Evaluating Dynamic Analysis Techniques for Program Comprehension

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus Prof. dr. ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op dinsdag 23 juni 2009 om 14:00 uur

door

Sebastiaan Gijsbert Marinus CORNELISSEN

doctorandus informatica
geboren te Amsterdam

Dit proefschrift is goedgekeurd door de promotor:

Prof.dr. A. van Deursen

Copromotor: dr. A.E. Zaidman

Samenstelling promotiecommissie:

Rector Magnificus	voorzitter
Prof.dr. A. van Deursen	Technische Universiteit Delft, promotor
Dr. A.E. Zaidman	Technische Universiteit Delft, copromotor
Dr.ing. L.M.F. Moonen	Simula Research Laboratory, Norway
Prof.dr.ir. J.J. van Wijk	Technische Universiteit Eindhoven
Prof.dr. M.J. de Vries	Technische Universiteit Delft
Prof.dr. T. Systä	Tampere University of Technology, Finland
Prof.dr. S. Ducasse	Institut National de Recherche en Informatique et en Automatique, France

The work in this thesis has been carried out at the Delft University of Technology, under the auspices of the research school IPA (Institute for Programming research and Algorithmics). The research was supported by the Netherlands Organization for Scientific Research through the Jacquard Reconstructor project (grant no. 638.001.408).

IPA Dissertation Series 2009-13

ISBN 978-90-799-8203-5

Depicted in the bottom right portion of the cover page is *The Thinker*, a famous sculpture by Auguste Rodin.

This dissertation was printed by Wöhrmann Print Service, and was typeset using the LaTeX Document Preparation System.

Acknowledgments

This is probably one of the parts I like best: putting on my nostalgic cap and looking back at the past few years, thinking of memorable events, and thanking certain people who "enabled" me in one way or another.

My utmost gratitude goes to my promotor, Arie van Deursen, whose patience and constructiveness enabled me to write this dissertation. Similar to my Master's thesis back in Amsterdam (where I first met Arie) I had a slow start, as initially I found it difficult to get a sense of direction – Surely, if so much research already exists in this area, how can there be room for my own contributions?!

Fortunately, Leon Moonen and my co-promotor Andy Zaidman turned out to be important factors in the past four years, as I could profit from Leon's experience in software engineering research in general and from Andy's expertise with dynamic analysis in particular. Thanks for teaching me how to do research and how to write it down. Andy, I'll be expecting an invitation to your daughter's Ph.D. defense roughly 25 years from now!

With respect to my publications, I would like to acknowledge the co-authors for their significant contributions over the years. In chronological order they are Bas Graaf, Danny Holten and Jack van Wijk (the visualization gurus from Eindhoven), Rainer Koschke (the German veteran), and Bart Van Rompaey. It's been a pleasure working with you guys! Thanks also to the 34 colleagues and students who participated in my final experiment, which in total robbed them of 72 hours of their precious time.

Aside from working on my research, several colleagues have made the past four years pleasant, funny, or downright hilarious, most notably Cathal Boogerd (the most relaxed housemate ever), Rui Abreu (whose many family names are beyond the scope of this section), and Marius Marin (whose shades have undoubtedly remained open to the present day). Let's continue our warrior tradition of eating spare ribs on a regular basis! I'd also like to mention the (platonic?) *ménage à trois* in room 08.080, consisting of Éric Piel, Michaela Greiler, and Albertini González (bring on the *lomo*!).

There are many more colleagues that deserve mentioning, including Gerd Gross, Ali Mesbah, Peter Zoeteweij, ... but the research group has grown so rapidly that I can barely keep up. (Most seem quite addicted to coffee, by the way – I don't know which is worse, a defective printer or a broken coffee machine.) Thanks also to Marco Dekker and Cathal for protecting me from my evil opponents during the defense!

Having saved the most important people for last, I would now like to thank my family. I am ever grateful to my parents for encouraging me to accept this challenge and the ones before, and to my brother Thijs for demonstrating how it should be done. Thijs, this dissertation contains fewer up quarks and interesting-looking formulas than yours, but the colorful images hopefully make up for it.

My final thanks go to a special someone who has been the subject of my reverse engineering efforts for several years. Miriam, thank you for your love and your support.

Bas Cornelissen
May 1, 2009
Delft

Contents

List of Acronyms

AOP Aspect-Oriented Programming

AST Abstract Syntax Tree

DOPG Dynamic Object Process Graph

DSDM Dynamic Systems Development Model

Extravis Execution Trace Visualizer

GQM Goal-Question-Metric

GUI Graphical User Interface

IBAA Importance-Based Anti-Aliasing

IDE Integrated Development Environment

JRET Java Reverse Engineering Tool

(k)LOC (kilo-)Lines of Code

NCD Normalized Compression Distance

NID Normalized Information Distance

OO Object-Oriented

SDR Sequence Diagram Reconstruction

TPTP Eclipse Test & Performance Tool Platform

UCM Use Case Map

UML Unified Modeling Language

XML Extensible Markup Language

XP Extreme Programming

Introduction

Chapter 1

Program comprehension is an essential part of software development and software maintenance, as software must be sufficiently understood before it can be properly modified. One of the common approaches in getting to understand a program is the study of its execution, also known as dynamic analysis. While many such approaches have been proposed in the literature, their empirical evaluation is, as we will see, often missing or inconclusive, and their added values to existing solutions are seldomly quantified.

This dissertation aims to characterize, and extend upon, the existing work on program comprehension through dynamic analysis. A strong emphasis in our research is put on empirical evaluation. To structure the current state of the art and to identify research opportunities, we first conduct a systematic survey of all publications on this topic in the past decades. We then propose both a traditional and a more advanced visualization technique, discuss the necessary abstraction techniques, and provide tool implementations. Our approaches are validated through extensive case studies and a controlled experiment. Finally, we conclude with a set of recommendations for future directions.

Software evolution has become an increasingly important aspect of the software development process. As software systems grow larger and their development becomes more expensive, they are constantly changed rather than rebuilt from scratch. The same holds for legacy systems, which in spite of their age are often essential in industry (Bennett, 1995); Moreover, it is argued that if no action is taken, evolving systems tend to become increasingly complex (Lehman and Belady, 1985).

As a result of these developments, a great deal of time and money is spent on performing *maintenance* activities. Software maintenance comprises a broad spectrum of activities: a common distinction is made between adaptive, perfective, corrective, and preventive maintenance (Lientz and Swanson, 1980). The focus in this thesis is on adaptive maintenance, which involves such activities as the addition of new functionalities and the modification of existing ones.

1.1 Program Comprehension

In order to properly maintain a software system, it must be sufficiently understood by its maintainers. If this knowledge is not readily available, they are faced with the challenging task of gaining an understanding of the system's inner workings. This process is known as *program comprehension*, which Biggerstaff et al. (1993) define as follows:

> *"A person understands a program when he or she is able to explain the program, its structure, its behavior, its effects on its operation context, and its relationships to its application domain in terms that are qualitatively different from the tokens used to construct the source code of the program."*

Following this definition, one should understand that `int z = x + y` actually corresponds to the addition of two numbers.

Program comprehension is at the center of this thesis, and typically comprises the study of such artifacts as source code and documentation. However, as dealing with source code involves a mental mapping between the system's code and its behavior, large amounts of source code are difficult to interpret directly because they result in a *cognitive overload* on the part of the maintainer. Furthermore, software documentation is often incomplete, outdated, or non-existent at all. As a consequence, program comprehension is a rather time-consuming activity: the literature reports that up to 60% of the software engineering effort is spent on understanding the software system at hand (Fjeldstad and Hamlen, 1979; Corbi, 1989; Pigoski, 1997).

1.2 Dynamic Analysis

Since program comprehension is so expensive, the development of techniques and tools that support this activity can significantly increase the overall efficiency of software development. The literature offers many such techniques: examples include execution trace analysis, architecture reconstruction, and feature location (an activity that involves linking functionalities to source code). Most approaches can be broken down into static and dynamic analyses (and combinations thereof).

Static approaches typically concern (semi-)automatic analyses of source code. An important advantage of static analysis is its completeness: a system's source code essentially represents a full description of the system. One of the major drawbacks is that static analyses often do not capture the system's behavioral aspects: in object-oriented code, for example, occurrences of late binding and polymorphism are difficult to grasp if runtime information is missing.

The focus of this thesis, on the other hand, is *dynamic analysis*, which concerns a system's runtime execution. It is defined by Ball (1999) as *"the analysis of the properties of a running software system"*. A specification of the properties at hand has been purposely omitted to allow the definition to apply to multiple problem domains. Figure 1.1 shows an overview of the main steps in dynamic analyses: they typically comprise the analysis of a system's execution through interpretation (e.g., using the Virtual Machine in Java) or instrumentation (e.g., using AspectJ (Kiczales et al., 2001)). The resulting data can be used for such purposes as reverse engineering and debugging, often in the form of execution traces. Program comprehension constitutes one such purpose, and over the years, numerous dynamic analysis approaches have been

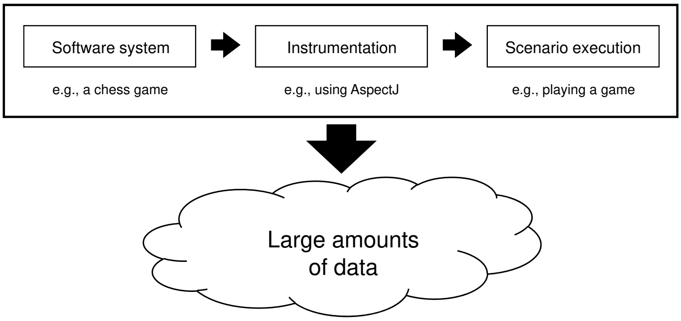


Figure 1.1 Principal steps of dynamic analysis.

proposed in this context, with a broad spectrum of different techniques and tools as a result.

Since the definition of dynamic analysis is rather abstract, we shall elaborate on the benefits and limitations of dynamic analysis for program comprehension in particular. The advantages that we consider are:

- The *precision* with regard to the actual behavior of the software system, for example, in the context of object-oriented software software with its late binding mechanism (Ball, 1999).

- The fact that a *goal-oriented strategy* can be used, which entails the definition of an execution scenario such that only the parts of interest of the software system are analyzed (Koenemann and Robertson, 1991; Zaidman, 2006).

The drawbacks that we distinguish are:

- The inherent *incompleteness* of dynamic analysis, as the behavior or execution traces under analysis capture only a small fraction of the usually infinite execution domain of the program under study (Ball, 1999). Note that the same limitation applies to software testing.

- The difficulty of determining which *scenarios* to execute in order to trigger the program elements of interest. In practice, test suites can be used, or recorded executions involving user interaction with the system (Ball, 1999).

- The *scalability* of dynamic analysis due to the large amounts of data that may be produced by dynamic analysis, affecting performance, storage, and the cognitive load humans can deal with (Zaidman, 2006).

- The *observer effect*, i.e., the phenomenon in which software acts differently when under observation, might pose a problem in multithreaded or multi-process software because of timing issues (Andrews, 1997).

In order to deal with these limitations, many techniques propose abstractions or heuristics that allow the grouping or program points or execution points that share certain properties, which results in more high-level representations of software. In such cases, a trade-off must be made between recall (are we missing any relevant program points?) and precision (are the program points we direct the user to indeed relevant for his or her comprehension problem?).

1.3 Evaluating Dynamic Analysis Techniques

In the literature, dynamic analysis techniques for program comprehension are evaluated in different manners. The findings of an extensive study that we conducted on this topic (discussed in Chapter 2) suggests a distinction between six evaluation types:

1. *Preliminary evaluations:* these involve toy examples, e.g., relatively small systems or traces.
2. *Regular evaluations:* these typically concern anecdotal evidence gathered through the study of a medium- or large-scale open source system.
3. *Industrial studies:* evaluations of this nature involve actual systems from industry.
4. *Comparisons:* the authors' approach is compared side-by-side with one or more existing solutions.
5. *Involvement of human subjects:* the measurement of a technique's impact from a cognitive point of view, e.g., through controlled experiments.
6. *Quantitative evaluations*: the assessment of a techniques' quantitative aspects, such as speed or recall.

In spite of their importance, industrial case studies, comparisons, and controlled experiments are rather uncommon in the literature. While industrial studies are indeed difficult to initiate because of several factors (discussed in Chapter 2), the latter two evaluation types certainly merit additional consideration for the following reasons.

Comparisons are important because they clearly highlight the improvements over existing work. While it is reasonable for a novel technique to be initially evaluated by itself through a case study, the next step must be to assess its added value to conventional solutions. Such an assessment should comprise measurements of both qualitative and quantitative aspects. As an example, consider a technique for the reduction of large execution traces: here, one should measure quantifiable aspects such as the speed and the reduction rate, but also the degree to which the reduced trace actually supports humans during program comprehension.

The cognitive impact of a program comprehension technique is ideally measured through the involvement of human subjects. This is often a difficult task, especially because a reasonable number of appropriate subjects is not easily found (Di Penta et al., 2007). Nevertheless, such an evaluation is vital in assessing the practical usefulness of a novel technique: examples include questionnaires that enable actual developers to give their opinions on the results of a technique, and controlled experiments, which allow for the accurate measurement of such a technique's added value (Wohlin et al., 2000).

1.4 Challenges & Research Questions

The field of program comprehension through dynamic analysis shows the development of numerous techniques every year, but the evaluations are often limited: they typically concern one case study, and most subfields exhibit little emphasis on comparisons. Therefore, our investigations into both existing and new techniques shall exhibit a strong focus on solid evaluations and comparisons. Specifically, in this thesis we distinguish a characterization of existing work, the study of a traditional visualization, and the proposition and evaluation of an advanced visualization technique.

1.4.1 Characterization of existing work

Program comprehension through dynamic analysis has been a popular research area for several decades, which has led to a large research body that distinguishes multiple subfields. The lack of a broad overview of the state of the art has made it increasingly difficult for researchers to identify future opportunities. There exist several surveys on this topic (Pacione et al., 2003; Hamou-Lhadj and Lethbridge, 2004; Greevy, 2007; Reiss, 2007), but they (1) do not constitute systematic approaches, which hinders the reproducibility of their results, (2) do not utilize common evaluation or characterization criteria, making it difficult to structure their collective outcomes, and (3) have restricted scopes rather than broad perspectives. This motivates us to conduct a literature survey that concerns the use of dynamic analysis in program comprehension contexts: an inventory and characterization of the research efforts to date will enable the comparison of existing work, and assists researchers in such tasks as finding related work and identifying new research opportunities.

Unfortunately, the design of such a survey is rather complicated as there exists no keyword standard within the field, and because the usefulness of search engines and the quality of abstracts is reportedly low in software engineering literature (Brereton et al., 2007). We thus formulate our first research question as follows:

Research Question 1

> *How can we structure the available literature on program comprehension and dynamic analysis, and which topics need additional consideration the near future?*

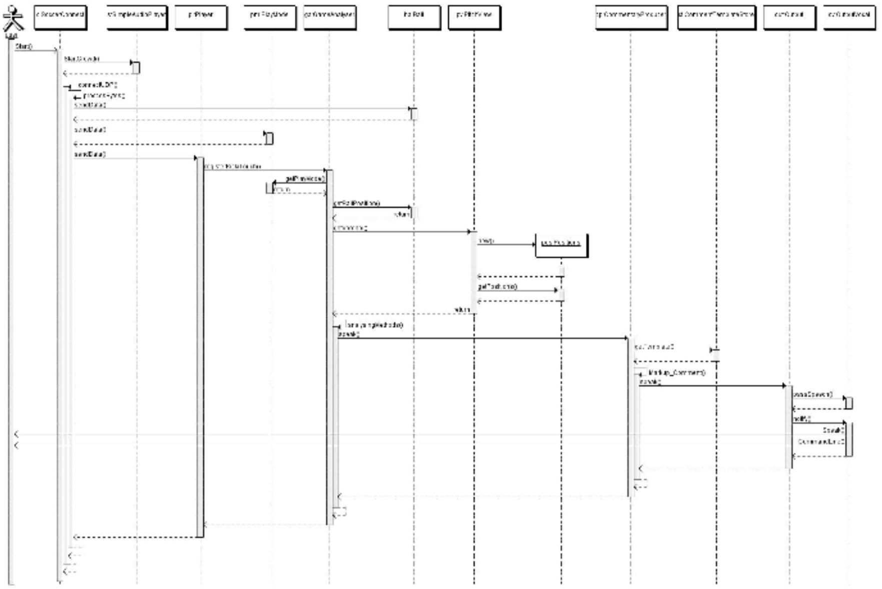


Figure 1.2 Example of a traditional visualization technique.

1.4.2 Suitability of traditional visualizations

Visualization is a popular means to convey information to the user. In particular, information gathered through dynamic analysis benefits greatly from visualization because of the large amounts of data that are involved: Reiss and Renieris, for example, report on an experiment in which one gigabyte of trace data was generated for every two seconds of executed C/C++ code or every ten seconds of Java code (Reiss and Renieris, 2001). Among the candidate visualization techniques in our context are UML sequence diagrams (OMG, 2003) (Figure 1.2). While these have been proposed in the dynamic analysis literature on several occasions (e.g., De Pauw et al. (1993); Koskimies and Mössenböck (1996); Briand et al. (2006)), they have not been extensively evaluated, with the exception of a recent article on the actual use of tool features for such diagrams (Bennett et al., 2008).

Furthermore, as dynamically reconstructed sequence diagrams inevitably suffer from scalability issues because they can only display limited numbers of events, any abstractions techniques used in these diagrams should be thoroughly evaluated, preferably in actual maintenance contexts. Unfortunately, the different techniques in the literature are generally not evaluated (1) in the same software engineering contexts, (2) by the same evaluation criteria, and (3) on the same test sets, which hinders the generalizability of their results.

Therefore, with respect to visualization, our initial focus shall be on the implementation and evaluation of reconstructed UML sequence diagrams and on the usefulness of the necessary abstraction techniques.

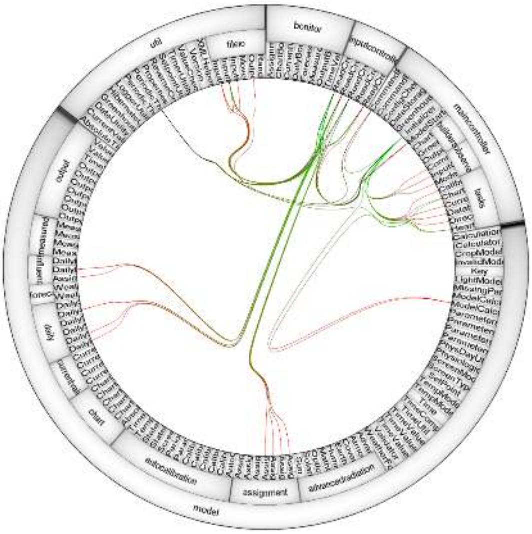


Figure 1.3 Example of an advanced visualization technique.

Research Question 2

> *Given the excessive amounts of data involved in dynamic analysis, are abstraction techniques sufficient to render traditional visualizations such as UML sequence diagrams useful for program comprehension?*

1.4.3 Suitability of advanced visualizations

In contrast to traditional visualizations, the literature also offers a number of advanced visualization techniques. Examples include information murals (Jerding and Stasko, 1998), polymetric views (Ducasse et al., 2004), and hierarchical edge bundling (Holten, 2006) (Figure 1.3). Most of these visualizations are quite scalable, rendering them potentially useful for dynamic analysis; on the other hand, the intended target audience may not be so easily accustomed to advanced visualizations as it is to traditional ones. Moreover, particularly on the subject of execution trace visualization, the existing work mostly provides anecdotal evidence of their usefulness, rather than evaluations that involve human subjects and actual comprehension tasks. For this reason, our focus shall not be merely on the *development* of an advanced visualization technique but also on its thorough empirical *validation*.

Research Question 3

> *How can program comprehension through dynamic analysis be effectively supported by more advanced visualizations?*

1.4.4 Evaluating program comprehension techniques

In order to raise the bar with respect to the evaluation of dynamic analysis techniques for program comprehension, our own manners of validation must meet certain standards. For example, we opt for multiple case studies that each involve different software systems and program comprehension activities. Moreover, if we are to compare our techniques to existing solutions and if fellow researchers are to compare their future techniques to ours, we need to introduce methodologies and reusable experimental designs to facilitate the comparison process. This is also emphasized by Sim et al., who argue that assessment frameworks and benchmarks can stimulate technical progress and community building (Sim et al., 2003). Thus, our final research question crosscuts the previous ones and is formulated as follows:

Research Question 4

> *How can we evaluate techniques for program comprehension and dynamic analysis, and how can we compare their added value to existing solutions?*

Table 1.1 shows the four research questions and the chapters in which they are addressed.

1.5 Thesis Outline

Chapter 2, "A Survey of Program Comprehension through Dynamic Analysis", is aimed at the structuring and interpretation of the entire research body on this topic. The relevant articles are decomposed into four facets: activity, target, method, and evaluation. Each of these facets contains a set of attributes. Using the facets, we identify the articles of interest, and characterize each article using an attribute framework. Based on the findings, we describe how the attention for each attribute is distributed across the relevant literature, and how the various activities are typically evaluated. We conclude with a series of recommendations as to directions that should receive additional consideration in the near future.

Chapter 3, "Sequence Diagram Reconstruction", describes the first of our efforts in understanding software through visualization. We show how a system's test suite can be used as a starting point for program comprehension by instrumenting and executing it and visualizing each test case as a UML sequence diagram. A series of abstractions and trace reduction techniques is proposed and implemented. Through a set of case studies we attempt to use the diagrams to build a sufficient understanding for several concrete maintenance tasks, and discuss the benefits and limitations of our approach.

Chapter 4, "An Assessment Methodology for Trace Reduction Techniques", addresses the need for automatic trace reduction techniques in program comprehension through dynamic analysis. Such techniques render large execution traces tractable and are omnipresent in the dynamic analysis literature, but are rarely (quantitatively) compared to one another. We attempt to fill this

Research question	Chapter(s)
1. How can we structure the available literature on program comprehension and dynamic analysis, and which topics need additional consideration the near future?	2
2. Given the excessive amounts of data involved in dynamic analysis, are abstraction techniques sufficient to render traditional visualizations such as UML sequence diagrams useful for program comprehension?	3, 4
3. How can program comprehension through dynamic analysis be effectively supported by more advanced visualizations?	5, 6
4. How can we evaluate techniques for program comprehension and dynamic analysis, and how can we compare their added value to existing solutions?	2, 3, 4, 5, 6

Table 1.1 The research questions and their corresponding chapters.

gap through the introduction of a methodology that enables a fair comparison of existing solutions. The framework is demonstrated on four reduction techniques from literature and a series of large traces from six different systems.

Chapter 5, "Advanced Trace Visualization with Extravis", continues our investigation of visualization techniques for program comprehension. We propose the use of two novel views to visualize large execution traces: the circular bundle view and the massive sequence view, both developed with an emphasis on scalability. These techniques are implemented in Extravis, which we validate through a series of three case studies, each involving different program comprehension activities and different software systems.

Chapter 6, "Trace Visualization: A Controlled Experiment", describes the design of a controlled experiment for the validation of trace visualization techniques. It discerns eight representative comprehension tasks that are to be performed by a control group (using a conventional tool) and an experimental group (using the tool under evaluation). We conduct one such experiment through the empirical validation of Extravis, of which we measure the added value to the Eclipse IDE using a group of 24 subjects.

Finally, in Chapter 7 we revisit our research questions, and evaluate the extent to which we have succeeded in answering them.

1.6 Origin of Chapters

The core chapters in this thesis are directly based on refereed publications and contain a certain degree of redundancy to ensure that they remain self-contained. In each chapter, the author of this thesis is responsible for most of the effort involved in implementing the approach (unless stated otherwise), executing the experiments, and writing the text.

Chapter 2 This chapter is based on our publication in the IEEE Transactions on Software Engineering (TSE) in 2009, and is referenced as (Cornelissen et al., 2009a).

Chapter 3 This chapter is a significant extension of our publication in the Proceedings of the 11^{th} European Conference on Software Maintenance and Reengineering (CSMR) in March 2007, referenced as Cornelissen et al. (2007b), and has been extended with work conducted with M.Sc. student Voets (2008).

Chapter 4 This chapter is based on our publication in the Proceedings of the 24^{th} International Conference on Software Maintenance (ICSM) in October 2008, referenced as Cornelissen et al. (2008a).

Chapter 5 This chapter is based on our publication in the Journal of Systems & Software (JSS) in December 2008, and is referenced as Cornelissen et al. (2008b). Earlier versions of this work appeared in the Proceedings of the 15^{th} International Conference on Program Comprehension (ICPC), cited as Cornelissen et al. (2007a), and in the Proceedings of the 4^{th} International Workshop on Visualizing Software for Understanding & Analysis (VISSOFT), cited as Holten et al. (2007), in June 2007. Credit for the tool implementation goes to Danny Holten.

Chapter 6 This chapter is based on our publication in the 17^{th} International Conference on Program Comprehension (ICPC) in May 2009, and is referenced as Cornelissen et al. (2009b).

Furthermore, our research has resulted in the following publications that are not explicitly included in this thesis:

- Dynamic Analysis Techniques for the Reconstruction of Architectural Views. In *Proceedings of the 14^{th} Working Conference on Reverse Engineering (WCRE)*, Doctoral Symposium, October 2007, referenced as Cornelissen (2007).
- Visualizing Similarities in Execution Traces. In *Proceedings of the 3^{rd} Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, October 2007, referenced as Cornelissen and Moonen (2007).
- Aiding in the Comprehension of Testsuites. In *Proceedings of the 2^{nd} Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, October 2006, referenced as Cornelissen et al. (2006).
- Identification of Variation Points using Dynamic Analysis. In *Proceedings of the 1^{st} Workshop on Reengineering towards Product Lines (R2PL)*, November 2005, referenced as Cornelissen et al. (2005).

A Survey of Program Comprehension through Dynamic Analysis*

Chapter 2

Program comprehension is an important activity in software maintenance, as software must be sufficiently understood before it can be properly modified. The study of a program's execution, known as dynamic analysis, has become a common technique in this respect and has received substantial attention from the research community, particularly over the last decade. These efforts have resulted in a large research body of which currently there exists no comprehensive overview. This chapter reports on a systematic literature survey aimed at the identification and structuring of research on program comprehension through dynamic analysis. From a research body consisting of 4,795 articles published in 14 relevant venues between July 1999 and June 2008 and the references therein, we have systematically selected 172 articles and characterized them in terms of four main facets: activity, target, method, and evaluation. The resulting overview offers insight in what constitutes the main contributions of the field, supports the task of identifying gaps and opportunities, and has motivated our discussion of several important research directions that merit additional consideration in the near future.

2.1 Introduction

One of the most important aspects of software maintenance is to understand the software at hand. Understanding a system's inner workings implies studying such artifacts as source code and documentation in order to gain a sufficient level of understanding for a given maintenance task. This *program comprehension* process is known to be very time-consuming, and it is reported that up to 60% of the software engineering effort is spent on understanding the software system at hand (Fjeldstad and Hamlen, 1979; Corbi, 1989; Pigoski, 1997).

Dynamic analysis, or the analysis of data gathered from a running program, has the potential to provide an accurate picture of a software system because it exposes the system's actual behavior. This picture can range from class-level details up to high-level architectural views (Richner and Ducasse, 1999; Walker et al., 1998; Schmerl et al., 2006). Among the benefits over static

*This chapter is based on our publication in the IEEE Transactions on Software Engineering in 2009 (Cornelissen et al., 2009a). It is co-authored by Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke.

analysis are the availability of runtime information and, in the context of object-oriented software, the exposure of object identities and the actual resolution of late binding. A drawback is that dynamic analysis can only provide a partial picture of the system, i.e., the results obtained are valid for the scenarios that were exercised during the analysis.

Dynamic analyses typically comprise the analysis of a system's execution through interpretation (e.g., using the Virtual Machine in Java) or instrumentation, after which the resulting data is used for such purposes as reverse engineering and debugging. Program comprehension constitutes one such purpose, and over the years, numerous dynamic analysis approaches have been proposed in this context, with a broad spectrum of different techniques and tools as a result.

The existence of such a large research body on program comprehension and dynamic analysis necessitates a broad overview of this topic. Through a characterization and structuring of the research efforts to date, existing work can be compared and one can be assisted in such tasks as finding related work and identifying new research opportunities. This has motivated us to conduct a systematic survey of research literature that concerns the use of dynamic analysis in program comprehension contexts.

In order to characterize the articles of interest, we have first performed an exploratory study on the structure of several articles on this topic. This study has led us to decompose typical program comprehension articles into four *facets*:

- The *activity* describes what is being performed or contributed [e.g., view reconstruction or tool surveys].

- The *target* reflects the type of programming language(s) or platform(s) to which the approach is shown to be applicable [e.g., legacy or web-based systems].

- The *method* describes the dynamic analysis methods that are used in conducting the activity [e.g., filtering or concept analysis].

- The *evaluation* outlines the manner(s) in which the approach is validated [e.g., industrial studies or controlled experiments].

Note that the goal of an article is captured in the activity facet.

Within each facet one can distinguish a series of generic *attributes*: the examples given above (in brackets) are in fact some of the attributes that we use in our framework. With this attribute framework, the papers under study can be characterized in a comprehensive fashion.

The goal of our survey is the systematic selection and characterization of literature that concerns program comprehension through dynamic analysis. Based on the four facets mentioned above, we derive attribute sets to characterize the articles of interest by following a structured approach that involves four main phases and two pilot studies. While our initial focus is on a selection of 14 relevant venues and on the last decade, we include additional literature by following the references therein. The resulting overview offers insight in what constitutes the main contributions of the field and supports the task of identifying gaps and opportunities. We discuss the implications of our findings and provide recommendations for future work. Specifically, we address the following research questions:

1. Which generic attributes can we identify to characterize the work on program comprehension through dynamic analysis?
2. How is the attention for each of these attributes distributed across the relevant literature?
3. How are each of the main activities typically evaluated?
4. Which recommendations on future directions can we distill from the survey results?

Section 2.2 presents an introduction on dynamic analysis for program comprehension. The protocol that lies at the basis of our survey is outlined in Figure 2.1, which distinguishes four phases that are described in Sections 2.3 through 2.6. Section 2.7 evaluates our approach and findings, and in Section 2.8 we conclude with a summary of the key contributions of this chapter.

2.2 Program Comprehension through Dynamic Analysis

To introduce the reader to the field of program comprehension through dynamic analysis, we first present a historical overview of the literature in the field, in which we distinguish between early literature and research conducted in the last decade. In doing so, we employ the definitions of program comprehension and dynamic analysis as provided in the introduction of this thesis in Chapter 1. We then motivate the need to perform a literature survey.

2.2.1 Early research

From a historical perspective, dynamic analysis was initially used for debugging, testing and profiling. While the purpose of testing is the verification of correctness and while profiling is used to measure (and optimize) performance, debugging is not used to merely locate faults, but also to understand the program at hand.

As programs became larger and more complex, the need to understand software became increasingly important. Originating from the discipline of debugging, the use of dynamic analysis for program comprehension purposes steadily gained more interest. As program comprehension is concerned with conveying (large amounts of) information to humans, the use of *visualization* attracted considerable attention.

Our study of this field showed that the first paper that can be labeled as "program comprehension through dynamic analysis" can be traced back to as early as 1972, when Biermann and Feldman synthesized finite state machines from execution traces (Biermann, 1972). (While their paper is not about reverse engineering, it does concern the creation of abstracted models from runtime behavior.) Since then, this type of research has steadily gained momentum, resulting in several important contributions throughout the 1980s and 1990s, which we summarize below.

In 1988, Kleyn and Gingrich (1988) proposed structural and behavioral views of object-oriented programs. Their tool, called TraceGraph, used trace information to animate views of program structures.

Five years later, De Pauw et al. (1993, 1994, 1998) started their extensive (and still on-going) research on program visualization, introducing novel views that include matrix visualizations, and the use of "execution pattern" notations to visualize traces in a scalable manner. They were among the first to reconstruct interaction diagrams (Jacobson, 1992) from running programs, and their work has later resulted in several well-known tools, most notably Jinsight and the associated Eclipse plug-in, TPTP[1].

Wilde and Scully (1995) pioneered the field of *feature location* in 1995 with their Software Reconnaissance tool. Feature location concerns the establishment of relations between concepts and source code, and has proven a popular research interest to the present day. Wilde et al. continued the research in this area in the ensuing years with a strong focus on evaluation (Wilde and Casey, 1996; Wilde et al., 2001, 2003). At the same time, Lange and Nakamura (1995b,a) integrated static and dynamic information to create scalable views of object-oriented software in their Program Explorer tool.

Another visualization was presented by Koskimies and Mössenböck (1996) in 1996, involving the reconstruction of scenario diagrams from execution traces. The associated tool, called Scene, offers several abstraction techniques to handle the information overload. Sefika et al. (1996) reasoned from a higher level of abstraction in their efforts to generate architecture-oriented visualizations.

In 1997, Jerding et al. proposed their well-known ISVis tool to visualize large execution traces (Jerding et al., 1997; Jerding and Rugaber, 1997). Two linked views were offered: a continuous sequence diagram, and the "information mural" (Jerding and Stasko, 1998): a dense, navigable representation of an entire trace.

Walker et al. (1998) presented their AVID tool a year later, which visualizes dynamic information at the architectural level. It abstracts the number of runtime objects and their interactions in terms of a user-defined, high-level architectural view (cf. Reflexion (Murphy et al., 2001)).

Finally, in 1999, Ball (1999) introduced the concept of frequency spectrum analysis. He showed how the analysis of frequencies of program entities in execution traces can help software engineers decompose programs and identify related computations. In the same year, Richner and Ducasse (1999) used

[1] The Eclipse Test & Performance Tools Platform Project, `http://www.eclipse.org/tptp/`

static and dynamic information to reconstruct architectural views. They continued this work later on (Richner and Ducasse, 2002), with their focus shifting to the recovery of collaboration diagrams with Prolog queries in their Collaboration Browser tool.

2.2.2 Research in the last decade

Around the turn of the millennium, we witness an increasing research effort in the field of program comprehension through dynamic analysis. The main activities in existing literature were generally continued, i.e., there do not seem to have emerged fundamentally new subfields. Due to the sheer size of the research body of the last decade, we limit ourselves to a selection of notable articles and discuss them in terms of their activities.

As program comprehension is primarily concerned with conveying information to humans, the use of *visualization* techniques is a popular approach that crosscuts several subfields.

One such purpose is *trace analysis*. A popular visualization technique in this respect is the UML sequence diagram, used by (e.g.) De Pauw et al. (2001), Systä et al. (2001), and Briand et al. (2006). Most of these approaches offer certain measures to address scalability issues, such as metrics and pattern summarization. Popular trace compaction techniques are offered by Reiss and Renieris (2001) and by Hamou-Lhadj et al. (Hamou-Lhadj et al., 2004; Hamou-Lhadj and Lethbridge, 2004, 2006).

From a higher level perspective, there have been several approaches toward *design* and *architecture recovery*. Among these efforts are influential articles by Heuzeroth et al. (2002, 2003), who combine static and dynamic analyses to detect design patterns in legacy code. Also of interest is the work on architecture reconstruction by Riva (Riva and Rodriguez, 2002; Riva and Yang, 2002), and the DiscoTect tool that constructs state machines from event traces in order to generate architectural views (Yan et al., 2004; Schmerl et al., 2006).

Another portion of the research body can be characterized as the study of *behavioral* aspects. The aforementioned work by Heuzeroth et al. analyzes running software by studying interaction patterns. Other notable approaches include a technique by Koskinen et al. (2006), who use behavioral profiles to illustrate architecturally significant behavioral rules, and an article by Cook and Du (2005) in which thread interactions are exposed in distributed systems. Furthermore, recently there has been considerable effort in the recovery of protocols (Quante and Koschke, 2007), specifications (Lo et al., 2008), and grammars (Walkinshaw et al., 2008).

The final subfield that we distinguish is *feature analysis*. While in this context there exist fundamental analyses of program features such as those by Greevy et al. (2005, 2006a) and by Kothari et al. (2007), particularly the activity of feature location has become increasingly popular since the aforementioned work by Wilde and Scully (1995). Influential examples include techniques by Wong et al. (2000) (using execution slices), Eisenbarth et al. (2003) (using formal concept analysis), Antoniol and Guéhéneuc (2006) (through statistical analyses), and Poshyvanyk et al. (2007) (using complementary techniques).

2.2.3 Structuring the field

The increasing research interest in program comprehension and dynamic analysis has resulted in many techniques and publications, particularly in the last decade. To keep track of past and current developments and to identify future directions, there is need for an overview that structures the existing literature.

Currently, there exist several literature surveys on subfields of the topic at hand. In 2004, Hamou-Lhadj and Lethbridge (2004) discussed eight trace exploration tools in terms of three criteria: trace modeling, abstraction level, and size reduction. In the same year, Pacione et al. (2003) evaluated five dynamic visualization tools on a series of program comprehension tasks. He later extended this selection in his Ph.D. thesis with 14 more tools and discussed a series of methods for evaluating such tools and techniques(Pacione, 2005). Greevy's Ph.D. thesis from 2007 summarized several directions within program comprehension, with an emphasis on feature analysis (Greevy, 2007). Also from 2007 is a study by Reiss (2007), who described how visualization techniques have evolved from concrete representations of small programs to abstract representations of larger systems.

However, the existing surveys have several characteristics that limit their usability in structuring the entire research body on program comprehension and dynamic analysis. First, they do not constitute a systematic approach because no explicit literature identification strategies and selection criteria are involved, which hinders the reproducibility of the results. Second, the surveys do not utilize common evaluation or characterization criteria, which makes it difficult to structure their collective outcomes. Third, their scopes are rather restricted, and do not represent a broad perspective (i.e., all types of program comprehension activities).

These reasons have inspired us to conduct a systematic literature survey on the use of dynamic analysis for program comprehension. In doing so, we follow a structured process consisting of four phases. Figure 2.1 shows the tasks involved, which are discussed in the following sections.

2.3 Article Selection

This section describes the first phase, which consists of a pilot study, an initial article selection procedure, and a reference checking phase.

2.3.1 Initial article selection

Since program comprehension is a broad subject that has potential overlaps with such fields as debugging, a clear definition of the scope of our survey is required.

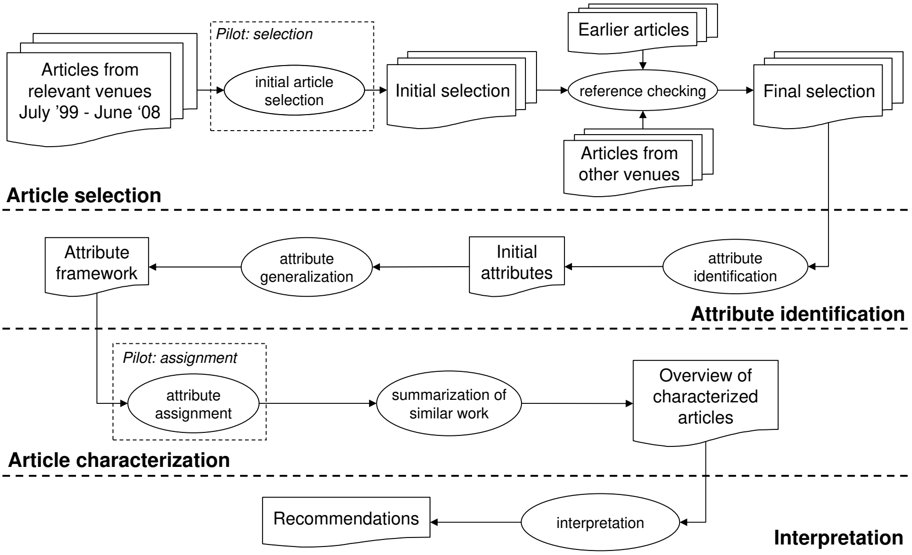


Figure 2.1 Overview of the systematic survey process.

Identification of research

Search strategies in literature surveys often involve automatic keyword searches (e.g., Beecham et al. (2008); Dyba and Dingsøyr (2008)). However, Brereton et al. (2007) recently pointed out that (1) current software engineering digital libraries do not provide good support for the identification of relevant research and the selection of primary studies, and that (2) in comparison to other disciplines, the standard of abstracts in software engineering publications is poor. The former issue exists because in software engineering and computer science, keywords are not consistent across different venues and organizations such as the ACM and the IEEE. Moreover, the field of program comprehension does not have a usable keyword standard.

Similar to Sjøberg et al. (2005), we therefore employ an alternative search strategy that involves the manual selection of articles from a series of highly relevant venues.

Given our context, we consider the five journals and nine conferences in Table 2.1 to be the most closely related to program comprehension, software engineering, maintenance, and reverse engineering. Our focus is primarily on the period of July 1999 to June 2008; the initial research body thus consists of 4,795 articles that were published at any of the relevant venues as a full paper or a short paper.

Selection criteria

Against the background of our research questions, we define two selection criteria in advance that are to be satisfied by the surveyed articles:

Type	Acronym	Description	Total no. art. July 1999 - June 2008
Journal	TSE	IEEE Transactions on Softw. Eng.	583
	TOSEM	ACM Transactions on Softw. Eng. & Methodology	113
	JSS	Journal on Systems & Softw.	965
	JSME	Journal on Softw. Maintenance & Evolution	159
	SP&E	Softw. – Practice & Experience	586
Conf.	ICSE	Int. Conf. on Softw. Eng.	429
	ESEC/FSE	European Softw. Eng. Conf. / Symposium on the Foundations of Softw. Eng.	240
	FASE	Int. Conf. on Fundamental Approaches to Softw. Eng.	198
	ASE	Int. Conf. on Automated Softw. Eng.	233
	ICSM	Int. Conf. on Softw. Maintenance	413
	WCRE	Working Conf. on Reverse Eng.	254
	IWPC/ICPC	Int. Workshop/Conf. on Program Comprehension	218
	CSMR	European Conf. on Softw. Maintenance and Reeng.	270
	SCAM	Int. Workshop/Working Conf. on Source Code Analysis and Manipulation	134

Table 2.1 Venues involved in the initial article selection.

1. The article exhibits a profound relation to program comprehension. The author(s) must state program comprehension to be a goal, and the evaluation must demonstrate the purpose of the approach from a program comprehension perspective. This excludes such topics as debugging and performance analysis.

2. The article exhibits a strong focus on dynamic analysis. For this criterion to be satisfied, the article must *utilize* and *evaluate* one or more dynamic analysis techniques, or concern an approach aimed at the support of such techniques (e.g., surveys).

The suitability of the articles is determined on the basis of these selection criteria, i.e., through a manual analysis of the titles, abstracts, keywords, and (if in doubt) conclusions (Brereton et al., 2007); borderline cases are resolved by discussion amongst the authors of this chapter.[2]

2.3.2 Selection pilot study

While the selection criteria being used may be perfectly understandable to the authors of this survey, they could be unclear or ambiguous to others. Following the advice of Kitchenham (2004) and Brereton et al. (2007), we therefore conduct a pilot study in advance to validate our selection approach against the opinion of domain experts. The outcomes of this study are used to improve the actual article selection procedure that is performed later on.

[2] In this chapter, "the authors" actually refers to the authors of the *publication* associated with this chapter (Cornelissen et al., 2009a).

To conduct the pilot study, the first two authors randomly pre-selected *candidate* articles, i.e., articles from relevant venues and published between July 1999 and June 2008, of which the titles and abstracts loosely suggest that they are relevant for the survey. Note that this selection also includes articles that are beyond the scope of the survey and should be rejected by the raters.[3]

The domain experts that serve as raters in the pilot are the last three authors of this survey. Since they were involved in neither the article selection procedure nor in the design thereof, they are unbiased subjects with respect to this study.

Each of the subjects is given the task of reading these articles in detail and identifying, on the basis of the selection criteria defined above, the articles that they feel should be included.

The outcomes are then cross-checked with those of the first two authors, who designed the selection procedure. Following these results, any discrepancies are resolved by discussion and the selection criteria are refined when necessary.

Pilot study results

The results of the pilot study were favorable: out of the 30 article selections performed, 29 yielded the same outcomes as those produced by the selection designers. These figures suggest that our selection criteria are largely unambiguous. The article that was assessed differently by a one subject concerned the field of impact analysis, which, following a discussion on its relation to program comprehension, was considered beyond the scope of this survey.

2.3.3 Reference checking

As previously mentioned, the initial focus of this survey is on selected venues in the period of July 1999 to June 2008. To cover articles of interest published before that time or in alternative venues, we (non-recursively) extend the initial selection with relevant articles that have been cited therein, regardless of publication date and venue but taking the selection criteria into account. This procedure minimizes the chance of influential literature being missed, and results in a *final* article selection.

2.3.4 Article selection results

The initial selection procedure resulted in 127 relevant articles that were published between July 1999 and June 2008 in any of the 14 venues in Table 2.1. The reference checking yielded another 45 articles (and 17 additional venues), which were subsequently included in the selection. This resulted in a research body that comprises 172 articles. The full listing of these articles is available online[4] and in a technical report (Cornelissen et al., 2008c). Figure 2.2 shows

[3] This latter characteristic intentionally makes the task more challenging for the raters.

