

---

### JFace and the Model-View-Controller Pattern

---

The widget design discussed in Section B.3 defines a view model for a sequence diagram. Each widget represents an element that is visible and interactive in the view. However, in Sections 3.1.3 and 3.1.3 we noted that just presenting a control is not enough to support our requirements. We must create a viewer that can interact with a data model so that the application can support communication between different representations of the underlying data, or between representations of differing, though related data. In our case, for instance, we are be interested in supporting communication between views of source code and views of execution traces.

Decoupling the data model and the visual control is achieved through the standard Model-View-Controller pattern originally created for use in the Smalltalk development environment [48]. In this pattern, the *model* corresponds to the data model. The *view* is a visual representation of an instance of the data model. It is analogous to the controls and widgets which we have been describing for SWT. The *controller* is a set of classes that can make changes to the data and update the view.

In this discussion, we are most concerned with the method by which we may create a mapping between the data model and the widgets that we have already defined. Eclipse offers a convenient solution in the form of JFace [90]. JFace supports the Model-View-Controller pattern by supplying a *viewer* which acts as a controller for the view model (instantiated as widgets) and the data model (represented by user data). The mapping between the user data and the widgets is supplied by two instantiations of what is essentially the adapter pattern as described by Gamma *et al.* [25]. These two adapters are a *content provider* and a *label provider*. The content provider is responsible for translating data model concepts into concepts that can be used to generate widgets. The label provider is responsible for translating data model concepts into visual attributes such as textual labels and colours. Figure C.1 shows the associations that are used in Diver for creating two different views of the same data using the JFace techniques. In this example, trace data is used to create both a tree visualization and a sequence diagram visualization.

To illustrate how the JFace techniques are used in our design to create a data model independent sequence diagram viewer, we include the Java interface for our content provider in listing C.1. For brevity and ease of explanation, the description of the methods of the interface are included as source code comments. JFace offers a number of standard label provider interfaces, which are supported by our viewer. The standard interfaces provide things such as textual labels and colours. Our implementation also offers several additional interfaces for styling the lines which represent message calls. All of the code for these interfaces is available as Open Source software which can be obtained as described in appendix A.

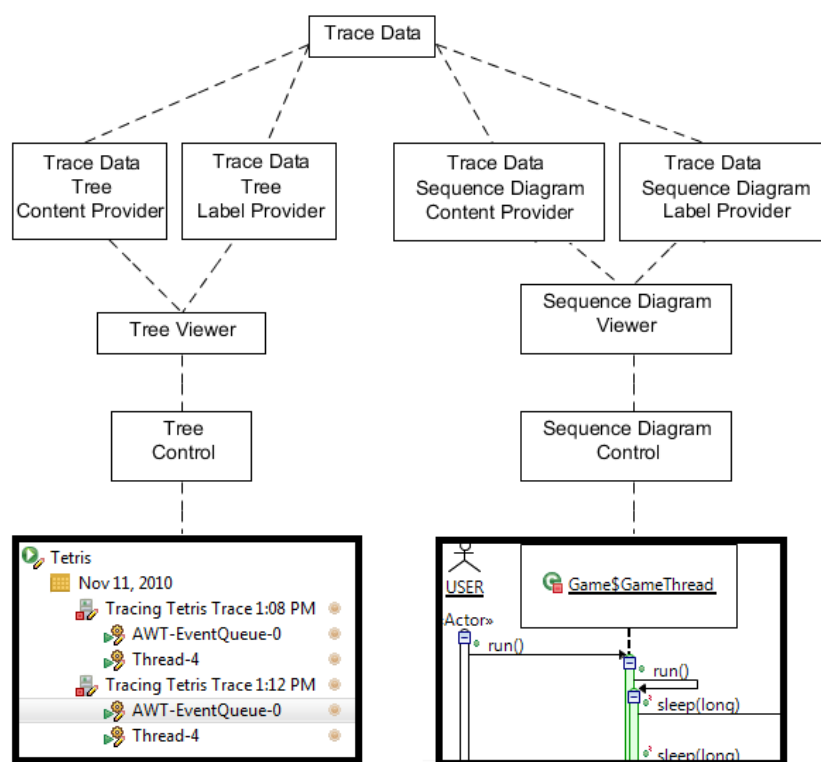


Figure C.1: Using the same data model for two viewers

---

```
public interface ISequenceChartContentProvider implements IStructuredContentProvider{
    /**
     * Inherited from IStructuredContentProvider
     * Returns an array of model objects representing the root activation of
     * the sequence diagram. Note, it is expected that this array is of length
     * one. It returns an array to conform with the IStructuredContentProvider
     * interface
     */
    Object[] getElements(Object input);

    /**
     * Returns all of the messages that originate from the given activation.
     * These may correspond to calls to other activations or returns to
     * previous activations.
     * activations.
     */
    Object[] getMessages(Object activation);

    /**
     * Returns an object corresponding to the lifeline that the given
     * activation is on.
     */
    Object getLifeline(Object activation);

    /**
     * Returns an activation that corresponds to the target activation
     * of the given message.
     */
    Object getTarget(Object message);

    /**
     * Returns true if the given message should be treated as a call. That is,
     * it will result in a new activation being created. Otherwise, it will
     * be treated as a return to a previous activation on the same call stack.
     * Note that if the object returned by {@link #getTarget(Object)}
     * is not on the call stack, and this method returns false, then a malformed
     * sequence diagram may result, and the diagram will not be displayable.
     */
    boolean isCall(Object message);
}
```

---

Listing C.1: The sequence diagram content provider interface

## APPENDIX D

---

### A $O(n)$ Layout Algorithm for Sequence Diagrams

---

In Chapter 3, we indicated that finding a solution to the “ideal” layout for a sequence diagram is an NP-Complete problem according to Poranen *et al.* [68]. They argue that there are a large number of constraints that must be considered in order to create an aesthetically pleasing diagram and deciding the best layout according to those constraints is very difficult. However, it is possible to create a much faster solution if we are selective about our constraints. This appendix presents an  $O(n)$  solution to the sequence diagram layout problem.

#### D.1 Layout Requirements

Before presenting the implementation of the layout algorithm, we will investigate the constraints that we should use to present a sequence diagram. Since this thesis is concerned with both technical and cognitive scalability in sequence diagrams, we will look for constraints that will offer some cognitive support. Wong *et al.* surveyed literature in perceptual

theory to detail several laws of perception [102]. They used these laws to define a number of criteria for laying out UML diagrams to support program comprehension. We use these criteria as a basis for our layout solution.