[4] `http://swerl.tudelft.nl/bin/view/Main/ProgCompSurvey`

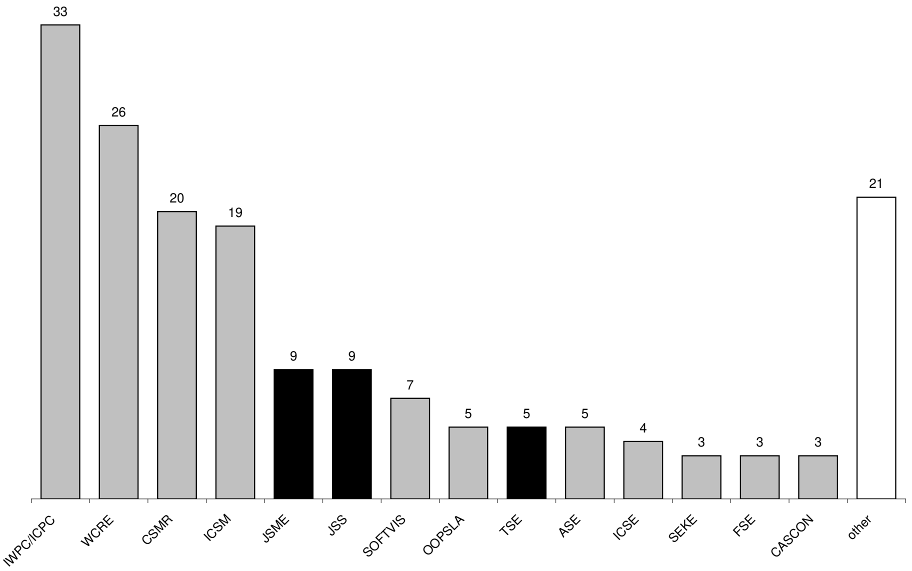


Figure 2.2 Distribution of the final article selection across the different venues. Bars in black denote journals; grey bars denote conferences.

the distribution of all surveyed articles across the venues from which at least three articles were selected.

2.4 Attribute Framework

As shown in Figure 1, the step after identifying the papers of interest is the construction of an attribute framework that can be used to characterize the selected papers. In this section we describe the process we used to arrive at such a framework, as well as the resulting framework.

2.4.1 Attribute identification

As stated in Section 2.1, our framework distinguishes four facets of interest: the *activity* performed, the type of *target* system analyzed, the *method* developed or used, and the *evaluation* approach used. The goal of our attribute identification step is to refine each of these four facets into a number of specific attributes.

In a first pass, we study all papers, and write down words of interest that could be relevant for a particular facet (e.g., "survey", or "feature analysis" for the activity facet). This data extraction task is performed by the first two authors of this survey. The result after reading all articles is a (large) set of initial attributes.

Note that to reduce the reviewer bias, we do not assume to know any attributes or keywords in advance.

2.4.2 Attribute generalization

After the initial attribute sets have been identified, we generalize them in order to render their number manageable and to improve their reusability. This is achieved through a discussion between the first three authors of this survey. Regarding the target facet, for example, the attributes "Java" and "Smalltalk" can intuitively be generalized to "object-oriented languages". After this data synthesis task, the resulting attribute sets are documented.

2.4.3 Resulting attribute framework

The use of our attribute framework on the article selection has resulted in seven different activities, six targets, 13 methods, and seven evaluation types. Table 2.2 lists the attributes and their descriptions.

The activity facet distinguishes between *five established subfields* within program comprehension: design and architecture recovery, visualization, feature analysis, trace analysis, and behavioral analysis. Each of these five attributes encapsulates a series of closely related activities, of which some were scarcely found: for example, very few authors propose new dynamic slicing techniques[5], and only a handful of articles aim at the reconstruction of state machines for program comprehension. In addition to the five major subfields, we have defined attributes for surveys and general purpose activities. The latter attribute denotes a broad series of miscellaneous activities that are otherwise difficult to generalize, e.g., solutions to the year 2000 problem, new dynamic slicing techniques, or visualizations with no specific focus.

The target facet contains six different types of programming platforms and languages. While we found it interesting to discern "legacy" software, this turned out to be difficult in practice, as such a classification depends greatly on one's perspective. For instance, a legacy system could have been written in Fortran or COBOL, lack any documentation, or simply be over 20 years old; on the other hand, it could also be a modern system that is simply difficult to maintain. Therefore, with respect to the legacy attribute, we rely on the type of the target platform as formulated by the authors of the papers at hand. Other targets include procedural languages, object-oriented languages, web applications, distributed systems, and software that relies heavily on multithreading.

The method facet is the most versatile of facets, and contains 13 different techniques. Note that we have chosen to distinguish between standard and advanced visualizations: the former denotes ordinary, widely available techniques that are simple in nature, whereas the latter represents more elaborate approaches that are seldomly used (e.g., OpenGL) or simply not publicly

[5] There exist numerous papers on dynamic slicing, but we found only two that use it in a program comprehension context.

Facet	Attribute	Description
Activity	survey	a survey or comparative evaluation of existing approaches that fulfill a common goal.
	design/arch.	the recovery of high-level designs or architectures.
	views	the reconstruction of specific views, e.g., UML sequence diagrams.
	features	the analysis of features, concepts, or concerns, or relating these to source code.
	trace analysis	the understanding or compaction of execution traces.
	behavior	the analysis of a system's behavior or communications, e.g., protocol or state machine recovery.
	general	gaining a general, non-specific knowledge of a program.
Target	legacy	legacy software, if classified as such by the author(s).
	procedural	programs written in procedural languages.
	oo	programs written in object-oriented languages, with such features as late binding and polymorphism.
	threads	multithreaded systems.
	web	web applications.
	distributed	distributed systems.
Method	vis. (std.)	standard, widely used visualization techniques, e.g., graphs or UML.
	vis. (adv.)	advanced visualization techniques, e.g, polymetric views or information murals.
	slicing	dynamic slicing techniques.
	filtering	filtering techniques or selective tracing, e.g., utility filtering.
	metrics	the use of metrics.
	static	information obtained through static analyses, e.g., from source code or documentation.
	patt. det.	algorithms for the detection of design patterns or recurrent patterns.
	compr./summ.	compression, summarization, and clustering techniques.
	heuristics	the use of heuristics, e.g., probabilistic ranking or sampling.
	fca	formal concept analysis.
	querying	querying techniques.
	online	online analysis, as opposed to post mortem (trace) analysis.
	mult. traces	the analysis or comparison of multiple traces.
Evaluation	preliminary	evaluations of a preliminary nature, e.g., toy examples.
	regular	evaluations on medium-/large-scale open source systems (10K+ LOC) or traces (100K+ events).
	industrial	evaluations involving software, or people, from industry.
	comparison	comparisons of the authors' approach with existing solutions.
	human subj.	the involvement of human subjects, i.e., controlled experiments & questionnaires.
	quantitative	assessments of quantitative aspects, e.g., speed, recall, or trace reduction rate.
	unknown/none	no evaluation, or evaluations on systems of unspecified size or complexity.

Table 2.2 Attribute framework.

available (e.g., information murals (Jerding and Stasko, 1998)). The remaining attributes represent a variety of largely orthogonal techniques that are often used in conjunction with others.

The evaluation facet distinguishes between seven types of evaluations. The "preliminary" attribute refers to early evaluations, e.g., on relatively small programs or traces. By contrast, the "regular" predicate indicates a mature validation that involves medium- to large-sized systems (typically open source) or answers actual research questions. Additionally, we have defined an attribute that captures case studies of an industrial nature: these typically relate to commercial software, legacy systems, or the involvement of actual developers for evaluation purposes. Furthermore, comparisons refer to evaluation types in which an approach is compared to existing solutions side-by-side; the involvement of human subjects measures the impact of an approach from a cognitive point of view; and quantitative evaluations are aimed at the assessment of various quantifiable aspects of an approach (e.g., the reduction potential of a trace reduction technique).

2.5 Article Characterization

The third phase comprises the assignment of attributes to the surveyed articles, and the use of the assignment results to summarize the research body.

2.5.1 Attribute assignment

Using our attribute framework from the previous section, we process all articles and assign appropriate attribute sets to them. These attributes effectively capture the essence of the articles in terms of the four facets, and allow for a clear distinction between (and comparison of) the articles under study. The assignment process is performed by the first two authors of this survey.

When assigning attributes to an article, we do not consider what the authors claim to contribute, but rather judge for ourselves. For example, papers on sequence diagram reconstruction are not likely to recover high-level architectures; and we consider an approach to target multithreaded systems if and only if this claim is validated through an evaluation or, at the very least, a plausible discussion.

2.5.2 Summarization of similar work

Certain articles might be extensions to prior work by the same authors. Common examples are journal publications that expand on earlier work published at conferences or workshops, e.g., by providing extra case studies or by employing an additional method, while maintaining the original context. While in our survey all involved articles are studied and characterized, in this report they are summarized to reduce duplication in frequency counts.

We summarize two or more articles (from the same authors) if they concern similar contexts and (largely) similar approaches. This is achieved by assigning the *union* of their attribute subsets to the most recent article, and discarding the other articles at hand. The advantage of this approach is that the number of articles remains manageable at the loss of virtually no information. The listing and characterization of the discarded articles are available in the aforementioned technical report and website.

2.5.3 Characterization pilot study

As previously mentioned, the attributes were defined and documented by the first two authors of this survey. Since the actual attribute assignment procedure is performed by the same authors, there is a need to verify the quality of the framework because of reviewer bias: the resulting attributes (and by extension, the resulting article characterization) may not be proper and unambiguous. In other words, since the process is subject to interpretation, different reviewers may envision different attribute subsets for one and the same article.

We therefore conduct another pilot study to assess the attribute quality and the attribute assignment procedure. The approach is similar to that of the first pilot. From the final article selection, a subset of five articles are randomly picked and given to the domain experts (the last three authors of this survey), along with (an initial version of) the attribute framework in Table 2.2.

The task involves the use of the given framework to characterize each of the five articles. A comparison of the results with those of the first two authors again yields a measure of the interrater agreement, upon which we discuss any flaws and strengthen the attribute sets and their descriptions.

2.5.4 Characterization pilot results & implications

The results of the characterization pilot resulted in generally high agreement on the activity, target, and evaluation facets. Most disagreement occurred for the method facet, which is also the one with the most attributes. This disagreement can be partly attributed to the fact that one rater tried to assign the single most suitable attribute only, whereas the others tried to assign as many attributes as possible. In the ultimate attribute assignments (discussed in the next section), we adopt the latter strategy: for each article, we select *all* attributes that apply to the approach at hand.

In several cases, the action taken upon interrater disagreement was to adjust the corresponding attributes or their descriptions. These adjustments have already been incorporated in Table 2.2. The following measures were taken:

Activity

- It was unclear what constitutes a framework, and whether the associated attribute covers contributions such as IDEs. We therefore extended the "framework" attribute to include models, environments, platforms, and architectures.
- At the request of the raters, the "communication" attribute was renamed to "behavior" and extended to include the recovery of state machines.

Target

- The raters had difficulty in assigning the "legacy" attribute, as the term "legacy software" is rather vague. For this reason, in each article we decided to rely on the classification of the target platform (legacy or non-legacy) as specified by the author(s).

Method

- The "trace comparison" attribute was renamed to "multiple traces", thus broadening its scope as it now covers all techniques that involve two or more traces.
- Whereas formerly there were distinct attributes for compression, merging, and clustering techniques, these are now covered by the newly created "compression/summarization" attribute since they are often hard to distinguish.
- Similarly, the distinct attributes for (a priori) "selective tracing" and (a posteriori) "filtering" were merged to "filtering", as the difference is generally subtle and largely dependent on the manner in which these techniques are described by their authors.
- The description of the "online analysis" attribute was refined to prevent this method from being confused with online approaches in machine learning (often used in state machine recovery), in which "online" denotes so-called active analysis algorithms.

Evaluation

- The former "performance" attribute was renamed to "quantitative analysis" to prevent confusion with software performance analysis, and was adjusted such that it not merely concerns measurements in terms of speed but also in such terms as recall and precision.
- The "human subjects" attribute was extended to include questionnaires.

- The distinction between open source and industrial studies was strengthened through the addition of the "open source" property to the "regular" attribute.

2.5.5 Measuring attribute coincidence

To further evaluate our attribute framework, we analyze the degree to which the attributes in each facet coincide. Against the background of our characterization results, we examine if there are certain attributes that often occur together, and whether such attributes in fact exhibit such an overlap that they should be merged.

We measure this by determining for each attribute how often it coincides with each of the other attributes in that facet. This results in a fraction between 0 and 1 for each attribute combination: 0 if they never coincide, and 1 if each article that has the one attribute also has the other.

2.5.6 Characterization results

The characterization and summarization of the 172 selected articles resulted in an overview of 110 articles, shown in Tables A.2 through A.4 in Appendix A. The second column denotes the number of underlying articles (if any) by the same author; the third column indicates whether we could find a reference to a publicly available tool in the article. In rare cases, none of our attributes fitted a certain aspect of an article; in such cases the value for the facet at hand can be considered "other", "unknown", or "none". The characterization of *all* 172 articles is available online and in the aforementioned technical report (Cornelissen et al., 2008c); in the remainder of this survey, however, we talk only in terms of the 110 *summarized* articles because they constitute unique contributions.

As previously mentioned, in each article we have focused on its achievements rather than its claims. On several occasions the titles and abstracts have proven quite inaccurate or incomplete in this respect. However, such occasions were not necessarily to the disadvantage of the author(s) at hand: for example, occasionally the related work section is of such quality that it constitutes a respectable survey (e.g., Briand et al. (2006); Liu et al. (2007)).

The overview in Tables A.2 through A.4 serves as a useful reference when seeking related work in particular subfields. For example, when looking for literature on trace visualization, one need only identify the articles that have both the "views" and the "trace analysis" attributes. In a similar fashion, one can find existing work on (e.g.) the use of querying and filtering techniques for architecture reconstruction, or learn how fellow researchers have assessed the quantitative aspects of state machine recovery techniques.

Our attribute coincidence measurements yielded no extraordinary results: while certain high fractions were found, none of these merited merges between the attributes involved because these attributes were obviously different in nature. Table A.1 in Appendix A shows the coincidences for the attribute combinations of which the fraction value exceeded 0.5.

Figure 2.3 shows for each facet the distribution of the attributes across the summarized articles, which we discuss in the next section.

2.6 Avenues for Future Research

Given the article selection and attribute assignments of Tables A.2 through A.4, our final survey step (see Figure 2.1) consists of interpreting our findings: what patterns can we recognize, what explanations can we offer, which lessons can we learn, and what avenues for further research can we identify? To conduct this step, we analyze the tables, looking for the most and least common attributes, and for interesting attribute combinations. In this section, we offer a selection of the most important outcomes of this analysis.

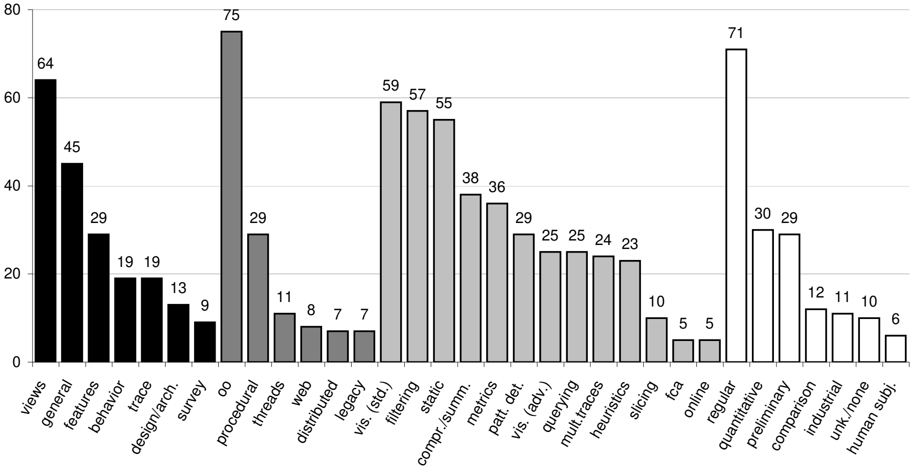


Figure 2.3 Attribute distribution in each facet across the summarized articles.

2.6.1 Most common attributes

Understanding the most common attributes (as displayed in Figure 2.3) gives an impression of the most widely investigated topics in the field.

Starting with the first facet, the activity, we see that the view attribute is the most common. This is not surprising as program comprehension deals with conveying information to humans, and particularly in dynamic analysis the amounts of information are typically large (Zaidman and Demeyer, 2004). We also found many articles to concern general activities, i.e., miscellaneous purposes that could not be generalized to a major subfield.

Moving on to the next facet, object-oriented software turns out to be the most common target in the research body: 75 out of the 110 articles propose techniques for, or evaluate techniques on, systems written in (predominantly) Java or Smalltalk. We are not sure why this is the case. Reasons might include ease of instrumentation, the suitability of certain behavioral visualizations (e.g., UML sequence diagrams) for OO systems, the (perceived) complexity of OO applications requiring dynamic analysis, or simply the fact that many researchers in this area have a strong interest in object-orientation.

Regarding the third facet, the method, we observe that standard visualizations occur more than twice as often as advanced ones. This may have several reasons, among which are the accessibility of standard tools (for graphs, sequence diagrams, and so forth) and possibly the belief that traditional visualizations should suffice in conjunction with efficient abstractions techniques (e.g., filtering). Furthermore, we observe that half of the surveyed articles employ static information. This is in accordance with Ernst's plea for a hybrid approach in which static and dynamic analysis are combined (Ernst, 2003).

Finally, within the evaluation facet, we note that regular evaluations (typically using open-source case studies) are the most typical, and that comparisons, industrial case studies, and involvements of human subjects (discussed later on) are rather uncommon. Furthermore, while the assessment of a technique's quantitative aspects is not very commonplace, this evaluation type does appear to be gaining momentum, as more than half (18 out of 30) such evaluations were carried out in the last three years. Interestingly, more than half of these evaluations involved the *feature location* activity; this is further discussed in Section 2.6.5.

Paraphrasing, one might say that the most popular line of research has been to work on dynamic visualization of open-source object-oriented systems. In the remainder of this section we will look at some of the less popular topics, analyze what the underlying causes for their unpopularity might be, and suggest areas for future research.

2.6.2 Least common activities

In the activity facet, surveys and architecture reconstruction occurred least.

As discussed in Section 2.2.3, the fact that a satisfactory survey of the field was not available was the starting point for our research, so this did not come as a surprise. Nevertheless, nine papers are labeled as survey, also since we marked papers containing elaborate discussions of related work as surveys (as explained in Section 2.5.6).

In our survey are 13 papers dealing with the use of dynamic analysis to reconstruct software architectures and designs. Some of these papers make use of fairly general traces capturing, e.g., method calls, from which occurrences of design patterns such as the Observer or Mediator can be identified (Heuzeroth et al., 2003).

Another line of research makes use of architecture-aware probing, and aims at visualizing system dynamics in terms of architectural abstractions, such as connectors, locks, dynamically loaded components, client-server configurations, and so on (Sefika et al., 1996; Schmerl et al., 2006; Israr et al., 2007). While there are not many papers addressing these topics, the initial results do suggest that successful application is possible. We expect that the importance of this field will grow: for complex adaptive systems or dynamically orchestrated compositions of web services, dynamic information may be the only way to understand the runtime architecture.

2.6.3 Least common targets

Web applications

We were surprised that web applications were encountered rather sporadically as a target platform. While traditional web sites consisting of static HTML pages can be easily processed using static analysis alone, modern web applications offer rich functionality for online banking, shopping, e-mailing, document editing, and so on. The logic of these applications is distributed across the browser and the web server, and written using a range of technologies (such as PHP, Javascript, CSS, XSLT, etc.). While this severely complicates static analysis, dynamic analysis might still be possible, for example by monitoring the HTTP traffic between the client and the server.

One complicating factor might be that web applications require user interaction, and hence, user input. Several solutions to this problem exist, such as the use of webserver log files of actual usage, or the use of capture and playback tools. Furthermore, techniques have been developed to analyze web forms and to fill them in automatically based on a small number of default values provided by the software engineer (Benedikt et al., 2002).

The growing popularity of Javascript in general and Ajax (Asynchronous Javascript and XML) in particular, is another argument in favor of dynamic analysis of web applications. With Javascript, events can be programmatically attached to any HTML element. In this setting, even determining the seemingly simple navigation structure of the web application can no longer be done statically, as argued by Mesbah et al. (2008). To deal with this problem, they propose a "crawler" capable of executing Javascript, identifying clickable elements, and triggering clicks automatically: a solution that can also serve as the starting point for dynamic analysis in which client-side logic is to be executed automatically.

Distributed systems

As it turns out, the understanding of distributed systems has received little attention in literature: no more than seven articles are concerned with this target type. Such systems are, however, becoming increasingly popular, e.g., with the advent of service-orientation. Gold et al. (2004) paraphrase the core issue as follows: "Service-oriented software lets organizations create new software applications dynamically to meet rapidly changing business needs. As its construction becomes automated, however, software understanding will become more difficult". Furthermore, distributed systems often behave differently than intended, because of unanticipated usage patterns that are a direct consequence of their dynamic configurability (Moe and Carr, 2001). This increases the need to understand these systems, and due to their heterogeneous nature, dynamic analysis constitutes a viable approach.

Multithreaded applications

In recent years, multicore CPUs have become mainstream hardware and multithreading has become increasingly important. The evolution towards multithreaded software is in part evidenced by the foundation of the International Workshop on Multicore Software Engineering (IWMSE), first held at ICSE in 2008: in the proceedings of this workshop it is stated by Pankratius et al. (2008) that in the near future "every programmer will be confronted with programming parallel systems", and that in general "parallel components are poorly understood".

The importance of understanding multithreading behavior is not reflected by the current research body: a total of 11 articles are explicitly targeted at multithreaded applications. The use of dynamic analysis on such systems has the important benefit that thread management and interaction can be understood at runtime. A problematic issue in multithreaded systems can be reproducing behavior: does replaying the same scenario result in the same trace? An interesting route to deal with this is to explore the use of multiple traces and suitable trace comparison techniques to highlight essential differences between traces. According to our findings, this is largely unexplored territory: there are only few papers combining the multithreading and trace comparison attributes in our tables.

Legacy systems

Legacy systems are often in need of a reverse engineering effort, because their internals are poorly understood. Nevertheless, our survey shows that very few papers explicitly mention legacy environments as their target, meaning that dynamic analysis is rarely applied to legacy software systems. This can be partly explained by (1) the fact that researchers do not have access to legacy systems, (2) a lack of available instrumentation tools for legacy platforms, or (3) the fact that instrumented versions of the application are difficult to deploy and, subsequently, run. Another hindering factor is the difficulty of integrating the instrumentation mechanism into the legacy build process, which is often heterogeneous, i.e., with several kinds of scripting languages in use, and few conventions in place (Zaidman et al., 2006).

2.6.4 Least common evaluations

Industrial studies

In our survey we have distinguished between evaluations on industrial and open-source systems. Industrial systems may differ from open-source systems in terms of the way of working, size, complexity, and level of interaction with other systems. Furthermore, industrial systems may share some of the problems of legacy systems as just discussed (Demeyer et al., 2003).

We found industrial evaluations to be uncommon, with a total of 11 articles involving industrial cases. Most of these are conducted within the context of research projects with industrial partners, in which the industrial partners have a particular need for reverse engineering.

We have also observed that the degree to which developers or maintainers are involved in the validation is generally low, as their feedback is often limited to answering several general questions, if given at all. This may be a consequence of a lack of time on the part of the developers, or because the industry is not fully aware of the potential benefits of dynamic analysis. This may be resolved by familiarizing practitioners with the benefits, e.g., through the development environment (IDE), as proposed by Röthlisberger et al. (2008) who provide dynamic information during programming tasks.

Another impediment for industrial involvement in publications can be fear for disclosing proprietary material. Apart from open discussions with management about the mutual interest, anonymizing traces or presenting aggregated data only might be an option, although obfuscated traces will be even harder to understand.

Finally, a more technical obstacle is the lack of resources, be it memory or processor cycles for the tracing mechanism or disk space for the storage of execution traces. A potential solution to these problems is found in lightweight tracing techniques (e.g., Reiss (2003b)) or capture/replay techniques (e.g., Joshi and Orso (2007); Xu et al. (2007)).

Involvement of human subjects

In the field of program comprehension, an evaluation that involves human subjects typically seeks to measure such aspects as the usability of a technique in practice. The involvement of human subjects is important in this field because it deals with conveying information to humans. Moreover, dynamic analyses are particularly notorious for producing more information than can be comprehended directly (Zaidman, 2006).

In spite of its importance, this type of evaluation was used in no more than six articles. Bennett et al. (2008) use four experts and five graduate students to assess the usefulness of reverse engineered UML sequence diagrams in nine specific comprehension tasks. Quante (2008) reports on a controlled experiment with 25 students that involves the use of "object process graphs" in a program comprehension context. Röthlisberger et al. (2008) preliminarily assess the added value of dynamic information in an IDE by having six subjects conduct a series of tasks; the authors remain unclear as to the background of the subjects and the nature of the tasks at hand. Hamou-Lhadj and Lethbridge (2006) report on a questionnaire in which the quality of a summarized execution trace is judged by nine domain experts; however, no real comprehension tasks are involved. Finally, Wilde et al. (2003) and Simmons et al. (2006) conduct experiments to assess the practical usefulness of different feature location techniques in legacy Fortran software and in a large 200 kLOC system, respectively.

The design and execution of a controlled experiment is quite elaborate, and requires a great deal of preparation and, preferably, a substantial number of test subjects. Nonetheless, such efforts constitute important contributions to the field of program comprehension and must therefore be encouraged, particularly in case of (novel) visualizations. On a positive note, the fact that three out of the six experiments mentioned above were conducted in 2008 could suggest that this type of evaluation is already gaining momentum.

Comparisons

Comparisons (or comparative evaluations) are similar to surveys in the sense that the article at hand involves one or more existing approaches. The difference in terms of our attribute framework is that the authors of side-by-side comparisons do not merely discuss existing solutions, but rather use them *to evaluate their own*. Such a comparison can be more valuable than the evaluation of a technique by itself through anecdotal evidence, as it helps to clarify where there is an improvement over existing work.

Our survey has identified a total of 12 comparative evaluations. The majority of these comparisons was conducted in the context of feature location. As an example, Eaddy et al. (2008) discuss two recently proposed feature location techniques, devise one of their own, and subject combinations of the three techniques to a thorough evaluation. Similar approaches are followed by Antoniol and Guéhéneuc (2006) and by Poshyvanyk et al. (2007); in the same context, Wilde et al. (2003) offer a comparison between a static and a dynamic technique.

Apart from the field of feature location, in which complementary techniques have already proven to yield the best results, the degree to which existing work is compared against is generally low. One can think of several causes (and solutions) in this context.

First, it must be noted that work on program comprehension cannot always be easily compared because the *human factor* plays an important role. The aforementioned feature location example is an exception, since that activity typically produces quantifiable results; Evaluations of qualitative nature, on the other hand, may require hard to get domain experts or control groups, as well as possibly subjective human interpretation and judgments.

Second, we have determined that only 14 out of the 110 articles offer publicly available *tools*. The lack of available tooling is an important issue, as it hinders the evaluation (and comparison) of the approaches by other researchers. In our discussion of existing trace reduction techniques in Chapter 4, for example, we resort to our own implementations, which may result in interpretation errors (thus constituting a threat to the internal validity of the experiments). We thus encourage researchers to make their tools available online, and advocate the use of these tools to compare new solutions against.

Third, the comparison of existing approaches is hindered by the absence of common assessment frameworks and benchmarks, which, as Sim et al. (2003) observed, can stimulate technical progress and community building. In the context of program comprehension through dynamic analysis, one could think of using common test sets, such as execution trace repositories, and common evaluation criteria, such as the precision and recall measures that are often used in the field of feature location (e.g., Eaddy et al. (2008)). Also of importance in this respect is the use of open-source cases to enable the reproducibility of experiments.

We will discuss examples of the above in Chapter 4.

2.6.5 How activities are evaluated

In the historical overview in Section 2.2 we identified five main subfields in program comprehension: feature analysis, visualization, trace analysis, design and architecture recovery, and behavioral analysis, which correspond to the activity facet of our attribute framework. Here we consider these fields from the perspective of the evaluation facet.

The literature on feature analysis mostly deals with feature location, i.e., relating functionality to source code. What is interesting is not only that this field has received significant attention from 1995 to the present day, but also that comparative evaluations are a common practice in this field, as noticed in Section 2.6.4. The introduction of common evaluation criteria (i.e., precision and recall) may have contributed to this development. Furthermore, feature analysis accounts for seven out of the 11 industrial evaluations identified in this survey, and for four out of the six evaluations that involve human subjects.

Visualization is a rather different story: for reasons mentioned earlier, the effectiveness of visualization techniques is more difficult to assess, which hinders their comparison and their involvement in industrial contexts. Furthermore, there is still a lot of experimenting going on in this field with both traditional techniques and more advanced solutions. As an example of the former, consider the reverse engineering of UML sequence diagrams: this has been an important topic since the earliest of program comprehension articles (e.g., De Pauw et al. (1993); Koskimies and Mössenböck (1996)) and has only recently been the subject of an actual user study (Bennett et al., 2008). In general, the evaluation of visualizations through empirical studies is quite rare, as are industrial studies in this context.

Execution trace analysis, and trace reduction in particular, has received substantial attention in the past decade. This has seldomly resulted in industrial studies and never in controlled experiments. Furthermore, while comparisons with earlier approaches are not very common either, Chapter 4 will discuss our first efforts at the (quantitative) evaluation of existing reduction techniques.

Finally, behavioral analysis and architecture recovery are somewhat difficult to assess: the latter has been treated in only five articles, while the former is a rather heterogeneous subfield that comprises various similar, but not equal, disciplines. They are mostly small and involve limited numbers of researchers, and generally these areas of specialization cannot be compared with each other. However, as a behavioral discipline receives more attention in literature, it may grow to become a subfield on its own: the automaton-based recovery of protocols, for example, is a recent development that is adopting common evaluation criteria and thorough comparisons (Quante and Koschke, 2007; Lo et al., 2008).

2.7 Evaluation

In the previous sections we have presented a series of findings based on our paper selection, attribute framework, and attribute assignments. Since conducting a survey is a largely manual task, most threats to validity relate to the possibility of researcher bias, and thus to the concern that other researchers might come to different results and conclusions. One remedy we adopted is to follow, where possible, guidelines on conducting systematic reviews as suggested by, e.g., Kitchenham (2004) and Brereton et al. (2007). In particular, we documented and reviewed all steps we made in advance (per pass), including selection criteria and attribute definitions.

In the following sections, we successively describe validity threats pertaining to the article selection, the attribute framework, the article characterization, and the results interpretation, and discuss the manners in which we attempted to minimize their risk.

2.7.1 Article selection

Program comprehension is a broad subject that, arguably, has a certain overlap with related topics. Examples of such topics are debugging and impact analysis. The question whether articles of the latter categories should be included in a program comprehension survey is subject to debate. It is our opinion that the topics covered in this survey are most closely related to program comprehension because their common goal is to provide a deeper understanding of the inner workings of software. Following the advice of Kitchenham (2004) and Brereton et al. (2007), we enforced this criterion by utilizing predefined selection criteria that clearly define the scope, and evaluated these criteria through a pilot study that yielded positive results (Section 2.3.2).

In the process of collecting relevant articles, we chose not to rely on keyword searches. This choice was motivated by a recent paper from Brereton et al. (2007), who state that "current software engineering search engines are not designed to support systematic literature reviews"; this observation is confirmed by Staples and Niazi (2007). For this reason, we have followed an alternative search strategy that comprises the manual processing of relevant venues in a certain period of time.

The venues in Table 2.1 were chosen because they are most closely related to software engineering, maintenance, and reverse engineering. While this presumption is largely supported by the results (Figure 2.2), our article selection is not necessarily unbiased or representative of the targeted research body. We have addressed the threat of selection bias by utilizing the aforementioned selection criteria. Furthermore, we have attempted to increase the representativeness of our selection by following the references in the initial article selection and including the relevant ones in our final selection. We found a non-recursive approach sufficient, as checking for citations within citations typically resulted in no additional articles. As a result, we expect the number of missed articles to be limited; particularly those that have proven influential have almost certainly been included the survey, as they are likely to have been cited often.

2.7.2 Attribute framework

We acknowledge that the construction of the attribute framework may be the most subjective step in our approach. The resulting framework may depend on keywords jotted down in the first pass, as well as on the subsequent generalization step. However, the resulting framework can be evaluated in terms of its usefulness: Specifically, we have performed a second pilot study, and measured the degree to which the attributes in each facet coincide. Both of these experiments yielded favorable results and demonstrate the applicability of our framework.

2.7.3 Article characterization

Similar to the construction of the attribute framework, the process of applying the framework to the research body is subjective may be difficult to reproduce.

We have addressed this validity threat through a second pilot study (Section 2.5.3), of which the results exposed some discrepancies, mostly within the method facet. The outcomes were discussed among the authors of this survey and resulted in the refinement of several attributes and their descriptions.

In order to identify topics that have received little attention in the literature, we counted the occurrences of all attributes in the selected articles (Figure 2.3). A threat to validity in this respect is duplication among articles and experiments: one and the same experiment should not be taken into account twice, which is likely to occur when considering both conference proceedings and journals. We have addressed this threat by summarizing the article selection and using the summarized articles for the interpretation phase, while making the full selection available online and in a technical report.

2.7.4 Results interpretation

A potential threat to the validity of the results interpretation is researcher bias, as the interpretation may seek for results that the reviewers were looking for. Our countermeasure has been a systematic approach towards the analysis of Tables A.2 through A.4: in each facet we have discussed the most common and least common attributes. In addition, we have examined the relation between activities and evaluations in particular, as this combination pertains to one of our research questions.

2.8 Conclusion

In this chapter we have reported on a systematic literature survey on program comprehension through dynamic analysis. We have characterized the work on this topic on the basis of four main facets: activity, target, method, and evaluation. While our initial focus was on nine conferences and five journals in the last decade, the use of reference checking to include earlier articles and alternative venues yielded a research body that comprises 31 venues, and relevant articles of up to thirty years old.

Out of 4,795 scanned articles published between July 1999 and June 2008 in 14 relevant venues, we selected the literature that strongly emphasizes the use of dynamic analysis in program comprehension contexts. The addition of relevant articles that were referenced therein resulted in a final selection of 172 articles. Through a detailed reading of this research body, we derived an attribute framework that was consequently used to characterize the articles under study in a structured fashion. The resulting systematic overview is useful as a reference work for researchers in the field of program comprehension through dynamic analysis, and helps them identify both related work and new research opportunities in their respective subfields.

In advance, we posed four research questions pertaining to (1) the identification of generic attributes, (2) the extent to which each of these attributes is represented in the research body, (3) the relation between activities and evaluations, and (4) the distillation of future directions.

The identified attributes are shown in Table 2.2. While being generic in the sense that they characterize all of the surveyed articles, they are sufficiently specific for researchers looking for related work on particular activities, target system types, methods, and evaluation types.

The characterization of the surveyed articles is shown in Tables A.2 through A.4. The frequencies of the attributes are provided by Figure 2.3, which clearly shows the distribution of the attributes in each facet across the research body. We discussed the results, highlighted research aspects that have proven popular throughout the years, and studied the manners in which the major subfields are evaluated.

Based on our analysis of the results, we report on three lessons learned that we consider the most significant. First, we have observed that the feature location activity sets an example in the way research results are evaluated: a great deal of effort is put in comparing and combining earlier techniques, which has led to a significant technical progress in the past decade (Section 2.6.5). Second, we conclude that standard object-oriented systems may be overemphasized in the literature, at the cost of web applications, distributed software, and multithreaded systems, for which we have argued that dynamic analysis is very suitable (Section 2.6.1 and Section 2.6.3). Third, with regard to evaluation, we have learned that in activities other than feature location, comparisons and benchmarking do not occur as often as they should. To support this process, we encourage researchers to make their tools publicly available, and to conduct controlled experiments for visualization techniques because these are otherwise difficult to evaluate (Section 2.6.4).

In summary, the work described in this chapter makes the following contributions:

1. A historical overview of the field of program comprehension through dynamic analysis.
2. A selection of key articles in the area of program comprehension and dynamic analysis, based on explicit selection criteria.
3. An attribute framework that can be used to characterize papers in the area of program comprehension through dynamic analysis.
4. An actual characterization of all selected articles in terms of the attributes in this framework.
5. A series of recommendations on future research directions.

Sequence Diagram Reconstruction[*] Chapter 3

In this chapter, we elaborate on our first efforts in supporting the program comprehension process by means of dynamic analysis. Our focus here is on traditional visualization techniques, as they are readily available and because we expect most people to be accustomed to them. Inspired upon the popularity of testing during, e.g., Agile software development methods, we focus on a system's test suite as a starting point for knowledge gathering. We propose to use a variant of UML sequence diagrams to visualize the information obtained from the execution of such a test suite, and employ a series of abstraction techniques to make the resulting diagrams more scalable. Our approach is implemented in two different tool implementations, and then validated through case studies that involve two different systems.

3.1 Introduction

When implementing and maintaining software systems, *testing* is of vital importance. The advent of Agile software development methods such as eXtreme Programming (XP) (Beck, 1999), Scrum (Schwaber and Beedle, 2001), and DSDM (Stapleton, 1997) has ensured that testing is given much attention. In these processes, testing serves two distinct purposes. Not only is testing considered essential when implementing and maintaining software to help increase the quality and correctness of code, but (unit) tests are also a means of documentation (van Deursen, 2001; Marick, 2004; Miller, 2004; Demeyer et al., 2003; Forward and Lethbridge, 2002). Test-driven development (Beck, 2003), which is related to Agile software development, implies creating and maintaining an extensive test suite in order to guarantee that the various components work correctly, both individually (by means of unit tests) and as a whole (through use of acceptance tests).

Various testing methodologies have been devised for a broad range of programming languages. Among these approaches are the well-known *xUnit* frameworks that were originally based on the JUnit design by Beck and Gamma (1998). JUnit allows for the specification of both unit tests and acceptance tests, and is relatively easy to use. A JUnit test case consists of several steps: the creation of a fixture, exercising the method(s) under test, comparing the results (i.e., assertions), and the teardown. Typically, a set of test cases is run as a test suite.

[*] This chapter is a significant extension of our publication in the Proceedings of the 11th European Conference on Software Maintenance and Reengineering (Cornelissen et al., 2007b). It is co-authored by Arie van Deursen, Leon Moonen, and Andy Zaidman.

3.1.1 Visualizing test suites

Although most test cases might seem easily comprehensible at first sight, they quickly tend to become relatively complex for someone who desires a quick look "under the hood". For example, when considering object-oriented systems, it is difficult to keep track of the numerous objects that are typically involved. Therefore, we propose to visualize test cases in a manner that is sufficiently detailed for comprehension purposes and that remains human readable at the same time. As such, the general approach we follow can be described as either *analyzing* or *tracing* the test cases, applying certain *abstractions* and, finally, *presenting* the results. Such visualizations can be particularly helpful in the context of Agile software development, in which tests serve as documentation.

UML sequence diagrams (OMG, 2003; Rumbaugh et al., 1998) are a useful means to visualize a system's behavior (De Pauw et al., 2001). A *scenario diagram* is a somewhat simplified version of a sequence diagram that describes a single scenario, i.e., depicting one particular execution path. Scenario diagrams provide detailed information on interactions at either the class level or the object level, and are easy to read because the chronological order is intuitive. However, if no abstractions are applied, scenario diagrams tend to become too large: the entire execution of a sizable software system could result in a scenario diagram that contains more information than the reader can handle (De Pauw et al., 1998).

3.1.2 Goal

The goal of this research is the design and evaluation of tools that support the understanding of unfamiliar software. Specifically, we focus on the support of change requests and feature requests. We propose that users build up the required knowledge by studying reconstructed scenario diagrams from JUnit test suites. Our choice for using test cases as scenarios is inspired by three major factors:

1. XP and Agile software development in general advocate the use of tests as a form of documentation (Marick, 2004; Miller, 2004; Demeyer et al., 2003; Forward and Lethbridge, 2002).

2. Finding relevant scenarios for executing software and performing dynamic analysis on them (e.g., constructing visualizations) is not straightforward when domain knowledge is lacking (Zaidman, 2006).

3. Scalability problems can in part be overcome by the careful selection of relatively concise execution scenarios, such as test cases. This also fits the as-needed reverse engineering strategy that is often advocated in dynamic analysis (Zaidman, 2006).

3.1.3 Research questions

To structure our investigation toward our goals, we define the following research questions:

- Are tests as typically written in JUnit a suitable starting point for program comprehension?
- Do scenario diagrams obtained from these tests offer insight in the way an application works?
- Which abstractions do we need to make these diagrams more easily understandable?
- JUnit can be used to write tests focused on one class or method (which are true *unit* tests), as well as for creating high level test cases which act more like system or acceptance tests. Should these be treated differently when leveraging test suites for program comprehension?

In order to address these questions and to validate our techniques, we design a sequence diagram reconstruction framework and propose two tool implementations. We then conduct case studies on two systems, JPacman and Checkstyle, both of which feature extensive test suites that render them suitable for our research.

This chapter is structured as follows. The next section discusses the issues and design choices. Based on this discussion, Section 3.3 presents our framework for sequence diagram reconstruction. This framework is implemented in two different prototype tools. Our first prototype, Sdr, is presented in Section 3.4 and evaluated in Section 3.5. Section 3.6 proposes our second tool implementation, called JRET, which is evaluated in Section 3.7. Related work is described in Section 3.8, and we conclude with final words in Section 3.9.

3.2 Design Options and Requirements

In the course of converting test suites to scenario diagrams, we face several challenges. This section addresses the most prominent issues and requirements.

3.2.1 Design options

Static versus dynamic approaches
In obtaining scenario diagrams from test cases, we can choose whether to capture the system's behavior by means of *static* analysis (i.e., analyzing the code) or through *dynamic* analysis (i.e., tracing the execution). The well-known benefits of a static approach are the genericity and compactness, whereas a dynamic technique potentially offers more details on important aspects such as *late binding*.

A frequently mentioned drawback of dynamic analysis is the need for scenarios to exercise the system. However, these scenarios come for free when using test cases to drive the comprehension process. Another drawback of dynamic analysis is that the information derived represents only part of the system's behavior. Yet, we argue that in our context, having more detailed information pertaining directly to test cases is to be favored over genericity.

Recognizing stages in a scenario
The second issue that arises is how one can distinguish between *particular stages* during the execution of a scenario. In particular, we are interested in recognizing the various stages in executing a test case, i.e., the test initialization, test execution, result validation and teardown[1]. Being able to separate these stages provides a more focused view on the execution of a test case and allows the viewer to skip a stage (if desired), or to put stages in separate diagrams. Furthermore, as we will see later, recognizing which methods and objects play a role in the various stages allows for particular filtering and abstraction techniques. For example, the assertions made during the result validation stage of a (unit) test are generally of less interest in the context of comprehending the inner workings of a software system.

Class versus object level
Another design choice concerns whether we want to trace the interactions at the *class* level or at the *object* level. The former is simpler because it does not require an administration of all runtime objects, whereas being able to distinguish between objects provides detailed information on object interactions, and exposes occurrences of polymorphism and late binding, for which scenario diagrams are very suitable. However, with diagrams at the class level already being susceptible to size problems, one will definitely encounter scalability issues with additional object information.

Scalability
Despite the fact that (in general) the execution of a unit test is relatively short, *scalability* problems are inevitable as systems and test case executions grow larger. Most simple unit tests will presumably fit on a single screen, but more complex test cases (i.e., acceptance tests) induce too many interactions to simply put in a diagram without applying any form of abstraction. Therefore, we need trace reduction techniques that are both efficient and useful: we must determine which interactions are presumably irrelevant and can therefore be combined or omitted for an initial view.

3.2.2 Requirements

In order to create a flexible tool that can aid in the understanding of a variety of software systems, we define the following requirements:

[1] The earlier naming conventions for these stages were replaced by annotations in JUnit 4.

Obliviousness: The system under analysis should not require any changes to its implementation to enable its examination.

Extensibility: The framework should support the addition and customization of (1) event listeners to capture (dynamic) system behavior, (2) reduction techniques to combine or omit information, and (3) output handling to facilitate flexible visualization and embedding in other environments (e.g., a software development environment like Eclipse).

Interactivity: The framework should allow the interactive exploration of reconstructed diagrams. We take the position that a human in the loop is a necessity as no automated analysis can capture which improvements in comprehensibility are achieved when trace reductions are employed. The user should remain in control of the level of detail.

Comprehensibility: When talking about generating "understandable" scenario diagrams, naturally we reason from the perspective of humans: we strive to minimize the effort involved in understanding the diagrams. To this end, we need criteria that capture *comprehensibility* to shape our visualization requirements. In the realm of visual programming languages, Yang et al. (1997) propose two properties to determine when a diagram is understandable:

1. *Accessibility of related information*. Viewing related information in close proximity is essential. When two screen objects are not in close proximity, there is the psychological claim that these objects are not tightly related to each other or are not dedicated to solving the same problem.
2. *Use of screen real estate*. The term "screen real estate" refers to the size of a physical display screen and connotes the fact that screen space is a valuable resource. Making optimal use of the available screen real estate is necessary to save the user time when locating information.

We aim for these properties to be essential criteria in our visualization design.

3.3 Framework

Based on the design options and issues discussed earlier, we have designed a framework for the reconstruction of sequence diagrams from execution traces. Figure 3.1 provides an overview of the approach. It is an extensible framework in that there is a clear separation between the tracing part, the abstraction part, and the rendering of scenario diagrams. In the remainder of this section we will discuss and motivate various features of the framework.

3.3.1 Tracing test cases

There exist several methods to obtain execution traces, among which the most commonly used are manually instrumenting code (e.g., using InsectJ[2]), using

[2] InsectJ, `http://insectj.sourceforge.net/`

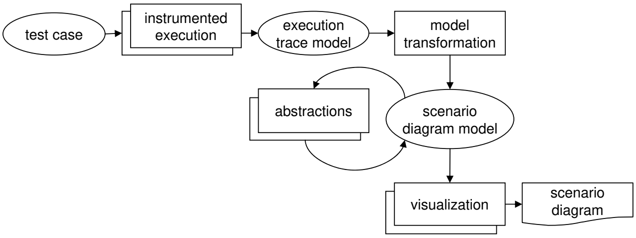


Figure 3.1 Overview of the reconstruction framework.

a debugger or profiler (Zaidman et al., 2005), and instrumentation through *aspects*[3]. The advantages and shortcomings these techniques are mostly well-known (and, for example, discussed by Zaidman (2006)).

Our framework uses Aspect Oriented Programming (AOP) because aspects are easily customizable and therefore constitute a flexible solution in our context (Kiczales et al., 2001). Moreover, the use of AOP satisfies our requirement of obliviousness: the system under analysis does not require any changes to enable its examination. Aspects allow us to specify very accurately which parts of the execution are to be considered, where tracing must start and stop, and what type of post-processing is needed. In addition, aspects can obtain detailed information on all interactions, such as the unique objects that are involved, the runtime arguments, and the actual return values in case of method and constructor calls.

Finally, most aspect languages allow for patterns in the definition of which classes and interactions are to be traced (i.e., pointcuts). This enables a distinction of the various stages in a test case by exploiting the *naming conventions* or *annotations* used for these stages within xUnit frameworks. These are illustrated in Listing 3.1.

```
public class MonsterMoveTest extends MoveTest {

   MonsterMove monsterMove;

   @Before
   public void setUp() {
      theMonster = (Monster) monsterCell.getInhabitant();
   }

   @Test
   public void testSimpleGetters() {
      ...
   }

   @Test
   public void testT1_outOfBorder() {
      ...
   }

   ...
}
```

Listing 3.1 Test stage annotations in JUNIT.

[3] AspectJ, http://www.eclipse.org/aspectj/

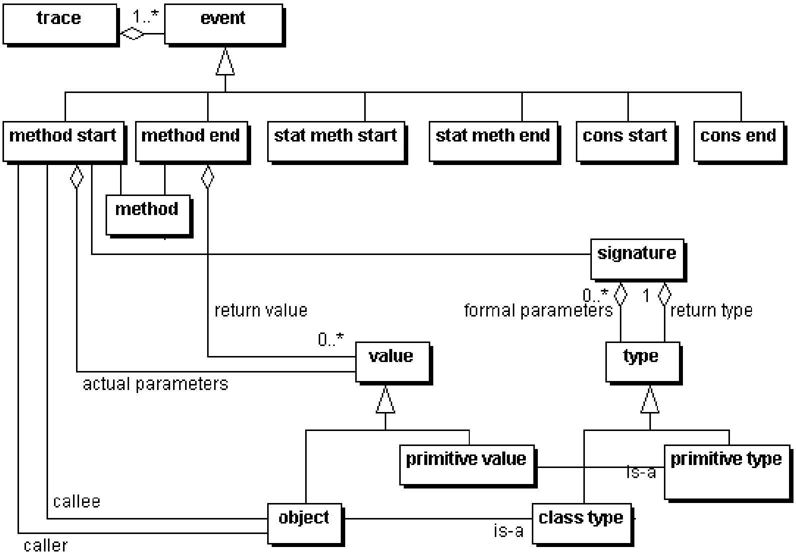


Figure 3.2 Trace metamodel.

Our traces are captured in a common format describing the *events* that can take place during the execution of a system. We maintain a distinction between the beginnings and endings of *method* calls, *static method* calls, and *constructor* calls. An execution trace consists of a series of events, each type of event having its own characteristics. Associated with each `method start`, for example, is a method identifier, two objects (i.e., the caller and the callee), a signature, and zero or more (runtime) parameters; a `method end`-event features a method identifier and the actual return value.

Figure 3.2 shows the model describing our traces. Details for static method calls and constructor calls are omitted, but these are similar to those of the regular calls that are shown in the figure.

3.3.2 Trace reduction

In order to make large scenario diagrams easier to read, we need several abstraction and filtering techniques to reduce the amount of information. In the context of scenario diagrams, one intuitively thinks of omitting or joining objects, classes or combining interactions to shrink the dimensions of the diagram. But which messages and objects can be removed or combined while ensuring that the essence of the test case is preserved? Below, we collect and discuss a catalog of reduction techniques that we have identified in the context of scenario diagrams based on test cases.

- *Constructor hiding.* Omit all constructors and their control flows. This is especially applicable in the initialization stages of complex test cases.

- *Selective constructor hiding.* Omit *irrelevant* constructors, i.e., those pertaining to objects that are never used. This enables the reduction of a diagram's dimensions without loss of essential information. The selection in one particular stage can also be based on objects that appear in certain other stages of the scenario. This can, for example, be used to filter objects created for result validation in test cases.

- *Maximum stack depth.* Omit all interactions that occur above a certain stack depth threshold. Intuitively, this filters low level messages that tend to be too detailed, at least for an initial viewing. This is illustrated by Figures 3.3 and 3.4. A similar abstraction was applied by Rountev and Connell (2005), who (in a static context) consider the length of a call chain.

- *Minimum stack depth.* Omit all interactions below a certain depth threshold, i.e., remove high level interactions. This can be used to omit "control messages" that start up a certain scenario.

- *Fragment selection by using stack depths.* Choose a set of methods of interest and, by selecting appropriate minimum and maximum stack depths, highlight its direct environment. Method selection can be done by, e.g., using the techniques described by Zaidman et al. (2005).

- *Fragment selection by zooming.* Zoom in on areas of interest (Sharp and Rountev, 2005).

- *Pattern recognition.* Summarize recurrent patterns (De Pauw et al., 2001; Systä et al., 2001). Patterns may be identical or very similar (sets of) method calls.

- *Object merging (clustering).* Merge lifelines of closely related (or manually selected) objects, thereby hiding their mutual interactions (Riva and Rodriguez, 2002).

- *Colors.* Use color techniques (e.g., graying out, or using different colors for various fragments) to "distinguish" between (ir-)relevant parts (Sharp and Rountev, 2005).

- *Getters and setters.* Omit simple getters and setters. We will see this abstraction applied in the case studies.

- *Textual techniques.* (1) Omit return values in case of void methods, (2) abbreviate full parameters, and/or (3) abbreviate return values.

3.4 First Implementation: SDR

The design in the previous section has been implemented in a prototype tool, which we call the *Sequence Diagram Reconstruction* (SDR) tool. It is implemented in Java, AspectJ, and Perl, and was used to conduct the first case study described later in the chapter.

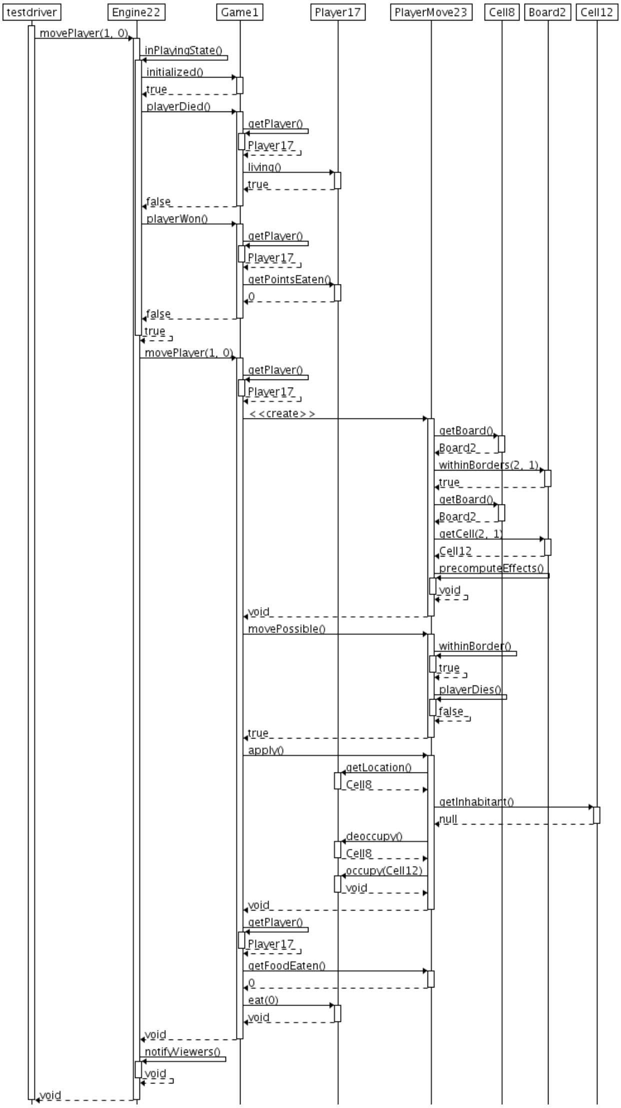


Figure 3.3 Reconstructed scenario diagram with a restricted stack depth.

3.4.1 Scenario diagram metamodel

The scenario diagram metamodel that we use in this implementation is loosely based on the UML 1.1 standard for sequence diagrams (OMG, 2003). It is a simplified version in that we do not focus on genericity: a scenario diagram corresponds to the execution of one particular *scenario* (one test case). As such, constructs such as conditions and repetitions are omitted in our model, whereas detailed information such as runtime parameters and actual return values are included in each diagram. A distinction is made between various

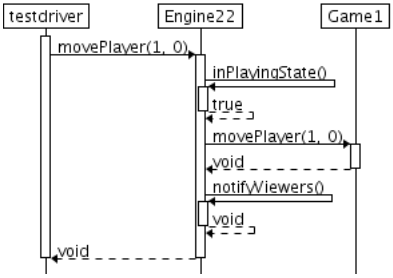


Figure 3.4 Diagram from Figure 3.3, with the stack depth reduced to 2.

types of events.

The model is depicted in Figure 3.5, and illustrates the definition of a scenario diagram as a sequence of *messages* between *objects*. Each message is associated with a corresponding *method*, has zero or more *actual parameters* and sometimes an actual *return value*.

3.4.2 Tracing

Tracers are implemented by specializing the abstract tracing aspect `AbstractTracer` that specifies the common (default) behavior. Upon initialization, it starts an *event listener* and contains only one advice: it notifies the current listener before and after the `startingPoint()` pointcut is encountered. In case of test suites, this pointcut is useful for defining a test case's top level methods. This pointcut, along with pointcuts for the events that need to be traced and the type of listener, are all abstract and can be defined in specific tracers for specific cases, so as to provide maximum flexibility.

The prototype currently offers several "stock" tracers that can be extended for specific cases, e.g., `SimpleTracer` for capturing basic method and constructor calls and `ObjectTracer` for doing so while distinguishing between objects. As an example, consider the situation in which one merely wants to capture all method and constructor calls. In this case, the `SimpleTracer` must be extended in a custom tracer that contains definitions for the abstract pointcuts (Listing 3.2).

```
import sdr.tracers.SimpleTracer;

public abstract aspect CustomTracer extends SimpleTracer {

   protected pointcut theConstructors() :
      call(org.package..*.new(..));

   protected pointcut theCallers() :
      call(* org.package..*.*(..));

   protected pointcut startingPoint() :
      execution(* org.package..*.myStartMethod(..));
}
```

Listing 3.2 Extending an existing tracer to capture method and constructor calls.

3.4.3 Listening to events

Attached to each tracer is an event listener. In event listeners, one can define how to process the captured events. They implement the EventListener interface and, as such, define routines for starting and stopping the listening phase, and for processing generic events.

Our prototype features listeners that perform various tasks, among which are simply printing all events, writing trace files, and gathering statistics. Selecting a certain event listener is done by extending the tracer containing the pointcuts and specifying the listener of choice.

3.4.4 Handling test cases

For the purpose of tracing test cases, our framework includes a tracer, PerStartTracer, that distinguishes between top level methods, i.e., the test cases in a test suite. It is an extension to the ObjectTracer and, as such, identifies unique objects. It is extended by a custom tracer, in which the startingPoint() is typically defined as a pattern that matches testclasses that have "Test" in their names. This way, the listener will be informed whenever a (new) test case in a testclass is executed.

The SequenceListener is responsible for turning trace events into scenario diagrams. It keeps track of all method and constructor calls and performs the mapping between the trace metamodel and the scenario diagram metamodel. The output consists of a collection of scenario diagrams: one diagram for each of the top level methods in the test cases that are being executed. The data is visualized as scenario diagrams using Sequence, a publicly available visualization tool.[4]

3.5 Case Study: JPacman

The objective of our first case study is the evaluation of Sdr. It is exploratory in nature: we attempt to verify the usefulness of test case visualization, and to determine which abstractions are needed and how they are to be applied.

This is achieved by studying the JPacman 1.8.1 application and its test suite. It is a fairly simple Java (1.5) application consisting of approximately 20 classes and 3000 LOC, which implements a rudimentary version of the well known Pacman game. JPacman is used for teaching purposes in a software testing course at Delft University of Technology. In their lab work, students have to extend both the test suite and the functionality of JPacman using a test-driven / test-first approach, and make use of such tools as Eclipse, Ant, and the code coverage tool Clover[5].

JPacman features a JUnit test suite of approximately 1000 LOC. The test cases have been derived systematically using various strategies mostly obtained from Binder (2000), including branch coverage (100%), exercising do-

[4] Sequence, http://sequence-src.dev.java.net/

[5] Clover, http://www.cenqua.com/

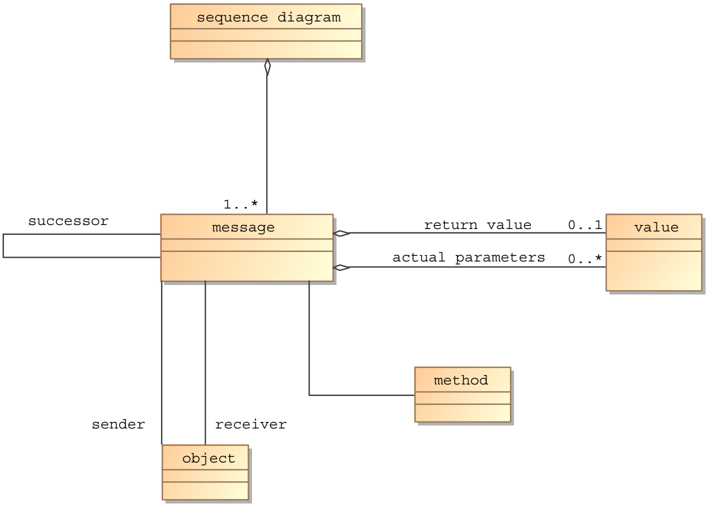


Figure 3.5 SDR's scenario diagram metamodel.

main boundaries at *on points* as well as *off points*, testing class diagram multiplicities, deriving tests from decision tables (describing, e.g., whether moves on the board are possible), and state and transition coverage (in the state machine describing the overall game behavior). The test suite consists of 60 test cases in total, divided over 15 high level acceptance tests exercising the top level public interface only, as well as 45 class level unit tests, all implemented in JUnit (version 4.1).

We have investigated the role that scenario diagrams recovered from these test cases can play when implementing two change requests into JPacman. The first is adding an *undo* feature to JPacman, so that users can get back to life if they missed a food element or unintendedly bumped into a monster. Relevant test cases include moving a player or a monster, as well as losing a game. The second change request is turning JPacman into a *multi-level* game including a stack of mazes growing in complexity. Relevant test cases here deal with winning and restarting the game with a different board.

We used Sdr to derive scenario diagrams from all 60 test cases, and analyzed them in light of the above change scenarios.[6] Below we describe our findings, formulate a number of observations, and establish a connection between the scenario diagrams and program comprehension.

3.5.1 Unit tests

Descriptive statistics for the JPacman case study are shown in Table 3.1, including such data as the number of constructors, objects and methods, as well as the maximum and most common stack depth.

[6] A selection of these diagrams has been included in the most recent JPacman distribution.

	Unit Tests				*Acceptance Tests*			
	Min	**Max**	**Median**	**0.75-Perc.**	**Min**	**Max**	**Median**	**0.75-Perc.**
# Constructors	0	22	1	1	0	543	1	3
# Methods	2	177	13	43	55	1,791	73	170
# Objects	1	32	4	8	39	578	13	14
Max. depth	1	7	4	5	7	9	8	8
Most common depth	1	5	1	3	3	6	5	6

Table 3.1 Descriptive statistics for the JPacman case study.

The data presented was derived from the actual tests themselves, i.e., they do not include the setup phases, in which the fixture is created in which the method under test can be invoked. For that reason, the number of constructors is much lower than the number of objects involved.

A first observation from this table is that most *unit tests* are fairly simple in terms of the number of objects and method calls. Half of the unit test cases use four or less objects (median 4), and three quarters of the test cases use fewer than 8 objects (0.75-percentile). Likewise, three quarters of the test cases have fewer than 43 method calls.

Observation $\mathbf{P_1}$: *Without any abstractions, three quarters of the unit test cases result in scenario diagrams that are small enough to be comprehensible.*

A typical example of such a small diagram is shown in Figure 3.6. The diagram shows how the test method first occupies a given cell, and then verifies that the result is as desired, i.e., that the inhabitant of the cell is indeed equal to the food item that was placed there.

A more complex example that just falls within the 0.75 percentile is shown in Figure 3.7. Although the diagram is large, it is still useful for one of our change requests, namely adding undo functionality. The diagram at hand illustrates a basic player move, which involves creating a `Move` object that is linked to the source and target cells, followed by the invocation of the `apply()` method to actually conduct the move (here, the *Command* design pattern is used (Gamma et al., 1994)). The diagram also shows how the particular subclasses are used, such as a `PlayerMove` (inheriting from `Move`) and a `Player` (inheriting from `Guest`). In this way, the actual method bodies executed are shown, rather than the abstract methods from the superclass.

While Figure 3.7 is understandable in itself, it also demonstrates several opportunities for applying abstractions:

- The diagram contains several simple getters, which merely result in an self-arrow to the currently active lifeline. Since these do not exhibit any object interactions (unless they result in nested calls), they can often be safely omitted.

- The diagram contains several recurrent patterns. For example, there are four fairly large blocks in which the first three objects interact in order to determine the current state of the game (which may affect whether the player move is allowed at all). All four blocks contain the same sequence of calls and interactions, and there is no need to unfold them all.

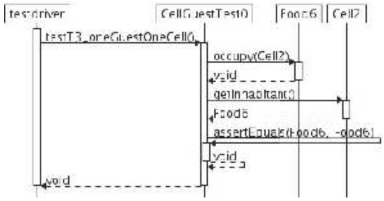


Figure 3.6 Scenario diagram for a typical unit test.

- The diagram includes the interactions for conducting the move, as well as for asserting that the move was indeed applied successfully. In understanding how moves are implemented, there is no need for the detailed interactions involved in checking the results. Omitting all interactions obtained from JUnit's *assert* methods results in the omission (or summarization) of two of the recurrent blocks mentioned above.

Observation P2 : *The abstractions "Omit Getters", "Collapse Recurrent Patterns", and "Omit Asserts" can be applied without loss of essential information required for change requests.*

With these abstractions in place, the scenario diagrams for most JPacman unit tests become sufficiently small. While this will not be the case for unit tests in general, we expect a similar result to hold for other systems:

Observation P3 : *The abstractions in Observation 2 are sufficient to make 90% of the scenario diagrams of unit test cases understandable.*

A final observation that should be made concerning the unit tests, is that according to the XP and test-driven methodologies, tests should run *instantaneously*. That is, it should be possible to run the full test suite within several seconds at most, in order not to discourage developers from running the tests continuously. Performance, scalability, system, or acceptance tests that take longer should not be included in the unit test suite, but in a different suite that should be run at regular intervals (one or more times per day) automatically on the basis of a checkout from the revision system.

As a result, we actually *expect* such metrics as the number of objects and methods involved in unit tests to be low. In the JPacman case, we encountered several test cases in which these metrics were unexpectedly high. The reason turned out to be an error in the loading of maps: the default map (inherited from a superclass of this test case) was always loaded first, and then immediately overwritten by a custom map. Since the default maze was large, and the custom map was deliberately chosen to be small for testing purposes, this was easily derived from the metrics (i.e., the number of constructors involved). While the code was exercised by the test cases, the unnecessary loading was never observed, so no failure was generated.

Observation P4 : *Dynamically obtained test case metrics can help to identify faults and refactoring opportunities.*

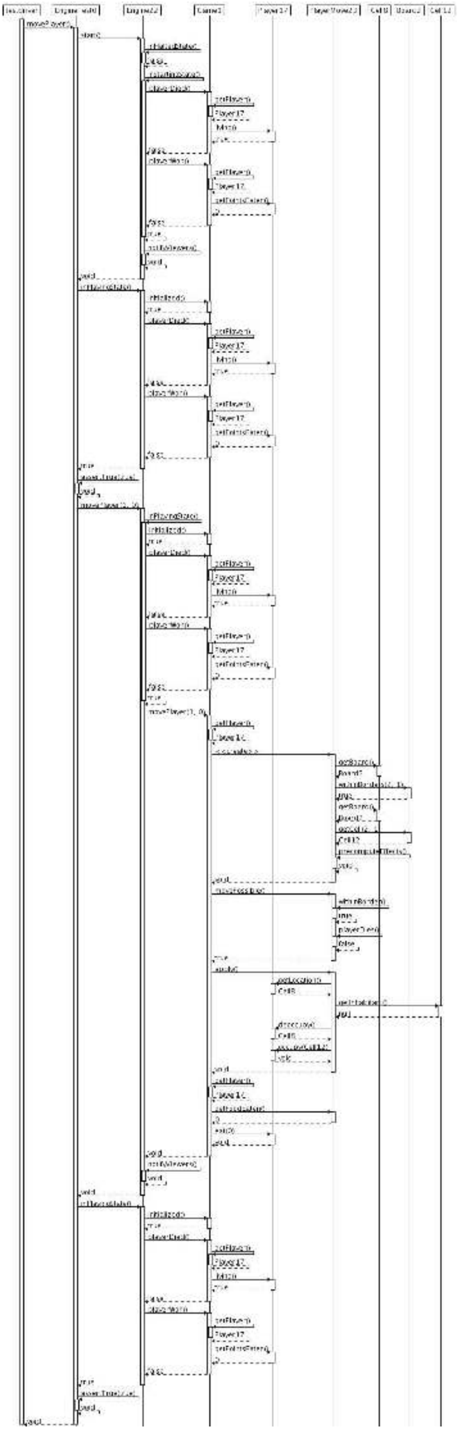


Figure 3.7 Scenario diagram for a player move.

Stack depth	1	2	3	4	5	6	7	8	9
Frequency	13	29	105	210	614	1035	245	88	4

Table 3.2 Depth frequencies in the longest acceptance test.

3.5.2 Acceptance tests

When looking at *acceptance tests*, obviously the figures are substantially larger than those of the unit tests. This is due to the many interactions taking place, and the fact that these tests require more elaborate fixtures, i.e., more objects. This is illustrated by the rightmost portion of Table 3.1.

However, the figures for the tests would have been far greater had the test stages not been separated. This holds true especially for the acceptance tests, as in JPacman these require the construction of large fixtures, e.g., a map with certain dimensions.

Observation P5 : *Separation of the test case stages leads to more readable diagrams without loss of essential information.*

By filtering out these phases, we obtain a smaller diagram that enables us to fully concentrate on the test case being executed. In case the viewer *is* interested in the initialization or the assertions, it can be viewed in a separate diagram.

Observation P6 : *In light of Observation P5, half of the acceptance tests lead to readable scenario diagrams without use of abstractions.*

Among the 15 acceptance tests is a relatively large test case (543 constructors and 1,791 methods) that involves several consecutive movements and game restarts, making it the most complex test case by far. If we look closer at the metrics for this test case – the stack depth frequencies in particular – we obtain the figures of Table 3.2.

Based on these metrics and an initial viewing of the scenario diagram, we applied two abstractions: hiding the control flow of constructors, and limiting the stack depth to 2 (Figure 3.8). This diagram effectively depicts the essence of the test case at hand: instantiating the `Pacman`-class (which would normally induce a large number of interactions), starting the `Engine`, and conducting the moves. For a more detailed view, one can increase the maximum stack depth to 3, resulting in a diagram that (according to Table 3.2) features approximately 100 more calls. It is from these detailed diagrams that the necessary information can be derived for gaining insight into the map loading procedure: this knowledge supports our change request regarding multiple maps.

Observation P7 : *Using more advanced abstractions such as stack depth limitation and hiding control flows of constructors, all of* JPacman*'s acceptance tests lead to readable scenario diagrams.*

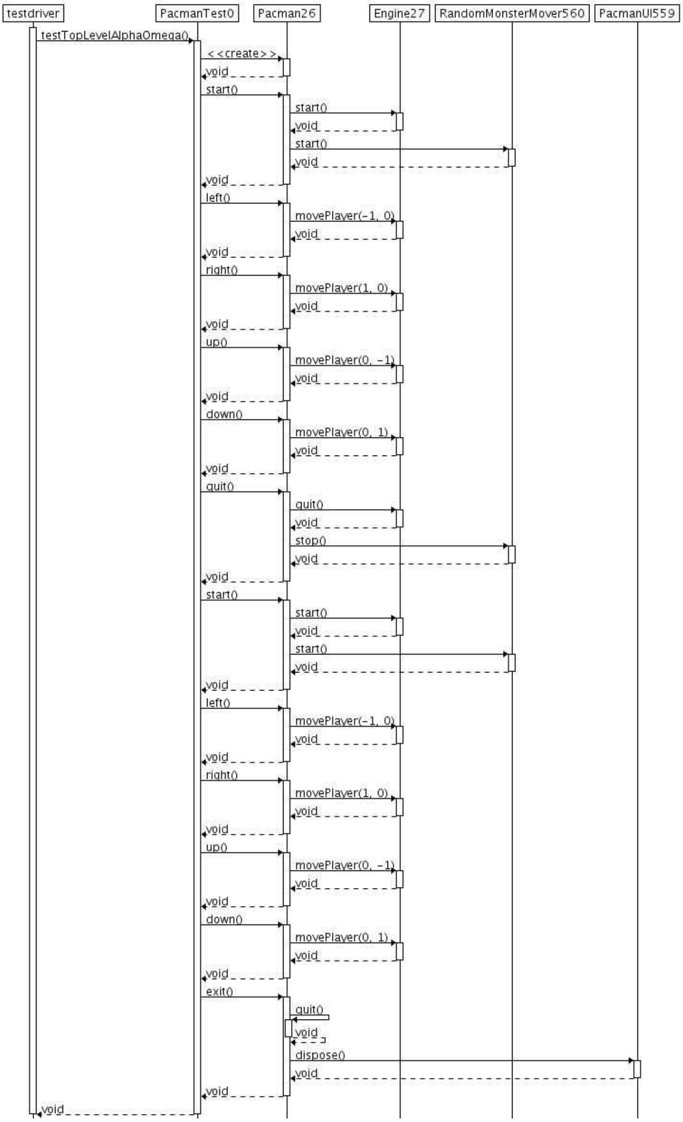


Figure 3.8 Scenario diagram with a max. stack depth of 2.

3.5.3 Discussion & future improvements

The JPacman study has taught us several lessons.

First of all, the dynamically obtained scenario diagrams contain lots of detail, such as runtime parameters, actual return values, and occurrences of late binding. While this is useful, especially for change requests that require detailed knowledge, we often obtain diagrams that are too large to comprehend. This holds true especially in the case of acceptance tests, making abstractions essential in this context.

Separating the test case stages was clearly a useful measure, as most scenario diagrams became significantly smaller when leaving phases (e.g., the setup) to distinct diagrams.

Several abstractions have been successfully applied in viewing the diagrams obtained from the JPacman test suite. Hiding constructors, omitting assertions and stack depth limitation have proven useful to shrink the diagrams without losing the essence of the test cases at hand. The viewer is constantly in control of these abstractions and, by changing the diagram's specifications, can always adjust the abstraction parameters and regenerate the diagram.

With many of the proposed abstraction techniques, however, it is preferable to use them in an interactive fashion, i.e., apply them on demand. One could think of making messages and objects clickable, and subject to (un-)collapsing. Using metrics to automatically apply abstractions is a good starting point, but the viewer must ultimately be in control of what information to show and what to hide.

The treatment of the test cases in our study was an iterative process, resulting in improvements in both our technique and the test cases at hand. Various mistakes (such as duplicate setups) were exposed and have been resolved in the next version.

3.6 Second Implementation: JRET

Following the findings of our earlier efforts on scenario diagram reconstruction, we set out to improve the tool design and to evaluate it on a more representative system two years later. This work was carried out in close collaboration with Roland Voets, whose Master's thesis (Voets, 2008) provides an extensive description and evaluation of the new tool, called JRET (Java Reverse Engineering Tool).

In this section we highlight the most important improvements. We then report on the new case study in Section 3.7.

3.6.1 Scenario diagram metamodel

JRET follows a similar approach to Sdr in that it maintains an internal representation of a scenario diagram that can be manipulated by use of abstraction mechanisms. The internal representation is an instance of a new scenario diagram metamodel. Several modifications have been made to the original metamodel described earlier, resulting in the model in Figure 3.9. There are two main improvements, which both relate to the need to represent abstractions for traces (see Section 3.3.2). First, the notion of *labels* is defined, which serves to annotate messages with such information as the corresponding test case stage. Second, the new model explicitly defines *patterns* of messages to support pattern summarization.

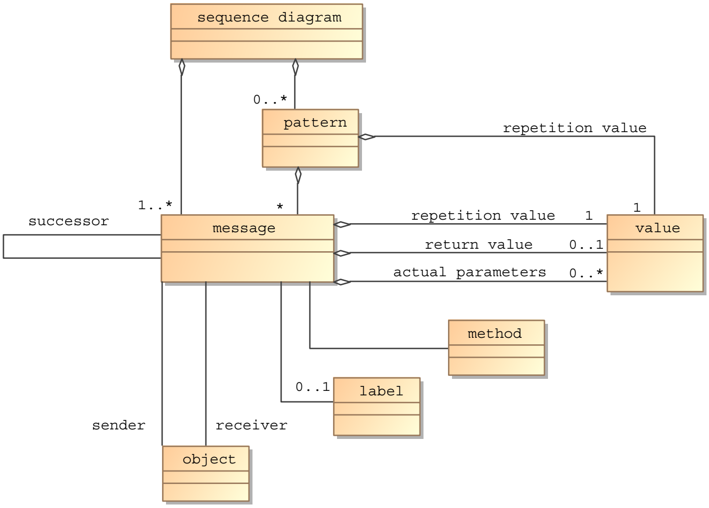


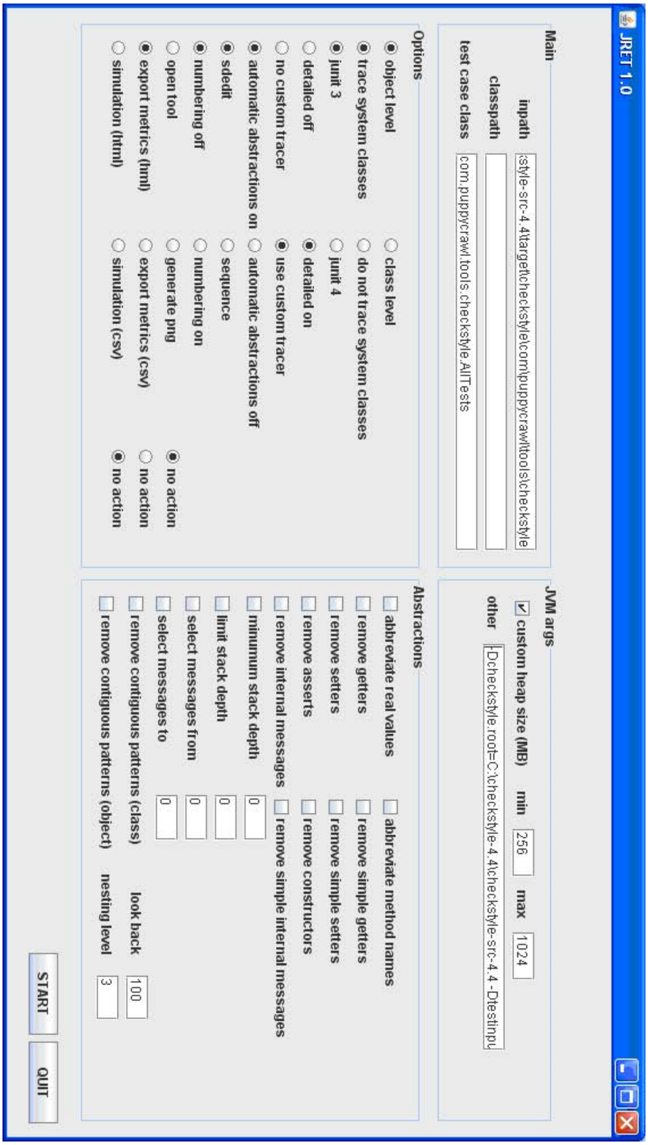
Figure 3.9 JRET's scenario diagram metamodel.

3.6.2 Tool implementation

Contrary to Sdr, JRET is entirely written in AspectJ and Java and no longer relies on Perl scripts. Furthermore, to render the new tool more accessible to the research community, JRET has a user-friendly interface and has been implemented as an Eclipse plugin. It is properly documented and is publicly available online[7]. Figure 3.10 shows JRET's user interface. The main advantage of the GUI is that it allows for an overview and easy customization of the available trace reduction techniques. The tool offers implementations of the following abstractions:

- Object or class level tracing.
- Filtering interactions with JDK classes.
- Abbreviation of method identifiers, runtime parameters, and return values.
- Removal of getters and/or setters (with or without control flows).
- Removal of constructors (with or without control flows).
- Removal of JUnit asserts.
- Removal of internal messages within classes (with or without control flows).
- Stack depth limitation.

[7] JRET 1.0, `http://sourceforge.net/projects/jret/`



JRET 1.0

Main

inpath :style-src-4.4\target\checkstyle\com\puppycrawl\tools\checkstyle

classpath

test case class com.puppycrawl.tools.checkstyle.AllTests

Options

● object level ○ class level

● trace system classes ○ do not trace system classes

● junit 3 ○ junit 4

○ detailed off ● detailed on

○ no custom tracer ● use custom tracer

● automatic abstractions on ○ automatic abstractions off

● sdedit ○ sequence

● numbering off ○ numbering on

○ open tool ○ generate png ● no action

● export metrics (hml) ○ export metrics (csv) ○ no action

○ simulation (html) ○ simulation (csv) ● no action

JVM args

☑ custom heap size (MB) min 256 max 1024

other -Dcheckstyle.root=C:\checkstyle-4.4\checkstyle-src-4.4 -Dtestinpu

Abstractions

□ abbreviate real values □ abbreviate method names

□ remove getters □ remove simple getters

□ remove setters □ remove simple setters

□ remove asserts □ remove constructors

□ remove internal messages □ remove simple internal messages

□ minumum stack depth 0

□ limit stack depth 0

□ select messages from 0

□ select messages to 0

□ remove contiguous patterns (class) look back 100

□ remove contiguous patterns (object) nesting level 3

Figure 3.10 JRET's graphical user interface.

- Pattern summarization.

In particular, the pattern summarization technique is a potentially powerful asset, as our earlier experiments suggested the presence of numerous repetitions in execution traces. The algorithm used in JRET detects recurring contiguous patterns (similar to Hamou-Lhadj and Lethbridge (2002)) and visualizes them as repetition constructs, which contain a number that signifies the number of repetitions involved (Figure 3.11). Additionally, JRET offers

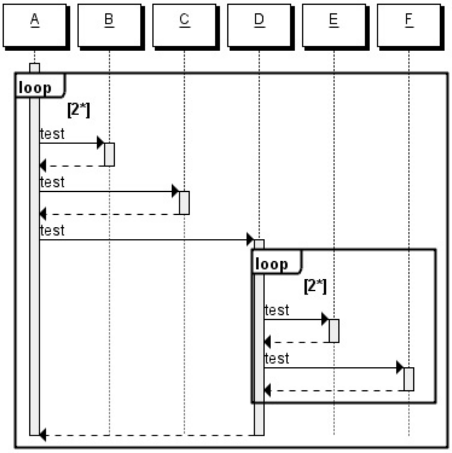


Figure 3.11 Summarizing recurrent patterns in JRET.

support for two different scenario diagram visualization tools: Sequence (also used in Sdr), and SDedit[8]. The latter visualization tool was found to have several important advantages during experimentation, the most important being its support for labels, pattern visualization, and zooming. It is also more flexible in terms of output formats, and allows for the visualization of multiple diagrams at once (using tabs). The latter feature may prove useful in studying several test case executions or test case stages at once.

The case study in the following section provides examples of scenario diagrams generated by JRET.

3.7 Case Study: Checkstyle

Checkstyle[9] is an open-source tool that validates Java code. It uses a set of coding standards that can be extended by third party developers. The program contains 57 kLOC and consists of 310 classes distributed over 21 packages. Checkstyle is an interesting case study not merely for its representative size, but also because of the test suite that contains almost 500 unit tests. Unfortunately, the developers have notified us that no acceptance tests are available, nor will they be in the short term.

The case study consists of two parts. First, we identify the need for abstractions through a brief study of certain *quantitative* aspects of Checkstyle, and apply the necessary abstractions to reduce the scenario diagrams in Section 3.7.1. Next, once more we take the perspective of a developer who is faced with two change requests, and assess how JRET's scenario diagrams support these tasks (Section 3.7.2).

[8] SDedit, `http://sourceforge.net/projects/sdedit/`

[9] Checkstyle 4.3, `http://checkstyle.sourceforge.net/`

Metric (per test case)	Min	Max	Median	0.75-Perc.
# Constructors	0	3,834	139	259
# Methods	3	69,031	2,571	5,512
# Objects	3	5,748	425	812
Max. stack depth	2	42	15	18
# Getters	0	31,205	1,189	2,781
# Simple getters	0	21,098	1,124	2,674
# Setters	0	598	13	14
# Simple setters	0	595	10	10
# Internal messages	0	14,389	521	1,218
# Simple internal messages	0	6,157	37	91

Table 3.3 Descriptive statistics for the CHECKSTYLE case study.

In the following, we provide a summarization of the most important findings: the reader is referred to (Voets, 2008) for more detailed descriptions and additional observations.

3.7.1 Quantitative evaluation

In this experiment we instrument and execute CHECKSTYLE's test suite, and measure key aspects such as the maximum stack depth and number of constructors and methods involved in the unit tests. These measurements may provide hints as to the abstraction techniques that could prove the most useful in reducing the resulting scenario diagrams. The next step is the actual application of JRET's abstraction mechanisms, and determining the extent to which they are successful.

Table 3.3 shows descriptive statistics that were derived from the instrumented execution of CHECKSTYLE's 493 unit tests. Note that we distinguish between getters and simple getters: the latter are trivial as they entail merely one event, whereas getters of the former type lead to multiple events.

From the measurements, it becomes clear that half of the test cases use more than 425 objects (median), and that one quarter of the test cases uses more than 812 objects (0.75-percentile). Likewise, one quarter of the test cases induces more than 5,512 method calls. These figures show that CHECKSTYLE is far more challenging than JPACMAN: in order to reduce the traces such that they can be visualized as scenario diagrams, we need powerful abstractions.

Observation C_1 : *Without any abstractions, the majority of* CHECKSTYLE*'s test cases result in scenario diagrams that are too large to be comprehensible.*

The table also shows that the test cases contain a relatively large number of getters compared to the number of setters: three quarters of the cases have fewer than 2,781 getters, compared to only 14 setters. This implies that the removal of getters could prove a potentially effective abstraction technique in this case.

Observation C$_2$: *The execution of* Checkstyle*'s test suite yields numerous getters, the omission of which could prove a powerful abstraction mechanism.*

To quantitatively evaluate JRET's abstraction mechanisms, we first apply them consecutively on each of the 493 traces. Table 3.4 shows the results for a representative sample, i.e., one test case for each package. It depicts the number of remaining calls in each test case after the consecutive application of several automatic abstractions.

As it turns out, only two out of the 18 test cases displayed in Table 3.4 are small enough to be comprehended without abstractions. The remaining test cases contain more than a hundred calls and thus require abstractions for them to become human-readable. Fortunately, the table shows that JRET's abstractions were successful for all 18 test cases. As for the entire test suite, only 54 out of the 493 test cases are sufficiently small to be comprehended directly. In other words, 89% of Checkstyle's test suite is in need of abstractions.

Observation C$_3$: *89% of* Checkstyle*'s test cases require abstractions to render them comprehensible.*

Table 3.5 shows the effectiveness of each abstraction technique by itself, i.e., when applied *separately* rather than consecutively. It shows the same sample of 18 test cases, and provides average values for the entire test suite.

We observe that the removal of getters is indeed a powerful abstraction technique for Checkstyle, yielding an average reduction of 51%. Removing internal messages and limiting the stack depth to 1 seem even more effective, yielding average reductions of 90% and 95%, respectively. However, we note that in contrast to getters and setters, the former techniques may remove too many essential calls: for example, the apparent effectiveness of internal message removal is presumably the result of many test cases being initiated by internal messages, meaning that their removal will filter the entire control flows as well. Therefore, a qualitative evaluation is needed to assess the value of reduction techniques *in practice*.

Observation C$_4$: *Stack depth limitation and the removal of internal messages and getters are the most effective abstraction techniques for* Checkstyle*'s test suite.*

3.7.2 Qualitative evaluation

With a proper set of scenario diagrams in place, we proceed with a qualitative evaluation of JRET. Specifically, we consider a maintenance context in which a sufficient knowledge of Checkstyle's *checks* is needed to implement a new check. We employ JRET to support this understanding process. (Another experimental context involved understanding Checkstyle's output handling, and is covered by Voets (2008).)

Test case	Total calls	Remove asserts	Remove constructors	Remove getters	Remove setters	Limit depth (=1)	Remove patterns (class)
PackageNamesLoaderTest.testNoFile	9	8	8	8	8	8	7
api.AbstractViolationReporterTest.testGetMessageBundleWithPackage	9	8	8	8	8	8	6
checks.NewlineAtEndOfFileCheckTest.testNewlineAtEndOfFile	296	295	204	120	102	96	87
checks.blocks.EmptyBlockCheckTest.testDefault	6,530	6,519	6,078	2,064	1,971	10	9
checks.coding.IllegalInstantiationCheckTest.testIt	7,407	7,400	6,929	2,123	2,054	33	29
checks.design.InterfaceIsTypeCheckTest.testDefault	996	994	844	361	304	92	82
checks.duplicates.StrictDuplicateCodeCheckTest.testSmallMin	626	625	525	341	323	63	58
checks.header.HeaderCheckTest.testNoHeader	258	258	183	76	76	76	61
checks.imports.ImportOrderCheckTest.testDefault	2,160	2,155	1,990	955	894	78	45
checks.indentation.IndentationCheckTest.testTabs	4,720	4,718	3,992	1,202	993	96	83
checks.j2ee.EntityBeanCheckTest.testCreate	6,030	6,018	5,452	2,124	2,008	18	16
checks.javadoc.JavadocMethodCheckTest.testTags	4,140	4,139	3,306	857	776	69	62
checks.metrics.JavaNCSSCheckTest.test	4,631	4,626	4,309	1,852	1,687	20	13
checks.naming.MethodNameCheckTest.testDefault	5,206	5,204	4,640	1,451	1,393	92	82
checks.sizes.FileLengthCheckTest.testOK	5,003	5,002	4,446	1,327	1,274	85	78
checks.whitespace.WhitespaceAroundTest.testIt	14,849	14,813	14,055	5,694	5,437	10	9
filters.SuppressElementTest.testDecideByLine	122	122	58	58	58	58	52
grammars.VarargTest.testCanParse	702	701	589	259	201	80	74

Table 3.4 Reducing the diagrams through the *consecutive* application of automatic abstractions.