Some criteria appear in multiple laws of perception. Two related criteria – *avoid overlapping* and *minimize crossing and bends* – appear in three of the laws, so we will try to optimize our algorithm for them. In sequence diagrams, overlapping and crossing occurs on the horizontal axis as when insufficient space is allotted for textual labels. Overlaps in the vertical direction are not a problem since all messages (in our implementation) are parallel. Unfortunately, allowing too much horizontal space conflicts with another important criteria according to Wong *et al.*: *reduce number of long edges*. So, our implementation will attempt to reduce overlaps, while also reducing the length of edges.

It is unlikely that we can come up with a truly optimal solution in any reasonable time over all possible layouts of a sequence diagram. However, it is possible to prevent overlap while reducing edge length *given an ordered list of lifelines*. For a given ordering of lifelines, it is impossible to prevent any other form of overlap concerning messages between the lifelines. Since the lifelines are fixed, the endpoints of the messages are fixed. Hence, messages will cross any lifeline that is between its two endpoints. However, it is possible to eliminate the overlap of textual labels, so our algorithm will focus on that problem.

To fix the ordering of lifelines, we consider Wong *et al.*'s criterion *exploit proximity*. Exploiting proximity means components that are closely related should be placed close together in the diagram. Since sequence diagrams are time-oriented, we optimize for time, ordering lifelines from left to right according to when they are first referenced in the sequence. In this way, it is possible to read the sequence diagram (mostly) from left to right,<sup>1</sup> and lifelines that are closely associated with one another early in the sequence will be drawn in close proximity. This can be done in  $O(n)$  time using a trivial walk on the call tree represented by the sequence diagram. We won't discuss it further here.

---

<sup>1</sup>Except in cases such as recursion or when a call is made on a lifeline that has previously been referenced.



## D.2 Implementation

The basic idea of the algorithm is to give as much horizontal room as is required to display textual labels without occlusion while also minimizing the length of the arcs that represent the messages. The pseudo-code for the algorithm is shown in Listings D.1, D.2, and D.3.

---

### Algorithm Layout-Sequence-Diagram

---

**Require:** A sequence diagram  $s$

---

```

1: Let  $L$  be an ordered list (size  $|L|$ ) of visible lifelines in  $s$ 
2: %initialize the horizontal positions of  $\exists l \in L$  so that we can tell the order
3: for  $i = 0 \dots |L| - 1$  do
4:    $L[i].x \leftarrow i$ 
5: end for
6: % $s.root$  is the visible root activation of the sequence diagram
7:  $a \leftarrow s.root$ 
8:  $a.y \leftarrow 0$ 
9: Recursive-Set-Spacing( $a$ )
10: %finalize the horizontal positions of the lifelines
11:  $L[0].x \leftarrow 0$ 
12:  $L[0].width \leftarrow textwidth(L[0].label)$ 
13:  $x \leftarrow L[0].x + L[0].width$ 
14: for  $i = 1 \dots |L| - 1$  do
15:    $L[i].width \leftarrow textwidth(L[i].label) + L[i].maxCallDepth$ 
16:    $labelWidths \leftarrow (L[i].width + L[i-1].width)/2$ 
17:   if  $distance(L[i-1], L[i]) > labelWidths$  then
18:      $x \leftarrow x + distance(L[i-1], L[i]) - labelWidths$ 
19:   end if
20:    $L[i].x \leftarrow x$ 
21:    $x \leftarrow x + L[i].width$ 
22: end for
23: %apply the final positions to all activations
24: Recursive-Apply-Layout( $a$ )

```

---

Listing D.1:  $O(n)$  Layout algorithm

**Algorithm** Recursive-Set-Spacing**Require:** An activation  $a$ 


---

```

1: %recursively iterate through all sub-calls, adjusting the distance between lifelines, and
   the y positions of activations
2:  $l \leftarrow a.lifeline$ 
3:  $a.callDepth \leftarrow l.callDepth$ 
4:  $l.callDepth \leftarrow l.callDepth + 1$ 
5: if  $l.callDepth > l.maxCallDepth$  then
6:    $l.maxCallDepth \leftarrow l.callDepth$ 
7: end if
8:  $top \leftarrow a.y$  %the current y position for a message
9:  $a.height = 1$ 
10: for each message  $m$  originating at  $a$  do
11:    $top \leftarrow top + 1$ 
12:    $t \leftarrow m.target$ 
13:    $l_t \leftarrow t.lifeline$ 
14:    $w \leftarrow textwidth(m.label)$ 
15:   %At this point, the x value of each lifeline is initialized to the lifeline's index in  $L$ 
16:   if  $l_t.x < l.x$  then
17:     %the message is pointing left: set the distance between this lifeline, and the one
       to the left
18:     if  $w > distance(l, L[l.x - 1])$  then
19:        $distance(l, L[l.x - 1]) \leftarrow w$ 
20:     end if
21:   else
22:     %similar for self-calls and messages pointing right
23:     if  $l.x < |L|$  and  $w > distance(l, L[l.x + 1])$  then
24:        $distance(l, L[l.x + 1]) \leftarrow w$ 
25:     end if
26:   end if
27:   if  $m$  is a call message then
28:      $t.y \leftarrow top$ 
29:     Recursive-Set-Spacing( $t$ )
30:     %the next message must have a y position below  $t$ 
31:      $top \leftarrow top + t.height$ 
32:   end if
33: end for
34:  $a.height \leftarrow (top - a.y) + 1$ 
35:  $l.height = l.height + a.height$ 
36:  $l.callDepth \leftarrow l.callDepth - 1$ 

```

---

Listing D.2: Setting the spacing for lifelines and activations

**Algorithm** Recursive-Apply-Layout**Require:** An activation  $a$ 


---

```

1: %recursively iterate through all sub-calls, adjusting the the x position of an activation
   according to the position of its lifeline, and its call depth
2:  $l \leftarrow a.lifeline$ 
3:  $a.x \leftarrow (l.x + l.width)/2 + a.callDepth$ 
4: for each message  $m$  originating at  $a$  do
5:   if  $m$  is a call message then
6:     Recursive-Apply-Layout( $m.target$ )
7:   end if
8: end for

```

---

Listing D.3: Applying the layout constraints to the x coordinates of the activations

There are several basic steps. After the orientation of the lifelines has been initialized (lines 3 to 5 in Listing D.1), the horizontal spacing between lifelines and the vertical location of activation boxes is set in the algorithm Recursive-Set-Spacing (Listing D.2). Each message originating at an activation  $a$  is visited (D.2, line 10). To set the spacing between lifelines, first the direction of the message is checked. If the message is directed to the left, there must be enough space to display the message's label between the lifeline of  $a$  and the lifeline to its left (line 19). Similar for right-directed messages (line 24). The conditions of this portion of the code ensure that by the time the algorithm is complete, for any index  $i \leq 0 < |L| - 1$ ,  $distance(L[i], L[i + i])$  will equal the minimum distance required between  $L[i]$  and  $L[i + 1]$  to prevent occlusion of a message label by a lifeline.

To set the vertical location of an activation, we simply ensure that each activation has enough height to encompass all of the messages that originate from it. This is achieved by incrementing a y location (*top* at line 11) for each message that is encountered. If a message is a call to another activation,  $t$ , then it signifies that another call tree rooted at  $t$  must also be laid out. Each activation may be treated as its own sub-sequence, so the algorithm Recursive-Set-Spacing sets the y position of  $t$  to *top* and recurses on  $t$ . Since  $a$  must have enough vertical height to encompass the sub-sequence rooted at  $t$ , the *top* indicator is increased to include the height of  $t$  (which will have its height set inductively). The final height of the activation is set at line 34, and the height of the lifeline is set to

encompass the height of  $a$  at line 35.

The algorithm Recursive-Set-Spacing also ensures that there is enough horizontal space to allow self-calls on lifelines. The current number of calls to a lifeline (the call depth) is recorded between lines 3 and 7.

After the distances have been set between lifelines by Recursive-Set-Spacing, their final x positions and widths are set in lines 11 through 22 of Layout-Sequence-Diagram (Listing D.1). This is done by iterating through the lifelines, making sure that there is enough space for each lifeline's label, the maximum number of activations that are going to be displayed on the lifeline ( $maxCallDepth$ ), and the  $distance(l, m)$  set by Recursive-Set-Spacing. Note that the width of the labels for adjacent lifelines may be greater than the total amount of distance required between them to avoid occluding message names. Line 17 checks for this condition, and line 18 minimizes the total distance between lifelines, and hence minimizes the length of edges representing messages while also removing label occlusion from the diagram. This allows us to achieve the goal of our algorithm.

However, before the layout can be completed, the x positions of the activations must be set. Recursive-Apply-Layout algorithm (Listing D.3) set x position for an activation to the center x position of its lifeline, plus its call depth on the lifeline (line 3). The call depth was set in the algorithm Recursive-Set-Spacing.

This concludes the algorithm. There are a number of small details left out. It is assumed that lines representing messages between activations can be laid-out in  $O(1)$  time by anchoring them to their source and target activations (this, in fact, is how it is achieved in the Diverimplementation). In our pseudo-code, we ignore the widths of activations, and the vertical distance between calls is only 1 using the algorithms as indicated here. Constants can be used to give appropriate widths and vertical distances. Using such constants would require slightly more complex arithmetic in the algorithms, though, so they are left out for the sake of simplicity. We also exclude the layout of combined fragments. The position of combined fragments can be set by making relatively minor modifications to the loop found at line 10 in Listing D.2. Diver includes a full Java implementation of this algorithm.

## D.3 Analyzing the Layout Algorithm

Demonstrating that the layout algorithm is  $O(n)$  where  $n$  is the total number of components (widgets) in the sequence diagram is relatively straight-forward. However, the algorithm makes use of several supporting procedures that must be explained first:

- *textwidth(label)*: calculates the horizontal space required to display the textual label. This can be approximated in  $O(1)$  using the number of characters in the label, times the average width of a character.
- *distance(l, m)*: set or get the horizontal distance between the lifelines  $l$  and  $m$ .  $distance(l, m) = distance(m, l)$ . This procedure can be implemented in  $O(1)$  time in several different ways. One possible way is to create a hash function such that  $hash(l, m) = hash(m, l)$  and store a numerical distance in a hash table, which would allow getting/setting  $distance(l, m)$  in amortized  $O(1)$  time. Alternatively, since the algorithm is concerned only with finding the distance between adjacent lifelines, we can duplicate the list of lifelines  $L$  as a linked structure, such that for each  $L[i], L[i + 1]$  ( $0 \leq i < |L| - 1$ ), there is an edge  $e = (L[i], L[i + 1])$  annotated with the distance  $distance(L[i], L[i + 1])$ . Then, create *left* and *right* pointers to  $e$  on the lifelines in  $L$  such that  $L[i].right = e = L[i + 1].left$ . In this way,  $distance(L[i], L[i + 1])$  may be set or retrieved in constant time by accessing the *left* or *right* pointers on the lifeline.

Since these two supporting procedures can be run in  $O(1)$  time, we are only really concerned with the loops and the recursions. Arithmetic statements, variable references, and array indexing can also be run in  $O(1)$  time.

There are two kinds of loops that occur in these algorithms. The algorithm Layout-Sequence-Diagram loops over the lifeline list  $L$  twice, and the algorithms Recursive-Set-Spacing and Recursive-Apply-Layout both iterate over messages, and recurse on activations. The iterations over  $L$  are easiest to analyze, so we will deal with them first.

Both of the loops (lines 3 to 5 and lines 11 to 22 in Listing D.1) iterate at most  $|L|$  times because the looping conditions define the invariant  $i < |L| - 1$  and  $i$  is not changed by the loop. The body of each loop is run in  $O(1)$  time. Since the lifelines contained in  $L$  are a subset of all of the components in the sequence diagram  $|L| \leq n$  and each loop is  $O(n)$  (by the definition of  $O$ ). Both of the loops combined, then, are at most  $O(2n) = O(n)$  (also by the definition of  $O$ ).