Test case	**Total calls**	**Remove constructors**	**Remove asserts**	**Remove getters**	**Remove simple getters**	**Remove setters**	**Remove simple setters**	**Remove internal msgs**	**Remove simple internal**	**Limit depth (=1)**	**Remove patterns (class)**
PackageNamesLoaderTest.testNoFile	9	88%	11%	11%	11%	0%	0%	0%	0%	88%	11%
AbstractViolationReporterTest.testGetMessageBundleWithPackage	9	11%	11%	88%	0%	0%	0%	0%	0%	77%	22%
checks.NewlineAtEndOfFileCheckTest.testNewlineAtEndOfFile	296	30%	0%	32%	21%	22%	2%	89%	3%	95%	90%
checks.blocks.EmptyBlockCheckTest.testDefault	6,530	6%	0%	61%	51%	3%	0%	99%	1%	99%	49%
checks.coding.IllegalInstantiationCheckTest.testIt	7,407	6%	0%	65%	44%	2%	0%	99%	0%	99%	49%
checks.design.InterfaceIsTypeCheckTest.testDefault	996	15%	0%	49%	36%	16%	1%	98%	1%	99%	34%
checks.duplicates.StrictDuplicateCodeCheckTest.testSmallMin	626	15%	0%	31%	12%	10%	0%	97%	2%	98%	38%
checks.header.HeaderCheckTest.testNoHeader	258	29%	0%	43%	29%	60%	1%	98%	0%	99%	13%
checks.imports.ImportOrderCheckTest.testDefault	2,160	7%	0%	48%	40%	7%	0%	99%	1%	99%	31%
checks.indentation.IndentationCheckTest.testTabs	4,720	15%	0%	63%	30%	15%	0%	99%	8%	99%	26%
checks.j2ee.EntityBeanCheckTest.testCreate	6,030	9%	0%	56%	42%	6%	0%	99%	1%	99%	50%
checks.javadoc.JavadocMethodCheckTest.testTags	4,140	20%	0%	63%	23%	4%	0%	99%	2%	99%	47%
checks.metrics.JavaNCSSCheckTest.test	4,631	6%	0%	53%	44%	7%	0%	99%	1%	99%	36%
checks.naming.MethodNameCheckTest.testDefault	5,206	10%	0%	61%	51%	3%	0%	99%	0%	99%	59%
checks.sizes.FileLengthCheckTest.testOK	5,003	11%	0%	62%	52%	3%	0%	100%	0%	99%	63%
checks.whitespace.WhitespaceAroundTest.testIt	14,849	5%	0%	56%	47%	3%	0%	99%	4%	99%	40%
filters.SuppressElementTest.testDecideByLine	122	50%	1%	28%	12%	48%	0%	22%	2%	90%	13%
grammars.VarargTest.testCanParse	702	15%	0%	48%	37%	24%	1%	93%	1%	99%	27%
Average (493 test cases)	5,021	13%	1%	51%	36%	8%	0%	90%	2%	95%	37%

Table 3.5 The effectiveness of JRET's abstraction techniques when applied *separately* to the test cases.

	Total Calls	Remove Asserts	Remove Constr.	Remove Getters	Remove Setters	Limit Stack Depth	Remove Patterns (Class)
DefaultComesLastCheckTest.testIt	1,451	1,449	1,305	491	434	96	84
UncommentedMainCheckTest.testDefaults	3,871	3,867	3,585	1,376	1,310	96	70
AvoidStarImportTest.testDefaultOperation	4,340	4,336	4,062	1,630	1,573	96	70
Newline [...] testNoNewlineAtEndOfFile	344	342	266	154	136	76	65

Table 3.6 The number of calls after the consecutive application of JRET's abstractions to four test cases.

Selecting suitable diagrams

Checks are the subject of most of Checkstyle's unit tests, so we may choose from many test cases. We have selected a sample of four test cases to study:

- `checks.coding.DefaultComesLastCheckTest.testIt`
- `checks.UncommentedMainCheckTest.testDefaults`
- `checks.imports.AvoidStarImportTest.testDefaultOperation`
- `checks.NewlineAtEndOfFileCheckTest.testNoNewlineAtEndOfFile`

These test cases concern checks that (1) have names that clearly suggest their functionality, and (2) reside in different packages. Specifically, they involve testing if `default` succeeds the last `case` in a `switch` statement, testing for uncommented `main()` methods, testing if there are `import` statements using an asterisk, and testing if a Java file ends with a newline character, respectively.

Comparing different checks

Table 3.6 shows the number of calls after the consecutive application of JRET's abstractions to our selection of test cases. Note that the first three diagrams contain identical numbers of calls after the stack depth limitation: we therefore study these diagrams and the fourth diagram separately.

Looking at the scenario diagrams of the first three test cases, they indeed appear to follow a similar pattern. This pattern is shown in Figure 3.12 and consists of 11 steps:

1. A `Checker` object is created and configured by the test class.
2. A listener is added to the `Checker` object.
3. The `Checker` object executes a `fireAuditStarted()` method on itself.
4. The `Checker` object notifies its listener using an `auditStarted(AuditEvent)` method call.
5. The `Checker` object executes a `process(File[])` method on a `TreeWalker` object.
6. The `TreeWalker` object invokes several calls, among which are `parse(FileContents)` and `walk(DetailAST, FileContents)`.
7. The `Checker` object destroys the `TreeWalker` object.

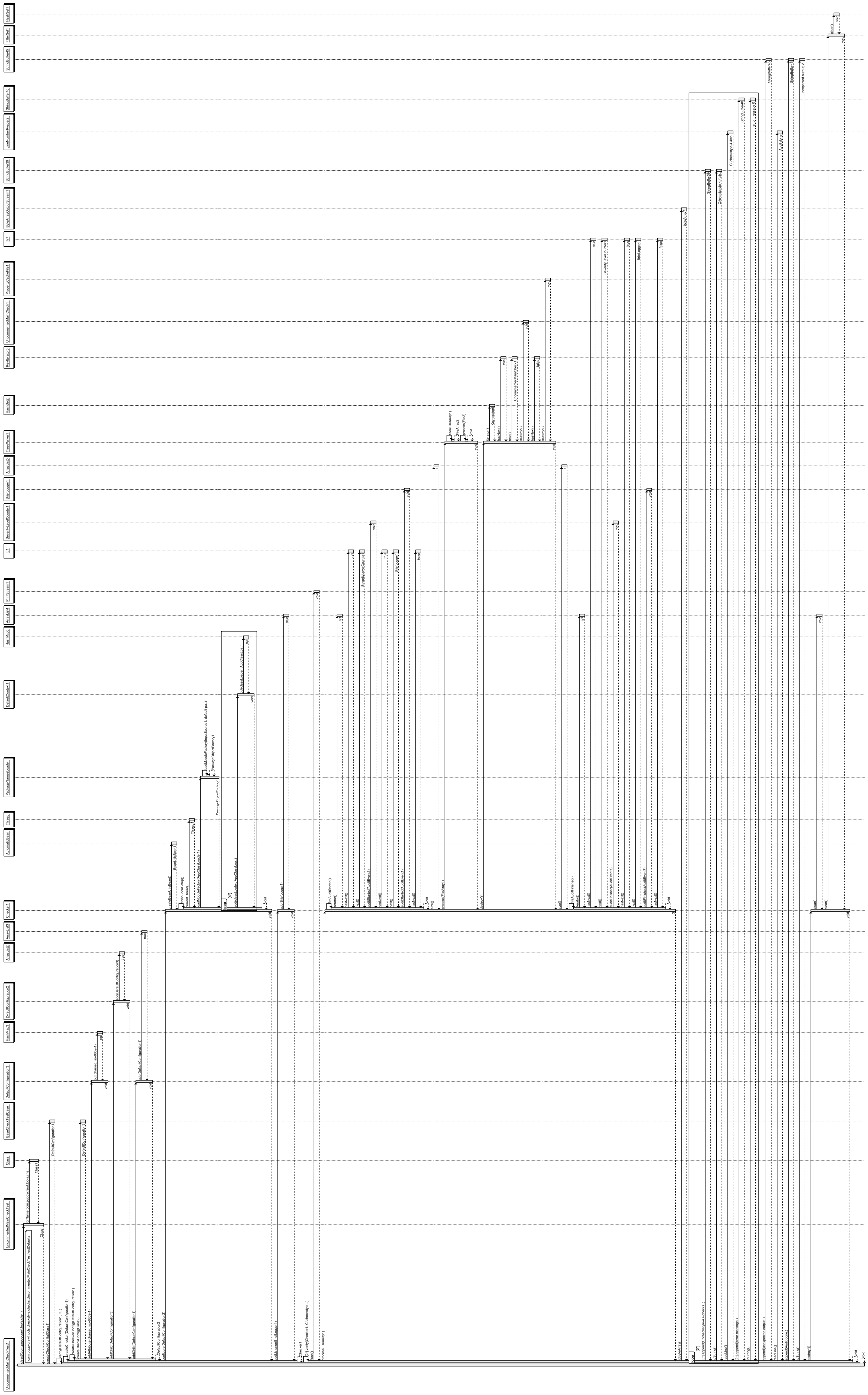


Figure 3.12 Visualization of the `UncommentedMainCheckTest.testDefaults` test case.

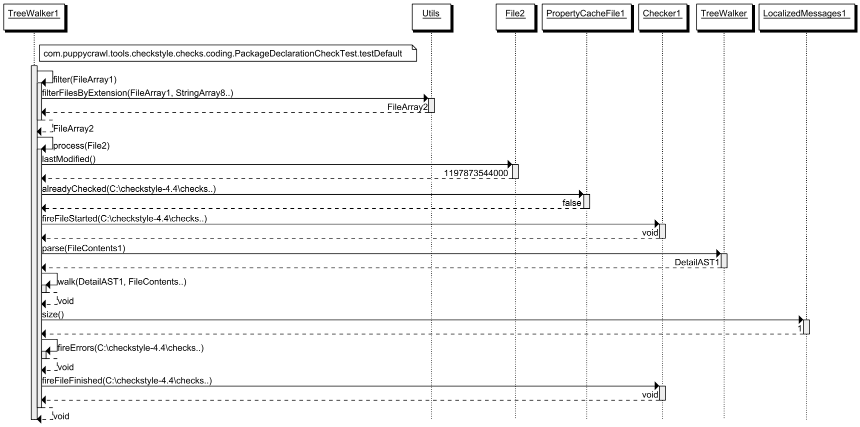

TreeWalker1
Utils
File2
PropertyCacheFile1
Checker1
TreeWalker
LocalizedMessages1
com.puppycrawl.tools.checkstyle.checks.coding.PackageDeclarationCheckTest.testDefault
filter(FileArray1)
filterFilesByExtension(FileArray1, StringArray8..)
FileArray2
FileArray2
process(File2)
lastModified()
1197873544000
alreadyChecked(C:\checkstyle-4.4\checks..)
false
fireFileStarted(C:\checkstyle-4.4\checks..)
void
parse(FileContents1)
DetailAST1
walk(DetailAST1, FileContents..)
void
size()
1
fireErrors(C:\checkstyle-4.4\checks..)
void
fireFileFinished(C:\checkstyle-4.4\checks..)
void
void

Figure 3.13 Control flow of `process(File)` within the first three test cases.

8. The `Checker` object performs a `fireAuditFinished()` method call.
9. The `Checker` object notifies its listener using an `auditFinished(AuditEvent)` method call.
10. An error message is built.
11. The `Checker` object is destroyed.

The execution of the fourth test case differs with respect to step 5: the `process(File)` method is invoked on the `NewLineAtEndOfFileCheck` (Figure 3.13) rather than on the `TreeWalker` object (Figure 3.14), resulting in an alternative pattern for step 6:

1. `filter(File[])`
2. `fireFileStarted(String)`
3. `endsWithNewLine(RandomAccessFile)`
4. `log(..)`
5. `fireErrors(String)`
6. `fireFileFinished(String)`

A closer inspection of the source code suggests that the former three checks involve walking an Abstract Syntax Tree (AST) that is generated from the contents of the file being audited, whereas the fourth check does not.

Observation C5 : *The side-by-side comparison of similar scenario diagrams can reveal essential differences.*

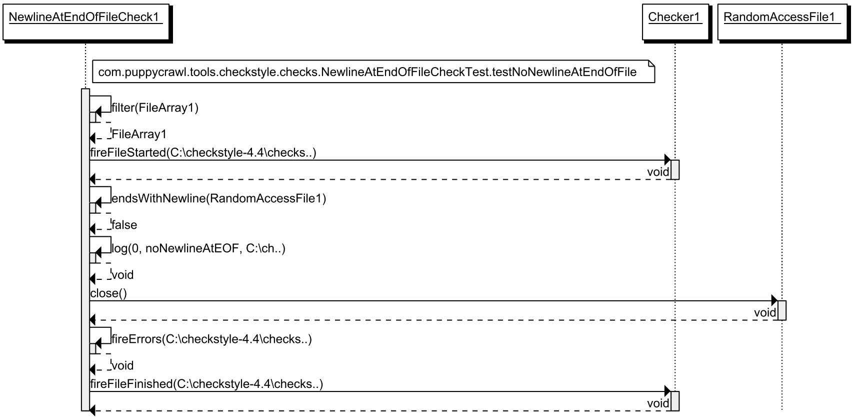

NewlineAtEndOfFileCheck1
Checker1
RandomAccessFile1
com.puppycrawl.tools.checkstyle.checks.NewlineAtEndOfFileCheckTest.testNoNewlineAtEndOfFile
filter(FileArray1)
FileArray1
fireFileStarted(C:\checkstyle-4.4\checks..)
void
endsWithNewline(RandomAccessFile1)
false
log(0, noNewlineAtEOF, C:\ch..)
void
close()
void
fireErrors(C:\checkstyle-4.4\checks..)
void
fireFileFinished(C:\checkstyle-4.4\checks..)
void

Figure 3.14 Control flow of `process(File)` within the fourth test case.

Summary of findings

Following our findings in the previous section, we studied additional test cases in a similar fashion. This study confirmed our assumption that there exist two types of checks:

- **AST checks:** checks that rely on ASTs. The AST is walked upon, thereby retrieving and inspecting tokens. Typical checks that belong to this category include checking if a package declaration exists, checking if `default` comes after the last `case` in a `switch` statement, and checking if there exists an uncommented `main()` method.

- **non-AST checks:** checks that can be performed directly on the contents of the Java files to be audited. Typical checks that belong to this category include checking if a file ends with a newline character, and checking if a file contains tab characters.

In order to implement a new check in Checkstyle, one must determine the appropriate check type. Furthermore, a closer investigation of the scenario diagrams involved (Voets, 2008) has resulted in a "recipe" for creating new checks:

- **AST checks:**

  1. Create a new class that extends the `Check` class.
  2. Define and implement the `getDefaultTokens()` method, which specifies the token types of interest.
  3. Define and implement the `visitToken(DetailAST)` method, in which the token types of interest are retrieved using methods of the `DetailAST` class. Consequently, this method can be used to check the tokens. Also, errors can be logged using the `log(..)` function.

4. *Optional*. Define and implement the `beginTree(DetailAST)` method, in which initialization is performed prior to walking the AST.
5. *Optional*. Define and implement the `finishTree(DetailAST)` method, which is executed after the AST is walked. For example, it can be used to log errors using the `log(..)` function.

- **non-AST checks:**

  1. Create a new class that extends the `AbstractFileSetCheck` class.
  2. Define a `process(File[])` method which has a void return type. All methods in the following steps should be defined within this `process(File[])` method.
  3. Add the methods that filter the `File[]` object, and retrieve the `MessageDispatcher` object.
  4. Create a loop that retrieves all files in the file array.
  5. Add the method that performs the `fireFileStarted(String)` method call prior to the actual check.
  6. Write the code that performs the check, preferably in a separate method, which should be executed on each retrieved file.
  7. Add the `log(..)` function that logs the errors in case the check returns false.
  8. Add the methods that perform the `fireErrors(String)` and the `fireFileStarted(String)` method calls.

This case study has illustrated how the use of reverse-engineered sequence diagrams can assist in understanding a system's inner workings: through a study of relevant test cases it can be derived how specific functionalities are implemented, which supports maintainers during modifications or change requests.

3.7.3 Threats to validity

The case studies in this chapter have certain limitations that hinder the generalization of our conclusions.

First, in spite of its extensive test suite, JPacman is a relatively *small* system, which means that the observations and conclusions made do not necessarily hold for larger systems. We addressed this threat by conducting another case study on a system that is more representative in terms of size. Still, real life software may involve traces of larger sizes and of a different nature: for example, whereas the stack depth limitation technique works quite well for recurrent patterns, it may remove too many essential events in case of recursion.

Second, our approach relies on test suites because they provide the necessary execution scenarios. JPacman and Checkstyle are well-maintained applications that feature test suites with a high code coverage. This does not necessarily hold true for real life cases: for example, if a certain industrial system does not provide test cases that pertain to a certain feature, then this functionality cannot be studied through test case visualization. Furthermore, the fact that Checkstyle only offers unit tests means that the applicability of our tools to *acceptance tests* remains unclear, at least for large programs. Additional research involving representative systems with both unit and acceptance tests is needed for further exploration.

3.8 Related work

Using dynamic analysis for program comprehension purposes is an active field of research. This section provides a short overview of related work.

3.8.1 General approaches

Various approaches reconstruct scenario and interaction diagrams based on static analysis of program code (Rountev and Connell, 2005; Kollmann and Gogolla, 2001a; Rountev et al., 2005; Kollmann et al., 2002). The techniques that are used vary from mapping of control flow graphs (Rountev et al., 2005) to interprocedural dataflow analysis (Rountev and Connell, 2005). A comparison of various approaches is presented by Kollmann et al. (2002).

3.8.2 Sequence diagrams

UML sequence diagrams (and variations thereof) have since long been used to visualize dynamically obtained information from a system. We now provide a selection of the existing research in this area.

De Pauw et al. (2001) propose several abstractions in reconstructing sequence diagrams. Among these techniques is pattern recognition, and they introduce the concept of execution pattern notation (De Pauw et al., 1998). Their techniques (formerly implemented in *Jinsight*) are used in a plugin that is part of Eclipse's Test & Performance Tools Platform (TPTP) project[10].

Systä et al. (2001) aid in the understanding of Java systems in an environment called Shimba, which uses both static analysis and dynamic information. The Shimba environment considers static and dynamic information to be complementary and uses static information to bring focus to dynamically generated diagrams, and vice versa.

Jerding et al. (1997) have developed the ISVis (Interactive Scenario Visualizer) environment. Its main purpose is to alleviate the architecture localization problem, or the problem of finding the exact location in a system's architecture where a specific enhancement can be inserted. Most of the abstraction techniques that are being used are not fully automatic.

Briand et al. (2006) reverse engineer UML sequence diagrams for (1) program understanding in the face of incomplete documentation, and (2) quality

[10] TPTP, `http://www.eclipse.org/tptp/`

assurance, when reverse engineered diagrams are compared with existing design sequence diagrams. Their major contribution lies in the fact that they are among the first to tackle the reverse engineering of UML sequence diagrams from distributed (Java) systems. However, they do not reason from the viewpoint of test suites, and abstractions play a minor role.

3.8.3 General visualizations

Riva and Rodriguez (2002) combine static and dynamic analysis to reconstruct message sequence charts. In their trace-based approach, they provide an abstraction mechanism based on the decomposition hierarchy that is extracted from the system's source code. It is not described how the scenarios are defined, and in dealing with large diagrams, they only offer manual abstraction techniques.

Furthermore, Pacione et al. (2003) make a comparison of dynamic visualization techniques, and Hamou-Lhadj and Lethbridge (2004) discuss a series of trace visualization tools from literature.

3.9 Conclusions and Future Work

Testing is an essential part of the software life cycle. Not only are tests important to guarantee correctness: in the context of Agile methods and eXtreme programming, test cases are used as documentation as well. Moreover, by understanding a system's test suite, one can gain a great deal of knowledge about its inner workings. It is for this reason that visualizing both units tests and acceptance tests can be of great help in program comprehension.

To this end, we have employed dynamic analysis and scenario diagrams. By gathering knowledge of a system's test suite, we gain more insight into the way the application works. We have discussed the issues and design choices that we encountered and, through several examples, elaborated on our choices for these techniques.

To address the scalability issues associated with dynamic analysis and scenario diagrams, we have established a set of metrics that recommend a number of abstractions that should be used to keep the scenario diagram (for each test case stage) readable, while preserving the desired amount of detail. In a second iteration, we developed a more robust and user-friendly tool and made it available online.

We performed two case studies in which we sought to answer the research questions that were posed in the introduction. Based on our experiences, we can now formulate answers to these questions:

- JUnit tests are a useful starting point for program comprehension. They induce scenarios that effectively decompose a system's functionalities.

- The scenario diagrams that are obtained from these tests offer knowledge about a system's inner workings. Our case studies suggest that through the application of several abstractions, scenario diagrams effectively visualize how test cases (and the associated functionalities) work, which aids in the planning of change requests.

- The number and types of abstractions that are necessary to make the scenario diagram of a test case sufficiently understandable, depend greatly on the type of tests. We have listed several abstraction techniques, implemented them in our tool implementations, and quantitatively evaluated their effectiveness in our second case study.

- Our experiences indicate that in case of unit tests, simple techniques such as the omission of getters and setters and the collapsing of recurrent patterns suffice, whereas more complex acceptance tests require more complicated filterings such as stack depth limitation and control flow hiding.

Having answered our research questions, we now conclude this section by listing our contributions:

- We have presented a technique to effectively trace the various stages of a JUnit test case.

- We have listed a broad range of abstractions in the context of scenario diagrams, some existing and some new. These have been implemented as a publicly available Eclipse plugin, called JRET.

- By means of two case studies, we have collected preliminary evidence of the usefulness of our approach for test suite and program understanding.

Future work

Developing visualizations that are specifically optimized for test suite comprehension is a relatively new field of research. Besides investigating existing general purpose visualizations, we have established a number of research directions that we wish to pursue in the future.

First, it could be argued that our approach provides an insight into the workings of test cases rather than of a system's functionalities: For example, certain features may only be exhibited by long sequences of user interactions that are not captured by any of the system's test cases. Therefore, more work is needed to investigate and characterize this duality.

Second, we proposed preliminary list of trace reduction techniques in (Section 3.3.2), but there is a need for further research into their comparative performance and applicability in different subject systems. As it stands, such assessments are not available in the literature. We will examine trace reduction techniques in more detail in the next chapter.

Third, while our case studies yielded promising results, focusing on more complex systems or adverse circumstances will undoubtedly raise new issues. Issues one could think of are (1) how to deal with threads, (2) how to handle incomplete test suites, and (3) how to deal with software systems that have no clear unit testing approach and rely merely on integration and acceptance tests (e.g., Apache Ant (Van Geet and Zaidman, 2006)).

An Assessment Methodology for Trace Reduction Techniques[*]

Chapter 4

In the previous chapter, we proposed the use of a traditional visualization technique to convey a system's runtime information to the user. Although various trace reduction techniques have been proposed in the literature to address the scalability concerns involved in such approaches, their applicability in different contexts often remains unclear because extensive comparisons are lacking. In this chapter, we propose an assessment methodology for the evaluation and comparison of trace reduction techniques. We illustrate the methodology using a selection of four types of reduction methods found in literature, which we evaluate and compare using a test set of seven large execution traces. Our approach enables a systematic assessment of trace reduction techniques, which eases the selection of suitable reductions in different settings, and allows for a more effective use of dynamic analysis tools in software maintenance.

4.1 Introduction

The use of dynamic analysis has become increasingly popular in various stages of the software development process. Among the areas of interest is program comprehension, which constitutes an essential part of many maintenance tasks (Basili, 1997; Corbi, 1989): the engineer must sufficiently understand the program at hand before any action can be undertaken. In doing so, a mental map is built that bridges the gap between the program's high-level concepts and its source code (von Mayrhauser and Vans, 1995; LaToza et al., 2006).

There exist various approaches to gain knowledge of a software system. Static analyses focus on such artifacts as source code and documentation, and potentially cover all of the program's execution paths. Dynamic analysis, on the other hand, concerns the examination of the program's behavior at runtime, which offers the ability to reveal object identities and occurrences of late binding (Ball, 1999). One of the main issues with dynamic techniques, however, is the huge amounts of data that need to be analyzed (Zaidman, 2006).

In recent years, many solutions have been proposed to tackle the scalability issues that are associated with large execution traces. Unfortunately, an effective comparison of such techniques is hampered by three factors. First,

[*] This chapter is based on our publication in the Proceedings of the the 24th International Conference on Software Maintenance (Cornelissen et al., 2008a). It is co-authored by Leon Moonen and Andy Zaidman.

the evaluations of the techniques by their authors mostly concern limited numbers of software engineering contexts. Second, the evaluation criteria being used across these evaluations are typically different. Third, different researchers use their own sets of execution traces to evaluate their techniques on, i.e., no two techniques have been tested on one and the same trace. As a consequence, the evaluation results have limited generalizability, which makes it unclear for an engineer which reduction technique best fits a particular context.

In this chapter, we propose an assessment methodology for trace reduction techniques. The purpose of this methodology is to enable the community to subject such techniques to a systematic evaluation process, in order to provide end-users with sufficient information to choose the most suitable technique in their respective contexts. We illustrate our methodology by applying it to a selection of trace reduction techniques encountered in literature, which we evaluate and compare using context-specific criteria. We argue how such assessments enable the reasoning about the applicability of a reduction technique in certain analysis contexts, which leads to a more effective use of dynamic analysis tools during maintenance tasks.

The remainder of this chapter is organized as follows. Section 4.2 provides an outline of the problem area and the challenges involved therein. Next, in Section 4.3, we elaborate on our assessment methodology. We then illustrate the use of this methodology by discussing four existing reduction techniques in Section 4.4, which we then assess in Section 4.5. Our findings are discussed in Section 4.6, after which we present conclusions and future directions in Section 4.7.

4.2 Background

Our intent to support software engineers in discerning the most effective reduction techniques in specific contexts is motivated by the research community's growing interest in dynamic analysis. These analyses are often characterized by huge amounts of data: Reiss and Renieris, for example, report on an experiment in which one gigabyte of trace data was generated for every two seconds of executed C/C++ code or every ten seconds of Java code (Reiss and Renieris, 2001).

Being able to cope with such amounts of run time data is beneficial to many areas in software engineering. These include such tasks as debugging and performance optimization, and tasks related to software understanding, such as feature analysis, trace understanding, and visualization. Unfortunately, in many such tasks, the analyses have upper bounds on the amount of data that can be handled. In Chapter 3, for example, we reconstructed UML sequence diagrams from event traces (Cornelissen et al., 2007b). Clearly, from a cognitive point of view, such diagrams in themselves do not scale up to thousands of events. In Chapter 5 we will propose novel visualization techniques with

Category	Examples
Summarization	Execution pattern notation (De Pauw et al., 1998) Pattern summarization (Reiss and Renieris, 2001; Systä et al., 2001; Hamou-Lhadj and Lethbridge, 2002; Safyallah and Sartipi, 2006) Object & event clustering (Gargiulo and Mancoridis, 2001; Riva and Rodriguez, 2002) Monotone subsequence summarization (Kuhn and Greevy, 2006)
Metrics-based filtering	Frequency spectrum analysis (Ball, 1999; Zaidman and Demeyer, 2004) Utilityhood measure (Hamou-Lhadj et al., 2005) Webmining (Zaidman et al., 2005) Stack depth limitation (De Pauw et al., 1998; Cornelissen et al., 2007b)
Language-based filtering	Package filtering (Gargiulo and Mancoridis, 2001; Salah and Mancoridis, 2004) Visibility specifiers (Hamou-Lhadj and Lethbridge, 2006) Getters & setters (Hamou-Lhadj and Lethbridge, 2006; Cornelissen et al., 2007b) Constructor hiding (Hamou-Lhadj and Lethbridge, 2006; Cornelissen et al., 2007b)
Ad hoc	Sampling (Chan et al., 2003; Dugerdil, 2007) Fragment selection (Salah and Mancoridis, 2004; Vasconcelos et al., 2005)

Table 4.1 Categories of automatic trace reduction techniques.

a strong focus on scalability; still, a tool's performance generally deteriorates as the amount of data being visualized exceeds certain thresholds.

The huge amounts of data involved in dynamic analysis necessitate the use of *trace reduction techniques* to render the information suitable for analysis. In this chapter we consider (very) large traces, and therefore focus on automatic rather than manual techniques in achieving initial data reductions. Many such techniques have been proposed in literature over the recent years, each targeting different aspects of execution traces. To provide an overview of the approaches in literature, we distinguish four different categories:

(a) **Summarization** techniques attempt to shorten a trace by replacing part of its contents by more concise notations. Typical summarization targets include recurrent patterns.

(b) **Metrics-based filtering** is centered around the use of certain metrics. Examples of such metrics are the stack depth, and degrees of fan-in and fan-out.

(c) **Language-based filtering** techniques are targeted at the omission of such constructs as getters and setters, private methods, and so forth.

(d) **Ad hoc** approaches concern the use of "black-box" techniques that do not consider the trace contents.

Table 4.1 shows the categories, along with various example techniques and pointers to literature.

4.3 Assessment Methodology

The main issue with the reduction techniques being offered is that they are seldomly compared side-by-side by their respective authors. For lack of a common assessment framework, the different techniques are generally not evaluated

- in the same software engineering contexts;
- by the same evaluation criteria; and
- on the same test set (i.e., execution traces).

The absence of a benchmark hinders technical progress in this field (Sim et al., 2003), and engineers faced with large amounts of trace data have the difficult task of selecting the most suitable reduction technique(s) in their specific contexts.

To address this issue, we propose an assessment methodology that is aimed at a thorough evaluation and comparison of trace reduction mechanisms. Such assessments are important because they enable a side-by-side comparison of both existing and future techniques. The key aspect of our methodology is the use of a common context, common evaluation criteria, and common test set.

Given a set of trace reduction techniques that are to be assessed, our methodology distinguishes the following steps:

1. **Context:** the establishment of a context, i.e., a certain task, and the role of reduction techniques therein.
2. **Criteria:** the definition of a set of evaluation criteria that are relevant to the context.
3. **Metrics:** the definition of set of metrics that enables the reasoning about the techniques in terms of the aforementioned criteria.
4. **Test set:** the selection of a series of execution traces on which to evaluate the techniques.
5. **Application:** the application of the techniques on the test set while extracting the previously defined metrics.
6. **Interpretation:** the interpretation and comparison of the measurements, in terms of the evaluation criteria.

Our methodology is applicable in any context that involves the need for trace reductions, and to any of the trace reduction techniques in Table 4.1. Furthermore, the evaluation criteria can be chosen such that the end-user's requirements are met. Note that the first three steps of our methodology correspond, respectively, to the goal, the question, and the metric in the Goal-Question-Metric (GQM) paradigm (Basili et al., 1994).

The methodology can be used in various cases. Examples are the development of new (or more effective use of existing) analysis tools that require the reduction of certain amounts and types of trace data, and the development of new reduction techniques that should be compared to existing solutions with respect to certain criteria. The use of our methodology in these cases ensures that the relevant aspects of reduction techniques can be properly compared, which helps end-users to estimate the applicability of those techniques in specific contexts.

4.4 Four Reduction Techniques

We demonstrate our methodology on a selection of four trace reduction techniques from the literature. Our choice for these particular techniques is motivated by the categorization in Section 4.2, in the sense that we select one technique from each of the four categories. The techniques under study are subsequence summarization, stack depth limitation, a combination of language-based filtering techniques, and sampling.

For lack of available implementations of these techniques, we have created versions of our own. These are based on their descriptions in literature; below we provide relevant implementation details to ensure the reproducibility of our experimental results. Furthermore, for reasons of scalability, in our implementations the traces are processed on the fly rather than read entirely into memory.

4.4.1 Subsequence summarization

The first reduction mechanism that we put to the test is a summarization technique by Kuhn and Greevy (2006). It is based on the grouping of trace events according to some criterion, with each group (or "subsequence") being represented in the output trace by that group's first event. The projected result is a trace that generally contains a significantly smaller number of events. The authors of this technique have named it *monotone subsequence summarization*, and while they use it to represent traces as signals in time, the technique is essentially a trace reduction mechanism.

The grouping criterion used by this technique is based on nesting level differences between trace events: the algorithm assigns consecutive events that have equal or increasing nesting levels to the same group. As soon as a level decrease is encountered and the difference exceeds a certain threshold, called the *gap size*, a new group is initiated. Considering the fact that the nesting level typically fluctuates during the execution of a system, the number of resulting events is smaller than the number of original events, and can be controlled by changing the gap size. Our implementation follows an iterative approach: initially setting the gap size to 0, the algorithm repeatedly increments its value until the projected output size meets the requirements.

Stack depth	0	1	2	3	4	5
Frequency	607	674	890	1,584	2,115	2,142

Table 4.2 Stack depth frequencies during an example scenario.

4.4.2 Stack depth limitation

The second technique is centered around metrics-based filtering and is called *stack depth limitation*. This form of reduction has been used both in static contexts (Rountev and Connell, 2005), and in Chapter 3 of this thesis, in which encouraging results were attained in the removal of implementation details from test case executions. The variant discussed here revolves around the definition of a *maximum* depth: events taking place at depths higher than this threshold are removed from the original trace. The maximum depth depends on the maximum size of the output trace and on the stack depth progression in the original trace, i.e., the program's nesting behavior.

For this technique to obtain the necessary stack depth information, the algorithm first collects the number of events at each depth. Next, given the maximum output size M, the value of the maximum depth d can be automatically determined, by use of which the trace is consequently reduced. For instance, the example stack depth progression in Table 4.2 implies that for M = 4,000, the d must be 3 (as the number of events until this depth is 3,755). The value of M is this technique's only parameter and can be increased if need be.

4.4.3 Language-based filterings

From the third category of reduction mechanisms we consider a combination of *language-based filtering* techniques. Since initial experiments have pointed out that these techniques by themselves are generally not successful in the significant reduction of large traces, we consider *three* consecutive filtering steps:

(i) Removal of getters/setters and their control flow.

(ii) Removal of private and protected method calls.[1]

(iii) Removal of constructors and their control flow.

Depending on the maximum output size, either of these mechanisms can be applied "on demand" in the given order.

4.4.4 Sampling

The fourth category of reduction techniques is represented in our experiment by *sampling*, an ad hoc reduction method that is used, among others, by Chan et al. (2003) in reducing the dynamic information used by their AVID visualizer. The variant that we use in our experiment is simple: given an execution

[1] Note that we preserve the control flows of private and protected methods since these are generally of interest, e.g., private initialization and processing methods within a main method.

trace, we keep every n-th event. We call n the sampling distance, which is automatically determined based on the maximum size of the output trace.

4.5 Experimental Setup

Our demonstration assessment aims at a thorough evaluation of the trace reduction techniques in the previous section. The design of the experiment follows our methodology, with each of the six steps being described in the following sections.

4.5.1 Context

In this experiment, we consider a use case in which an engineer is faced with the task of understanding a system's execution through the *visualization* of its execution traces. We assume his main interest to concern events taking place at high and medium abstraction levels, i.e., low-level details are considered less important. To make this example representative of real life situations, we assume the traces at hand to contain several tens of thousands or even millions of events. Furthermore, the intended visualization offers the opportunity to understand the temporal aspects of the trace, and is interactive in the sense that one can dynamically alter the size of the input data if need be.

4.5.2 Evaluation criteria

The context of our experiment entails a set of requirements that must be sufficiently met by a candidate reduction technique. In particular, we distinguish three evaluation criteria: reduction success rate, performance, and information preservation. These criteria are largely representative of actual use cases in the sense that they are often applicable in practice, particularly the first and third criteria.

Reduction success rate: the degree to which the techniques attain the desired reductions. We say that a reduction fails if the size of a reduced trace does not satisfy some threshold on the output size. The reduction success rate is relevant, as it depends greatly on the trace aspects exploited by a technique, and the degree to which these aspects occur in the trace.

Performance: a measure for the computational effort that is involved in the reduction. This is relevant in our context because the interactive nature of the reference visualization implies that modifications of the trace data should be processed as quickly as possible. For example, if during an interactive session the engineer decides that the trace data should be reduced further, it is not desirable if it takes several minutes for the visualization to refresh its views.

Information preservation: the extent to which information from the original trace is kept after reduction. While the application of a reduction generally implies that certain information is lost, it is important to quantify this loss and to study how it relates to the information needed for the context.

We explore two directions for measuring information preservation. The first route involves a generic approach from information theory that does not use background information regarding the data that is compared; the second route concerns a domain-specific analysis of information preservation that is tailored to the comparison of traces, with respect to the context sketched earlier. In the latter case, we distinguish three *event types* in a trace: (1) high-level events, which intuitively correspond to the control routines in a trace; (2) low-level events, which intuitively correspond to implementation details (e.g., utilities); and (3) medium-level events that comprise the remainder and intuitively concern business logic.

4.5.3 Metrics

In order to reason about the relevant aspects of the reduction techniques in terms of the criteria discussed above, we define a set of metrics. The first two metrics below are directly related to the measurement of reduction success rate and performance, respectively. The last two metrics correspond to the two routes to measuring information preservation.

Actual output size: the actual size of the output dataset after reduction, in calls. This metric allows for a discussion on the reduction success rate in each run. The measurements reflect the degree to which the reduction was successful (if at all), on the basis of which an *average success rate* can be calculated for each technique. For example, if a trace must be reduced to 1,000 events, the success rate is 90% in case of an output of 900 events, and 0% if the reduction fails.

Computation time: the amount of time spent on the reduction, in seconds. This metric allows for a comparison of the techniques in terms of performance. Since the reduction techniques represent different approaches, in each run we measure the total time spent on *all* subtasks. These include such tasks as reading the trace (multiple times if need be), determining the appropriate value for the technique's parameter, and the actual reduction.

Normalized compression distance (NCD): a generic similarity metric (Cilibrasi and Vitányi, 2005) that uses standard compression algorithms to compute a practical approximation of the non-computable but optimal "normalized information distance" (NID) (Li et al., 2004). This metric has its origins in the field of information theory and is based on the notion of Kolmogorov complexity. NCD has been successfully applied in various areas, ranging from text corpora to handwriting recognition, genome sequences, and pieces of music. The NCD can be used to measure information preservation: a reduced trace that is shown to have a high similarity to the original trace implies that little information has been lost.

Preservation of events per type: for each event type, we measure the percentage of events that remains after reduction, relative to the number of events in the original trace for that type. While there are various options for defining such types (e.g., utilityhood (Hamou-Lhadj and Lethbridge, 2006)), we define the high-, medium-, and low-level types without loss of generality as (1) events with no fan-in, (2) events with no fan-out, and (3) remaining events for our demonstration experiment. As events we consider the method signatures, and fan-in/fan-out rates are determined on the basis of the original trace.[2]

4.5.4 Test set

Systems under study

The test set in our example assessment consists of seven different execution traces from six different Java systems. For this test set to be as representative as possible, in our systems selection we have taken into account such characteristics as system size, typical trace size, and multithreading.

JPacman is a small application used for educational purposes at Delft University of Technology. The program is an implementation of the well-known Pacman game in which the player can move around on a graphical map while eating food and evading monsters.

Cromod is a medium-size, multithreaded industrial system that regulates the environmental conditions in greenhouses. Given a set of sensor inputs at the command line, it calculates for a series of discrete points in time the optimal values for such parameters as heating, windows, and shutters. Since these calculations are performed for a great number of points in time, a typical scenario involves massive numbers of events.

Checkstyle[3] is a medium-size source code validation tool. From the command line it takes a set of coding standards to process one or more input files, while systematically looking for violations and reporting these to the user.

JHotDraw[4] is a medium-size tool for graphics editing. It was developed as a showcase for design pattern usage and is acknowledged to be well-designed. It provides a GUI that offers various graphical features such as the insertion of figures and drawings.

Azureus[5] is a large-size, multithreaded peer-to-peer client that implements the BitTorrent protocol. Its GUI can be used to exchange files by use of so-called torrents, which are files containing metadata on the files being exchanged.

Apache Ant[6] is a medium-size, Java-based build tool. It is command line based and owes much of its popularity to its ability to work cross-platform.

[2] Alternatively, one could use the system's *static* call graph, as in (Hamou-Lhadj and Lethbridge, 2006).

[3] Checkstyle 4.3, `http://checkstyle.sourceforge.net/`

[4] JHotDraw 6.0b, `http://www.jhotdraw.org/`

[5] Azureus 2.5.0.0, `http://azureus.sourceforge.net/`

[6] Apache Ant 1.6.1, `http://ant.apache.org/`

Trace	Sys. kLOC	# calls	# threads	Description
checkstyle-short	57	31,237	1	The processing of a small input file that contains 50 lines of commented Java code.
pacman-death	3	139,582	1	The start of a game, several player and monster movements, player death, start of a new game, and quit (Cornelissen et al., 2007a).
jhotdraw-3draw5fig	73	161,087	1	The creation of a new drawing in which five different figures are inserted, after which the drawing is closed. This process is repeated two times (Cornelissen et al., 2008b).
cromod-assignment	51	266,337	11	The execution of a typical assignment that involves the calculation of greenhouse parameters for two days for one greenhouse section (Cornelissen et al., 2008b).
checkstyle-3c	57	1,173,914	1	The processing of three Java source files that are between 500 and 1000 lines in size each.
azureus-newtorrent	436	3,144,785	172	The program's initialization, and invocation of the "new torrent" functionality on a small file before exiting.
ant-selfbuild	99	12,135,031	1	The execution of the program, having specified the non-trivial task of building Apache Ant itself (Zaidman et al., 2005).

Table 4.3 Description of the test set.

The execution trace for this system was obtained through fellow researchers (Zaidman et al., 2005).

Execution scenarios

For each system we define a typical execution scenario. We then instrument the systems, run the scenarios, and register the entries and exits of all constructor and (static) method calls on the class level, and the threads in which these events take place. This results in seven execution traces that size from several tens of thousands of events to several millions of events.[7] The descriptions and sizes of the traces are given in Table 4.3. Figure 4.1 shows the progression of the stack depth in each of the traces, on which the subsequence summarization and stack depth limitation technique are dependent.

4.5.5 Application

Each of the four techniques is applied on all seven traces. The task being performed during each run is the reduction of the input trace while conforming

[7] The traces are available online and may be downloaded from `http://swerl.tudelft.nl/bin/view/Main/TraceRepos`.

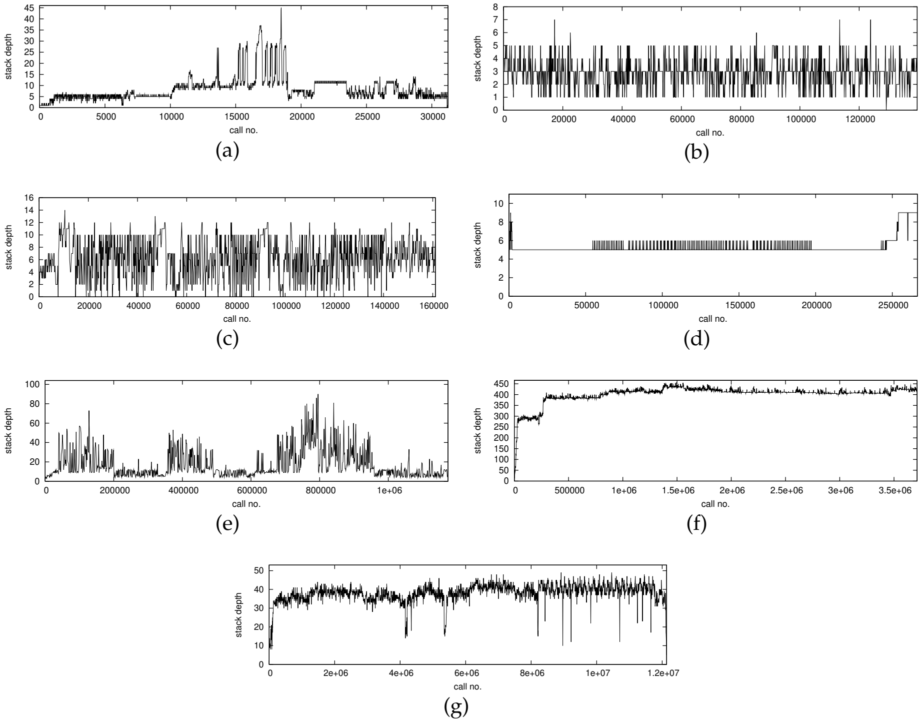


Figure 4.1 Progression of the stack depth in each of the seven traces. (a) checkstyle-short; (b) pacman-death; (c) jhotdraw-3draw5fig; (d) cromod-assignment; (e) checkstyle-3c; (f) azureus-newtorrent; (g) ant-selfbuild.

to a certain *threshold*. The threshold is the maximum output size of the trace, and reflects use cases in which a certain degree of reduction is necessary for a certain task.

As an example of such a use case, consider the trace visualization in Figure 4.2. It shows the similarities within a trace as a matrix: the horizontal and vertical axes symbolize the trace (progressing from left-to-right and top-to-bottom, respectively), and similar events are denoted by colored dots (with each color symbolizing a certain recurring event).[8]. A major issue with this trace comparison technique was that it could handle at most 50,000 events (approx.) in our experimental setup, because it has complexity $O(n^2)$ with respect to the trace size n. Therefore, the trace must be reduced prior to processing and visualization.

We employ seven different output thresholds with values between 1,000 and 1,000,000 calls. This yields a total of 196 runs, which we perform on

[8] A detailed description of this visualization technique is provided in (Cornelissen and Moonen, 2007).