The other loops to be concerned with are the ones found in Recursive-Set-Spacing (Listing D.2) and Recursive-Apply-Layout (Listing D.3). Both of them have the same form, which is easiest to see between lines 4 and 8 of Listing D.3. These loops iterate over each message originating at an activation  $a$  doing a little bit of  $O(1)$  work for each message. If the message is a call, then the algorithm recurses on the target activation,  $t$ , of the message. These recursions will certainly terminate. Figure B.3 in Appendix B shows that an activation has exactly one caller, so an activation cannot be reached by more than one call message. Therefore, each activation in the sequence diagram will be visited at most once through these recursive calls. The loop condition causes it to iterate exactly once over each message originating at an activation (line 4 of Listing D.3). By Figure B.3, we can see that each message has exactly one originating activation. Therefore, since each activation is visited at most once, each message is visited at most once. Therefore, given that all other statements in the algorithms may run a  $O(1)$  time, these recursive calls run at most  $O(|M| + |A|)$  where  $M$  is the set of messages in the diagram and  $A$  is the set of activations. But since  $A$  and  $M$  are both subsets of the components of the diagram,  $O(|A| + |M|) = O(n + n) = O(2n) = O(n)$  by the definition of  $O$ .

We have shown that all statements, loops, and recursions in these algorithms are bounded by  $n$ : the total number of components in the sequence diagram. The time complexity of the algorithm is therefore  $O(n)$ .

## APPENDIX E

---

### User Study Documents

---

#### **E.1 Dynamic Interactive Views For Reverse Engineering (Diver) User Study Consent Form**

##### **Introduction**

Thank you for offering your participation 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 Del Myers of the CHISEL group at the University of Victoria under the supervision of Dr. Margaret-Anne Storey and by Dr. Jim Buckley of the University of Limerick in Ireland. 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 at [mstorey@uvic.ca](mailto:mstorey@uvic.ca) or Del Myers at [delmyers.cs@gmail.com](mailto:delmyers.cs@gmail.com). For more information about our research group, please see our website: <http://www.>

thechiselgroup.org.

### **What is Involved**

This study involves the performance of two 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 perform a brief interview after you have completed the tasks. 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 Del Myers at [delmyers.cs@gmail.com](mailto:delmyers.cs@gmail.com).

### **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 and Dr. Jim Buckley. 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. 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 gift card in the amount of \$25 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://diver.sf.net>. 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 Del Myers at [delmyers.cs@gmail.com](mailto:delmyers.cs@gmail.com) or Dr. Margaret-Anne Storey at [mstorey@uvic.ca](mailto:mstorey@uvic.ca). If you have questions or concerns about the ethical implications of this study, you may contact the University of Victoria Human Research Ethics Office at [ethics@uvic.ca](mailto:ethics@uvic.ca). This study is part of a larger research investigation titled Reverse Engineered Sequence Diagrams to Support Software Evolution.

## **E.2 General Information**

In this study, we will ask you to use Diver to perform software feature location and analysis. The software that you will be analyzing is Diver itself. You have been provided an installation of Eclipse version 3.6. When you load Eclipse, you should be presented with the Diver perspective which will contain all of the views that you should need to complete your analysis. You will be given two tasks. Each task will ask you to investigate a particular functionality of the Diver tool. We ask you to try each task until you feel that you would be able to explain the functionality associated with that task. If at any point you feel that the task has become too frustrating or difficult to continue, feel free to stop and

let us know. For each task, you will be asked to write your answers to several questions so that we can gain insight into how well you have been able to comprehend the functionality under investigation. You will be asked the same set of questions for each task.

While performing each task, we ask that you attempt to use the features of Diver wherever possible. You may use any of the other features of the Eclipse IDE, but we ask that you refrain from using the Eclipse Java Search features. This includes such things as Java Element Search, Open Type (Ctrl+T), and Search for References (Ctrl+Shift+G). This will help us to narrow the scope of our investigation to the features of the Diver tool itself. This study is in a “Talk-aloud” form. We ask you to speak the thoughts that come to mind while you perform the tasks given to you. Try not to filter your thoughts. Just say what comes to mind.

You have been given a list of features of the Diver Tool. You may refer to this list whenever you like as a reminder of the features available to you. If you forget how to use a feature, you may ask the investigator about how to use it. Think of the investigator as a kind of “live documentation.” We cannot help you answer the questions.

Please turn the page to begin your first investigation.

## E.3 Available Diver Features

Feature	Purpose	Found In...
Launch Trace	Run a target application for tracing	Launch Menu
Pause/ Resume Trace	Tell Diver to either stop or start recording trace data	Debug view (pencil button)
Package Pane	Displays the packages in which the classes for each lifeline are contained	Top of sequence diagram
Clone Pane	Duplicates the sequence diagram so that more than one part of it can be seen at a time	Left-hand side of sequence diagram
Expand/ Collapse	Displays or removes the items contained in a package, activation, lifeline, loop, error handling block, or conditional block	Sequence diagram

Timeline	Displays when a trace is paused and when a selected Java element was executed during a trace	Bottom of Sequence Diagram
Search	Keyword and wildcard search within traces	Search menu
Keyboard Navigation	Navigate the sequence diagram without a mouse	Sequence diagram
Link To Source	Display the source code associated with a sequence diagram element	Sequence diagram (double click on a method call, activation, or timeline)
Activate Trace	Sets a selected trace to be "active" which causes the workbench to be filtered according to the Java elements present in the selected trace	Program Traces View (amber circle icon)
Hide Trace	Sets a selected trace to "hidden" which removes the Java elements in the "hidden trace" from the Eclipse Workbench	Program Traces View (eye icon)
Reveal (In..)	Shows a selected Java element or activation in the sequence diagram	Timeline (right click on vertical bar), Search Results View (right click on class, method, or activation)
Focus On...	Re-roots the sequence diagram to the given activation	Sequence diagram (right click on activation), Timeline (right click on vertical bar), Breadcrumb bar (select a method name)
Breadcrumb bar	Displays the method calls that lead to the root of the sequence diagram	Top of the sequence diagram
Annotate	Add a personal message to a sequence diagram element	The Eclipse Properties View

## E.4 Tasks

### E.4.1 Task 1 – Linking to Source Code

#### Instructions

Diver offers the ability to link to source code from the Sequence Diagram View. Double clicking on an element in the view will display its corresponding source code in the Java Editor (if the source code is available). You are asked to locate and analyze this functionality.

For this task, please use the software reconnaissance technique. Software reconnaissance is performed in Diver by collecting several traces of the target software. The first execution trace will exercise the functionality under investigation. The others will be similar to the first, but will not exercise the functionality. Once the traces have been captured, you may “Activate” the first trace using the Program Traces View. You can also use the Program Traces View to “Hide” the other traces, and refine the filters used in the Package Explorer. You may use the sequence diagram and the trace search facilities for your investigation.

For this investigation, please perform the following steps.

1. Use the Mylar Task List to activate the task Link To Source.
2. A launch configuration called Link To Source has been created for you. Use this configuration to capture each of your traces. You may change the default settings for the configuration if you like, but do not alter the location of the workspace data in the main tab.
3. After you have started the launch, you may use the trace called Trace Example located in the target workspace to complete this investigation. That is, there is no need to generate a trace using your “Runtime Workbench”. You only need to generate traces using your initial Eclipse instance.
4. Answer the questions on the given questionnaire.

### E.4.2 Task 2 – Exchanging Repetitions

**Instructions** Diver attempts to keep sequence diagrams down to a manageable size by com-

pacting repetitive portions of the execution. Repetitive blocks are grouped into “combined fragments” (blue boxes with the labels, “for”, “while”, or “do...while”). Only one repetition is displayed at a time. It is possible for the user to display other repetitions through a right-click context menu, however. Please investigate the way in which Diver exchanges such repetitions.

You are asked to not use the software reconnaissance technique. That is, no trace is allowed to be “Active” at any point during your investigation. You may use any of the other Diver features, however.

For this investigation, please perform the following steps.

1. Use the Mylar Task List to activate the task Exchange Repetitions.
2. A launch configuration called Exchange Repetitions has been created for you. Use this configuration to your traces. You may change the default settings for the configuration if you like, but do not alter the location of the workspace data in the main tab.
3. After you have started the launch, you may use the trace called Trace Example located in the target workspace to complete this investigation. That is, there is no need to generate a trace using your “Runtime Workbench”. You only need to generate traces using your initial Eclipse instance.
4. Answer the questions on the given questionnaire.

## E.5 Task 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 involved in the execution of the functionality?

4. Please describe how the classes/methods interact in order to perform the functionality. Please note concrete implementations (i.e. no interfaces).
5. Under what conditions do the interactions occur (i.e. what are some of the conditions that must be true for the functionality to execute)
6. Please take a screen-shot of the sequence diagram, and use your own words to explain how it describes the execution of the feature.

**Optional Questions, if you have time.**

1. Why do you think the developer implemented the functionality this way?
2. How would you implement the functionality?

## **E.6 Interview Questions**

**Note:** Interviews were free-form, and participants answers could direct the course of the interview. The following were used as a guide only.

1. Please Describe your general experience using the tools to carry out the assigned tasks. Did you have enough time to finish the tasks? Why not.
2. What helped you to complete the tasks?
3. What hindered you from completing the tasks?
4. What tool features did you find most useful?
5. What tool features did you find to be a hindrance?
6. Any suggestions for improvement?
7. On a scale from 1-5, how would you rate the software reconnaissance technique?
8. Closing remarks?

## APPENDIX F

---

### User Stories

---

Through analyzing audio/video recordings, as well as logged data about each participant's usage of the views in the Eclipse IDE, we were able to extract rich information about the ways that the participants approached the sessions. Figure 9.1 displays the usage trends for all participants between their sessions including a percentage of how much time the participants spent in each view. In this appendix, we give a more detailed description of each participant's usage of the Diver tool in order to give a rich context surrounding each participant's usage patterns and the frustrations that they encountered.

#### F.1 Participant P1

Participant P1 began the first session by generating a trace and trying to analyze the software. Before attempting to gather a second trace, he attempted two keyword searches in his first trace, both of which failed. After spending approximately ten minutes familiarizing himself with the tool and the task at hand, he proceeded to attempt to solve the prob-

lem using software reconnaissance. After recording a second trace, he used the Program Traces View to select different threads, which updated the filter in the Package Explorer. He looked in the Package Explorer for meaningful words associated with the feature under investigation. After approximately 9 minutes, he found a method of interest and relied on the sequence diagram to analyze the feature, verifying his findings using source code.

For session **S2**, P1 used various methods to locate the feature. He tried browsing the sequence diagram as well as source code. He tried two trace searches, but both failed. In the end, he found his first foothold using mnemonic reasoning and by browsing the Package Explorer. His first foothold was found 10.6 minutes after capturing an execution trace

## F.2 Participant P2

This participant began session **S1** by recording a single trace and using Diver's filter on the single trace. The Package Explorer showed the elements involved in the trace, but not unique to the feature. He noted that there is "far too much to actually look through," and so he recorded a second trace. Approximately 3.5 minutes after he completed his second trace, he was able to locate his first foothold in the Package Explorer. He verified his findings using the trace search mechanism. From then, he made use of the Package Explorer to navigate into the sequence diagram and the sequence diagram to navigate to code. He used the sequence diagram and the source code editor equally for his analysis of the behavior of the software. He finished the session in short order, using the remaining time to investigate details of the feature, such as the low-level user interface functionality. He tended to verify his findings using the trace search facility.

In the second session, participant P2 began by anticipating the difficulty of the task stating, "So much happens in Eclipse," and, "The filter was hugely beneficial." After capturing the trace, he spent several moments exploring some sequence diagrams and noted, "Expanding the sequence diagram is not going to work." He resorted to using Diver's search facilities with mixed results. After 14 minutes and several failed searches, he found a first



foothold. This behaviour differed from his first session in which he only used the search mechanism to verify his findings and not to discover new information. He also sighed numerous times during this session, a phenomenon which did not occur in the first session. After the first foothold was found, he followed his previous pattern using a combination of the sequence diagram and source code to analyze the feature.

### F.3 Participant P3

Session **S1** was started by immediately generating two traces. Participant P3 began his investigation of the traces by using software reconnaissance and the Diver filters in the Package Explorer. He selected different threads in the Program Traces View and looked at the effect that they had on the Package Explorer. After locating a method of interest in the Package Explorer, he first investigated its source code and then revealed the first call to the method in the sequence diagram. He used the sequence diagram extensively to answer the first four understanding tasks, but relied on source code to answer the fifth. It took Approximately 7.6 minutes to find his first foothold.