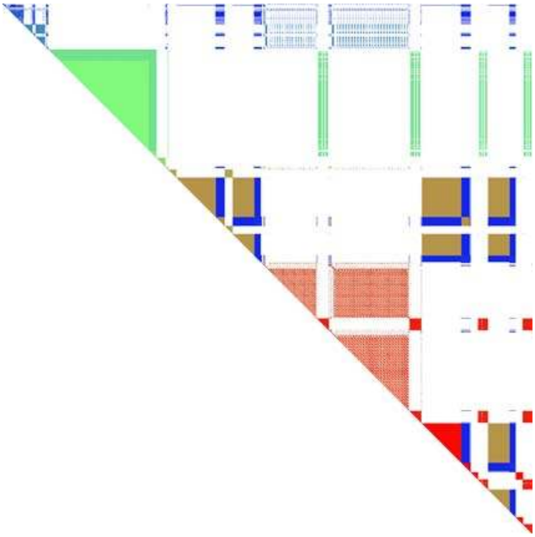


Figure 4.2 Visualization of an execution trace as a similarity matrix.

a Linux system with an Intel Pentium M 1.6 GHz processor and 2 GB of memory.

4.5.6 Interpretation

The final stage of the assessment concerns the interpretation of the results. By focusing on the measurements in each of the 196 runs, we discuss the results of the techniques in terms of our evaluation criteria. Finally, based on our observations, we conclude with a comparison of the techniques.

4.6 Results & Discussion

The results of the experiment are shown in Table 4.4, which shows the measurements for each of the four techniques across the relevant runs.[9] Reductions that were unsuccessful are denoted by dashes; furthermore, the percentages of preserved events have been rounded upwards so as to distinguish very small fractions from zeroes. Finally, the NCD values for information preservation were omitted in this table since they proved unreliable for the trace sizes used in the experiment; this is discussed in Section 4.6.3.

Figure 4.3 shows the average reduction success rate of each technique across the entire test set. The percentages are based on the measurements of all relevant runs.

Figures 4.4 through 4.6 demonstrate the performance of each technique in terms of computation time. We have selected the cases that exhibit the clearest

[9] Runs with thresholds higher than the input trace sizes were omitted.

		Subsequence summarization					Stack depth limitation					Language-based filterings					Sampling				
	max.output size	output size	comp.time	high %	medium %	low %	output size	comp.time	high %	medium %	low %	output size	comp.time	high %	medium %	low %	output size	comp.time	high %	medium %	low %
checkstyle-short	10,000	6,417	2	59	29	12	4,251	0	100	12	16	-	-	-	-	-	7,809	0	12	25	26
31,237 calls	5,000	781	4	36	4	2	4,251	0	100	12	16	-	-	-	-	-	4,462	0	0	16	13
	1,000	781	4	36	4	2	730	0	100	1	5	-	-	-	-	-	976	0	6	4	4
pacman-death	100,000	73,402	4	100	52	54	44,743	1	100	67	1	37,343	3	87	54	3	69,791	0	47	50	51
139,582 calls	50,000	26,374	7	86	34	6	44,743	1	100	67	1	37,343	3	87	54	3	46,527	0	22	33	34
	10,000	7,080	18	34	11	1	172	1	100	1	1	-	-	-	-	-	9,970	0	6	8	8
	5,000	37	22	12	1	1	172	1	100	1	1	-	-	-	-	-	4,985	0	4	4	4
	1,000	37	21	12	1	1	172	1	100	1	1	-	-	-	-	-	997	0	3	1	1
jhotdraw-3draw5fig	100,000	25,965	12	45	26	10	83,584	1	100	58	48	61,906	5	80	59	27	80,543	0	51	51	50
161,087 calls	50,000	25,965	12	45	26	10	42,748	1	100	26	25	44,383	17	42	47	17	40,271	0	26	26	25
	10,000	7,559	23	27	5	4	5,491	1	100	2	2	-	-	-	-	-	9,475	0	6	6	6
	5,000	2,659	29	18	4	1	-	-	-	-	-	-	-	-	-	-	4,881	0	4	4	4
	1,000	773	40	17	1	1	-	-	-	-	-	-	-	-	-	-	1,000	0	1	1	1
cromod-assignment	100,000	10,983	22	83	49	3	363	2	99	2	1	443	35	22	1	1	88,779	1	28	34	34
266,337 calls	50,000	10,983	22	83	49	3	363	2	99	2	1	443	36	22	1	1	44,389	1	12	12	17
	10,000	207	33	74	1	1	363	2	99	2	1	443	35	22	1	1	9,864	1	4	4	4
	5,000	207	33	74	1	1	363	2	99	2	1	443	35	22	1	1	4,932	1	2	2	2
	1,000	207	33	74	1	1	363	2	99	2	1	443	35	22	1	1	997	1	1	1	1
checkstyle-3c	1,000,000	922,603	49	100	89	66	990,514	9	100	79	92	769,277	40	99	88	38	586,957	3	52	50	51
1,173,914 calls	500,000	209,263	92	12	19	17	454,657	9	100	37	42	-	-	-	-	-	391,304	3	34	34	34
	100,000	31,233	135	7	3	3	49,834	9	100	4	5	-	-	-	-	-	97,826	2	8	9	9
	50,000	31,233	135	7	3	3	49,834	9	100	4	5	-	-	-	-	-	48,913	2	4	5	5
	10,000	8,032	177	2	1	1	2,061	9	100	1	1	-	-	-	-	-	9,948	2	2	1	1
	5,000	1,650	220	2	1	1	2,061	9	100	1	1	-	-	-	-	-	4,995	2	0	1	1
	1,000	1	304	2	0	0	743	9	100	1	1	-	-	-	-	-	1,000	2	0	1	1
azureus-newtorrent	1,000,000	-	-	-	-	-	657,657	25	100	20	54	217,671	185	69	6	17	786,196	7	26	25	25
3,144,785 calls	500,000	-	-	-	-	-	451,133	25	100	13	33	217,671	183	69	6	17	449,255	7	15	15	15
	100,000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	98,274	7	4	4	4
	50,000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	49,917	7	2	2	2
	10,000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	9,983	7	1	1	1
	5,000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	4,999	7	1	1	1
	1,000	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1,000	7	1	1	1
ant-selfbuild	1,000,000	921,549	1,589	100	14	4	768,246	86	100	8	6	111,650	193	100	1	2	933,463	24	0	8	8
12,135,031 calls	500,000	34,432	2,338	100	1	1	434,088	86	100	4	4	111,650	191	100	1	2	485,401	24	0	4	5
	100,000	34,432	2,326	100	1	1	94,396	86	100	1	1	84,588	579	100	1	1	99,467	22	0	1	1
	50,000	34,432	2,326	100	1	1	47,793	87	100	1	1	-	-	-	-	-	49,938	22	0	1	1
	10,000	4,830	3,085	100	1	1	4,742	87	100	1	1	-	-	-	-	-	9,995	22	0	1	1
	5,000	4,830	3,073	100	1	1	4,742	87	100	1	1	-	-	-	-	-	5,000	22	0	1	1
	1,000	534	3,867	100	1	1	343	87	100	1	1	-	-	-	-	-	1,000	22	0	1	1

Table 4.4 Measurement results for all four techniques.

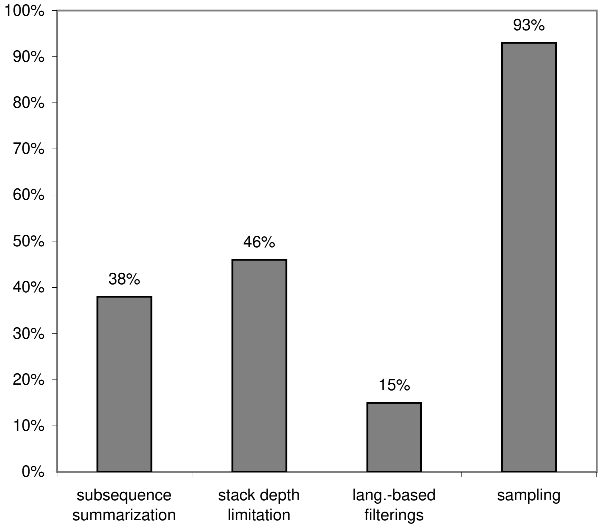


Figure 4.3 Average reduction success rates.

differences (i.e., the largest traces) and that have high numbers of successful reductions, being `cromod-assignment`, `checkstyle-3c`, and `ant-selfbuild`. Note that the latter two diagrams employ logarithmic scales for the computation time.

Finally, Table 4.5 summarizes each technique's achievements relative to those of the other techniques.

In the following sections we discuss our findings, which are structured according to the three criteria.

4.6.1 Reduction success rate

In terms of the first evaluation criterion, we observe that the sampling technique achieved the best results: it is the only method that yielded successful reductions under all circumstances, with the output sizes mostly being close to the thresholds. This is presumably due to its ad hoc nature, as execution traces can always be sampled such that the maximum output size is satisfied, regardless of the size and composition of the trace.

The summarization and stack depth limitation techniques are both dependent on stack depth progression (Figure 4.1), and show results that are similar to one another, with both methods mostly having difficulties with the `azureus-newtorrent` trace. The cause is most likely found in the abundance of active threads during this program's execution, in which (1) there occur many thread interactions, which hinders the grouping algorithm used during summarization; and (2) many threads exhibit low nesting levels, which renders depth limitations less effective. Furthermore, when faced with strict trace size

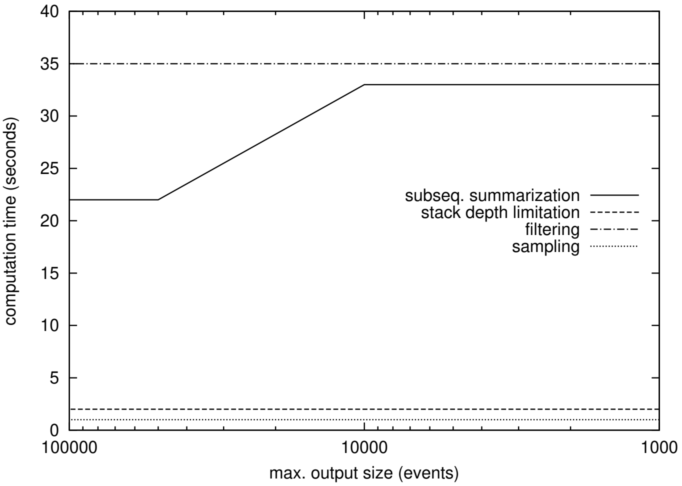


Figure 4.4 Performance for the Cromod trace.

limits, the summarization technique occasionally produces very small traces because in such cases the gap size is large out of necessity.

Finally, the combination of language-based filtering techniques proves disappointing with nearly half of the reductions having been unsuccessful. A noteworthy exception is the `cromod-assignment` trace, in which 98 out of every 100 events concern constructors, which are all filtered given any of the thresholds in Table 4.4.

Note that alternative definitions may be considered: e.g., a reduction may not necessarily have "failed" in case the result contains only several events too many.

4.6.2 Performance

With regard to performance, our measurements show that all four techniques were capable of reducing traces smaller than one million events within one minute (Figure 4.4). When looking at larger traces, however, there exist clear differences: here we observe that sampling easily outperforms any of the other techniques (Figures 4.5 and 4.6). We assume the principal cause to be that the sampling distance can be determined a priori, after which the trace needs to be processed only once. For the same reason, the computational effort involved in this approach is independent of the thresholds.

The same holds for the stack depth limitation technique, but here a trace must be processed twice because the stack depth frequencies must first be collected. Moreover, the interpretation of the stack depth at each event requires additional parsing effort in comparison to the black-box approach used by the sampling technique.

Concerning the language-based filtering techniques, there is little timing data available due to the many failed reductions. The data that is available,

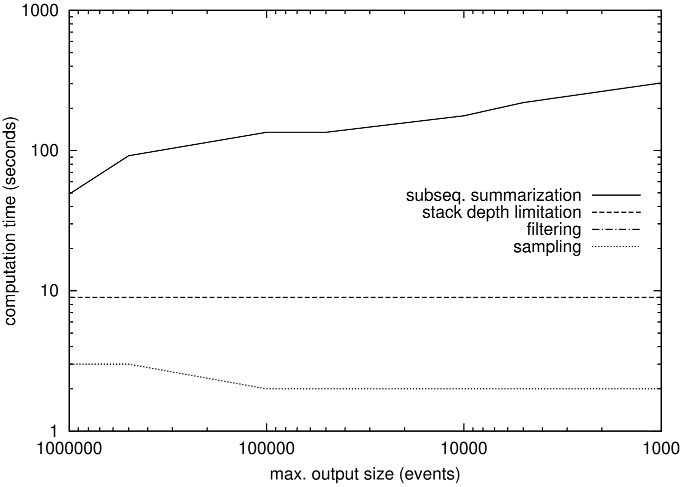


Figure 4.5 Performance for the checkstyle-3c trace.

however, suggests that this approach is significantly slower than the aforementioned techniques. One can think of several reasons for this slowdown. Since the filterings are applied on demand and one by one, low thresholds require that the trace at hand is processed up to four times, i.e., once for each filter type, and once more to read and write the traces. Moreover, the stack depths and signatures of each event must be parsed in order to acquire the information that is targeted by the filters.

The subsequence summarization technique typically requires a trace to be processed multiple times, as the gap size must be repeatedly incremented (starting at 0) until a suitable projected output size has been found. This iterative process yields significant overheads if the threshold is much smaller than the size of the trace, with the effort involved in each iteration being proportional to the trace size. Moreover, the number of necessary iterations also depends on the stack depth progression in the trace. The overall result is that the summarization approach is clearly the slowest technique in our experiment, particularly for large traces.

4.6.3 Information preservation

The assessment of our final criterion yields mixed results. Unfortunately, the values computed by the NCD metric proved unreliable in practice due to the trace sizes that were used in our experiment. To explain the issue, we need to provide some background on this metric. NCD is based on the notion that two objects are close to each other if we can significantly compress one object given the information in the other (Cilibrasi and Vitányi, 2005). In practice, this translates to compressing the concatenation of the original and reduced traces and comparing its size to that of the compressed original trace. However, it turns out that standard compression tools split their input in "com-

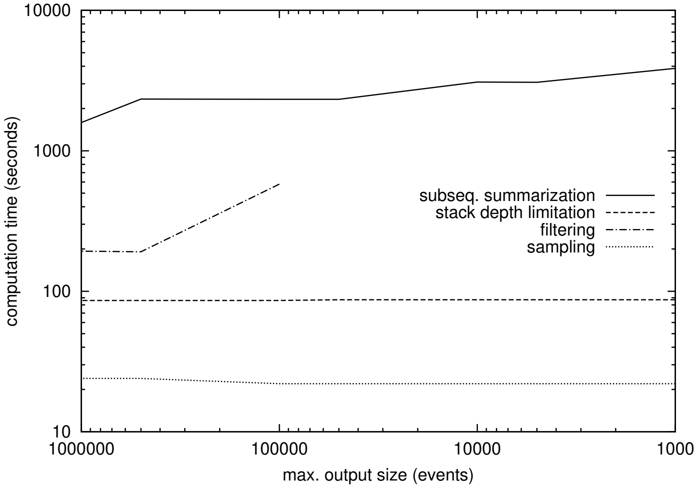


Figure 4.6 Performance for the ant-selfbuild trace.

pression windows" within which the compression information is shared. As the size of the concatenation of the original and reduced trace exceeds the size of the compression window, that particular compressor can no longer be used to determine the NCD between those traces (since we are no longer compressing one object given the information in the other). Personal communication with R. Cilibrasi, the metric's first author (Cilibrasi and Vitányi, 2005), confirmed these issues and suggested their circumvention by writing a dedicated compressor. Since the metric only serves as an example in our experiment, this is left as a direction for future work.

For the domain-specific assessment of information preservation, we focus on high-level *and* medium-level events since those are required by our context (Section 4.5.1).

Subsequence summarization typically attains the best results: the percentages of preserved high-level events are significantly higher than those of the medium-level events which, in turn, are often higher than those of the low-level events. This is because each group is represented by its first event, and using our depth-based grouping criterion this event is likely to reside at relatively high levels.

The stack depth limitation and language-based filtering techniques show comparable results: the percentages of preserved high-level events are generally higher than those of other event types, with the depth limitation technique attaining the highest percentages in this respect. In several reductions, however, the fractions of preserved medium-level events are not always higher than those of low-level events. Examples are `checkstyle-3c` for depth limitation, and `azureus-newtorrent` for filtering. This implies that the use of these two techniques sometimes causes the preservation of low-level events at the cost of those at the medium-level, which is undesirable in the given context.

	Subseq. summariz.	Stack depth lim.	Lang.-based filt.	Sampling
reduction success rate	o	o	−	+
performance	−	o	o	+
information preservation	+	o	o	−

Table 4.5 Assessment summary with respect to the example context.

The sampling technique mostly exhibits similar fractions of preserved events across all three event types, particularly in large traces. This means that all event types are equally represented in the reduced traces. We attribute this to the technique's ad hoc nature, which implies that low-level events are neither identified, nor removed. This makes sampling the least useful technique in preserving high-level and medium-level events in our context.

On an interesting note, the measurements for the `ant-selfbuild` trace suggest that all of its high-level events are often preserved. However, it turns out that our definition of high-level events implies that this trace has only *one* high-level event.

4.6.4 Threats to validity

A potential threat to the internal validity concerns the test set in our experiment. As with most evaluations in literature, certain implications were based on the properties of our test set, e.g., systems with multiple threads running the risk that stack depth-based reductions may have limited applicability (Section 4.6.1). Such observations do not necessarily hold true for any program or trace, as threading and nesting behavior can vary from system to system. We have addressed this issue by using a test set that is above average in terms of size and composition, and that contains systems and traces with different sizes and characteristics.

An additional threat to the internal validity concerns the fact that reduction techniques in literature could be subject to different interpretations. To address this threat, we have described our implementation choices to allow validation by others and to ensure the reproducibility of our results.

Concerning the external validity, we note that the reduction techniques considered in this chapter are automatic in nature. The assessment of reduction methods is more difficult if other factors come into play; e.g., when a technique relies heavily on additional information, such as domain knowledge. Furthermore, most reduction techniques can be implemented in different manners: for instance, in terms of performance, the summarization algorithm used in our experiment could benefit from a higher initial gap size in case of large traces or low thresholds.

Finally, alternative contexts may require other evaluation criteria in addition to those used in our example assessment. For example, the evaluation of a memory-intensive technique warrants a discussion on spatial complexity. However, we argue that our example criteria are generic to a great extent: in particular, the notions of reduction success rate and information preservation are applicable in many alternative assessment contexts, which renders our experimental results useful in those cases.

4.7 Concluding Remarks

Program comprehension is an important aspect of the software development process. While the use of dynamic analysis in this process has become increasingly popular, such analyses are often associated with large amounts of trace data, which has led to the development of numerous trace reduction techniques in recent years. Unfortunately, the different techniques being offered are generally not evaluated (1) in the same software engineering contexts, (2) by the same evaluation criteria, and (3) on the same test sets. As a result, it is often unclear to which extent a certain technique is applicable in a particular context, if at all.

We addressed this challenge by proposing an assessment methodology that uses a common context, common evaluation criteria, and a common test set to ensure that the reduction techniques under study can be properly compared. To illustrate its use in practice, we applied the methodology on a selection of four types of reduction techniques, being subsequence summarization, stack depth limitation, language-based filtering, and sampling. Using a test set of seven large execution traces (made available online), we evaluated and compared these approaches in terms of context-specific criteria, leading to an overview (Table 4.5) that is valuable for software maintainers in similar contexts.

In summary, the work described in this chapter makes the following contributions:

- An assessment methodology for the evaluation and comparison of trace reduction techniques.

- The demonstration of this methodology through the implementation, evaluation, and comparison of four types of trace reduction techniques used in literature.

Future work

As a direction for future work one could consider the application of our methodology to additional traces and reduction techniques. In particular, the extent could be determined to which the assessment results can be generalized, i.e., whether the achievements of a technique are representative for other techniques in the same category (Table 4.1). This includes the use of larger test sets and the consideration of alternative contexts, which could involve different evaluation criteria (e.g., with more emphasis on *qualitative* aspects).

Another direction for future work concerns adapting the compressor that is used to compute the NCD metric, such that it no longer suffers from the "compression window" limitations that were discussed in Section 4.6.3. This enables its applicability to realistically-sized traces, and renders it an interesting alternative for measuring information preservation.

Finally, an interesting future direction could be to investigate whether certain trace characteristics (similar to those in (Hamou-Lhadj and Lethbridge, 2005)) can help in predicting the effectiveness of certain reduction techniques. The Cromod trace in our experiment is a good example, as its many constructors were the key to the success of the constructor filtering technique in that case.

Advanced Trace Visualization with Extravis[*]

Chapter 5

An alternative to traditional visualization techniques is the use of more advanced visualizations. Such techniques are a potentially powerful means to tackle the scalability issues associated with program comprehension and dynamic analysis, particularly when designed with these challenges in mind. In this chapter, we propose two such visualizations: the massive sequence view and the circular bundle view, both reflecting a strong emphasis on scalability. They have been implemented in a tool called EXTRAVIS*. By means of distinct usage scenarios that were conducted on three different software systems, we show how our approach is applicable in three typical program comprehension tasks: trace exploration, feature location, and top-down analysis with domain knowledge.*

5.1 Introduction

Software engineering is a multidisciplinary activity that has many facets to it. In particular, in the context of software maintenance, one of the most daunting tasks is to *understand* the software system at hand. During this task, the software engineer attempts to build a mental map that relates the system's functionality and concepts to its source code (Renieris and Reiss, 1999; LaToza et al., 2006).

Understanding a system's behavior implies studying existing code, documentation, and other design artifacts in order to gain a level of understanding that is sufficient for a given maintenance task. This *program comprehension* process is known to be very time-consuming, and Basili reports that 50 to 60% of the software engineering effort is spent on understanding the software system at hand (Basili, 1997). Thus, considerable gains in overall efficiency can be obtained if tools are available that facilitate this comprehension process. The greatest challenge for such tools is to create an accurate image of the entities and relations in a system that play a role in a particular task.

Dynamic analysis

Dynamic analysis, or the analysis of data gathered from a running program, has the potential to provide an accurate picture of a software system: e.g., in the context of object-oriented systems it can reveal object identities and

[*] This chapter is based on our publication in the Journal of Systems & Software in December 2008 (Cornelissen et al., 2008b). It is co-authored by Andy Zaidman, Danny Holten, Leon Moonen, Arie van Deursen, and Jarke J. van Wijk. Danny Holten was responsible for the implementation of the tool as well as the design of the visualization techniques involved.

occurrences of late binding. Furthermore, through the careful selection of an execution scenario, a goal-driven program comprehension strategy can be followed (Zaidman et al., 2005). The data is obtained through the instrumentation and execution of a system, which (in the case of a post mortem analysis) results in one or more *execution traces* that are to be analyzed.

Challenges & goal

The main issue in the context of dynamic analysis approaches for program comprehension is the enormous amount of data that is collected at runtime, since it gives rise to scalability issues (Zaidman, 2006). Particularly in the case of a sizable program, the main challenge for any dynamic analysis based technique is to convey both the program's large structure *and* its many interrelationships to the user, such that the available screen real estate is used efficiently. This is not a trivial task, and straightforward visualizations typically do not suffice because they often require two-dimensional scrolling, thus hindering the comprehension process (Yang et al., 1997). An example of such visualizations are UML sequence diagrams which we reconstructed in Chapter 3.

The goal of this chapter is the development of new techniques that allow the visualization of dynamically gathered data from a software system in a condensed way, while still being highly scalable and interactive.

Visualization approach

We attempt to achieve our goal by presenting two synergistic views of a software system. The first view is the *circular bundle view* that projects the system's structural elements on the circumference of a circle and visualizes the call relationships in between. The second view, the *massive sequence view*, provides an interactive, high-level overview of the traced events. These techniques are implemented in our tool Extravis (EXecution TRAce VISualizer).

To characterize our approach, we use the framework introduced by Maletic et al. (2002):

1. *Task: Why is the visualization needed?* The amount of trace data that often results from dynamic analysis, calls for a visualization that represents an execution trace in a concise and interactive manner. More specifically, we describe how our approach is useful for three representative program understanding scenarios:

    - trace exploration;
    - feature location; and
    - top-down program comprehension with domain knowledge.

2. *Audience: Who will use the visualization?* The target audience consists of software developers and re-engineers who are faced with understanding (part of) an unknown software system.

3. *Target: What low level aspects are visualized?* Our main aim is to represent information pertaining to call relationships, and the chronological order in which these interactions occur. This information is augmented with static data to establish the system's structural decomposition.

4. *Representation: What form of representation best conveys the target information to the user?* We strive for our visualization to be both intuitive and scalable. To optimize the use of screen real estate, we represent a system's structure in a circular view. Moreover, our massive sequence view presents an interactive overview.

5. *Medium: Where is the visualization rendered?* The visualization is built up from two synchronized views that are rendered on a single computer screen.

Our approach enables software engineers to quickly gain an understanding of unfamiliar software systems, thus enabling related tasks such as software maintenance to be performed more efficiently. We illustrate this through the application of our tool implementation in the context of trace exploration, feature location, and top-down analysis. These studies involve two open source programs and an industrial system.

Contributions

This chapter makes the following contributions.

- A novel approach to execution trace visualization that is based on two linked views: (1) the circular bundle view that displays the structural elements and bundles their call relationships, and (2) the massive sequence view that provides an interactive overview.

- The application of our tool implementation on three distinct software systems in three different program comprehension contexts.

Structure of this chapter

In Section 5.2 we discuss the existing visualization techniques that our approach relies on, while Section 5.3 provides a detailed description of our own approach. Section 5.4 introduces our experimental setup and case studies, while Sections 5.5 through 5.7 deal with the three case studies that we have performed. Section 5.8 discusses benefits and drawbacks of our approach and Section 5.9 outlines related work. Section 5.10 summarizes our work and provides pointers for future work.

5.2 Existing work

The two synergistic views that we propose in our approach are based on a number of existing information visualization techniques. In this section we briefly introduce these existing techniques and point out how our approach differs from their original application.

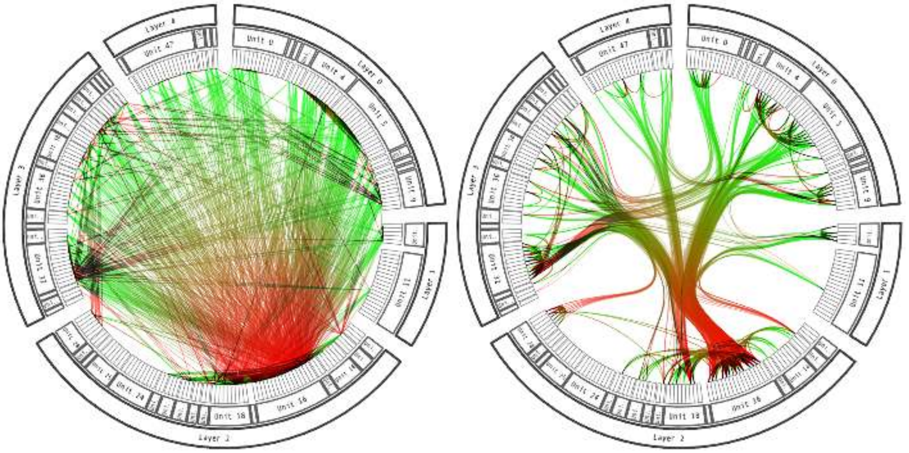


Figure 5.1 Call relations within a program shown using linear edges (left) and using hierarchical edge bundles (right).

5.2.1 Circle approach

In order to address the issue of visualizing the large number of structural entities that constitute a software system, we propose to employ a circle approach in which all structural elements are projected on the circumference of the circle.

Hierarchical edge bundles

To depict a system's structural information we use the circle approach from Holten (2006), who proposed to project all of a system's structural entities on the outline of a circle and to draw their relations in the middle. The entities are presented in a nested fashion on the circle in order to convey their hierarchical properties, e.g., package structures or architectural layers.

Within the circle the relationships between the structural elements are drawn. These relations are depicted by bundled splines (Figure 5.1): the visual bundling of relations helps to reduce visual clutter, and also shows the implicit relationships that exist between parent elements resulting from explicit calls between their respective children. These *hierarchical edge bundles* were used by Holten (2006) to depict static dependencies; we enrich this visualization so that it can show dynamic information.

5.2.2 Message sequence charts

The technique that we propose to use for the interactive visualization of time-ordered events builds upon the notion of message sequence charts. Message sequence charts are commonly used to visualize a series of chronologically ordered interactions between the entities of a system (Briand et al., 2006). Their main advantage is readability: the fact that the events are ordered from top to bottom makes the diagrams intuitive for humans. Scalability, however, is an important limitation for this technique: the charts rapidly become hard to navigate when dealing with too much information, which makes them less suitable for large amounts of entities and interactions.

Information murals

To tackle the scalability issue in visualizing large amounts of dynamic information, Jerding and Stasko (1998) proposed the “information mural”. This technique creates a two-dimensional miniature version of the information space and is appropriately scaled to fit on one screen. In this chapter, we use a similar technique to visualize large-scale message sequence charts: we show a system’s *entire* (nested) structure on the horizontal axis, while plotting the interactions as rectangles along the vertical axis. The rectangles are appropriately colored to indicate the directions of the calls. The purpose of the resulting view is to provide a navigable overview of an execution trace.

5.2.3 Visualization criteria

When constructing new techniques to visualize large amounts of data in a comprehensible way, we need criteria that capture “comprehensibility”. Taken from the realm of visual programming languages, we discuss two properties that express these criteria and represent a set of important requirements (Yang et al., 1997).

Accessibility of related information

It is essential that related information is viewed in close proximity. When considering two objects in a visualization that are not close to each other, there is the psychological claim that these objects are not closely related, or are not dedicated to solving the same problem. Translated to the field of software visualization, we observe two dimensions that pertain to this criterion:

1. Structural entities that are related according to some criterion should be visualized in close proximity to each other. Example criteria include parent-child and composition relationships.

2. Structural entities that participate in the execution in a particular time interval should be visualized in close proximity to each other.

Use of screen real estate

The term “screen real estate” refers to the size of a physical display screen and connotes the fact that screen space is a valuable resource. During information comprehension tasks, it is of great importance to make optimal use of the available screen real estate in order to prevent excessive scrolling and to reduce the effort from the user’s part.

Keeping these criteria in mind, any *trace visualization* technique faces a twofold challenge: it must depict (1) a potentially large number of structural entities, all the while keeping related entities relatively close together, and (2) massive amounts of runtime information without confusing the viewer, e.g., preferably without the need for scrolling.

5.3 Our Approach

The techniques that we described in the previous section have been implemented in a prototype tool called Extravis. Given an execution trace (or part thereof), Extravis presents two linked views:

- The *circular bundle view* shows the system's structural decomposition (using a parent-child relationship) and the nature of its interactions during (part of) an execution trace.
- The *massive sequence view* provides a concise and navigable overview of the consecutive calls between the system's entities in a chronological order.

The tool's user interface is depicted in Figure 5.2, which illustrates the actual views in the context of a large execution trace[1]. Both views offer multiple interaction methods and detailed textual information on demand, and a synchronized mode of operation ensures that changes in the one view are propagated to the other. In this section, we discuss the metamodel used by Extravis and describe the two views in more detail.

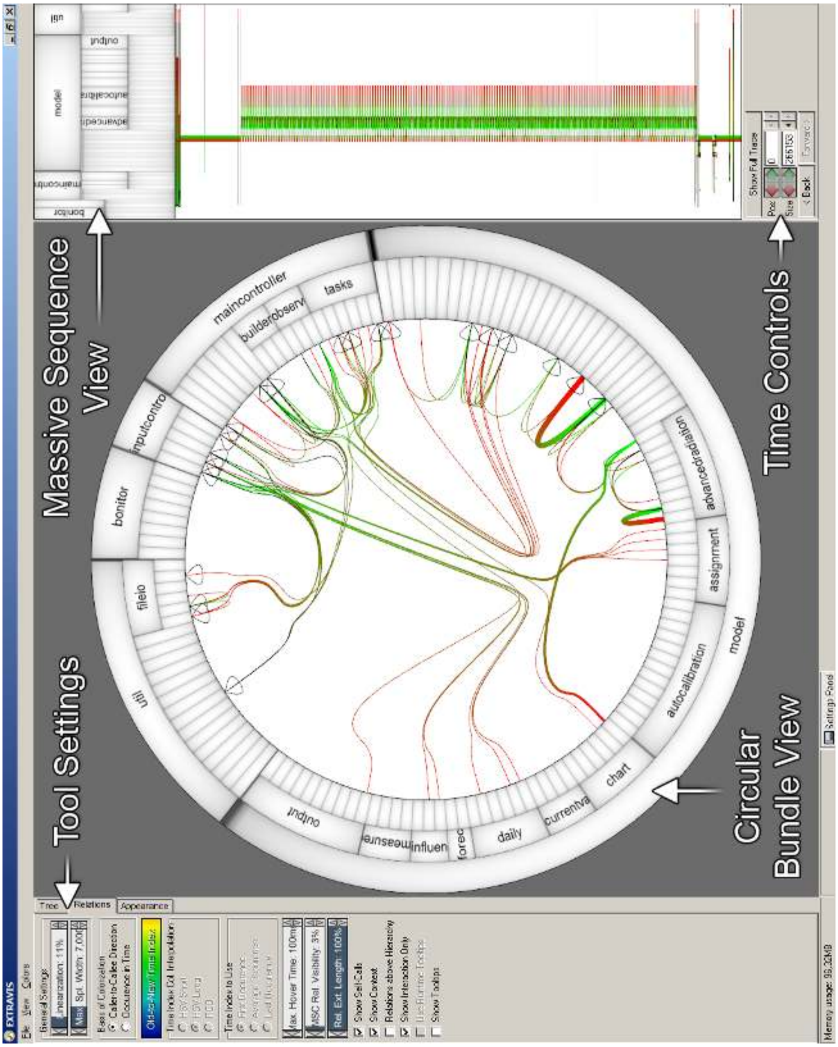
5.3.1 Metamodel

Extravis is based on a metamodel that describes the structural decomposition of the system (a parent-child relationship or *contains*-hierarchy) and a timestamped call relation. Optionally, additional relations can be supplied which contain more detailed information.

Structural information
To visualize the structure of a program, the tool requires a type of relation that defines the system's structural decomposition. In this context, one could think of package or directory structures, or architectural layers. The remainder of this chapter assumes a parent-child relationship.

Basic call relations
The second mandatory part of the metamodel is a series of *call* relations which are extracted from an execution trace. The input thus contains information on the caller and callee's classes, the method signatures, and the chronological order of the calls (by means of an increment). Additionally, to link with the

[1] The figures in this chapter are also available in hi-res at `http://swerl.tudelft.nl/extravis/`.



General Settings
Linearization: 11%
Max. Spl. Width: 7,000

Basis of Coloration
● Caller-to-Callee Direction
○ Occurence in Time

Old-to-New Time Index

Time Index Col. Interpolation
○ HSV Short
● HSV Long
○ RGB

Time Index to Use
● First Occurence
○ Average Occurence
○ Last Occurence

Max. Hover Time: 100ms
MSC Rel. Visibility: 3%
Rel. Ext. Length: 100%

☑ Show Self-Calls
☑ Show Context
☐ Relations above Hierarchy
☑ Show Interaction Only
☐ Use Runtime Tooltips
☐ Show Tooltips

Figure 5.2 Extravis' interface. A full view of an entire Cromod trace is shown.

source code, the method signatures contain pointers to the source files (if available) and include the relevant line numbers.

Detailed call relations

In case the execution trace is rich in the sense that detailed call information is available, the metamodel also allows the specification of such data as object identifiers, runtime parameters, and actual return values.

5.3.2 Circular bundle view

The first of the two views, the circular bundle view, offers a detailed visualization of the system's structural entities and their interrelationships. At the basis of this view lie the techniques that were proposed in Section 5.2.1:

- The projection of the software system's structural elements on the circumference of a circle, including their hierarchical structuring.
- The visual bundling of the relationships between these elements in the circle.

Furthermore, we have made several enhancements to further facilitate the comprehension process.

First, the high-level structural entities can be *collapsed* to enable focusing on specific parts of the system. As is illustrated in Figure 5.4, collapsing an element hides all of its child elements and "lifts" the relations pertaining to these child elements to the parent element, thus providing a straightforward abstraction mechanism. The (un-)collapsing process is fully animated for the user to maintain a coherent view of the system, i.e., to facilitate the cognitive linking of the "pre" and "post" view.

Secondly, the circular bundle view provides a snapshot in time that corresponds to the part of the execution trace that is currently being viewed. As such, the hierarchical edge bundles visualize the interactions occurring during a certain time interval. Edges are drawn between the elements that communicate with each other, and the *thickness* of an edge indicates the number of calls between two elements, thus providing a measure for their degree of coupling. Furthermore, textual information related to the underlying source code is provided by means of call highlighting (i.e., by hovering over an edge) and by providing direct links to the relevant source parts.

Finally, with respect to the coloring, the user can choose from either the *directional* or the *temporal* mode. In the former case, a color gradient along the edge indicates its direction. The latter mode colors the edges such that the calls are ordered from least recent (light) to most recent (dark).

5.3.3 Massive sequence view

To support users in navigating through traces and identifying parts of interest, Extravis offers the massive sequence view. Being a derivative of the information mural, it provides an overview of (part of) an execution trace in which the directions of the relations (caller-to-callee) are color coded using a gradient (green-to-red; see Figure 5.2). Additionally, the massive sequence view allows to *zoom in* on parts of the execution trace by the selection of a fragment that needs closer inspection.

Importance-Based Anti-Aliasing

When visualizing information murals and large-scale message sequence charts in particular, it becomes apparent that the number of available pixel lines on a

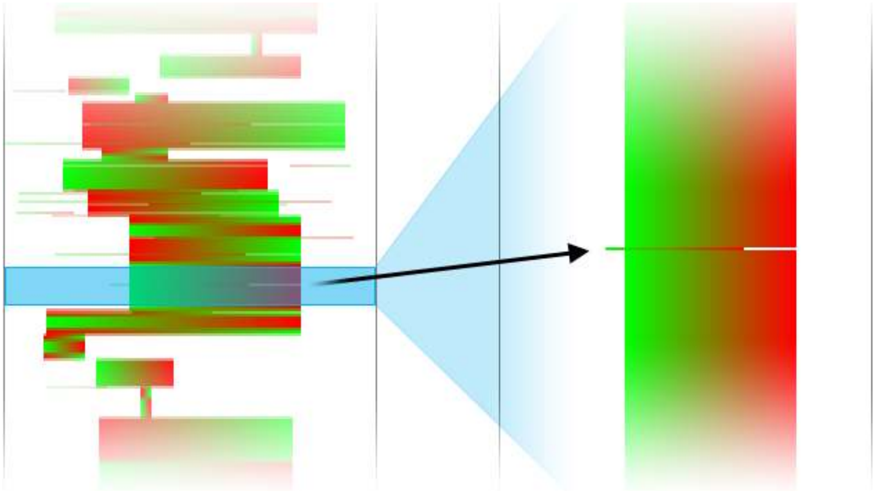


Figure 5.3 Using Importance-Based Anti-Aliasing (IBAA) to visualize outlier calls.

normal display is not sufficient. Rather than visualizing every event, one typically resorts to such measures as abstraction and linear anti-aliasing (Jerding et al., 1997). While being useful in maintaining the big picture, certain potentially useful *outlier calls* may not "survive" these measures.

To keep the loss of such calls to a minimum, we propose a new anti-aliasing technique that uses an importance-based blending mode to calculate a pixel line's average color. Calls are weighted depending on the frequency with which they appear within a certain time frame, and calls with small frequencies are emphasized. This technique is called *Importance-Based Anti-Aliasing* (IBAA) and ensures that outlier calls remain visible in the proximity of thousands of calls, thus ensuring that potentially important calls are not missed by the user. Figure 5.3 illustrates the technique; a detailed description is provided by Holten et al. (2007).

5.3.4 View interaction

An important strength of our approach is the *synergy* between the two views. The views are linked in the sense that user interactions in the one view are visible in the other. This ensures that the user maintains a coherent view of the system during all view interactions.

An example is the collapsing process that was described earlier. Collapsing a structural entity in the circular view hides its interrelationships and aggregates its relations with other entities; this results in an abstraction that is propagated to the massive sequence view, in which the structural hierarchy and the series of calls are modified accordingly. Additionally, the user may zoom in on part of the massive sequence view, thus reducing the time frame under consideration in both views.

Another example that illustrates the usefulness of the linked views concerns highlighting. In the circular view, the viewer can select a structural element, upon which the massive sequence view shows the interactions involving this element by graying out the other calls. Selecting two elements, on the other hand, highlights their mutual interactions. Hovering over a call in either of the two views shows its occurrence(s) in the other view, and spawns a tooltip that describes its nature (e.g., the method signature and call site information).

5.3.5 Implementation

This section focuses on the technologies that were used to build Extravis.

The front-end of Extravis is written in Delphi and makes heavy use of OpenGL. For extracting a system's class decomposition from its directory structure, we make use of a simple Perl script. As for the dynamic part, we trace a system's execution by monitoring for (static) method and constructor invocations, and registering the objects that are involved. We achieve this by extending the Sdr framework from Chapter 3, which incorporates a tracer that employs AspectJ[2] for the instrumentation. During an execution, this tracer registers unique objects, method and constructor names, information on the call sites (i.e., source filenames and line numbers), runtime parameters, and actual return values, while a custom-built event listener converts the events to our tool's input format.

As it stands, Extravis does not support multithreaded systems. While our tracer is capable of registering thread information, our visualization technique is not currently attuned to multiple threads.

Extravis is available for download and requires a modern Windows PC to run[3]. A recent graphics card is highly recommended in order to fully benefit from the importance-based anti-aliasing.

5.4 Case Studies

To illustrate the effectiveness of our techniques, we have conducted three distinct case studies. Each of these studies is centered around a specific use case for our techniques, and is representative for program comprehension challenges that are faced by software engineers in everyday life. While in certain cases there is some a priori knowledge regarding the user-level functionality of the system at hand, no implementation details are known in advance.

Trace exploration (Section 5.5)

- *Context:* The system is largely unknown, but an execution trace is available. No (or little) up-front knowledge is present.

[2] AspectJ, `http://www.eclipse.org/aspectj/`

[3] Extravis, `http://swerl.tudelft.nl/extravis/`.

- *Task:* Identify the phases that constitute the system's execution, and study its fan-in and fan-out characteristics.
- *Goal:* Get an initial feeling of how the system works, as a basis for a more focused examination.

Feature location (Section 5.6)

- *Context:* The user-level functionality of the system is known. The nature of the system is such that the features can be invoked at the user's discretion.
- *Task:* The execution of a scenario in which a set of features is invoked, and the visual detection of these features in the resulting trace.
- *Goal:* The establishment of relations between feature invocations and the corresponding source code elements.

Top-down program comprehension with domain knowledge (Section 5.7)

- *Context:* The system is unknown, and the user has little control over its functionality since it concerns a batch execution. However, the system's description provides clues as to the behavioral aspects (i.e., the execution phases) that are to be expected.
- *Task:* Using domain knowledge, formulate a hypothesis describing a set of conceptual phases, and validate it through the analysis of a typical trace.
- *Goal:* The use of a top-down approach to gain and refine (detailed) knowledge of a system's inner workings.

Each of these use cases is exemplified by means of a typical usage scenario that involves a medium-scale Java system[4].

5.5 Case Study 1: Trace Exploration

5.5.1 Motivation

When a system is largely unknown and an execution trace is available, being able to globally understand the control flow in the trace can be of great help in understanding the system. Particularly in the context of a legacy system that lacks a thorough documentation, any information on the system's inner workings is welcome. However, since execution tracing tends to result in large amounts of data, the *exploration* of such traces is by no means a trivial task. To illustrate how our techniques facilitate this process, we explore an industrial system called Cromod.

[4] Note that although these experiments involve Java because our tool-chain is optimized for Java systems, we are confident that our technique is applicable to other (non-object-oriented) languages.

5.5.2 Cromod

Cromod is an industrial Java system that predicts the environmental conditions in greenhouses. The system is built up from 145 classes that are distributed across 20 packages. According to the manual, it takes a greenhouse configuration (e.g., four sections, 15 shutters, and 40 lights) and a weather forecast as its input; it then calculates the optimal conditions and determines how certain parameters such as heating, lights, and shutters should be controlled; and then writes the recommended values. Since the calculations are done for a large number of discrete time frames, they typically induce massive numbers of interactions, which makes this system an interesting subject for trace visualization.

5.5.3 Obtaining the trace

The trace that results from a typical Cromod execution contains millions of events, of which a large part can be attributed to logging. At the recommendation of the developers, we have run the program at a log level such that the resulting trace contains roughly 270,000 method and constructor calls. This trace captures the essence of the execution scenario, and in terms of size, the visualization and comprehension of this trace remain a challenge. We conclude the setup with the extension of the trace with information on the system's hierarchical decomposition in terms of its package structure.

5.5.4 Analyzing the trace

Loading the trace into Extravis provides us with the initial views that are shown in Figure 5.2.

Studying fan-in and fan-out behavior

The circular view of the trace shows Cromod's structural decomposition and (the frequency of) the calls that occurred during execution. For example, we can see several edges that are thicker than usual, which suggests that most of the activity is centered around these particular calls. What is also noteworthy is that in the vicinity of certain packages the edges are predominantly colored red, which indicates a high degree of fan-in: the only outgoing calls here seem to be directed toward classes within the package. Examples of such packages are `util.fileio` and `model`: Figure 5.4 confirms our assumption by collapsing these packages, which results in aggregated relations that are clearly incoming in nature. From this observation, we draw the conclusion that these packages fulfill a library or utility role.

Identifying phases

The massive sequence view indicates that there are three major "phases" in the execution scenario. The first and third phases are characterized by two

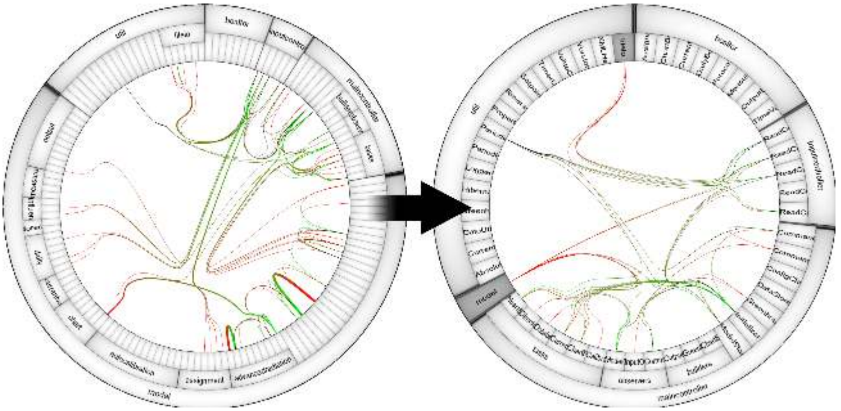


Figure 5.4 Collapsing the `model` and `util.fileio` packages in the Cromod trace.

small beams; in between we observe a long segment that appears to be somewhat broader, and shows a very consistent coloring. At this point in time, we made the hypothesis that the three stages concern (1) an input phase, (2) a calculation phase, and (3) an output phase. We attempt to validate this hypothesis in the following steps.

The first phase that we can visually discern looks like an almost straight vertical "beam". We zoom in on this phase by selecting a suitable interval, thus reducing the time frame under consideration. Now, Extravis only visualizes the interactions within the chosen time frame in both views. Turning our attention to the circular view (Figure 5.5), we learn that this first phase merely involves a limited number of classes and packages, and judging by the names of the packages and classes involved (e.g., `inputcontroller.ReadCromodForecast` and `fileio.InputFileScanner`), this phase mainly concerns I/O activity with an emphasis on input processing.

A quick glance through the second phase reveals a massive number of recurrent calls within the `model` package. The phase is mainly made up from the construction of `model.advancedradiation.Sun` objects by `model.advancedradiation.SolarModel`, and the creation of `model.Time` instances by `model.TimeValue`. Indeed, out of the 270,000 events in this execution scenario, 260,000 events concern calls to constructors. From these findings we conclude that this phase is concerned with the Cromod's main functionality, i.e., the model calculations.

Another noteworthy observation in the second phase is the occasional appearance of very thin lines in the massive sequence view of Figure 5.2. A closer look reveals that the aforementioned construction processes are incidentally interleaved with short fragments that involve readers of Cromod's XML inputs. Focusing on one such fragment (Figure 5.6), we learn that `util.PeriodicThread.run()` is invoked on `inputcontroller.ReadCromodForecast`, which results in a call to `util.fileio.InputFileScanner.scanForFiles()`. This example illustrates the purpose of our anti-aliasing technique: without the

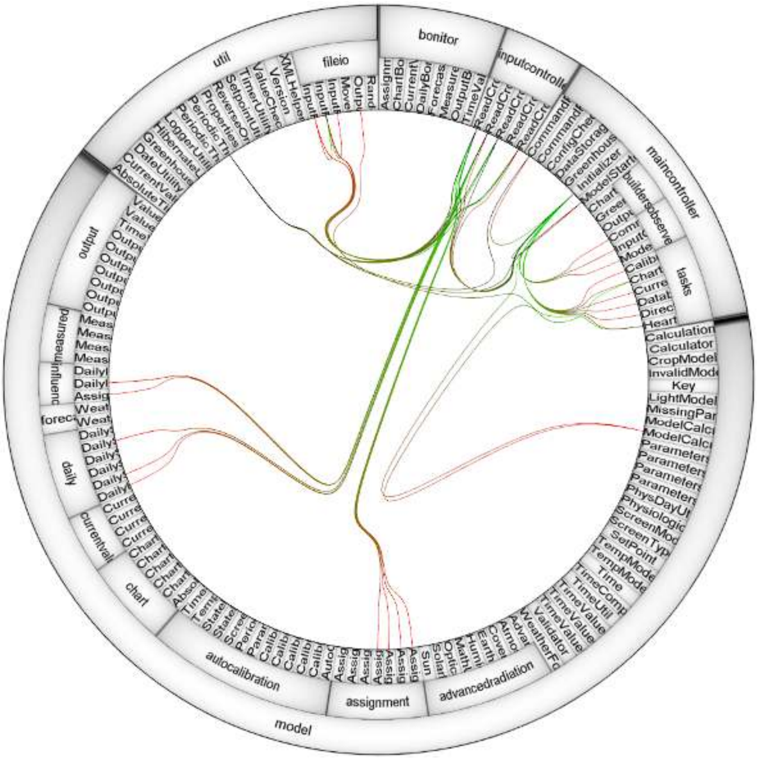


Figure 5.5 Circular view of Cromod's initialization phase.

use of IBAA, on a typical display that offers 1,200 pixel lines, and given a trace size of 270,000 calls, the odds of two calls ending up in the resulting massive sequence view would have been less than 1%.

The third phase is similar to the first phase: here we observe interactions between `maincontroller.tasks.CurrentValuesGenerator` and `util.fileio-.OutputFile` which again suggests I/O activity, focused on output rather than input. This provides strong evidence for the validation of the hypothesis formulated earlier.

5.5.5 Discussion

In this experiment, we explored a typical execution trace of an industrial system of which we had very little knowledge in advance. Our visualization techniques proved to be very useful in this context, and rapidly helped us gain a certain degree of knowledge of the system.

More specifically, the circular bundle view showed (1) Cromod's fan-in and fan-out behavior, and (2) the distribution of the events across the system. Moreover, the collapsing of packages aggregates the interrelationships

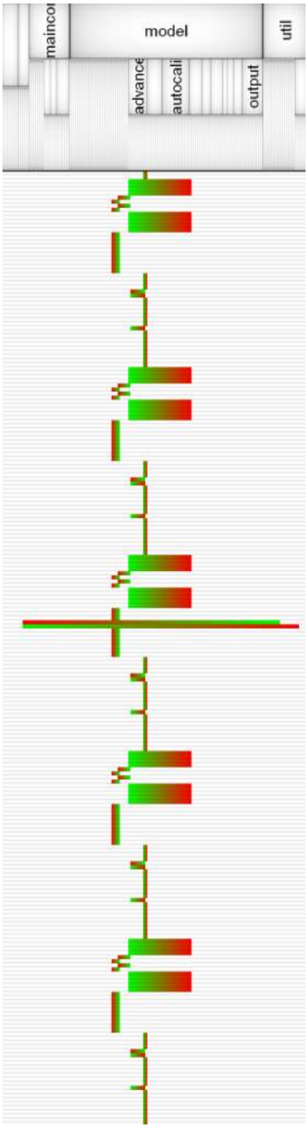


Figure 5.6 Zoomed massive sequence view of Cromod's second phase, focusing on a periodic fragment.

of these packages, rendering both the circular and the massive sequence view easier to read.

The massive sequence view provided an overview of the trace and indicated the existence of three major phases in the execution. We used zooming and call highlighting to learn more about the nature of these phases. The use of IBAA pointed out a series of outlier calls.

5.6 Case Study 2: Feature Location

5.6.1 Motivation

A significant portion of the effort in a maintenance task is spent on determining *where* to start looking within the system. In the context of specific features, this process is called *feature location* (Wilde et al., 1992). Its purpose is to relate a system's features to its source code, which enables an engineer to focus on the code fragments that are relevant to a feature's implementation (e.g., to handle change requests). While research into feature location is often concerned with automatic techniques such as concept analysis (Eisenbarth et al., 2001) and Latent Semantic Indexing (Poshyvanyk et al., 2007), we will attempt to *visually* locate certain features in an execution trace (similar to Kuhn and Greevy (2006)). In this context, we consider a feature to be a user-triggerable unit of functionality (Eisenbarth et al., 2003).

5.6.2 JHotDraw

JHotDraw[5] is a well-known, highly customizable Java framework for graphics editing. It was developed as a "design exercise" and is generally considered to be well-designed. It comprises 344 classes and 21 packages. Running the program presents the user with a GUI in which there is a set of features that may be invoked at the user's discretion, such as opening a file, inserting predefined figures or manual sketches, and adding textual information. While the authors are familiar with JHotDraw's user-level functionality, its inner workings are unknown in advance.

5.6.3 Obtaining the trace

To generate a suitable feature trace, we use an execution scenario that involves several major features that we want to detect: the creation of a new drawing, and the insertion of *five* different types of figures therein. These figures include rectangles, rounded rectangles, ellipses, triangles, and diamonds. To make the localization of the "new drawing" and the "insert figure" features easier, we invoke the aforementioned scenario a total of *three* times (Figure 5.7).

Since JHotDraw registers all mouse movements, the trace that results from our scenario is bound to contain a lot of noise. We have therefore filtered these mouse events to obtain a trace that is somewhat cleaner. The resulting trace contains a little over 180,000 events.

5.6.4 Analyzing the trace

Figure 5.8(a) shows the massive sequence view of the entire execution trace, in which we can immediately observe several recurrent patterns.

[5] JHotDraw 6.0, `http://www.jhotdraw.org/`

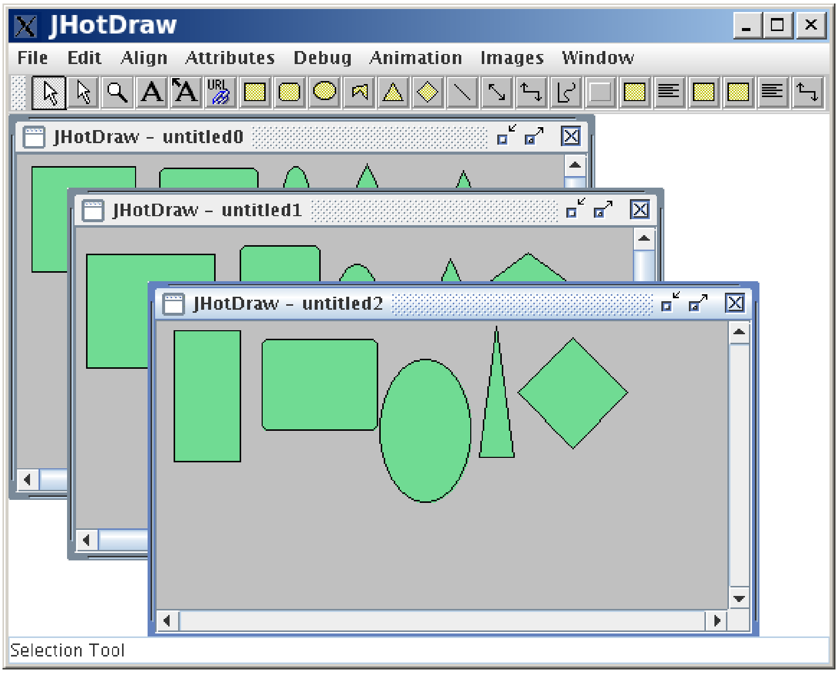


Figure 5.7 Execution scenario for JHotDraw, in which five different figures are inserted in three distinct drawings.

Locating the "new drawing" feature

Since in our trace scenario we invoked the "new drawing" feature three times, we are looking for a pattern that has the same number of occurrences. Finding these patterns in the massive sequence view is not very difficult: we can discern three similar blocks, all of which are followed by fragments of roughly the same length. This leads us to the assumption that the blocks concern the initialization of new drawings, and that the subsequent fragments pertain to the figure insertions. As a means of verification, we zoom in on these patterns to gather more evidence.

Locating the "insert figure" feature

Figure 5.8(b) presents a zoomed view of such a fragment, in which we can see the alleged initialization of the drawing in the top fraction. What follows is a series of five patterns that look very similar: a closer look reveals the most prominent difference to be the destinations of certain outgoing calls, which seem to differ especially in the fourth and fifth patterns. These calls are part of an intermitting series and in the latter patterns they are directed toward classes in *different* packages. To determine the identities of these classes, we zoom in on the third and fourth patterns and, in each case, refer to the circular view. Comparing the two views points out the differences: at the bottom of

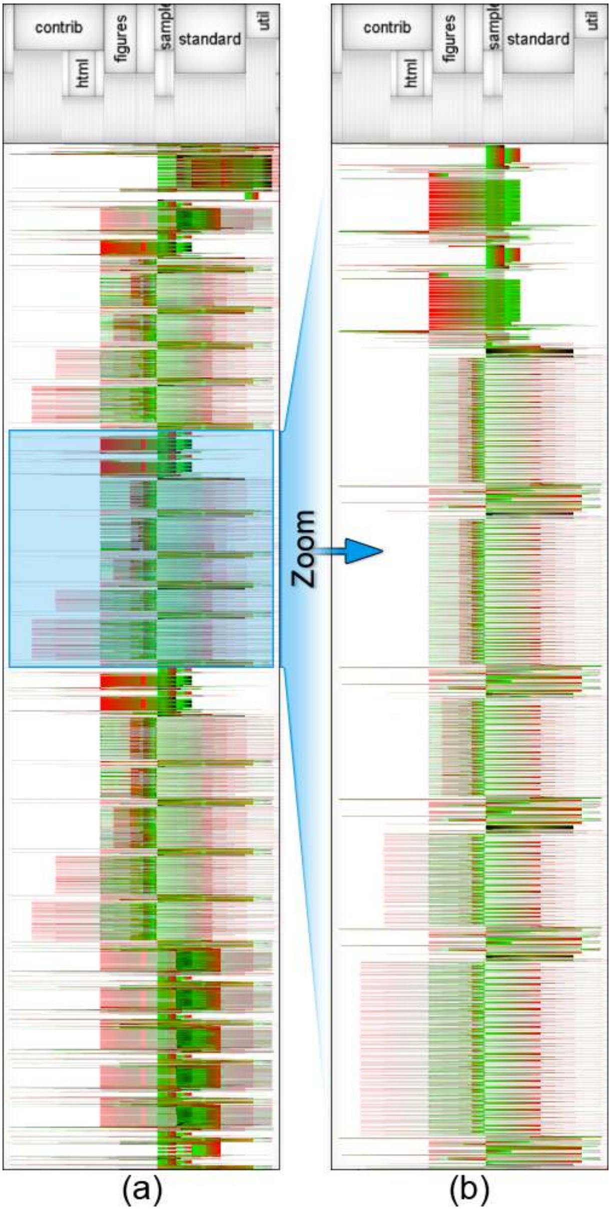
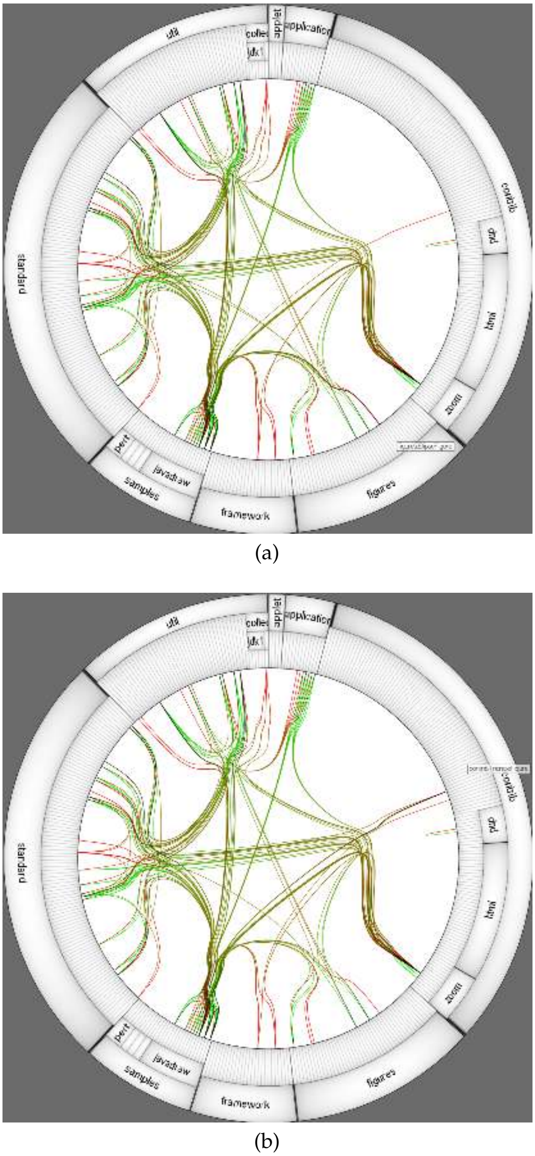


Figure 5.8 (a) Full trace of the JHotDraw scenario. (b) Zooming in on the "new drawing" feature and the subsequent figure insertions.

the first figure in Figure 5.9(b) we observe calls toward three different figure types in the `figures` package (the result of the three figures that are drawn at this point) whereas in the rightmost portion of the second figure there is also an *additional* call toward `contrib.TriangleFigure`. Repeating this task for



(a)

(b)

Figure 5.9 Circular bundle views of two alleged figure drawings, indicating very subtle differences.

the fifth pattern reveals another new figure: `contrib.DiamondFigure`, the fifth figure that was drawn in our scenario. These observations confirm that each of the five fragments concerns a figure insertion.

Taking into consideration that JHotDraw is an open-source project, the division between the figures can be explained as follows: `RectangleFigure`, RoundRectangleFigure, and EllipseFigure are standard figures in JHotDraw and therefore reside in figures, whereas TriangleFigure and DiamondFigure are in the contrib package because they were contributed by third parties.

5.6.5 Discussion

In this experiment, we have instrumented and executed a medium-sized, GUI-based program according to a scenario that involves certain user-triggerable features. We then visualized the resulting execution trace and pinpointed the locations of these features. The results are promising: choosing an appropriate scenario establishes a strong focus in recognizing the patterns associated with the features, which consequently requires little effort. The feature location process serves as an important first step toward understanding how the system's features are implemented (Wilde et al., 1992).

Our study has pointed out how the massive sequence view serves as an excellent basis for the (visual) recognition of recurrent patterns. While it does not allow for an easy extraction of more detailed information, selecting a suitable interval requires little effort. Consequently, the circular bundle supports the user in learning more about the events at hand, e.g., by enabling the identification of rather subtle differences in recurrent patterns.

5.7 Case Study 3: Top-down Program Comprehension with Domain Knowledge

5.7.1 Motivation

A common situation that developers often find themselves in is when they are not familiar with a specific software system, but that they do have a general knowledge concerning the system's background. Such domain knowledge may have been gained through experience with similar projects in the past. The existence of this up-front domain knowledge means that, a priori, a number of hypotheses about the software system can be formulated and then validated and refined in a top-down fashion (von Mayrhauser and Vans, 1995). In the process, we form auxiliary hypotheses and receive help from beacons that can direct us. Examples of such beacons are design patterns, and identifiers that have meaningful names. The findings lead to a greater understanding of the system under study, and form a basis for the refinement of the initial hypotheses. The subject system in this experiment is Checkstyle.

5.7.2 Checkstyle

Checkstyle[6] is an open-source tool that validates Java code. At the basis of this process lies a set of coding standards that can be extended by third parties, and while formerly the focus was on code layout issues, nowadays it also

[6] Checkstyle 4.3, http://checkstyle.sourceforge.net/

addresses such issues as design problems and bug patterns. The program consists of 21 packages that contain a total of 310 classes and offers both a graphical and a command line user interface. From a user's perspective, its functionality comprises taking a set of coding standards and a Java file as its input, and subsequently presenting a report. Since the system utilizes a batch execution, unlike in the previous case study we can only control the execution very indirectly.

5.7.3 Obtaining the trace

The generation of an execution trace is achieved by instrumenting and running Checkstyle from the command line. The input for our scenario consists of an XML file specifying Sun's Java coding conventions [7], and a typical, well-documented Java file that defines one class with 20 methods (300 LOC). Note that while Checkstyle also offers a GUI, in this case study we focus on its command line interface in order to fully concentrate on its core functionality. Executing and tracing this scenario yields an execution trace of nearly 200,000 events.

5.7.4 Comprehension hypothesis

The main advantage in this case study is the presence of domain knowledge: based on our knowledge of typical source code analysis tools we can speculate about certain properties of Checkstyle. In this experiment, the focus is on its *phases of execution*. In particular, we specify a set of characteristic phases that we expect during the execution of our scenario:

1. **Initialization.** As with most programs, we expect the initial phase to be concerned with initializations tasks such as command line parsing and input reading.

2. **AST construction.** Since tools dealing with source code structures typically make use of Abstract Syntax Trees (ASTs), we anticipate that Checkstyle exhibits a similar behavior. The first step in such approaches is the creation of an AST.

3. **AST traversal.** Once the AST has been generated, the standard procedure for programs in this context is to traverse its nodes and to take action when necessary.

4. **Report generation.** We expect the final phase to involve the generation of a report and its presentation to the user.

The hypothesis can be considered as a definition of *conceptual* phases. The aim in this experiment is to map these phases onto Checkstyle's actual execution.

[7] Code Conventions for the Java Programming Language, `http://java.sun.com/docs/codeconv/`

5.7.5 Analyzing the trace

The leftmost view in Figure 5.10 shows the massive sequence view for the full execution trace. Based on the initial view (i.e., at the highest level of granularity) we can roughly discern *five* major phases. For each of these phases, we report and interpret our findings in the next sections.

First major phase

Based on the initial massive sequence view, we choose to zoom in on the first phase (events 1 through 6,400). As it turns out, we are actually dealing with *two* subphases: this is demonstrated by the first of the zoomed views in Figure 5.10.

First subphase
A quick glance through the first subphase reveals a strong activity among a limited set of objects, being `ConfigLoader` and `DefaultConfiguration`.

Second subphase
The second subphase appears to be more interesting: here we witness an interleaving of sequences of similar calls. The main differences between these sequences are the objects on the receiving end, all of which seem to be located within the `checks` package. This is where the circular view proves helpful: in its temporal mode, browsing through the sequences reveals how all of the checks are processed one by one. This is illustrated in Figure 5.11, in which the calls are shown in a yellow-to-black (old-to-recent) fashion. Indeed, upon highlighting these calls, the tooltips tell us that the calls pertain to interactions between `DefaultConfiguration` and such checks as `checks.naming.PackageNameCheck` and `checks.header.HeaderCheck`. As a side note, we suspect the reason for the seemingly clockwise trend of the calls in Figure 5.11 to be the alphabetic order in which the checks are processed, which corresponds to the similarly alphabetically ordered packages and classes in the circular view.

Interpretation
From the many interactions involving configuration classes, we conclude that we are indeed dealing with initialization and configuration tasks. As such, we map Checkstyle's first phase onto the "initialization" phase in our hypothesis.

Second major phase

The second major phase that we consider is a series of 91,000 events, of which the largest part displays a very local behavior (second zoomed view in Figure 5.10). Referring to the circular view in this interval, we learn that most of the activity is concentrated in the `grammars` package, and in `grammars.GeneratedJavaLexer` in particular. To determine what caused this chain of events, we focus on the transition between this phase and the previous phase (Figure 5.12). Browsing through a limited set of events while

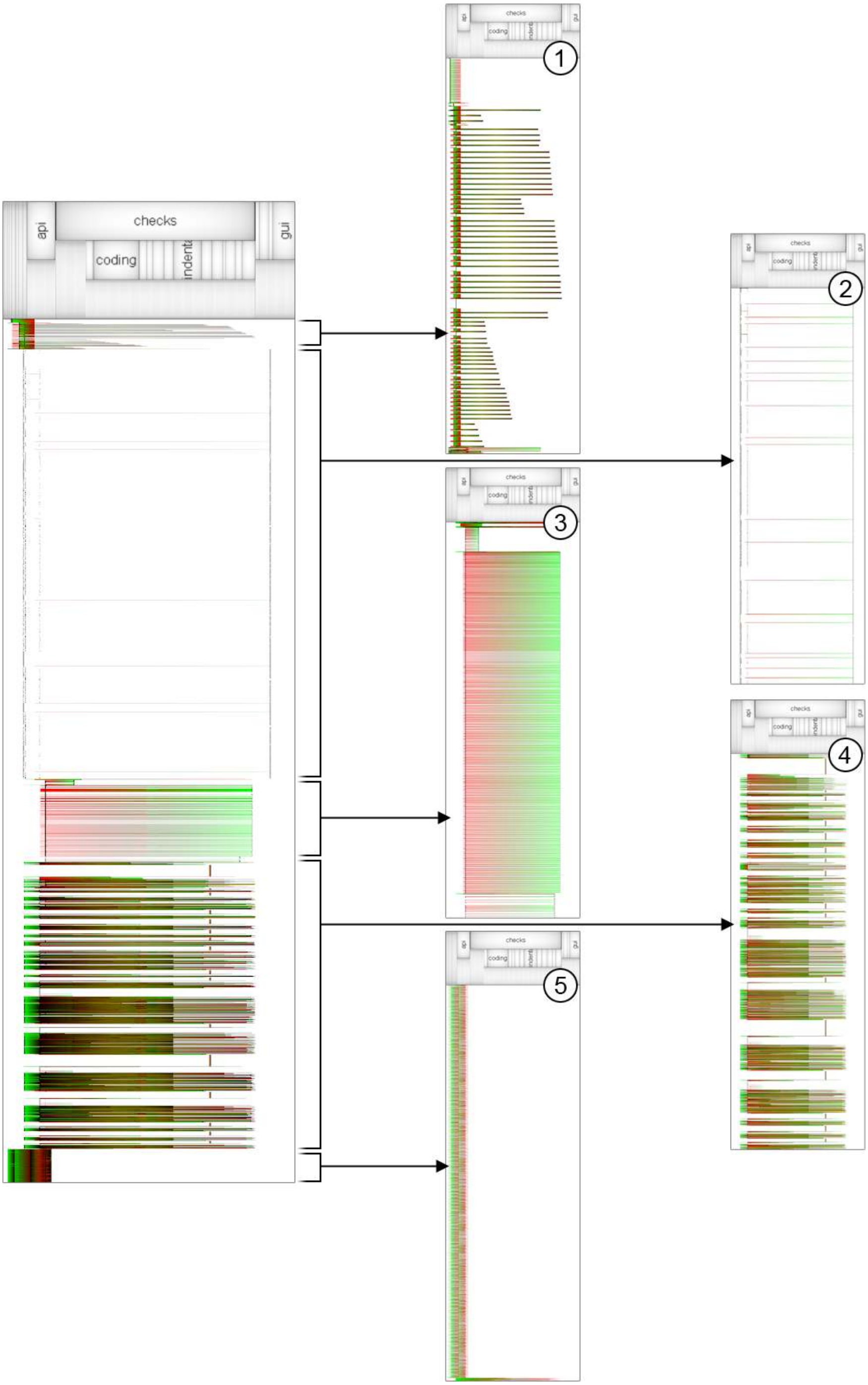


Figure 5.10 Massive sequence view of the entire CHECKSTYLE trace (left), and zoomed views of each of its five phases (right).

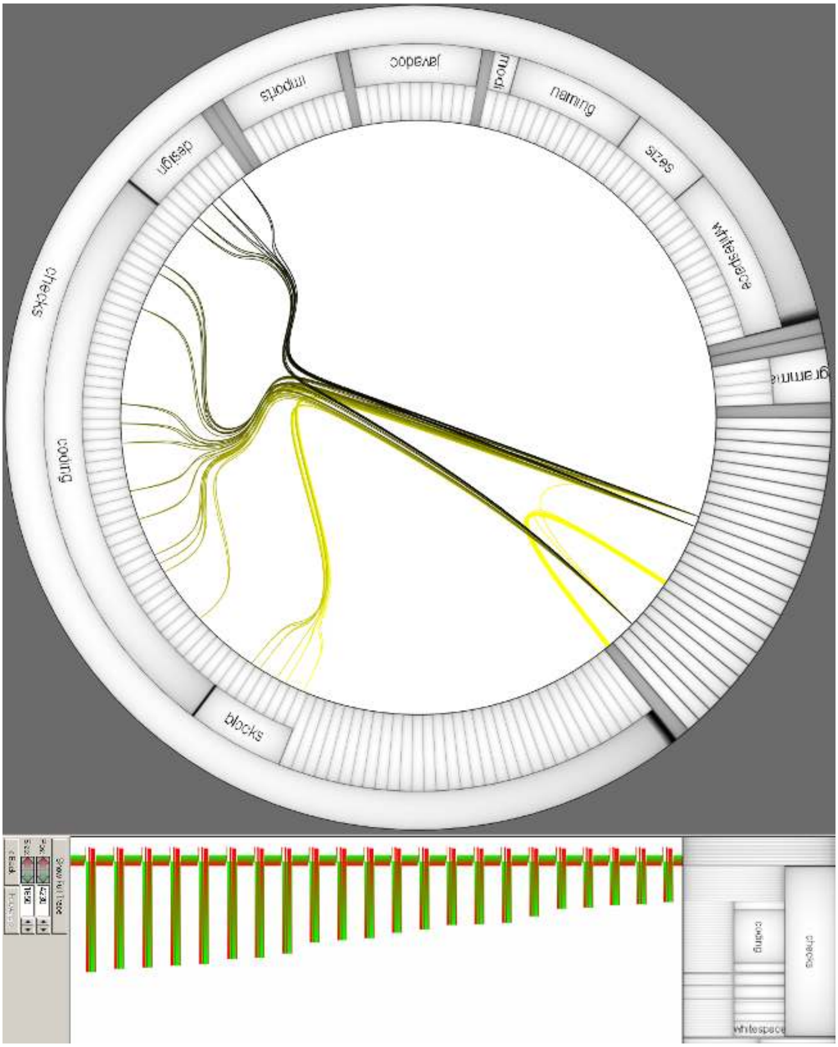


Figure 5.11 Full view of a recurrent sequence in CHECKSTYLE's first phase, in which the circular view indicates the receiving classes and the order in which they are processed.

scanning for events that have interesting identifier names, leads us to a call that bears the signature `api.DetailAST TreeWalker.parse(api.FileContents)`.

Interpretation

The signature of the aforementioned call suggests that this phase is concerned with parsing the input file and building an AST. The reason that we are not witnessing the explicit creation of the tree (i.e., node creations) is assumably because we have not instrumented external libraries such as the ANTLR

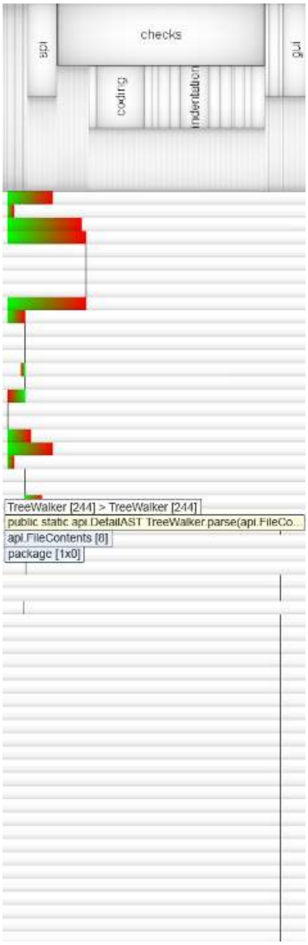

api
checks
gui
coding
indentation
TreeWalker [244] > TreeWalker [244]
public static api.DetailAST TreeWalker.parse(api.FileCo...
api.FileContents [8]
package [1x0]

Figure 5.12 Focusing on the transition between CHECKSTYLE's first and second phase. The tooltips provide information on the call sites.

Parser Generator. In conclusion, this phase maps seamlessly onto the "AST construction" phase in our hypothesis.

Third major phase

The next phase is a sequence that is characterized by a very consistent shape and coloring, which indicates a great degree of similarity between its 18,000 calls. By examining the earliest calls, we learn that the signatures of the initial calls are `void TreeWalker.walk(api.DetailAST, api.FileContents)` and `void TreeWalker.notifyBegin(api.DetailAST, api.FileContents)`. However, as pointed out by the zoomed view of this third phase in Figure 5.10, we are actually dealing with roughly *three* subphases: two relatively short parts, and a much longer one in between.

First subphase
This subphase consists of 1,300 events and starts off with a series of double calls, in which the TreeWalker repeatedly invokes void api.Check.setFileContents(api.FileContents) and void api.Check.beginTree(api.DetailAST) on a series of different Check subclass instances. Most of these calls lead to no further interactions, with the exception of checks.TodoCommentCheck (very short, broad "box" at the beginning of the zoomed view) and checks.GenericIllegalRegexpCheck (somewhat longer box).

Second subphase
In the second subphase, the receiver of the aforementioned calls is checks-.whitespace.TabCharacterCheck. The result is the involvement of this class in no less than 15,000 similar interactions, constituting most of the events in this phase. Judging by the method names and the return values that are being passed (e.g., getLines() and getTabWidth()), we are witnessing the processing of tab characters in the input Java file.

Third subphase
The third subphase is initially similar to the first subphase: more Check subclasses receive double calls. Furthermore, in the case of checks.sizes.LineLengthCheck the result is a total of 1,100 comparable calls of a line processing nature.

Interpretation
The rather meaningful names of the first few calls in this phase lead to the initial assumption that we are witnessing the traversal of the AST. However, this does not fully explain the peculiar shape of the massive sequence view in this stage. Class highlighting provides the answer as it reveals that api.DetailAST is completely absent in the interactions, which means that the aforementioned checks are not reliant on ASTs to fulfill their tasks. This can be explained by the fact that such formatting-oriented checks merely require a *lexical* analysis of the input file. In terms of our hypothesis, we conclude that this phase is *not* covered.

Fourth major phase

The fourth zoomed view in Figure 5.10 shows Checkstyle's fourth major phase, amounting to a total of 78,000 events. The phase is initiated by a call with the signature boolean TreeWalker.useRecursiveAlgorithm(), which returns false. The next call is void TreeWalker.processIter(api.DetailAST). By highlighting the third event – void TreeWalker.notifyVisit(api.DetailAST) – we learn that this particular call frequently occurs throughout the entire phase.

Another observation is the interleaving of two types of activities: a recurrent pattern with interactions spreading across a broad range of classes in the checks package, and an intermitting series of 16 vertical "beams". By highlighting the calls that constitute these beams and by referring to the circular view, we learn that these interactions are exclusively concerned with classes in the `checks.javadoc` package.

Interpretation
Judging by the source code associated with `boolean TreeWalker.useRecursiveAlgorithm()`, the decision of the program to use an iterative algorithm rather than a recursive algorithm can be attributed to Checkstyle's configuration during the execution. Furthermore, the frequent calls to `void TreeWalker.notifyVisit(api.DetailAST)` (and the ensuing interactions involving `Check` subclasses) lead to believe that the "Visitor" design pattern (Gamma et al., 1994) is used during the AST traversal. Finally, we discovered that 16 out of 20 method definitions in our input file are preceded by Javadoc entries, which we presume accounts for the 16 Javadoc call sequences that we encountered.

Based on our observations in this phase, we conclude that this phase maps to the "AST traversal" phase in our hypothesis.

Fifth major phase

The fifth and final phase as seen in the last zoomed view in Figure 5.10 contains nearly 15,000 events and actually starts with a main subphase and finishes with a small endphase.

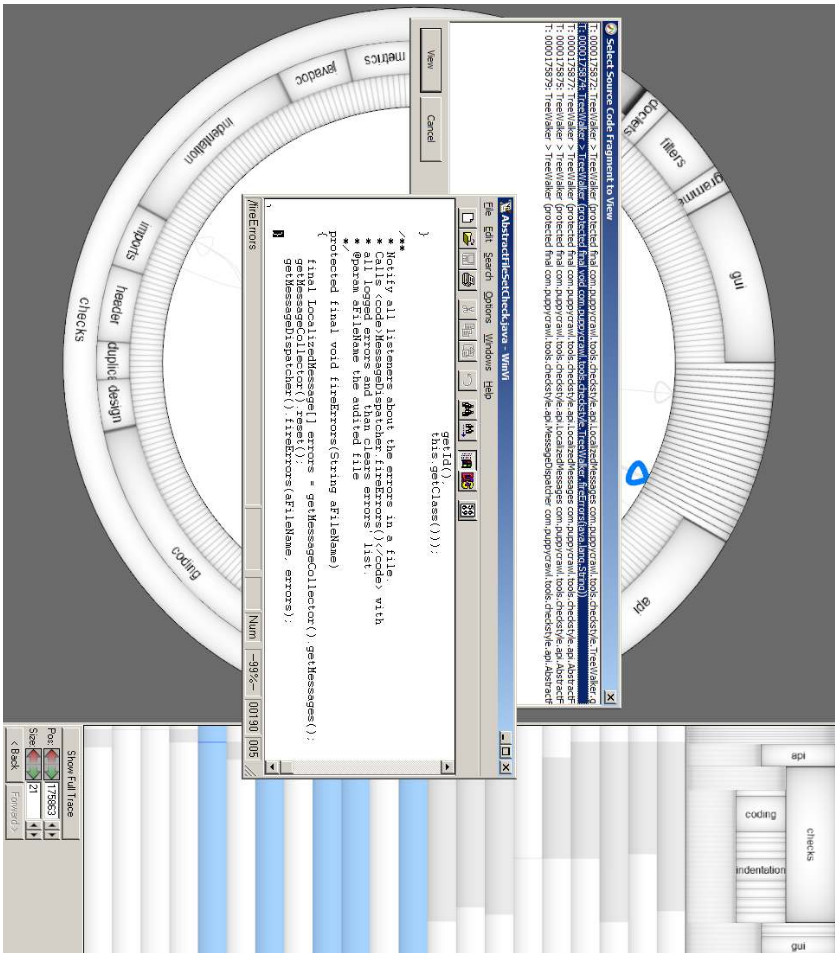
First subphase
The initiating call here is `void TreeWalker.fireErrors(java.lang.String)`, with the actual parameter being "Game.java", our input Java file. Since a brief glance through the ensuing interactions does not prove very meaningful, we double-click on the initial call to view the source code at this call site (Figure 5.13), in which the Javadoc entry states "Notify all listeners about the errors in a file".

Second subphase
Checkstyle's final phase starts off with `Checker` invoking the `void api.FileSetCheck.destroy()` method on `TreeWalker`. What follows is a series of `api.Check.destroy()` calls toward each of the `Check` subclasses; near the end, the most noteworthy event is `DefaultLogger.closeStreams()` as it precedes the writing of Checkstyle's output to the shell.

Interpretation
The source code associated with `void TreeWalker.fireErrors(java.lang.String)` leads us to believe that the first subphase is responsible for presenting the accumulated errors to the user, whereas the second subphase handles the program's termination. This phase maps perfectly onto the "report generation" phase in our hypothesis.

5.7.6 Discussion

In this case study we have focused on a medium-sized software system that, while being unknown to us in advance, has a functionality that we are familiar



```
T: 0000175872: TreeWalker > TreeWalker (protected final com.puppycrawl.tools.checkstyle.api.LocalizedMessages com.puppycrawl.tools.checkstyle.TreeWalker.g
T: 0000175874: TreeWalker > TreeWalker (protected final void com.puppycrawl.tools.checkstyle.TreeWalker.fireErrors(java.lang.String))
T: 0000175877: TreeWalker > TreeWalker (protected final com.puppycrawl.tools.checkstyle.api.LocalizedMessages com.puppycrawl.tools.checkstyle.api.AbstractF
T: 0000175875: TreeWalker > TreeWalker (protected final com.puppycrawl.tools.checkstyle.api.LocalizedMessages com.puppycrawl.tools.checkstyle.api.AbstractF
T: 0000175879: TreeWalker > TreeWalker (protected final com.puppycrawl.tools.checkstyle.api.MessageDispatcher com.puppycrawl.tools.checkstyle.api.AbstractF
```

```
                                getId(),
                                this.getClass()));
    }

    /**
     * Notify all listeners about the errors in a file.
     * Calls <code>MessageDispatcher.fireErrors()</code> with
     * all logged errors and than clears errors' list.
     * @param aFileName the audited file
     */
    protected final void fireErrors(String aFileName)
    {
        final LocalizedMessage[] errors = getMessageCollector().getMessages();
        getMessageCollector().reset();
        getMessageDispatcher().fireErrors(aFileName, errors);
    }
}
```

Figure 5.13 Looking up associated source code fragments.

with. We have used this domain knowledge to formulate a hypothesis that specifies a set of conceptual phases in a typical execution scenario. Table 5.1 summarizes the results.

The experiment was quite successful: with the exception of Checkstyle's third phase, each of the five phases that we discerned through the use of Extravis could be mapped onto a conceptual phase. In particular, the massive sequence view allows to rapidly identify the essential events, i.e., calls that are responsible for phase transitions. The circular bundle view is an aid when a more detailed visualizations of certain call sequences are required, and offers an easy link to the system's source code. While the third phase came unexpected, in retrospect it is perfectly understandable that certain aspects of an input file's source code are treated differently since the associated checkers have no need of ASTs.

Phase	#Calls	Notable calls	Prominent classes	Description	Maps to
1	6.4K	-	ConfigurationLoader DefaultConfiguration checks.*	Interactions between various configuration classes and checks.	Initialization
2	91K	TreeWalker.parse()	grammars.-GeneratedJavaLexer TreeWalker	Local activity in the grammars package, presumably involving external libraries.	AST construction

3	18K	TreeWalker.walk()
		TreeWalker.notifyBegin()
			TreeWalker
			FileContents
			4 Check-subclasses
				Non-AST related activities involving four specific check subclasses.	-

4	60K	TreeWalker.processIter()
		TreeWalker.notifyVisit()
			TreeWalker
			api.DetailAST
			remaining checks
				Interleaving between various check-related events and Javadoc processing.	AST traversal

5	8K	TreeWalker.fireErrors()
		TreeWalker.destroy()
			TreeWalker
			checks.*
				Processing of errors and termination of the program.	Report generation

Table 5.1 Results of the CHECKSTYLE experiment.

The experiment took only a few hours, and illustrates how domain knowledge and a top-down approach can lead to a significant level of understanding of the system under study. Moreover, additional (sub-)phases can be observed by means of zooming, which in turn can be used to refine initial hypotheses in an iterative fashion (cf. Reflexion models (Murphy et al., 2001)) until a sufficient level of understanding has been obtained for the task at hand.

5.8 Discussion

The case studies in Sections 5.5 through 5.7 have pointed out a series of potential applications of our approach in the context of understanding large execution traces and, by extension, understanding software systems. This section lists a number of important characteristics of our techniques and discusses both the advantages and limitations.

5.8.1 Advantages

Common trace visualization tools use UML sequence diagrams (or variants thereof) to display a system's structure and the detailed interactions between its components (e.g., De Pauw et al. (1993) and Briand et al. (2006)). Although sequence diagrams are very intuitive, they typically become difficult to navigate when the number of components or the time period under consideration become too large: situations where two-dimensional scrolling is necessary to grasp even relatively simple functionalities can rapidly occur, which easily disorients and confuses the user. EXTRAVIS, on the other hand, uses a scalable circular view that fits on a single screen. All of the system's components are hierarchically projected on a circle, and entities that are of no immediate interest can be collapsed, which improves readability and ensures that the user is not overwhelmed by too much information.

Moreover, the calling relationships between elements are visualized using bundling, which greatly improves the overall readability in case of many simultaneous relations. Through the use of colors, there is the ability to either (1) show these relationships in a chronological order, or (2) indicate the fan-in and fan-out behavior of the various entities.

The massive sequence view, which provides a concise overview of an entire execution trace, allows the user to easily zoom in on parts of the trace. This reduces the time period under consideration in *both* views and eases the navigation. Another benefit of this view is that it is easy to recognize patterns and phases on the macroscopic level and, by use of zooming, on the fine-grained level as well.

Finally, our techniques are aimed at the optimal use of screen real estate. The observation that a circular representation does not fit on a standard (rectangular) screen is valid; however, it is a matter of positioning the tool controls and settings in the unused space for the screen to be optimally used. Such improvements could be included in future versions of Extravis.

5.8.2 Limitations

While our techniques effectively visualize large execution traces that are normally too difficult to understand, the size of the input trace is limited in terms of our prototype tool. The reason for this is twofold: not only does Extravis require a substantial amount of *computational* resources – i.e., memory to keep track of all elements and relations, and CPU cycles to perform calculations, counts etc. – but visualizing large systems also requires a considerable amount of *screen real estate*. The latter problem exists because not all events can be visualized in the massive sequence view in a non-ambiguous fashion in case there are more events than there are horizontal pixel lines. It must be noted, however, that Extravis is not necessarily a stand-alone tool; it could well be used as part of a tool chain, e.g., after some abstraction phase.