Participant P3 had significant difficulty with session **S2**. He started with a very large trace which took Diver a long time to analyze. After a number of minutes waiting for the analysis, he became frustrated and attempted to create a smaller trace, noting, “No real world developer is going to wait ten minutes before trying to solve a problem.” Once the smaller trace was captured, he began exploration. He attempted to turn on the Diver filters, but was instructed that it was not allowed for the second session. He attempted to expand all activations in a sequence diagram which caused it to be very large. He stated that he did not like the sequence diagram for this session and abandoned it.

## F.4 Participant P4

The participants were instructed to try and keep their traces as small as possible. However, P4 recorded extremely large traces during session **S1**. This impaired the performance of the software reconnaissance filters during the session. After becoming frustrated with the tool's performance, participant P4 stopped using the filters and resorted to using keyword searches in the trace. After several attempts, he was able to locate the feature in the sequence diagram and complete session **S1**. He attempted the same strategy for session **S2** (using keyword searches in the traces), but was unsuccessful in this instance. He was not able to locate the feature and complete the session.

## F.5 Participant P5

Participant 5 began with some confusion. He started session **S1** by gathering a single trace and trying to browse it. He was prompted to review the instructions for the session to recognize that session **S1** requires at least two traces. After reviewing the instructions, he began again and recorded two traces. His traces contained the information needed to complete the session, but he began to look for the wrong feature. After looking back at his instructions, he realized his mistake and was able to find his first foothold using the filtered Package Explorer after 5.6 minutes. Once it was found, he primarily used the sequence diagram to investigate the feature, noting it was "easy to see from the sequence diagram."

To perform session **S2**, the P5 recorded a trace and began to explore it in the sequence diagram. After a little time, he decided to use Diver's search facilities, but the search failed to return any results. He re-ran the program (without tracing) to ensure that he understood what his actions were that caused the feature to execute. He then browsed the Package Explorer to try and find classes that had meaningful names, and was not able to find any. He tried several more trace searches using Diver before he found one that yielded good results. After 10.3 minutes, he found his first foothold and started to answer the assigned questions. He relied mostly on the sequence diagram and his search history to complete

the sessions, with some reference to source code.

## F.6 Participant P6

Participant 6 began his first session with some time spent familiarising himself with the tool. After 5 minutes, he finished 2 traces and began software reconnaissance. He mentioned several times that he was tempted to do a code search, but refrained from it and used the Package Explorer and sequence diagram to find some foothold into the feature. He found it 3.5 minutes after capturing his second trace, though he said that he may have “found that by accident”, but, “not totally”. From then, he attempted to answer the questions using the sequence diagram, but had difficulty understanding the visualization and got lost several times while trying to use the tooling. Particularly, he expressed some confusion about how the Package Explorer was filtered when a thread was selected in the Program Traces View, stating, “I don’t really understand why it says, ‘Reveal in Main’, when I have ‘Modal-Context selected.’” The issue was that an anonymous inner-class declared in selected method was called in a different thread (‘Modal-Context’) than the selection (‘Main’). Although he was able to complete the questions mostly using the sequence diagram, he did not find it very intuitive.

For session **S2**, participant P6 captured a single trace and tried to activate his trace to use the filters, but was advised that it was not allowed for this session. He was quickly struck by the size of the sequence diagram, and noted that he was “really looking for a way to filter”. He tried to solve the problems using the sequence diagram, expanding many elements in it. He attempted to expand all of the elements under a particular activation, but was frustrated with how long it took and the size of the diagram after doing so. Contrary to session **S1**, he avoided performing searches either in code or using Diver, stating several times that such searches would be “random”. In the end, he attempted several searches with one gaining him a first foothold after 37 minutes of investigation. He did not have enough time to answer any questions, though, and the session was incomplete.’

## F.7 Participant P7

Participant P7 was a junior programmer with less than one year of experience. During his attempt at session **S1**, it was evident to the investigator observing the experiment that he had great difficulty understanding the software reconnaissance process and the design of Diver. Diver requires that each execution trace be captured within separate executions of the target application. Participant P7 consistently tried to capture everything within a single execution. In other words, he was unable to perform software reconnaissance because all of his captured traces included the feature under investigation as well as the features that were not part of the session. Seeing that the participant was having extraordinary difficulty understanding the process, the investigator attempted to re-train the participant, but the participant was unable to complete the session within the time constraints. However, having been given the extra training, he was able to complete session **S2**.

## F.8 Participant P8

Participant P8 attempted software reconnaissance, but approached the problems differently than most participants. During **S1**, he started by gathering two traces and filtering based on them. However, he never used the Package Explorer to navigate to sequence diagrams. He navigated to source code with the Package Explorer, or he opened sequence diagrams using the Program Traces View. While browsing the Package Explorer, he never expanded the tree representation beyond the file level. During the interviews, he noted that this was a learned behaviour. He knew that the Package Explorer was filtered, but he, from past experience, still expected it to be very large so he tended to avoid it. He spent most of his time either reading source code or sequence diagrams. He was never able to find a foothold into the feature.

For **S2**, participant P8 gathered a single execution trace and primarily used a browsing strategy in the sequence diagrams. He was able to gain a first foothold after approximately two minutes. He attributed this to learning effects. He said that during session **S1**, he spent

a lot of time browsing sequence diagrams and learned that “Worker threads really are just workers.” That is, those threads with the name “Worker” were likely uninteresting, so he avoided them during his investigation and was able to narrow his search.

## **F.9 Participant P9**

Participant P9 began his investigation by capturing two traces, and using a combination of the sequence diagram and source code. After several minutes, the investigator noticed a bug in the software, and the participant had to capture a new trace. After that third trace was captured, the participant filtered the Package Explorer to look through the sequence diagram and source code. He was very thorough and was even able to recognize when the feature’s functionality was spread across multiple threads (this feature executes in 2 threads, but for us to consider it completed, we only ever required the participants to find one of the threads). One point of confusion for him, though, was that he wasn’t always able to keep things in the source code editor synchronized with the sequence diagram. He expressed that he would like to navigate from the source code directly into the sequence diagram, rather than using the Package Explorer as intermediary.

He experienced much more difficulty in the second session. After some initial difficulty in the sequence diagram, he resorted to searching, but the searches failed. He then resorted to exclusively browsing source code. He expressed that he was aware that the session involved invoking an action from the context menu, and that Eclipse developers often use terms like “action” and “command” when implementing context menu items, so he started browsing for packages that had those words. He eventually found a class and he started reading through pages of source code. After some browsing of source code, he ran a trace search for some of the keywords that he had found in the source. Then he tried to understand the software using the sequence diagram. He expressed that what he was seeing in the sequence diagram did not match what he thought he understood from reading the code. He quit the session before his time was exhausted, without finding a foothold or

completing any of the understanding tasks.