Moreover, while the circular bundle view is a useful means to display certain characteristics of a program without the need for scrolling, it can be fairly difficult to grasp the temporal aspect. When considering a small time frame (e.g., 50 calls), the circular approach's temporal mode does not make for a visualization that is as readable and intuitive as a sequence diagram, since it requires the interpretation of colors rather than a top-to-bottom reading. In other words, while the display of the system's *entire* structure is useful in such applications as fan-in and fan-out analysis, this information is not always needed.

Finally, threads are currently not supported. While adding a functionality to Extravis for switching between threads seems feasible, actually being able to show *interactions* between these threads is not trivial. As threads typically convey important information on the (interleaving of) distinct processes

Criterion	Extravis implementation
Overview	Massive sequence view
Zooming	Zooming in the massive sequence view
Filtering	Collapsing of elements
Details-on-demand	Highlighting of elements / relations
Relate	Circular view (with bundling)
History	Forward / back buttons
Extract	Save / load current state

Table 5.2 Shneiderman's GUI criteria.

within running software, an effective visualization thereof certainly warrants future investigations.

5.8.3 Shneiderman criteria

Shneiderman (1996) introduced seven criteria for assessing the graphical user interfaces of information visualizations. Table 5.2 outlines how the two synchronized views of Extravis satisfy each of these seven criteria.

5.8.4 Threats to validity

The case studies that we have presented are representative for real-life situations that software developers encounter on a daily basis. The trace exploration, feature location, and top-down analysis scenarios that we used to study and understand the subject software systems are realistic and, as was mentioned in the motivational sections of the studies, occur in various contexts. Nevertheless, there are a number of aspects in which our experiments may differ from real-world situations. We now address the factors that we feel are the most influential.

First, in our experiments we have occasionally relied on identifiers having meaningful names. In Checkstyle, for example, we often used the method's signatures to get an indication of the intended functionality. It must be noted that the presence of meaningful identifier names is by no means a guarantee in everyday software systems.

Secondly, with respect to our feature location study, we have stated that our definition of a feature is a user-triggerable unit of functionality. While this is a common assumption in this problem area (e.g., (Eisenbarth et al., 2003)), our visual form of feature location is more difficult if the features at hand can *not* be invoked directly. When considering Cromod, for example, it is hard to control the execution of its distinct features because it concerns a batch execution based on a set of complex input files. In other words, the applicability of our techniques in feature location tasks depends on the nature of the system's execution.

Finally, the initial traces in two of the case studies were inexplicably huge. Closer inspection revealed that these traces contained massive numbers of events that can be attributed to non-functional requirements, such as logging (e.g., the Cromod case) or registering mouse events (e.g., in JHotDraw). Assuming that mouse movements and logging are not particularly interesting in grasping a system's general functionality, we carefully filtered out these particular events in a preprocessing step, so as to prevent the traces and the resulting visualizations from becoming unnecessarily complex. It should be noted that this task is rather delicate, and in performing similar experiments one must be careful not to accidentally filter any events that pertain to functionalities that the user considers to be relevant.

5.9 Related Work

Research into trace visualization has resulted in various techniques and tools over the years. Most related articles are concerned with explaining the visualization tools and techniques by example; in contrast, we have reported on the use of our techniques in several real-world scenarios.

De Pauw et al. (1993) are known for their work on IBM's Jinsight, a tool for visually exploring a program's runtime behavior. Many features of this prototype tool have since found their way into Eclipse as plug-ins, more specifically, the *Test & Performance Tools Platform* (TPTP). Though being useful for program comprehension purposes, scalability remains worrisome. To this end, the authors have introduced the *execution pattern notation* (De Pauw et al., 1998), which unfolds the graph from a typical sequence diagram (or any variant of a Jacobson interaction diagram (Jacobson, 1992)) into tree structures. This layout emphasizes the progression of time and not so much the thread of control.

Lange and Nakamura (1995b) report on Program Explorer, a trace visualization tool targeted at C++ software. Several views are available, of which the *class graph* plays a central role. Through such abstractions as merging, pruning, and slicing, the tool attempts to reduce the search space when studying execution traces; however, the degree of automation of these abstractions is unclear. Furthermore, the tool does not offer a comprehensive view of all the packages and classes that are involved, and selecting a trace interval for detailed viewing does not seem feasible.

Jerding et al. (1997) present ISVis, a tool that features two simultaneous views of a trace: a continuous sequence diagram, and a *mural* view that is similar to our massive sequence view. ISVis' main strength lies in automatic pattern detection, which allows to summarize common execution patterns, and reduces the size of the trace considerably. Our approach differs from ISVis in that the latter deals from the perspective of sequence diagrams (which cannot contain a large number of structural elements), whereas our tool is centered around a scalable circular view.

AVID, a visualization tool by Walker et al. (1998), aims at exploring a system's behavior by manually defining a high-level model of a system and then enriching it with trace data collected during the system's execution. This is a manual step that involves multiple iterations, thus incrementally improving the user's comprehension of the system. At the basis of this operation lies the Reflexion process (Murphy et al., 2001). Although there is support for the (sampling-based) selection of a scenario fragment, the tool faces a significant scalability issue as scenarios still induce a potentially large amount of trace data that cannot be directly visualized.

Reiss and Renieris (2001) note that execution traces are typically too large to visualize directly and therefore propose to select, compact, and encode the trace data.

Jive, also by Reiss (2003b), is a Java front-end that visualizes a program's behavior while it is running, rather than analyzing its traces in a post mortem fashion. While the runtime visualization and relatively small overheads render it an attractive tool, it is hard to visualize entire executions. It does, however, provide a view on the classes that are active during a specific phase of the software's execution, and it also allows to perform a rudimentary performance analysis.

Systä et al. (2001) present Shimba, an environment that uses sequence diagrams to visualize interactions between user-specified components. Pattern recognition is applied to cope with the scalability problems that are often associated with these diagrams: in particular, the authors have employed the Boyer-Moore string matching algorithm. Furthermore, the structural entities of the system under analysis are clustered according to some criterion: For example, by clustering classes to packages, less structural entities and relationships need to be shown, which renders the trace far more tractable.

Richner and Ducasse (2002) propose to use their Collaboration Browser to reconstruct the various object collaborations and roles in software systems. This is achieved by selecting a class and then specifying queries to learn more about the interactions in which this class is involved. Iteratively studying the results of these queries and refining or adding new queries leads to a deeper understanding of the subject software system.

Ducasse et al. (2004) use polymetric views to visualize certain metrics that are collected at runtime, resulting in a significant reduction of information. Their approach is mainly aimed at recognizing those entities in a system that are actively allocating new objects, or that are frequently calling other classes. Although their approach works *offline*, there are similarities with the way in which Reiss projects dynamic metrics in his Jive-tool (Reiss, 2003b).

Kuhn and Greevy (2006) correlate feature traces with the help of "signals in time": they visualize traces as signals, of which the amplitude is determined on the basis of the stack depth at points during the execution. The idea is that similar traces exhibit comparable sequences of amplitude values, and that these similarities can be visually detected. Their work focuses solely on feature location and not so much on more general program comprehension. Similar work by Zaidman and Demeyer (2004) uses the relative frequency of method executions to compare regions in traces, as opposed to using stack depths.

Greevy et al. (2006b) present a 3D visualization of a software system's execution. The visualization metaphor that they use to display large amounts of dynamic information is that of growing towers, with towers becoming taller as more instances of a type are created. The authors aim to (1) determine which parts of the system are actively involved in a particular (feature) scenario execution, and (2) identify patterns of activity that are shared among different features of the system.

Hamou-Lhadj et al. (2005) report on a technique to recover behavioral design models from execution traces. Starting with a complete trace, they determine which classes are utility classes, or classes having a high level of fan-in and low (or non-existent) fan-out. Once these classes are removed from the trace, the resulting trace is visualized in the Use Case Map (UCM) notation. UCMs provide a compact and hierarchical view of the main responsibilities per class combined with architectural components. However, UCMs do not provide a global overview of the application, are not easily navigable, and are more targeted toward understanding very specific parts of a system.

5.10 Conclusions

Dynamic analysis is generally acknowledged to be a useful means to gain insight about a system's inner workings. A major drawback of dynamic analysis is the huge amounts of trace data that are collected and need to be analyzed. As such, designing an effective trace visualization that (1) is able to cope with these huge amounts of data, and (2) does not confuse the viewer, remains a challenge.

The solution that we propose to tackle this scalability issue is centered around two synchronized views of an execution trace. The first view, which we call the circular bundle view, shows all the system's structural elements (e.g., classes and packages) and their dynamic calling relationships in a bundled fashion. The second view, the massive sequence view, shows a large-scale message sequence chart that uses a new anti-aliasing technique and that provides an interactive overview of an entire trace. The linking of the two views creates a synergy that ensures the easy navigation and analysis of large execution traces. Our approach is implemented in a publicly available tool called Extravis.

To illustrate the broad range of potential usage contexts of our approach, we conducted three typical usage scenarios on three different software systems. More specifically, we performed (1) trace exploration, (2) feature location, and (3) top-down program comprehension with domain knowledge. For each of these scenarios, we have presented anecdotal evidence on how our approach helped us to gain different levels of understanding of the software systems under study. Finally, we have reported on the strengths and limitations of our tool, discussed the threats to validity in our case studies, and outlined the added value over related work.

To summarize, our contributions in this chapter are:

- A novel approach to visualizing execution traces that employs two synchronized views, namely (1) a circular bundle view for displaying the structural elements and bundling their call relationships, and (2) a massive sequence view that provides an interactive overview.

- The application of our tool, based on this approach, on three distinct software systems in three program comprehension contexts: trace exploration, feature detection, and top-down analysis.

5.10.1 Future work

There are many potential directions for future work, primarily in terms of improving our techniques and subjecting them to more thorough evaluations.

Among the *improvements* is to facilitate the comparison of execution traces: for example, observing two traces side by side (and thereby detecting correlations) might make feature location considerably easier.

Furthermore, we want to investigate the role of threads in our visualization, and come up with techniques to effectively display both the threads and their interactions.

Future applications include not only the visualization of larger execution traces, but also the detection of outliers. Outlier detection concerns the revelation of call relationships that are not allowed to exist for some reason, e.g., because the elements at hand belong to non-contiguous layers. The circular view, with its ability to show relations from entire traces in a bundled fashion, provides an excellent basis for the detection of such relationships.

With respect to further *evaluations*, we need to collect real evidence of our techniques' usefulness in practice. Specifically, in the context of a software system with large traces, we envision a controlled experiment that involves Extravis, a series of comprehension tasks, and several test subjects who are not familiar with the system. These tasks would have to be performed by the subjects, part of whom have access to our tool whereas others have not. The results of such an experiment will provide valuable information with respect to the practical applicability of our techniques. The next chapter discusses the design and execution of such an experiment.

Trace Visualization: A Controlled Experiment*

Chapter 6

The previous chapter presented EXTRAVIS, *our tool for supporting program comprehension through the visualization of large traces. Similar to other trace visualization techniques in the literature, our tool was validated through anecdotal evidence, but should also be quantitatively evaluated to assess its added value to existing solutions. In this chapter, we report on a controlled experiment concerning trace visualization for program comprehension. We designed eight typical tasks aimed at gaining an understanding of a representative subject system, and measured how a control group (using the Eclipse IDE) and an experimental group (using both Eclipse and* EXTRAVIS*) performed in terms of correctness and time spent. The results are statistically significant in both regards, showing a 21% decrease in time and a 43% increase in correctness for the latter group.*

6.1 Introduction

A major challenge in software maintenance is to understand the software at hand. As software is often not properly documented, up to 60% of the maintenance effort is spent on gaining a sufficient understanding of the program (Corbi, 1989; Basili, 1997). Thus, the development of techniques and tools that support the comprehension process can make a significant contribution to the overall efficiency of software development.

Common approaches in the literature can be roughly broken down into static and dynamic approaches (and combinations thereof). Whereas static analysis relies on such artifacts as source code and documentation, dynamic analysis focuses on a system's execution. An important advantage of dynamic analysis is its preciseness, as it captures the system's actual behavior. Among the drawbacks are its incompleteness, as the gathered data pertains solely to the scenario that was executed; and the well-known scalability issues, due to the often excessive amounts of trace data.

To cope with the issue of scalability, a significant portion of the literature on program comprehension has been dedicated to the reduction (Reiss and Renieris, 2001; Hamou-Lhadj and Lethbridge, 2006) and visualization (De Pauw et al., 1993; Jerding et al., 1997) of execution traces. Among our share of these techniques and tools is EXTRAVIS, a tool that offers two interactive views of large execution (Chapter 5). Through a series of case studies we illustrated

*This chapter is based on our publication in the Proceedings of the 17th International Conference on Program Comprehension (Cornelissen et al., 2009b). It is co-authored by Andy Zaidman, Bart van Rompaey, and Arie van Deursen.

how Extravis can support different types of common program comprehension activities. However, in spite of these efforts, there is no quantitative evidence of the tool's usefulness in practice: to the best of our knowledge, no such evidence is offered for any of the trace visualization techniques in the program comprehension literature.

The purpose of this chapter is the design of a controlled experiment to assess the usefulness of trace visualization for program comprehension, and the execution of this experiment to validate Extravis. Furthermore, to gain insight into the nature of its added value, we attempt to identify which types of tasks benefit most from trace visualization and from Extravis. To fulfill these goals, we perform a controlled experiment in which we measure how the tool affects (1) the time that is needed for typical comprehension tasks, and (2) the correctness of the answers given during those tasks.

The remainder of this chapter is structured as follows. Section 6.2 provides a background on dynamic analysis and trace visualization, and motivates our intent to conduct controlled experiments. Section 6.3 offers a detailed description of the experimental design. Section 6.4 discusses the results, and threats to validity are treated in Section 6.5. Section 6.6 outlines related work, and Section 6.7 offers conclusions and future directions.

6.2 Background

6.2.1 Execution trace analysis

The use of dynamic analysis for program comprehension has been a popular research activity in the last decades. In our literature survey in Chapter 2, we identified a total of 172 articles on this topic that were published between 1972 and June 2008. More than 30 of these papers concern *execution trace analysis*, which has often shown to be beneficial to such activities as feature location, behavioral analysis, and architecture recovery.

Understanding a program through its execution traces is not an easy task because traces are typically too large to be comprehended directly. For example, Reiss and Renieris (2001) report on an experiment in which one gigabyte of trace data was generated for every two seconds of executed C/C++ code or every ten seconds of Java code. For this reason, there has been significant effort in the automatic *reduction* of traces to make them more tractable (e.g., Reiss and Renieris (2001); Zaidman and Demeyer (2004); Hamou-Lhadj and Lethbridge (2006)). Another common approach is the *visualization* of execution traces: key contributions on this subject include Jinsight by De Pauw et al. (1993), Scene from Koskimies and Mössenböck (1996), ISVis by Jerding et al. (1997), and Shimba from Systä et al. (2001).

6.2.2 Extravis

Our own contributions to the field of trace understanding include Extravis, a publicly available[1] tool for the visualization of large execution traces (Chapter 5). Extravis provides two linked, interactive views. The *massive sequence view* is essentially a large-scale UML sequence diagram (similar to the Information Mural by Jerding and Stasko (1998)), and offers an overview of the trace and the means to navigate it (Figure 5.8 on page 108). The *circular bundle view* hierarchically projects the program's structural entities on a circle and shows their interrelationships in a bundled fashion (Figure 5.1 on page 94). We qualitatively evaluated the tool in various program comprehension contexts, including trace exploration, feature location, and top-down program comprehension. The results confirmed Extravis' benefits in these contexts, the main advantages being its optimal use of screen real estate and the improved insight into a program's structure. However, we hypothesized that the relationships in the circular view may be difficult to grasp.

6.2.3 Validating trace visualizations

Trace visualization techniques in the literature have been almost exclusively evaluated using anecdotal evidence: there has been no effort to quantitatively measure the usefulness of trace visualization techniques in practice, e.g., through controlled experiments. Moreover, most existing approaches involve *traditional* visualizations (Chapter 2), i.e., they rely on UML, graph, or tree notations, to which presumably most software engineers are accustomed. By contrast, Extravis uses non-traditional visualization techniques, and Storey argues that advanced visual interfaces are not often used in development environments because they tend to require complex user interactions (Storey, 2005). These reasons have motivated us to empirically validate Extravis through a controlled experiment, in which we seek to assess its added value in concrete maintenance contexts.

6.3 Experimental Design

The primary purpose of this experiment is a first quantitative evaluation of trace visualization for program comprehension. To this end, we define a series of typical comprehension tasks and measure Extravis' added value to a traditional programming environment: in this case, the Eclipse IDE[2]. Our choice for Eclipse as a baseline is motivated by its popularity among researchers and practitioners, who use it for both development and maintenance purposes. Similar to related efforts (e.g., Lange and Chaudron (2007); Quante (2008)) we maintain a distinction between *time spent* and *correctness*.

Furthermore, we seek to identify the types of tasks to which the use of Extravis, and trace visualization in general, is the most beneficial.

[1] Extravis, `http://swerl.tudelft.nl/extravis`

[2] Eclipse IDE, `http://www.eclipse.org`

6.3.1 Research questions & hypotheses

Based on our motivation in the previous section, we distinguish the following research questions:

1. Does the availability of Extravis reduce the *time* that is needed to complete typical comprehension tasks?

2. Does the availability of Extravis increase the *correctness* of the answers given during those tasks?

3. Based on the results, which *types* of tasks can we identify that benefit most from the use of Extravis?

Associated with the first two research questions are two null hypotheses, which we formulate as follows:

- **Hypothesis H1$_0$:** The availability of Extravis does not impact the time needed to complete typical comprehension tasks.

- **Hypothesis H2$_0$:** The availability of Extravis does not impact the correctness of answers given during those tasks.

The alternative hypotheses that we use in the experiment are the following:

- **Hypothesis H1:** The availability of Extravis reduces the time needed to complete typical comprehension tasks.

- **Hypothesis H2:** The availability of Extravis increases the correctness of answers given during those tasks.

The rationale behind the first alternative hypothesis is the fact that Extravis provides a broad overview of the subject system on one single screen, which may guide the user to his or her goal more easily.

The second alternative hypothesis is motivated by the inherent preciseness of dynamic analysis with respect to actual program behavior: For example, the resolution of late binding may result in more accurate answers.

To test hypotheses H1$_0$ and H2$_0$, we define a series of comprehension tasks that are to be addressed by both a control group and an experimental group. The difference in treatment between these groups is that the former group uses a traditional development environment (the "Eclipse" group), whereas the latter group also has access to Extravis (the "Ecl+Ext" group). We maintain a between-subjects design, meaning that each subject is either in the control or in the experimental group.

Sections 6.3.2 through 6.3.6 provide a detailed description of the experiment.

Activity	Description
A1	Investigating the functionality of (a part of) the system
A2	Adding to or changing the system's functionality
A3	Investigating the internal structure of an artifact
A4	Investigating dependencies between artifacts
A5	Investigating runtime interactions in the system
A6	Investigating how much an artifact is used
A7	Investigating patterns in the system's execution
A8	Assessing the quality of the system's design
A9	Understanding the domain of the system

Table 6.1 Pacione's nine principal activities.

6.3.2 Object & task design

Designing a controlled experiment for evaluating a visualization technique is not trivial because of several important challenges. (North, 2006): for example, while multiple choice questions may steer a test subject toward a certain answer, the answers to open questions are more difficult to score; and time constraints that are too strict may leave little room for deep insights. In the following, we describe how we attempt to meet these challenges.

The system that is to be comprehended by the subject groups is Checkstyle, an open source tool that employs "checks" to verify if source code adheres to specific coding standards. Our choice for Checkstyle as the object of this experiment was motivated by the following factors:

- Checkstyle comprises 310 classes distributed across 21 packages, containing a total of 57 KLOC.[3] This makes it tractable for an experimental session, yet representative of real life programs.

- It is written in Java, with which many potential subjects are sufficiently familiar.

- The researchers involved in the design and execution of our experiment are familiar with its internals as a result of earlier experiments (Zaidman et al., 2008; Van Rompaey and Demeyer, 2008; Cornelissen et al., 2008b). Furthermore, the lead developer was available for feedback.

To obtain the necessary trace data for Extravis, we instrument Checkstyle and execute it according to two scenarios. Both involve typical runs with a small input source file, and only differ in terms of the input configuration, which in one case specifies 64 types of checks whereas the other specifies only six. The resulting traces contain 31,260 and 17,126 calls, respectively, and are too large to be comprehended without tool support.

With respect to the comprehension tasks that are to be tackled during the experiment, the main criteria are for them to be (1) representative of real maintenance contexts, and (2) not biased towards any of the tools being used. To

[3] Measured using `sloccount` by David A. Wheeler, `http://sourceforge.net/projects/sloccount/`.

Task	Activities	Description
T1	A{1,7,9}	globally understanding the main stages in a typical CHECKSTYLE scenario
T2.1	A{4,8}	identifying three classes with a high fanin and a low fanout
T2.2	A{4,8}	identifying a class in package `X` with a strong coupling to package `Y`
T3.1	A{1,2,5,6}	describing the life cycle of check `X` during execution
T3.2	A{3,4,5}	listing the identifiers of all interactions between check `X` and class `Y`
T3.3	A{3,4,5,9}	listing the identifiers of additional interactions in case of check `Z`
T4.1	A{1,3}	providing a detailed description of the violation handling process
T4.2	A{1,5}	determining whether check `X` reports violations

Table 6.2 Descriptions of the comprehension tasks.

this end, we use the framework by Pacione et al. (2004), who argue that *"a set of typical software comprehension tasks should seek to encapsulate the principal activities typically performed during real world software comprehension"*. They distinguish between nine principal activities that focus on both general and specific reverse engineering tasks and that cover both static and dynamic information (Table 6.1). The latter aspect significantly reduces a bias towards either of the two tools used in this experiment.

Guided by these criteria, we created four representative tasks (subdivided into eight subtasks) that highlight many of CHECKSTYLE's aspects at both high and low abstraction level. Table 6.2 provides outlines of the tasks and shows how each of the nine activities from Pacione et al. is covered by at least one task: for example, activity A1, *"Investigating the functionality of (part of) the system"*, is covered by tasks T1, T3.1, T4.1, and T4.2; and activity A4, *"Investigating dependencies between artifacts"*, is covered by tasks T2.1, T2.2, T3.2, and T3.3.

To render the tasks even more representative of real maintenance situations, we have opted for open questions rather than multiple choice. The researchers can award up to four points for each task to accurately reflect the (partial) correctness of the subjects' answers. While at the same time open questions prevent the subjects from guessing, it should be noted that the answers are more difficult to judge, especially because the authors of this chapter were not involved in CHECKSTYLE's design or development. For this reason, we called upon CHECKSTYLE's lead developer, who was willing to review and refine our concept answers. The resulting answer model is provided in Appendix B. Following the experiment, the first two authors of this chapter select the answers of five random subjects, review them using the answer model, and compare the scores to verify the soundness of the reviewing process.

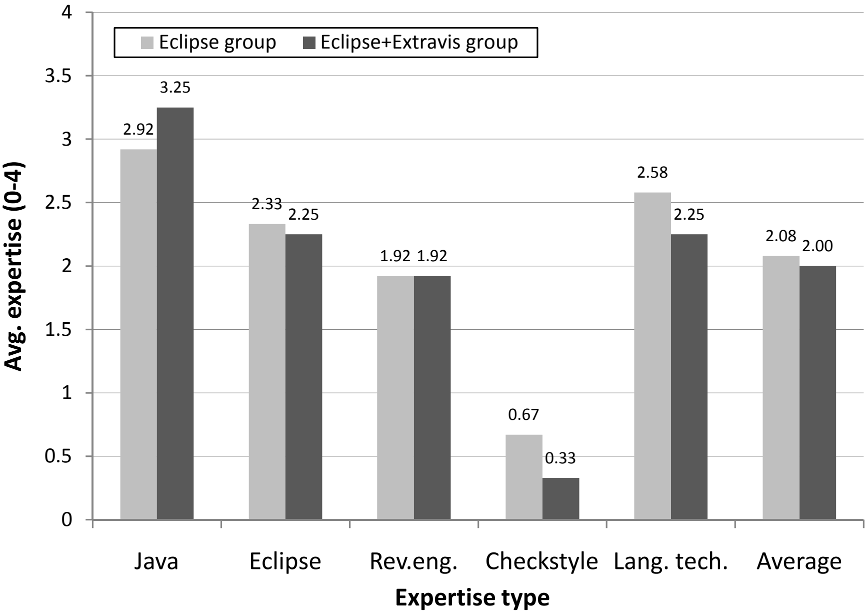


Figure 6.1 Average expertises of the subject groups.

6.3.3 Subjects

The subjects in this experiment are 14 Ph.D. candidates, five M.Sc. students, three postdocs, one associate professor, and one participant from industry. The resulting group thus consists of 24 subjects, and is quite heterogeneous in that it represents eight different nationalities, and M.Sc. degrees from thirteen universities. The M.Sc. students are in the final stage of their study, and the Ph.D. candidates represent different areas of software engineering, ranging from software inspection to fault diagnosis. Our choice of subjects aims at mitigating concerns from Di Penta et al. (2007), who argue that *"a subject group made up entirely of students might not adequately represent the intended user population"* (discussed in further detail in Section 6.5.2). Participation is on a voluntary basis, so the subjects can be assumed to be properly motivated. None of them have experience with Extravis.

In advance, we distinguished five fields of expertise that could strongly influence the individual performances. They represent variables that are to be controlled during the experiment, and concern knowledge of Java, Eclipse, reverse engineering, Checkstyle, and language technology (i.e., Checkstyle's domain). The subjects' levels of expertise in each of these fields were measured through a (subjective) a priori assessment: we used a five-point Likert scale, from 0 (*"no knowledge"*) to 4 (*"expert"*). In particular, we required minimum scores of 1 for Java and Eclipse (*"beginner"*), and a maximum score of 3 for Checkstyle (*"advanced"*). Appendix B provides a characterization of the subjects.

The assignments to the control and experimental group were conducted manually to evenly distribute the available knowledge. This is illustrated by Figure 6.1: in each group, the expertises are chosen to be as equal as possible, resulting in average expertises of 2.08 for the Eclipse group and 2.00 for the Ecl+Ext group.

6.3.4 Experimental procedure

The experiment is performed through eight sessions, most of which take place at Delft University of Technology. The sessions are conducted on workstations that have similar characteristics, i.e., at least Pentium 4 processors and more or less equal screen resolutions (1280x1024 or 1600x900).

Each session involves three subjects and features a short tutorial on Eclipse, highlighting the most common features. The experimental group is also given a 10 minute Extravis tutorial that involves a JHotDraw execution trace used in Chapters 4 and 5. All sessions are supervised, enabling the subjects to pose clarification questions, and preventing them from consulting others and from using alternative tools. The subjects are not familiar with the experimental goal.

The subjects are presented with a fully configured Eclipse that is readily usable, and are given access to the example input source file and Checkstyle configurations described in Section 6.3.2. The Ecl+Ext group is also provided with two Extravis instances, each visualizing one of the execution traces mentioned earlier. All subjects receive handouts that provide an introduction, Checkstyle outputs for the two aforementioned scenarios, the assignment, a debriefing questionnaire, and reference charts for both Eclipse and Extravis. The assignment is to complete the eight comprehension tasks within 90 minutes. The subjects are required to motivate their answers at all times. We purposely refrain from influencing how exactly the subjects should cope with the time limit: only when a subject exceeds the time limit is he or she told that finishing up is, in fact, allowed. The questionnaire asks for the subjects' opinions on such aspects as time pressure and task difficulty.

6.3.5 Variables & analysis

The independent variable in our experiment is the availability of Extravis during the tasks.

The first dependent variable is the *time spent* on each task, and is measured by having the subjects write down the current time when starting a new task. Since going back to earlier tasks is not allowed and the sessions are supervised, the time spent on each task is easily determined.

The second dependent variable is the *correctness* of the given answers. This is measured by applying our answer model on the subjects' answers, which specifies the required elements and the associated scores (between 0 and 4).

To test our hypotheses, we can choose from parametric and non-parametric tests. Whereas the former are more reliable, the latter are more robust: common examples include Student's t-test and the Mann-Whitney test, respectively. For the t-test to yield reliable results, two requirements must be met: the sample distributions must (1) be normal, and (2) have equal variances. These conditions can be tested using, e.g., the Kolmogorov-Smirnov test and Levene's test, respectively. These requirements are tested during our results analysis, upon which we decide whether to use the t-test or the more robust Mann-Whitney test.

Following our alternative hypotheses, we employ the one-tailed variant of each statistical test. For the time as well as the correctness variable we maintain a typical confidence level of 95% (α=0.05), which means that statistical significance is attained in cases where the p-value is found to be lower than 0.05. The statistical package that we use for our calculations is SPSS.

6.3.6 Pilot studies

Prior to the experimental sessions, we conducted two pilots to optimize several experimental parameters. These parameters included the number of tasks, their clarity, feasibility, and the time limit. The pilot for the control group was performed by one of the authors of this chapter, who had initially not been involved in the experimental design; the pilot for the experimental group was conducted by a colleague. Both would not take part in the actual experiment later on.

The results of the pilots have led to the removal of two tasks because the time limit was too strict. The removed tasks were already taken into account in Section 6.3.2. Furthermore, the studies led to the refinement of several tasks in order to make the questions clearer. Other than these ambiguities, the tasks were found to be sufficiently feasible in both the Eclipse and the Ecl+Ext pilot.

6.4 Results & Discussion

This section describes our interpretation of the results. We first discuss the time and correctness aspects in Section 6.4.1 and 6.4.2, and then take a closer look at the scores from a task perspective in Section 6.4.3.

Table 6.3 shows descriptive statistics of the measurements, aggregated over all tasks.[4]

Wohlin et al. (2000) suggest the removal of *outliers* in case of extraordinary situations, such as external events that are unlikely to reoccur. We found two outliers in our correctness data, but could identify no such circumstances.

As an important factor for both time and correctness, we note that one of the subjects gave up when his 90 minutes had elapsed with one more task to go, resulting in two missing data points in this experiment (i.e., the time spent by this subject on task T4.2 and the correctness of his answer). Seven others did finish, but only after the 90 minutes had expired: i.e., six subjects from

[4] The measurements themselves are available as a spreadsheet on `http://www.st.ewi.tudelft.nl/~cornel/results.xlsx`.

							one-tailed Student's t-test					one-tailed M.-W.	
Group	**Mean**	**Diff.**	**Min**	**Max**	**Median**	**Stdev.**	**K.-S. *Z***	**Lev. *F***	**df**	**t**	**p-val.**	**U**	**p-val.**
Time													
Eclipse	74.75		38	102	78	18.34	0.512						
Eclipse+Extravis	59.42	-20.51%	36	72	67	14.19	0.908	0.467	22	2.291	0.016	32.50	0.011
Correctness													
Eclipse	12.75		5	19	14	4.18	0.984						
Eclipse+Extravis	18.25	+43.14%	11	22	19	3.25	1.049	1.044	22	3.598	0.001	22.00	0.002

Table 6.3 Descriptive statistics of the experimental results.

the Eclipse group and one subject from the Ecl+Ext group spent between 97 and 124 minutes to complete all tasks.

For this reason, we shall *disregard the last two tasks* in our quantitative analyses: not taking tasks T4.1 and T4.2 into account, only two out of the 24 subjects still exceeded the time limit (by 7 and 12 minutes, respectively), which is acceptable. At the same time, this strongly reduces any ceiling effects in our data that may have resulted from the increasing time pressure near the end of the assignment. The remaining six tasks still cover all of Pacione's nine activities (Table 6.2).

6.4.1 Time results

We start off by testing null hypothesis $H1_0$, which states that the availability of Extravis does not impact the time that is needed to complete typical comprehension tasks.

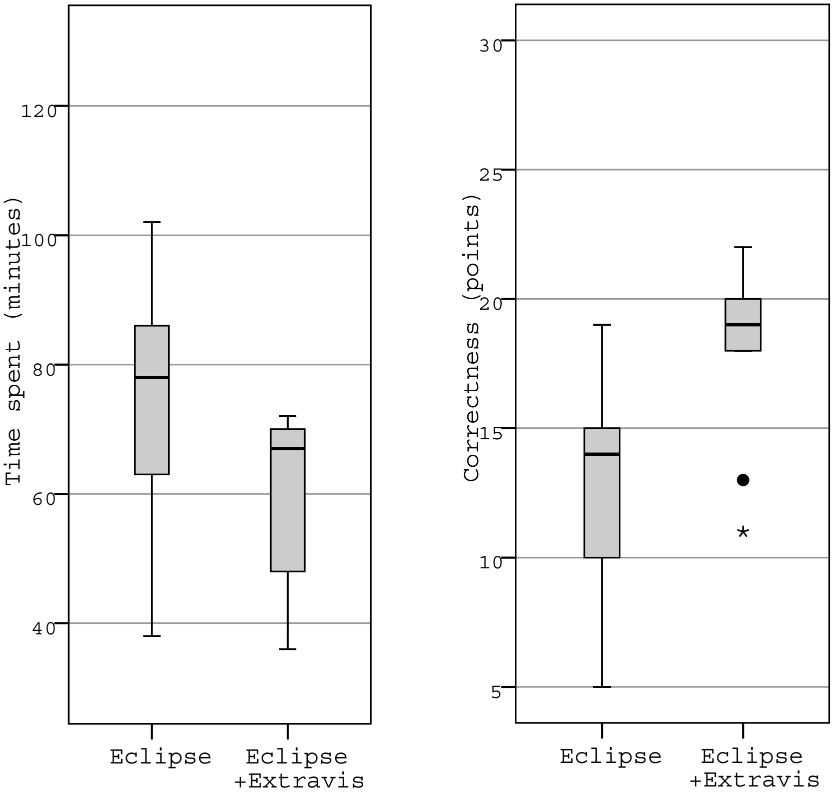
Figure 6.2(a) shows a box plot for the total time that the subjects spent on the first six tasks. Table 6.3 indicates that on average the Ecl+Ext group required 20.51% less time.

The Kolmogorov-Smirnov and Levene tests succeeded for the timing data, which means that Student's t-test may be used to test $H1_0$. As shown in Table 6.3, the t-test yields a statistically significant result. The average time spent by the Ecl+Ext group was clearly lower and the p-value 0.016 is smaller than 0.05, which means that $H1_0$ can be rejected in favor of the alternative hypothesis H1, which states that the availability of Extravis reduces the time that is needed to complete typical comprehension tasks. The non-parametric Mann-Whitney test confirms these findings.

The lower time requirements for the Extravis users could be attributed to several factors. First, all information offered by Extravis is shown on a single screen, which negates the need for scrolling. In particular, the overview of the entire system's structure saves much time in comparison to conventional environments, in which typically multiple files have to be studied at once. Second, the need to imagine how certain functionalities or interactions work at runtime represents a substantial cognitive load on the part of the user. This is alleviated by trace analysis and visualization tools, which show the actual runtime behavior. Examples of these assumptions are discussed in Section 6.4.3.

On the other hand, several factors may have had a negative impact on the the time requirements of Extravis users. For example, the fact that Extravis is a standalone tool means that context switching is necessary, which may yield a certain amount of overhead on the part of the user. This could be solved by integrating the trace visualization technique into Eclipse (or other IDEs), with the additional benefit that the tool could provide direct links to Eclipse's source code browser. However, it should be noted that Extravis would still require a substantial amount of screen real estate to be used effectively.

Another potential factor that hindered the time performance of the Ecl+Ext group is that these subjects may not have been sufficiently familiar with Ex-



(a)

(b)

Figure 6.2 Box plots for time spent and correctness.

travis' features, and were therefore faced with a time-consuming learning curve. This is partly supported by the debriefing questionnaire, which indicates that four out of the 12 subjects found the tutorial too short. A more elaborate tutorial on the use of the tool could help alleviate this issue.

6.4.2 Correctness results

We now test null hypothesis $H2_0$, which states that the availability of Extravis does not impact the correctness of answers given during typical comprehension tasks.

Figure 6.2(b) shows a box plot for the scores that were obtained by the subjects on the first six tasks. Note that we consider overall scores rather than scores per task (which are left to Section 6.4.3). The box plot shows that the difference in terms of correctness is even more explicit than for the timing aspect. The answers given by the Ecl+Ext subjects were 43.14% more accurate

(Table 6.3), averaging 18.25 out of 24 points compared to 12.75 points for the Eclipse group.

Similar to the timing data, the requirements for the use of the parametric t-test were met. Table 6.3 therefore shows the results for Student's t-test. At 0.001, the p-value is very low and implies statistical significance. Since the difference is clearly in favor of the Ecl+Ext group, it follows that hypothesis $H2_0$ can be easily rejected in favor of our alternative hypothesis H2, which states that the availability of Extravis increases the correctness of answers given during typical comprehension tasks. The Mann-Whitney test confirms our findings.

We attribute the added value of Extravis to correctness to several factors. First, the inherent preciseness of dynamic analysis could have played a crucial role: the fact that Extravis shows the actual objects involved in each call makes the interactions easier to understand. Section 6.4.3 discusses this in more detail through an example task.

Second, the results of the debriefing questionnaire (Table 6.4) show that the Ecl+Ext group used Extravis quite often: the subjects estimate the percentage of time they spent in Extravis at 60% on average. While in itself this is meaningless, we also observe through the questionnaire that on average, Extravis was used on 6.8 of the 8 tasks, and that on average the tool proved useful in 5.1 of those tasks (75%). This is a strong indication that the Ecl+Ext subjects generally did not experience a resistance to using Extravis (resulting from, e.g., a poor understanding of the tool) *and* were quite successful in their attempts.

The latter assumption is further reinforced by the Ecl+Ext subjects' opinions on the speed and responsiveness of the tool, which averaged a score of 1.4 on a scale of 0-2, which is between *"pretty OK: occasionally had to wait for information"* and *"very quickly: the information was shown instantly"*. Furthermore, all 24 subjects turned out to be quite familiar with dynamic analysis: in the questionnaire they indicated an average knowledge level of 2.4 on a scale of 0-4 on this topic, which is between *"I'm familiar with it and can name one or two benefits"* and *"I know it quite well and performed it once or twice"*.

Note that similar to a related study (Quante, 2008), we could not identify a correlation between the subjects' performances and their (subjective) expertise levels.

6.4.3 Individual task scores

To determine if there are certain types of comprehension tasks that benefit most from the use of Extravis, we examine the performances per task in more detail. Figure 6.3 shows the average scores and time spent by each group from a task perspective. While we focus primarily on correctness, timing data is also considered where appropriate.

The groups scored equally well on tasks T1 and T3.1 and required similar amounts of time. According to the motivations of their answers, for task T1 the Extravis users mostly used the massive sequence view for visual phase

	Eclipse		Eclipse+Extravis	
	Mean	**Stdev.**	**Mean**	**Stdev.**
Misc.				
Time pressure (0-4)	2.17	1.27	2.08	0.51
Dynamic analysis expertise (0-4)	2.33	1.15	2.50	1.24
Task difficulty (0-4)				
T1	1.00	0.74	1.58	0.67
T2.1	2.67	1.23	1.08	0.67
T2.2	2.50	1.24	1.50	0.90
T3.1	2.08	0.90	2.25	0.75
T3.2	2.08	0.90	1.50	0.80
T3.3	1.92	0.90	1.50	1.00
T4.1	2.50	0.67	2.83	0.83
T4.2	1.58	1.00	1.64	1.12
Average	2.04		1.74	
Use of Extravis				
No. of tool features used			6.42	2.68
No. of tasks involved (0-8)			6.75	1.14
No. of tasks success (0-8)			5.08	1.31
% of time spent in tool (est.)			60.00	26.71
Perceived tool responsiveness (0-2)			1.42	0.51

Table 6.4 Results of the debriefing questionnaire. Ratings shown are subjective.

detection, whereas the Eclipse group typically studied the `main()` method. The results of the latter approach were generally a little less accurate, because such important phases as the building and parsing of an AST are not directly visible in `main()`. As for task T3.1, both groups often missed the explicit destruction of each check at the end of execution, which is not easily observed in Eclipse nor in Extravis.

The only task on which the Ecl+Ext group was outperformed is T4.1, in terms of time as well as correctness. The Eclipse group rated the difficulty of this task at 2.5, which is between *"intermediate"* and *"difficult"*, whereas Extravis users rated the difficulty of this task at 2.8, leaning toward *"difficult"*. An important reason might be that Extravis users did not know exactly *what to look for* in the trace, whereas most Eclipse users used one of the checks as a starting point and followed the violation propagation process from there. The latter approach is typically faster: the availability of Extravis may have been a distraction rather than an added value in this case.

The Extravis users scored significantly higher on five tasks: the differences for tasks T2.1, T2.2, T3.2, T3.3, and T4.2 ranked between 0.8 and 1.6 points.

The rather decent results from the Ecl+Ext group on tasks T2.1 and T2.2 are presumably explained by Extravis' circular view, from which *all* classes and their interrelationships can be directly interpreted. In T2.1, the Eclipse group mostly went looking for utility-like classes, while in T2.2 a common approach was to search for specific imports. The former task required quite

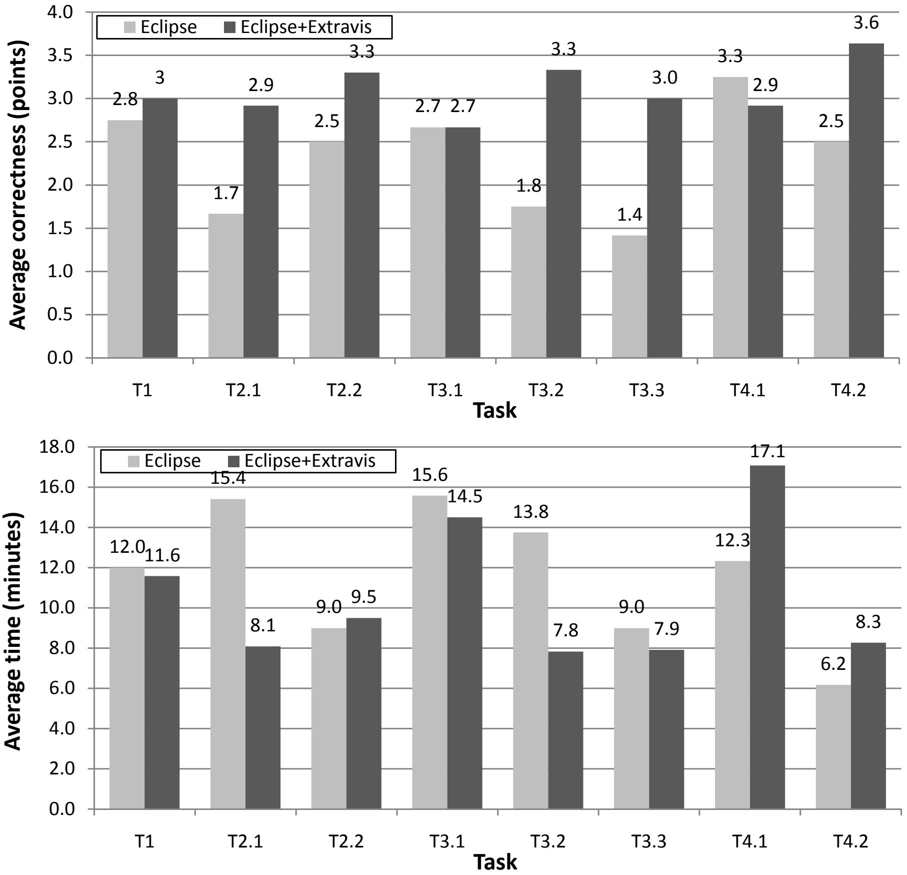


Figure 6.3 Averages per task.

some exploration for Eclipse users and was therefore time-consuming, and the approach does not necessarily yield optimal results. The latter task required less time, presumably because a more specific search was possible.

Task T3.2 involved *inheritance*: the fact that the check at hand is an extension of a superclass that is an extension in itself, forced the Eclipse group to distribute its focus across each and every class in the check's type hierarchy. Extravis users often selected check X and class Y in the tool, which highlights all mutual interactions. As evidenced by Figure 6.3, the latter approach is both faster and more accurate. In task T3.3, the Extravis users could follow the same routine whereas in Eclipse the required elements are easily missed.

In task T4.2, the Ecl+Ext group therefore mostly searched the execution traces for communication between check X and the violation container class. The Eclipse group had several choices. A few subjects tried to understand the check and apply this knowledge on the given input source file; others tried to relate the check's typical warning message (once it was determined) to the given example outputs; yet others used the debugger, e.g., by inserting breakpoints or print statements. With the exception of debugging, most of the latter approaches are quite time-consuming, if successful at all. Still, we observe no large difference in time spent: the fact that six members of the Eclipse group had already exceeded the time limit at this point may have reduced the amount of effort invested in this task.

6.5 Threats to Validity

This section discusses the validity threats in our experiment and the manners in which we have addressed them. We maintain the common distinction between internal validity, which refers to the cause-effect inferences made during the analysis, and external validity, which concerns the generalizability of the results to different contexts.