## F.10 Participant P10

Participant P10 started his investigation in session **S1** by launching the target application and familiarizing himself with the problem by trying various different interactions (opening a sequence diagram, swapping loops several times, etc.). He then captured his first trace in which he recorded a swap of a loop (he was investigating feature F2) as well as several mouse clicks, selections, etc. During his second trace, he tried several mouse clicks in the sequence diagram, but did not exercise any functionality that would cause the sequence diagram to draw any figures. He performed software reconnaissance, by hiding the second trace. When he was unable to find a foothold after several minutes, he attempted several searches, all of which failed. He spent much of his time, then, trying to browse source code and sequence diagrams to find something of interest. Unable to find anything, he decided to start his investigation again by capturing two more traces. While investigating the results of those traces, he “hid” both of the traces from his previous investigation which, in fact, filtered out the software elements involved in feature F2. He never found a foothold.

This participant was able to find a foothold for session **S2** in under eight minutes. To gather the trace for **S2**, he exercised the feature under investigation several times, pausing the trace for several seconds between each time he exercised the feature. He indicated that this was so that the timeline would contain “markers” that displayed where he exercised the function (Diver indicates that a trace was paused using a yellow marker in the timeline). After capturing his trace, he opened up a sequence diagram, and noticed that there was a blue marker in the timeline because he had a method selected in the Package Explorer. He right-clicked on the marker to reveal what method was called. He then browsed the expanded sequence diagram and after several minutes found a class of interest, which was his first foothold into the feature. He then proceeded to complete the understanding tasks using a combination of the sequence diagram, source code, and search.

## APPENDIX G

---

### Diver: Dynamic Interactive Views for Reverse Engineering (User Survey)

---

This is a survey for the research project Dynamic Interactive Views for Research Engineering (Diver). Diver is a set of tools that brings reverse engineering techniques into your Eclipse IDE. This survey is meant for users of Diver. It will help us to understand your needs as a software developer and it will be used to aid in future development of reverse engineering and IDE tools.

By participating in this study, you are agreeing that your responses may be published in academic works such as conference proceedings, journal papers, and thesis work. All data published will be aggregated and anonymous so that your identity will not be discernible from the published works.

All of your responses are anonymous unless you choose to give us your email address in order to stay in contact with us and be informed about the latest Diver research and development. In this case, your contact information will not be shared outside of the CHISEL group at the University of Victoria unless you give us explicit permission to do so.

Welcome! Thank you for coming to participate in our survey. This survey is designed to help us to better understand how you use the Diver tools. It will guide us in improving the software development process for users of integrated development environments. Please take a few minutes to fill out this survey and aid us in our research. There are 26 questions in this survey

## **General Questions**

These questions will be used to gain an understanding of your general programming and reverse engineering experience. It will help us to interpret our results.

### **How would you describe your programming/development work?**

Please choose all that apply:

- ☐ Software developer
- ☐ Software architect/engineer
- ☐ Software tester
- ☐ Manager
- ☐ Academic: researcher or instructor
- ☐ Academic: graduate student
- ☐ Academic: undergraduate student
- ☐ Hobbieist
- ☐ Other: \_\_\_\_\_

### **How many years of software development experience do you have?**

Please choose only one of the following:

- ☐ < 1 year
- ☐ 1-4 years
- ☐ 5-9 years
- ☐ 10-15 years
- ☐ > 15 years



**What is your primary programming language?**

Please choose only one of the following:

- ☐ Java
- ☐ C/C++
- ☐ C#
- ☐ Javascript/CSS
- ☐ Lisp/Scheme
- ☐ Python
- ☐ Perl
- ☐ Ruby
- ☐ Scala
- ☐ PHP
- ☐ Other \_\_\_\_\_

**What operating systems do you primarily develop for?**

Please choose all that apply:

- ☐ Windows
- ☐ MacOS
- ☐ Linux
- ☐ Solaris
- ☐ Other: \_\_\_\_\_

**How many years of experience do you have programming using the Eclipse IDE?**

Please choose only one of the following:

- ☐ < 1 year
- ☐ 1-3 years
- ☐ 4-6 years
- ☐ 7-9 years

**Reverse Engineering**

These questions are used to discern the different ways that you use reverse engineering to understand the software you are working with.

## Static Analysis

The following questions are about static analysis in reverse engineering. This includes methods of understanding your programs by inspecting source code, or compiled machine code. That is, static analysis deals with the static structure or definition of the software, not how it behaved/is behaving at run-time.

### What static analysis techniques do you use?

Please number each box in order of preference from 1 to 8

- ☐ Browsing source code
  - ☐ Searching source code (using queries or search dialogs)
  - ☐ Reading documentation
  - ☐ Tracing documentation to source code implementation
  - ☐ Using static slices
  - ☐ Generating diagrams from source code
  - ☐ Dependency analysis
  - ☐ Class hierarchy analysis
- 
- Browsing code includes actions such as reading source codes and comments, and following method invocations while reading source code
  - Searching code includes actions such as using wild-card searches to find source code elements that match different patterns. The Eclipse Open Type dialog would be an example of a source code search.
  - Reading documentation includes any documentation artifacts such as API or design diagrams
  - Tracing documentation to source includes processes which involve trying to discover where feature requirements in documentation are implemented in source code
  - Static Slices are portions of source code that may be executed to affect a certain outcome. For example: all of the source code that may change the value of a variable. There are various tools which search code for static slices; it is not a typical IDE feature
  - Generating diagrams from source code includes using tools which examine source code and attempt to generate graphical representations of the interactions in the source code such as sequence diagrams, or architecture diagrams
  - Dependency analysis involves using tools which expose dependencies between software artifacts, such as plug-in dependencies, call dependencies, and containment dependencies
  - Class hierarchy analysis is involved in object oriented programming languages, in which it is useful to understand what classes implement various interfaces or extend various other classes

---

**Please describe any other static analysis techniques that you may use**

Please write your answer here:

**Dynamic Analysis**

The following questions are designed to give us insight into how you use dynamic analysis in your reverse engineering. Dynamic analysis pertains to processes which use program run-time information in order to aid in understanding. These include memory dumps, trace logs, etc.

**Please rank the following dynamic analysis techniques in order of importance**

Please number each box in order of preference from 1 to 9

- ☐ Generating diagrams from program traces
- ☐ Program profiling
- ☐ Stepping through code using a debugger
- ☐ Memory/heap analysis
- ☐ Code coverage
- ☐ Test-case analysis
- ☐ Print statements
- ☐ Dynamic slices
- ☐ Dynamic dices

- Generating diagrams from program traces includes graphical representations of how a program ran such as sequence diagrams or state diagrams

- Program profiling includes methods such as logging program events and enumerating their occurrences or recording the time that it takes to execute them
- Stepping through code using a debugger is the standard technique found in most IDEs
- Memory/heap analysis includes saving the program's memory state at a particular point in execution and analyzing its content
- Code coverage includes running a piece of software to check which parts of the source code were executed
- Test-case analysis includes creating and running test cases to find out whether the program behaves as expected
- Print statements are an ad-hoc method of printing out program state in selected parts of the program
- Dynamic slices are much like static slices. They expose source code that was involved in affecting the value of a variable, but only for a particular run or series of runs for the program
- Dynamic dices are like slices. They are a set difference between code that was run to produce a correct value in a variable and code that was run to produce an incorrect value in a variable

### **Please indicate other dynamic analysis techniques that you use**

Please write your answer here:

## **Sequence Diagrams**

Diver offers support for analyzing your software using sequence diagrams. These questions are designed to help us understand the way that you used these sequence diagrams.

---

**Before using Diver, how would you rate your knowledge of UML sequence diagrams?**

Please choose the appropriate response for each item:

	None	Limited	Good	High
Experience with sequence diagrams	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

**How long have you been using Diver?**

Please choose only one of the following:

- ☐ < 1 week
- ☐ 1-2 weeks
- ☐ 2-3 weeks
- ☐ 3-4 weeks
- ☐ 1-2 months
- ☐ > 2 months

**Please describe any extra features that you would like to see in the sequence diagram**

Please write your answer here:

**Please describe a scenario in which you would have used sequence diagrams before you used Diver**

Please write your answer here:

**The Diver sequence diagram offers many features that are designed to help you navigate through large call traces. Please rate the following features by the level of importance they have in aiding you to navigate the sequence diagram and understand software behavior. 0 represents not important at all; 5 represents essential.**

Please choose the appropriate response for each item:

	1	2	3	4	5
Collapsing/expanding activation boxes	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Collapsing/expanding lifelines	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Collapsing/expanding combined fragments (loops/try-catch blocks/conditionals, etc.)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Swapping loop iterations	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Selecting elements (i.e. clicking on items in the sequence diagram)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Focusing on an activation box	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Moving through the call hierarchy using the breadcrumb bar	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Navigating to source code using the sequence diagram	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The sequence overview	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The "clone" pane	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

**The Diver sequence diagram offers numerous visual features that are designed to enhance your experience and understanding. Please rate the following features based on how useful they are in aiding your understanding of the sequence diagram. 0 represents not at all useful; 5 represents essential.**

Please choose the appropriate response for each item:

	1	2	3	4	5
Sequence diagram layout	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Colours	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Label names	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Label placement	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Icons	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Animated layouts	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Combined conditional fragments (if/else, etc.)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Combined error handling fragments (try/catch)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Combined loop fragments (while, for, do..while)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Compacted loop fragments (displaying only one iteration at a time)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Automated saving of the view state	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

**The sequence diagram viewer includes a timeline that allows you to filter messages in the sequence diagram based on time, as well as reveal activations of methods or open the source code of a method. Please rate the following timeline features in terms of how useful they were in aiding you in your understanding of the sequence diagram and your software. 0 represents not at all useful; 5 represents essential.**

Please choose the appropriate response for each item:

	1	2	3	4	5
Filtering the sequence diagram based on time	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Revealing activations using the timeline	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Focusing on activations using the timeline	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Opening source code from the timeline	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Visualizing times at which the trace was paused	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Visualizing method and class activations in time	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

## Diver IDE Integration

This group of questions deals with questions concerning the best ways to integrate reverse engineering techniques into an IDE. You will be asked to rate various features of the enhanced IDE based on how well they helped to integrate reverse engineering into your normal work.

**Diver includes a number of features related to launching, running, and analyzing your software. Please rate the following features based on their usefulness. 0 represents not at all useful; 5 represents essential.**

Please choose the appropriate response for each item:



	1	2	3	4	5
Trace launch configurations (Java Application Trace and Eclipse Trace Launches)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
The trace configuration tab (Setting analysis filters before a launch)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Pausing and restarting traces	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Viewing the launch time dialog	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Reanalyzing traces (resetting filters after a launch)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Tracing during debug sessions (eg. resuming a trace when a breakpoint is hit)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

**Diver includes a number of features that attempt to unify the views and editors in your IDE to help expose pertinent parts of software with respect to the traces that you have run. Please rate the following features based on their usefulness. 0 represents not at all useful; 5 represents essential.**

Please choose the appropriate response for each item:

	1	2	3	4	5
Combining views into the Diver perspective	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Organizing traces in the Program Traces View	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Filtering the package explorer based on the active trace	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Excluding previous traces from the current filter (i.e. using the "eye" icons)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Revealing activations in the sequence diagram using the package explorer	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Annotating elements of the trace (using the properties view)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Displaying executed code in the Java editor	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

## **Closing remarks**

Here, you have the opportunity to offer any closing remarks concerning Diver. You may offer more insight into what was useful about Diver and what may be improved.

### **Please describe how Diver helped you in your work.**

Please write your answer here:

### **Please indicate how often you used Diver. What did you use Diver for?**

Please write your answer here:

### **Please describe ways in which Diver may have hindered your work.**

Please write your answer here:

**Are there any enhancements or new features that you would like to see in Diver as a whole?**

Please write your answer here:

**Any closing remarks?**

Please write your answer here:

**Would you like to stay in contact with us? Please give us your email address.**

Please write your answer here: \_\_\_\_\_

Note: email addresses will be used for Diver related correspondence only. We will not communicate your email address outside of the CHISEL group at the University of Victoria unless you give us explicit permission to do so. By giving us your email you agree that we may assume that any correspondence from the given email address may be reasonably assumed to originate from you, the user of this survey.