6.5.1 Internal validity

Subjects

There exist several internal validity threats that relate to the subjects used in this experiment. First of all, the subjects may not have been sufficiently competent. We have reduced this threat through the a priori assessment of the subjects' competence in five relevant fields, which pointed out that all subjects had at least an elementary knowledge of Eclipse and no expert knowledge of Checkstyle.

Second, their knowledge may not have been fairly distributed across the control group and experimental group. This threat was alleviated by grouping the subjects such that their expertise was evenly distributed across the groups.

Third, the subjects may not have been properly motivated, or may have had too much knowledge of the experimental goal. The former threat is mitigated by the fact that they all participated on a voluntary basis; as for the latter, the subjects were not familiar with the actual research questions or hypotheses (although they may have guessed).

Tasks

The comprehension tasks were designed by the authors of this chapter, and therefore may have been biased towards Extravis (as this tool was also designed by several of the authors). To avoid this threat, we have involved an established task framework (Pacione et al., 2004) to ensure that many aspects of typical comprehension contexts are covered: as a result, the tasks concerned both global and detailed knowledge, and both static and dynamic aspects.

Another threat related to the tasks is that they may have been too difficult. We refute this possibility on the basis of the correctness results, which show that maximum scores were occasionally awarded in both groups for all but one task (T3.1), which in the Eclipse group often yielded 3 points but never 4. However, the average score for this task was a decent 2.67 (stdev. 0.7) in both groups. Our point of view is further reinforced by the subjects' opinions on the task difficulties: the task they found hardest (T4.1) yielded good average scores, being 3.25 for the Eclipse group and 2.92 for the Ecl+Ext group.

Also related to the tasks is the possibility that the subjects' answers were graded incorrectly. This threat is often overlooked in the literature, but was reduced in our experiment by creating concept answers in advance and by having Checkstyle's lead developer review and refine them. This resulted in an answer model that clearly states the required elements (and corresponding points) for each task. Furthermore, to verify the soundness of the reviewing process, the first two authors of this chapter independently reviewed the answers of five random subjects: on each of the five occasions the difference was no higher than one point (out of the maximum of 32 points).

Miscellaneous

The results may have been influenced by time constraints that were too loose or too strict. We have attempted to circumvent this threat by performing two pilot studies, which led to the removal of two tasks. Still, not all subjects finished the tasks in time; however, the average time pressure (as indicated by the subjects in the debriefing questionnaire) was found to be 2.17 in the Eclipse group and 2.08 in the Ecl+Ext group on a scale of 0-5, which roughly corresponds to only a *"fair amount of time pressure"*. Furthermore, in our results analysis we have disregarded the last two tasks, upon which only two out of the 24 subjects still exceeded the time limit.

Furthermore, our statistical analysis may not be completely accurate due to the missing data points that we mentioned in Section 6.4. This concerned only one subject, who did not finish task T4.2. Fortunately, the effect of the two missing timing and correctness data points on our calculations is negligible: had the subject finished the task, his total time spent and average score could have been higher, but this would only have affected the analysis of all eight tasks whereas our focus has been on the first six.

Lastly, it could be suggested that Eclipse is more powerful if additional plugins are used. However, as evidenced by the results of the debriefing questionnaire, only two subjects named specific plugins that would have made the tasks easier, and these related to only two of the eight tasks. We therefore expect that additional plugins would not have had a significant impact.

6.5.2 External validity

The generalizability of our results could be hampered by the limited representativeness of the subjects, the tasks, and Checkstyle as a subject system.

Concerning the subjects, the use of professional developers rather than (mainly) Ph.D. candidates and M.Sc. students could have yielded different results. Unfortunately, motivating people from industry to sacrifice two hours of their precious time is quite difficult. Nevertheless, against the background of related studies that often employ students, we assume the expertise levels of our 24 subjects to be relatively high. This assumption is reinforced by the (subjective) a priori assessment, in which the subjects rated themselves as being *"advanced"* with Java (avg. 3.08, stdev. 0.6), and *"regular"* at using Eclipse (avg. 2.29, stdev. 0.8). We acknowledge that our subjects' knowledge of dynamic analysis may have been greater than in industry, averaging 2.42 (Table 6.4).

Another external validity threat concerns the comprehension tasks, which may not reflect real maintenance situations. This threat is largely neutralized by our reliance on Pacione's framework (Pacione et al., 2004), that is based on activities often found in software visualization and comprehension evaluation literature. Furthermore, the tasks concerned open questions, which obviously approximate real life contexts better than do multiple choice questions.

Finally, the use of a different subject system (or additional runs) may have yielded different or more reliable results. Checkstyle was chosen on the basis of several important criteria; finding an additional system of appropriate size and of which the experimenters have sufficient knowledge is not trivial. Moreover, an additional case (or additional run) imposes twice the burden on the subjects or requires more of them. While this may be feasible in case the groups consist exclusively of students, it is not realistic in case of Ph.D. candidates (or professional developers) because they often have little time to spare, if they are available at all.

6.6 Related Work

To the best of our knowledge, there exist no earlier studies in the literature that offer quantitative evidence of the added value of trace visualization techniques for program comprehension. We therefore describe the experiments that are most closely related to our topic.

In a recent article, Bennett et al. (2008) summarized the state of the art in tool features for dynamic sequence diagram reconstruction. Based on this survey, they proposed a new tool that implemented these features. Rather than measuring its added value, they sought to characterize the *manner* in which the tool is used in practice. To this end, they had six subjects perform a series of comprehension tasks, and measured when and how the tool features were used. Among their findings was that tool features are not often formally evaluated in literature, and that heavily used tool features may indicate confusion among the users. Another important observation was that much time was spent on *scrolling*, which supports our hypothesis that Extravis saves time as it shows all information on a single screen.

Quante (2008) performed a controlled experiment to assess the benefits of Dynamic Object Process Graphs (DOPGs) for program comprehension. While these graphs are built from execution traces, they do not actually *visualize* entire traces. The experiment involved 25 students and a series of feature location[5] tasks for two subject systems. The use of DOPGs by his experimental

[5] Feature location is a reverse engineering activity that concerns the establishment of relations between concepts and source code.

group led to a significant decrease in time and a significant increase in correctness in case of the first system; however, the differences in case of the second system were *not* statistically significant. This suggests that evaluations on additional systems are also desirable for Extravis and should be considered as future work. Also of interest is that the latter subject system was four times smaller than the former, but had three DOPGs associated with it instead of one. This may have resulted in an information overload on the part of the user, once more suggesting that users are best served by as little information as possible.

Hamou-Lhadj and Lethbridge (2006) proposed the notion of summarized traces, which provide an abstraction of large traces to grasp a program's main behavioral aspects. The paper presents quantitative results with regard to the effectiveness of the algorithm. The traces were also qualitatively evaluated through a questionnaire among software developers. The actual usefulness *in practice*, i.e., its added value to conventional techniques in actual program comprehension contexts, was not measured.

6.7 Conclusion

In this chapter, we have reported on a controlled experiment that was aimed at the quantitative evaluation of Extravis, our tool for execution trace visualization. We designed eight typical tasks aimed at gaining an understanding of a well-known code validation program, and measured the performances of a control group (using the Eclipse IDE) and an experimental group (using both Eclipse and Extravis) in terms of correctness and time spent.

The results clearly illustrate Extravis' usefulness for program comprehension. With respect to time, the added value of Extravis was found to be statistically significant: on average, the Extravis users spent 21% less time on the given tasks. In terms of correctness, the results turned out even more convincing: Extravis' added value was again statistically significant, with the Extravis users scoring 43% more points on average. These results testify to Extravis' benefits compared to conventional tools: in this case, the Eclipse IDE.

To find out which types of tasks are best suited for Extravis or for trace visualization in general, we looked in more detail at the group performances per task. While inferences drawn from one experiment and only eight tasks cannot be conclusive, the experimental results do provide a first indication as to Extravis' strengths. First, questions that require insight into a system's structural relations are solved relatively easily due to Extravis' circular view, as it shows *all* of the system's structural entities and their call relationships on one single screen. Second, tasks that involve inheritance seem to benefit greatly from the fact that Extravis shows the actual objects involved in each interaction. Third, questions that require a user to envision a system's runtime behavior are clearly easier to tackle when traces are provided (in a comprehensible manner). The latter two observations presumably hold for most trace visualization techniques.

This chapter demonstrates the potential of trace visualization for program comprehension, and paves the way for other researchers to conduct similar experiments. The work in this chapter makes the following contributions:

- The reusable design of a controlled experiment for the quantitative evaluation of trace visualization techniques for program comprehension.
- The execution of this experiment on a group of 24 representative subjects, demonstrating a 21% decrease in time effort and a 43% increase in correctness.
- A first indication as to the types of tasks for which Extravis, and trace visualization in general, are best suited.

Directions for future work include replications of the experiment on different subject systems. Furthermore, we seek collaborations with researchers to evaluate other existing trace visualization techniques, i.e., to assess and compare their added values for program comprehension.

Acknowledgments

Many thanks to the 24 volunteers, without whom this experiment could not have been conducted, and to Checkstyle's lead developer, Oliver Burn, who assisted in the design of our task review protocol.

Conclusion

Chapter 7

Program comprehension is a challenging task for developers who are faced with maintaining a software system. In particular, the lack of sufficient knowledge of a system's internals means that a time-consuming examination is required in order to properly understand it. The study of a system's runtime behavior, known as dynamic analysis, has the potential to facilitate this process. Unfortunately, most of the literature on this topic does not provide thorough empirical validations.
In this thesis, we sought to raise the bar with respect to the evaluation of techniques for program comprehension through dynamic analysis. We systematically surveyed the existing work, defined reusable frameworks, conducted extensive case studies, provided publicly available tools, and quantitatively measured the added value of one of these tools compared to a traditional IDE. Our contributions demonstrate the benefits of dynamic analysis for program comprehension and enable fellow researchers to better evaluate their own solutions in this field.

7.1 Summary of Contributions

Each of the core chapters in this thesis offers several contributions, which we summarize as follows.

- A systematic survey of the literature on program comprehension through dynamic analysis, with a detailed characterization of 172 articles and a reusable attribute framework as a result.
- A thorough analysis of the survey results, resulting in the identification of underemphasized topics of which we motivated the importance in the near future.
- A novel approach toward the understanding of program behavior, that involves tracing and abstracting test case executions and their visualization as UML sequence diagrams.
- The implementation of this approach into a publicly available tool, and its validation through two case studies involving different subject systems and program comprehension tasks.
- A methodology for the assessment of execution trace reduction techniques, enabling a side-by-side comparison of both new techniques and existing solutions from the literature.
- The application of this methodology on a set of four reduction techniques from the literature, using a test set of seven large traces (made available online).

- A novel approach toward the understanding of program behavior using Extravis, a tool for the advanced visualization of large execution traces (made available online).

- The evaluation of this approach by means of three different case studies, each involving different subject systems and distinct program comprehension activities.

- The reusable design of a controlled experiment aimed at assessing the benefits of trace visualization during typical program comprehension tasks (fully documented in this thesis).

- The execution of this experiment on Extravis with a group of 24 representative subjects, demonstrating a 21% decrease in time effort and a 43% increase in correctness.

7.2 Research Questions

Research Question 1

> *How can we structure the available literature on program comprehension and dynamic analysis, and which topics need additional consideration the near future?*

We addressed our first research question by conducting a systematic literature survey (Chapter 2). We started off with the decomposition of typical program comprehension articles into four facets: activity, target, method, and evaluation, each representing an important aspect of such papers on the basis of which they can be distinguished and compared. Next, we defined explicit selection criteria, and systematically selected 127 out of 4,795 articles from 14 relevant venues in the period of July 1999 to June 2008. The references in these articles were checked, resulting in another 45 papers (from 17 additional venues) that were subsequently included in the selection.

Through a detailed reading of this research body, we derived an attribute framework that lists the different attributes that were distinguished in each facet (Table 2.2). This framework was used to characterize all 172 articles in a structured fashion, and to summarize their contributions into 110 articles. Tables A.2 through A.4 show the characterization of the summarized papers, whereas the full research body has been made available online.

The resulting overview is useful as a reference for researchers in the field of program comprehension through dynamic analysis, and helps them identify related work and new research opportunities in their respective subfields.

Our own study of the results focused on the identification of attributes that have received relatively little attention in the literature (Figure 2.3). Among the findings was the apparent lack of comparisons and benchmarks in most subfields, and the observation that very few articles offer publicly available tools, which hinders the use of existing solutions in current developments and the adoption of such tools in industry. We also expressed our concerns about the lack of controlled experiments, which we feel are crucial in the field of program comprehension. Finally, we identified four types of target applications that have been underemphasized in literature, being web applications, distributed systems, multithreaded applications, and legacy systems. We stressed the importance of the above aspects, discussed the issues involved, and offered a series of potential solutions and recommendations.

Research Question 2

> *Given the excessive amounts of data involved in dynamic analysis, are abstraction techniques sufficient to render traditional visualizations such as UML sequence diagrams useful for program comprehension?*

In our first attempt to support software understanding through the visualization of dynamically gathered data, we designed a framework for the reconstruction of UML sequence diagrams from test case executions in Chapter 3. The approach involved explicit metamodels for execution traces and sequence diagrams. Since the traces are often too large to visualize directly, we identified a series of trace reduction techniques, of which several were implemented in Sdr, our initial tool implementation. The prototype was evaluated on a small subject system that involved two concrete maintenance tasks. In a second iteration, we proposed JRET, a more mature tool implementation that features several improvements over its predecessor, and evaluated it on a medium-scale open source application. While the use of test cases as execution scenarios and their visualization as sequence diagrams proved useful as a starting point for program comprehension, we opted for a more thorough investigation of trace reduction techniques because their effectiveness and applicability in general remained unclear.

The goal of Chapter 4, therefore, was the evaluation of such techniques. To enable side-by-side comparisons of the numerous techniques offered in literature, we proposed an assessment methodology that distinguishes six steps: context, criteria, metrics, test set, application, and interpretation. The use of our methodology was demonstrated through an experiment that involved a test set of seven large traces – three of which were larger than one million events in size – and the implementation of four different reduction techniques from the literature (for which no publicly available tools were available). These included subsequence summarization, stack depth limitation, language-based filterings, and sampling. Our example experiment showed how a fair comparison of existing techniques can be achieved as long as similar contexts, criteria, and test sets are used.

Research Question 3

> *How can program comprehension through dynamic analysis be effectively supported by more advanced visualizations?*

Another manner in which we sought to tackle the scalability issues associated with dynamic analysis was the design and evaluation of more advanced visualizations, i.e., techniques not based on traditional visualizations such as trees and UML. To this end, we employed two novel views in the context of large execution traces. The circular bundle view, originally used for the visualization of static relationships (Holten, 2006), hierarchically projects a program's structural entities on a circle and shows their interrelationships in a bundled fashion (Figure 5.1 on page 94). The massive sequence view provides a navigable overview of a trace by chronologically visualizing each event as a colored line (Figure 5.8 on page 108). The views are linked and fully interactive, fit on a single screen, and are implemented in a tool called Extravis.

We initially evaluated the tool through a series of three case studies. The studies involved different subject systems (one industrial and two open source) and distinct program comprehension contexts, including trace exploration, feature location, and top-down program comprehension. The results provided a strong indication as to Extravis' benefits, the main advantages being its optimal use of screen real estate and the easy insight into a program's structure. However, we did hypothesize that the relationships in the circular view could be difficult to grasp: indeed, whereas the use of traditional visualizations such as UML is widespread, more advanced techniques warrant extensive validations to assess their understandability and usefulness (Storey, 2005).

For this reason, we conducted a controlled experiment to measure Extravis' added value to an existing solution, the Eclipse IDE (Chapter 6). We designed eight typical tasks aimed at gaining an understanding of a representative subject system, and measured how a control group (using the Eclipse IDE) and an experimental group (using both Eclipse and Extravis) performed in terms of correctness and time spent. The results were statistically significant in both regards, showing a 21% decrease in time and a 43% increase in correctness for the latter group. The design and execution of this experiment have demonstrated the potential of trace visualization for program comprehension, and have paved the way for other researchers to conduct similar experiments.

Research Question 4

> *How can we evaluate techniques for program comprehension and dynamic analysis, and how can we compare their added value to existing solutions?*

One of our main objectives was to raise the bar with respect to the manner in which dynamic analysis techniques for program comprehension should be evaluated. To this end, we have put a strong emphasis on the evaluation quality in our experiments. As described below, this objective crosscuts all of the core chapters in this thesis.

Our literature survey in Chapter 2 involved a systematic approach with an emphasis on reproducibility. The selection of relevant articles through explicit selection criteria and the derivation of a reusable attribute framework enables fellow researchers to replicate our findings and compare existing work. Moreover, both of these tasks involved pilot studies to ensure their soundness. The resulting literature overview allows researchers to easily identify relevant articles in their respective subfields.

The reconstruction of UML sequence diagrams in Chapter 3 was initially evaluated through a case study on a relatively small program. Later on, we proposed a more robust tool to perform another case study on a more representative system. Both studies revolved around comprehension tasks that stem from realistic software maintenance problems. The tool, called JRET, is implemented as an Eclipse plugin and is publicly available online for fellow researchers to download and experiment with.

In Chapter 4 we sought to address the lack of (comparative) evaluations in the literature on trace reduction techniques. We introduced an assessment methodology, the first of its kind, that enables a fair side-by-side comparison of such techniques. The methodology is fully documented in Chapter 4. Furthermore, the public availability of the traces and reduction technique implementations involved in our example experiment (1) ensures the reproducibility of our findings, and (2) supports fellow researchers in performing similar assessments of different reduction techniques.

Chapter 5 presented two advanced execution trace views, which were implemented in Extravis. The tool was thoroughly evaluated through a series of three extensive case studies, each involving distinct program comprehension contexts and representative subject systems of which one was industrial. Furthermore, the tool is documented and publicly available online to enable its involvement in similar experiments.

Finally, the rather exotic nature of Extravis' visualizations motivated us to measure its cognitive impact on humans, and led to the design and successful execution of a reusable controlled experiment, involving eight comprehension tasks based on a framework in the literature. Chapter 6 and Appendix B contain all the necessary information for the replication our experiment, and allow for similar experiments with different visualization techniques.

7.3 Evaluation

Our study of the literature, combined with our own experiences, have led to us to believe that dynamic analysis techniques for program comprehension should be more rigorously evaluated than is typically the case: Indeed, the scientific progress in this community could benefit greatly from improvements to the evaluation quality. With respect to the manner in which new techniques should be validated, we envision the following process:

- **Multiple case studies.** The proposition of a new dynamic analysis technique should be accompanied by multiple case studies. Depending on the activities it aims to support, these studies should involve different program comprehension contexts. To ensure a certain degree of generalizability, the studies should concern multiple subject systems that are representative in terms of the (typically large) amounts of data that are to be processed in dynamic analysis. The results provide a first indication as to the technique's benefits and limitations.

- **Comparison to existing solutions.** After a new dynamic analysis technique has been sufficiently evaluated in itself through anecdotal evidence, the next step should be to compare its performance to existing solutions. Such a comparison allows the authors to reason about the technique's added value for the program comprehension community. In particular, in a program comprehension context the following should be taken into account:
    - Most approaches in program comprehension require controlled experiments for their quantitative evaluation. This especially holds for dynamic analysis techniques, as they often involve an information overload that humans cannot cope with directly. The design of a controlled experiment can be elaborate, but can be reused if it is sufficiently documented and generic in nature.
    - Comparisons require existing solutions to have publicly available tools associated with them, as this saves researchers a great deal of reimplementation effort and prevents interpretation mistakes (e.g., in case of complex algorithms). In certain cases, such as advanced visualizations, the duplication of an existing technique may not even be impossible.
- **Industrial involvement.** Ideally, the final stadium of the validation of a program comprehension technique should involve a case study in an industrial context. Examples include evaluations on industrial software, or the involvement of actual developers in (controlled) experiments. This ensures that the added value of novel techniques can be expressed in industrial terms, which could support the adoption of such tools in industry.

In light of the contributions in this thesis, we believe our work to be a significant step toward such a rigorous evaluation process. However, we acknowledge that several aspects have not yet received the attention they deserve.

First, while we feel that our Extravis tool has been sufficiently validated, the evaluation of our efforts on UML sequence diagram reconstruction needs more work. Two case studies involving different tasks and subject systems were conducted with encouraging results, but the usefulness of the approach in practice remains unclear, and we do not yet know how it performs relative to other solutions. On the other hand, the fact that our approach was implemented in a publicly available tool means that the research community can involve it in comparative evaluations in the near future.

Second, our work does not exhibit a strong link with industry: we had only one industrial application at our disposal, which we used as a subject system in Chapter 5 and as an execution trace in Chapter 4. While our project involved several industrial partners, having them actively participate in the research turned out to be difficult. We can think of several potential reasons, some technical (e.g., difficulties in instrumenting and tracing industrial software, as discussed in Section 2.6.4) and some motivational (e.g., lack of time, focus, or interest). Possible solutions to the technical issues were proposed in Chapter 2, but are subject to future investigation.

7.4 Future Work

With the evolution of software systems and their regular modification in practice, program comprehension has become an increasingly important activity in software development. The literature and this thesis have shown dynamic analysis to be a powerful asset during such activities, as the extension of static (source code) information with behavioral knowledge provides developers with a very complete picture of the software they are working on. As systems grow ever larger and more complex in practice, we expect the need for techniques and tools to effectively support program comprehension activities to grow ever more in the near future.

Program comprehension through dynamic analysis is ultimately about conveying (large amounts of) information to users, and should therefore be evaluated on actual human subjects. It is for this reason that user studies and controlled experiments should be an important focus in the near future to advance the state of the practice. As an example, we believe that the experimental design proposed in Chapter 6 can be used to measure the added value of any tool that aims to support program comprehension.

Since it is often difficult to obtain sufficient numbers of suitable subjects for such experiments, it must be investigated how potential test subjects can be identified and properly motivated. Ideally, such groups should not be composed exclusively of students but also of practitioners (Di Penta et al., 2007). As an example solution, we suggest that a separate budget be defined in research projects on program comprehension to enable (financial) compensations for test subjects, as is already customary in the fields of medicine and psychology. Moreover, the participation of actual developers from industry (who actually constitute the target audience) could be established in this manner.

Another important focus in the near future should be the adoption of dynamic analysis tools in industry. While the research community has been working on dynamic analysis for several decades, they are seldomly used in practice. It could be argued that software developers are generally unfamiliar with the benefits; however, runtime debugging (e.g., using breakpoints) is essentially a form of dynamic analysis and is often used. Future research should be aimed at identifying new ways to stimulate the use of dynamic analysis in practice: For example, we hypothesize that additional (successful) controlled experiments may help convince industry of the added value of dynamic analysis tools.

Finally, several activities and target application types have received relatively little attention in the literature thus far, in spite of their importance in the near future (Section 2.6). These activities are architecture recovery, providing valuable views of applications on the architectural level; and surveys, which provide overviews of the state of the art. The target applications that deserve more attention are (1) web applications, because they have grown increasingly difficult to analyze, particularly using static methods; (2) distributed systems, which are receiving increasing interest from industry due to the advent of service-oriented architectures; (3) multithreaded applications, because multicore CPUs are becoming mainstream; and (4) legacy systems, which are typically in need of reverse engineering efforts because their internals are often poorly understood. We expect these aspects to play an important role in the near future, and therefore opt for additional efforts into their investigation.

Literature Survey: Coincidence Measurements & Article Characterization

Appendix A

This appendix presents the results of the attribute coincidence experiment in Chapter 2. The ensuing pages provide an extensive characterization of the 110 summarized articles.

Facet	Attribute #1	Attribute #2	Fraction
Activity	general	views	0.72
	design/arch.	views	0.71
	survey	views	0.60
	behavior	views	0.52
	views	general	0.50
	trace	views	0.50
	survey	general	0.50
Target	threads	oo	0.68
Method	compr./summ.	vis. (std.)	0.60
	fca	filtering	0.50
	fca	mult. traces	0.60
	fca	static	0.70
	fca	vis. (std.)	0.90
	filtering	vis. (std.)	0.58
	heuristics	metrics	0.61
	mult. traces	filtering	0.54
	mult. traces	metrics	0.57
	online	compr./summ.	0.62
	online	static	0.62
	online	vis. (adv.)	0.62
	patt. det.	vis. (std.)	0.60
	querying	filtering	0.54
	querying	static	0.59
	querying	vis. (std.)	0.59
	slicing	static	0.88
	slicing	vis. (std.)	0.76
	static	vis. (std.)	0.60
	vis. (adv.)	filtering	0.54
	vis. (std.)	static	0.54
Evaluation	comparison	quantitative	0.82
	comparison	regular	0.82
	human subj.	regular	0.62
	industrial	regular	0.53
	quantitative	regular	0.95

Table A.1 Attribute coincidence measurements.

			activity							target						method													evaluation						
	add'l articles	tool avail.	survey	design/arch.	views	features	trace	behavior	general	legacy	procedural	oo	threads	web	distributed	vis. (std.)	vis. (adv.)	slicing	filtering	metrics	static	patt. det.	compr./summ.	heuristics	fca	querying	online	mult. traces	preliminary	regular	industrial	comparison	human subj.	quantitative	unknown/none
(Antoniol and Di Penta, 2004)			.	.	.	.	o	.	.	.	o	o	.	.	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	o	.	.	.	o	.
(Antoniol et al., 2004)			.	o	o	.	.	.	.	.	.	.	.	o	.	o	.	.	o	.	o	.	.	.	.	.	.	.	.	o	.	.	.	o	.
(Antoniol and Guéhéneuc, 2006)	1		o	.	.	o	.	.	.	.	o	o	o	.	.	.	.	.	o	.	o	.	.	.	.	.	.	o	.	o	.	o	.	o	.
(Ball, 1996)			.	.	o	.	.	.	o	.	o	.	.	.	.	.	o	o	.	o	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(Ball, 1999)			.	.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.	o	o	.	.	.	o	.	.	.	o	.	.	.	.	.	.
(Bennett et al., 2008)			o	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	.	o	o	.	.	.	.	.	.	o	.	.	o	.	.
(Bojic and Velasevic, 2000)			.	o	o	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	.	.	o	.	o	.	.	.	o	.	.	.	.	.	.
(Briand et al., 2006)	2		o	.	o	.	.	.	o	.	.	o	o	.	o	o	.	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	o	.
(Chan et al., 2003)		o	.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	o	.	.	.	.	.
(Cook and Du, 2005)	2		.	.	.	.	.	o	.	.	o	.	o	.	.	o	.	.	.	o	.	o	.	o	.	.	.	.	.	o	.	.	.	.	.
(Cornelissen et al., 2007b)			.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.
(Cornelissen et al., 2008b)	1	o	.	.	o	o	o	.	o	.	.	o	.	.	.	o	o	.	o	.	o	.	o	.	.	.	.	.	.	o	o	.	.	.	.
(Cornelissen et al., 2008a)			o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	o	o	.	o	o	o	.	.	.	o	.	o	.	.	.	o	.
(Dalton and Hallstrom, 2008)			.	.	o	.	.	.	o	.	.	.	.	.	.	o	.	.	o	.	o	.	.	.	.	.	.	.	o	.	.	.	.	o	.
(Deprez and Lakhotia, 2000)			.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	.	o	o	.	.	.	.	.	.
(Ducasse et al., 2004)		o	.	.	o	.	.	.	o	.	.	o	.	.	.	.	o	.	o	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.
(Eaddy et al., 2008)		o	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	o	o	.	o	.	.	o	.	o	.	o	.	o	.	o	.	o	.
(Edwards et al., 2006)			.	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	o	o	.	.	.	.
(Eisenbarth et al., 2003)	4		.	.	o	o	.	.	.	.	o	.	.	.	.	o	.	.	o	.	o	.	.	.	o	.	.	o	.	o	o	.	.	.	.
(Eisenberg and Volder, 2005)			.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	.	.	o	o	.	.	o	.	o	.	.	.	o	.	o	.	o	.
(El-Ramly et al., 2002)	3		.	.	.	o	.	o	.	o	.	.	.	.	.	o	.	.	.	o	.	o	o	.	.	.	.	.	.	.	o	o	.	o	.
(Fischer et al., 2005)			.	.	o	.	o	.	.	.	o	.	o	.	.	.	o	.	.	o	.	.	.	.	.	.	.	o	.	o	.	.	.	.	.
(Fisher II et al., 2007)			.	.	.	.	.	o	.	.	.	.	.	o	.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	o	.	.	.	o	.
(Gargiulo and Mancoridis, 2001)		o	.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	.
(Greevy et al., 2006a)	1		.	.	o	o	.	.	.	.	.	o	.	o	.	.	o	.	o	o	.	.	o	.	.	.	.	o	.	o	.	.	.	.	.
(Greevy et al., 2005)			.	.	.	o	.	.	.	.	.	o	.	o.	.	.	.	.	o	o	o	.	.	.	.	.	.	o	.	o	.	.	.	.	.
(Greevy et al., 2006b)	1		.	.	o	o	.	.	.	.	.	o	.	o	.	.	o	.	o	o	o	.	.	.	.	o	.	o	.	o	.	.	.	.	.
(Gschwind et al., 2003)	1		.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.
(Guéhéneuc et al., 2002)		o	.	.	.	.	.	.	o	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.	o	o	.	.	o	.	.	.	o	.
(Guéhéneuc, 2004)		o	.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	o	.	o	.	.	o	o	.	.	o	.	.	.	.	.
(Hamou-Lhadj et al., 2004)	5		.	.	o	.	o	.	.	.	o	o	.	.	.	o	.	.	o	o	.	o	o	.	.	.	.	.	.	o	.	.	.	o	o
(Hamou-Lhadj and Lethbridge, 2004)			o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o
(Hamou-Lhadj and Lethbridge, 2006)	1		.	.	o	.	o	o	.	.	.	o	.	.	.	o	.	.	o	o	o	.	o	.	.	.	.	.	.	o	.	.	o	o	.
(Hendrickson et al., 2005)	1		.	.	o	.	.	o	.	.	o	.	o	.	o	o	.	.	.	.	.	.	.	o	.	o	.	.	.	o	.	.	.	.	.
(Heuzeroth et al., 2003)	1		.	o	.	.	.	o	.	o	.	o	.	.	.	.	.	.	o	.	o	o	.	.	.	.	.	.	.	o	.	.	.	o	.
(Huang et al., 2005)			.	o	.	.	.	o	.	.	.	o	.	.	.	.	.	.	o	.	o	o	.	.	.	o	.	o	.	o	.	.	.	o	.
(Israr et al., 2007)			.	o	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	.	o	o	.	.	.	.	o	.	o	.	.	.	.	.
(Jerding and Rugaber, 1997)	2		.	.	o	.	o	.	.	.	o	o	.	.	.	o	o	.	o	.	o	o	.	.	.	.	.	.	.	o	.	.	.	.	.
(Jiang et al., 2007)			.	.	.	.	.	o	.	.	o	.	.	.	.	o	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o
(Jiang et al., 2006)			.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.
(Jiang et al., 2008)			.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	o	o	.	.	.	.	.	o	o	o	.	o	.
(Kelsen, 2004)			.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	o
(Kleyn and Gingrich, 1988)			.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.	.
(Kollmann and Gogolla, 2001b)			.	.	.	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(Korel and Rilling, 1998)	1		.	.	.	.	.	.	o	.	o	.	.	.	.	o	o	o	.	.	o	.	o	.	.	.	.	.	o	.	.	.	.	.	.

Table A.2 Article characterization results.

			activity							target						method													evaluation						
	add'l articles	tool avail.	survey	design/arch.	views	features	trace	behavior	general	legacy	procedural	oo	threads	web	distributed	vis. (std.)	vis. (adv.)	slicing	filtering	metrics	static	patt. det.	compr./summ.	heuristics	fca	querying	online	mult. traces	preliminary	regular	industrial	comparison	human subj.	quantitative	unknown/none
(Koschke and Quante, 2005)		o	.	.	o	o	.	.	.	.	o	.	.	.	.	o	.	.	o	o	o	.	.	.	o	.	.	o	.	o	.	o	.	o	.
(Koskimies and Mössenböck, 1996)		o	.	.	o	.	o	.	o	.	.	o	.	.	.	o	.	.	o	.	o	o	o	.	.	.	.	.	o	.	.	.	.	.	.
(Koskinen et al., 2006)			.	.	o	.	.	o	.	.	.	o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(Kothari et al., 2007)	1		.	.	.	o	.	.	.	.	o	o	.	.	.	.	o	.	.	o	.	.	o	.	.	.	.	o	.	o	.	.	.	.	.
(Kuhn and Greevy, 2006)			.	.	o	o	o	.	.	.	.	o	.	.	.	.	o	.	o	o	.	.	o	.	.	.	.	o	.	o	.	.	.	.	.
(Lange and Nakamura, 1995a)	1		.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	o	o	.	.	.	o	.	.	o	.	.	.	.	.	.
(Lange and Nakamura, 1997)			.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	o	o	.	o	o	.	.	.	.	.	.	o	.	.	.	.	.	.
(Licata et al., 2003)			.	.	.	o	.	.	.	.	.	o	.	.	.	o	.	.	.	.	o	.	o	.	.	.	.	o	.	o	.	.	.	.	.
(Lienhard et al., 2007)			.	.	o	o	.	.	.	.	.	o	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.	.	.	o	.	o	.	o	.
(Liu et al., 2007)			o	.	.	o	.	.	.	.	.	o	.	.	.	.	.	.	o	.	o	.	.	o	.	o	.	.	.	o	.	o	.	o	.
(Lo and Khoo, 2006a)			o	.	o	.	.	o	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	o	.
(Lo and Khoo, 2006b)			.	.	o	.	.	o	.	.	.	o	.	.	.	o	.	.	o	o	.	o	o	o	.	.	.	.	.	o	.	o	.	o	.
(Lo et al., 2008)			.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	o	.	.	o	.	o	.	.	.	.	.	o	.	.	.	.	.
(Lucca et al., 2003)	3		.	.	o	.	.	.	o	.	.	.	.	o	.	o	.	.	.	.	o	.	o	o	.	o	.	.	o	o	.	.	.	.	.
(Lukoit et al., 2000)			.	.	o	o	.	.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	.	.	.	.	o	.	.	o	o	.	.	.	.
(Martin et al., 2002)			.	.	o	.	.	o	.	.	.	.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o
(Moe and Sandahl, 2002)	2		.	.	.	.	.	.	o	.	.	.	.	.	o	.	o	.	o	o	.	o	.	.	.	o	.	.	.	o	o	.	.	.	.
(Oechsle and Schmitt, 2001)			.	.	o	.	.	.	o	.	.	o	o	.	.	o	.	.	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(Pacione et al., 2003)			o	.	o	.	.	.	o	.	.	o	.	.	.	o	o	.	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.
(Pacione et al., 2004)			.	.	.	.	.	.	o	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(De Pauw et al., 1994)	1		.	.	o	.	.	.	o	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	o	.	.	o	.	.	.	.	.	.
(De Pauw et al., 2001)	1	o	.	.	o	.	o	.	o	.	.	o	o	.	.	o	o	.	o	.	.	o	.	o	.	.	.	.	o	.	.	.	.	.	.
(De Pauw et al., 2006)		o	.	.	o	.	.	.	o	.	.	.	.	o	o	.	o	.	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.	.	.
(Pheng and Verbrugge, 2006)			.	.	.	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	o	.	.	o	.	.	.	.	.	o	.	.	.	.	.
(Poshyvanyk et al., 2007)	1		.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	.	o	o	o	.	.	o	.	.	.	o	.	o	.	o	.	o	.
(Qingshan et al., 2005)			.	o	o	.	.	.	.	.	.	o	.	.	.	o	.	o	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.	.
(Quante and Koschke, 2007)	3		.	.	o	.	.	o	o	.	o	o	o	.	.	o	.	o	o	o	.	o	o	.	.	.	o	.	.	o	.	o	.	o	.
(Quante, 2008)			.	.	o	o	.	.	o	.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	.	.	o	.	.	o	.	.
(Reiss and Renieris, 2001)			.	.	.	.	o	.	.	.	.	o	.	.	.	o	.	.	.	.	.	o	o	.	.	.	.	.	.	o	.	.	.	o	.
(Reiss, 2003a)			.	.	o	.	.	.	o	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(Reiss, 2006)	3	o	.	.	o	.	.	.	o	.	.	o	o	.	.	.	o	.	.	.	o	.	o	.	.	.	o	.	.	o	.	.	.	.	.
(Reiss, 2007)			o	.	o	.	.	.	o	.	o	o	o	.	.	o	o	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	.	o
(Renieris and Reiss, 1999)			.	.	o	.	o	.	.	.	o	.	.	.	.	.	o	.	o	.	o	.	.	.	.	o	.	.	o	.	.	.	.	.	.
(Richner and Ducasse, 1999)			.	o	o	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	o	.	o	.	.	o	.	.	.	o	.	.	.	.	.
(Richner and Ducasse, 2002)			.	.	o	.	.	.	o	.	.	o	.	.	.	.	o	.	o	.	.	o	.	.	.	o	.	.	.	o	.	.	.	.	.
(Rilling, 2001)			.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	o	o	o	o	.	o	.	.	.	.	.	.	.	.	.	.	.	o
(Ritsch and Sneed, 1993)			.	.	.	.	.	.	o	o	.	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(Riva, 2000)			.	o	o	.	.	.	.	.	o	.	.	.	.	o	.	.	.	o	o	.	o	.	.	.	.	.	.	.	o	.	.	.	.
(Riva and Yang, 2002)	1		.	o	o	.	.	.	.	.	o	.	.	.	.	o	.	o	o	.	o	.	o	.	.	.	.	.	o	.	.	.	.	.	.
(Rohatgi et al., 2008)			.	.	.	o	.	.	.	.	.	o	.	.	.	.	.	.	o	o	o	.	.	o	.	.	.	.	o	.	.	.	.	.	.
(De Roover et al., 2006)			.	.	.	.	.	o	.	.	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.	.
(Röthlisberger et al., 2008)			.	.	.	.	.	.	o	.	.	o	.	.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.	o	.	.	o	.	.
(Safyallah and Sartipi, 2006)			.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	.	o	o	.	.	.	.	.	o	o	.	.	.	.	.	.
(Salah and Mancoridis, 2003)			.	.	o	.	.	.	o	.	.	.	.	.	o	o	.	.	.	.	o	.	.	.	.	o	.	.	o	.	.	.	.	.	.
(Salah and Mancoridis, 2004)			.	.	o	o	.	.	.	.	.	o	.	.	.	o	.	.	o	.	o	.	o	.	.	o	.	o	.	o	.	.	.	.	.

Table A.3 Article characterization results (continued).

			activity							target						method													evaluation						
	add'l articles	tool avail.	survey	design/arch.	views	features	trace	behavior	general	legacy	procedural	oo	threads	web	distributed	vis. (std.)	vis. (adv.)	slicing	filtering	metrics	static	patt. det.	compr./summ.	heuristics	fca	querying	online	mult. traces	preliminary	regular	industrial	comparison	human subj.	quantitative	unknown/none
(Salah et al., 2006)	2		.	.	o	o	.	.	.	.	.	o	.	.	.	o	.	.	.	o	.	o	o	.	.	o	.	o	.	o	.	.	.	.	.
(Sartipi and Dezhkam, 2007)	2		.	o	o	o	.	.	.	.	o	.	.	.	.	o	.	.	o	o	o	o	o	.	o	o	.	.	.	o	.	.	.	o	.
(Schmerl et al., 2006)	1		.	o	o	.	.	o	.	.	.	o	.	.	.	o	.	.	o	.	.	o	.	.	.	o	.	.	.	o	.	.	.	.	.
(Sefika et al., 1996)			.	o	o	.	.	.	.	.	.	o	.	.	.	o	.	.	o	.	o	.	.	.	.	o	.	.	.	o	.	.	.	o	.
(Shevertalov and Mancoridis, 2007)			.	.	o	.	.	o	.	o	.	.	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	.	.	.	.	.	.	.	o
(Simmons et al., 2006)			.	.	o	o	.	.	.	.	o	.	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	o	.	o	.	.	o	o	.
(Smit et al., 2008)			.	.	o	.	.	o	.	.	.	o	.	.	.	o	.	.	o	o	.	.	o	.	.	.	.	o	.	o	.	.	.	.	.
(Souder et al., 2001)			.	.	o	.	.	.	o	.	.	o	.	.	o	o	.	.	o	.	.	.	.	.	.	.	.	.	o	.	.	.	.	.	.
(de Sousa et al., 2007)			.	.	.	.	.	.	.	.	.	o	.	o	.	o	.	.	o	.	.	o	.	.	.	.	.	.	.	o	.	.	.	.	.
(Stroulia et al., 1999)			.	.	.	.	.	o	.	o	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	o
(Systä et al., 2001)	3		.	.	o	.	o	.	o	.	.	o	o	.	.	o	.	o	.	o	o	o	.	.	.	o	.	.	.	o	.	.	.	.	.
(Tonella and Potrich, 2002)			.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	o	.	.	.	.	.	.	.	.	o	.	.	.	.	.
(Walker et al., 2000)	1		.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	.	.	o	.	.	o	.	.	o	.	.	.	.	.	o
(Walkinshaw et al., 2008)	1		.	.	.	.	.	o	.	.	.	o	.	.	.	o	.	.	.	o	o	o	o	.	.	o	.	.	.	o	.	.	.	.	.
(Wang et al., 2005)			.	o	.	.	.	.	.	.	.	o	.	.	.	.	.	.	.	.	o	o	.	.	.	o	.	.	.	o	.	.	.	o	.
(Wilde and Scully, 1995)	4	o	.	.	.	o	.	.	o	o	o	.	.	.	.	.	.	.	.	o	.	.	.	o	.	.	.	o	o	o	o	o	o	.	.
(Wong and Gokhale, 2005)	2	o	.	.	.	o	.	.	.	.	o	.	.	.	.	o	.	o	.	o	o	.	.	o	.	.	.	.	.	o	.	.	.	o	.
(Zaidman and Demeyer, 2004)			.	.	.	.	o	.	.	.	.	o	.	.	.	.	o	.	o	o	.	.	o	o	.	.	.	.	.	o	.	.	.	.	.
(Zaidman and Demeyer, 2008)	2		.	.	.	.	.	.	o	.	.	o	.	.	.	.	.	.	.	o	o	.	.	o	.	.	.	.	.	o	.	.	.	o	.
(Zaidman et al., 2006)			.	.	.	.	o	.	o	o	o	.	.	.	.	.	o	.	.	o	.	.	.	o	.	.	.	.	.	.	o	.	.	.	.

Table A.4 Article characterization results (continued).

Controlled Experiment: Subject Characterization & Handouts

Appendix B

This appendix provides the expertise assessment presented to all 24 subjects who participated in the controlled experiment in Chapter 6, a characterization of the subjects, the handouts that they received, and the answer model used to grade their answers.

B.1 Expertise assessment

Please rate your expertise in the following fields.

- Java
  1. none: difficulty in reading Java, and never written a line
  2. beginner: can read Java, but has trouble in writing
  3. regular: can easily read Java and write basic programs
  4. advanced: familiar with most paradigms; worked on more advanced Java programs
  5. expert: years of experience
- Eclipse
  1. none: never used it
  2. beginner: incidentally used it, mostly for browsing
  3. regular: familiar with several features (e.g., checking for dependencies)
  4. advanced: experience with most features
  5. expert: years of experience
- Checkstyle
  1. none: never used it, not familiar with its user-level functionality
  2. beginner: used it one or two times, or familiar with user-level functionality
  3. regular: used it occasionally, familiar with a few implementation details
  4. advanced: very frequent user, wrote own extensions/checkers
  5. expert: years of experience

- Language technology, parsing, grammars, etc.
  1. none: no knowledge
  2. beginner: only some theoretical knowledge
  3. regular: applied it once or twice during student assignments
  4. advanced: frequently applied the theory in practice
  5. expert: years of experience

- Reverse engineering, software inspection/understanding
  1. none: no knowledge
  2. beginner: only some theoretical knowledge
  3. regular: did it once or twice during student assignments
  4. advanced: frequently performed these activities
  5. expert: years of experience

B.2 Subject characterization

			Expertise (0-4)						Performance (T1-T3)	
Subject #	**Affil.**	**M.Sc.**	**Avg**	**Java**	**Eclipse**	**Checkstyle**	**Lang.tech.**	**Rev. Eng.**	**Time**	**Correctness**
1	CWI	UvA	3.2	4	4	1	4	3	56	7
2	SIG	UU	3.0	3	3	3	3	3	55	14
3	CWI	UvA	2.8	4	3	3	2	2	38	16
4	TUD	UU	2.8	4	4	0	4	2	68	20
5	TUD	TUD	2.8	4	3	2	2	3	70	18
6	TUD	KUN	2.2	3	3	1	3	1	59	18
7	TUD	(Por)	2.2	3	2	0	3	3	37	11
8	UA	(Bel)	2.0	3	2	0	2	3	84	14
9	TUD	(Fra)	2.0	3	2	0	2	3	81	19
10	MCQ	(Aus)	2.0	2	2	0	4	2	77	15
11*	UT	UT	2.0	3	1	1	3	2	70	14
12*	TUD	TUD	2.0	3	3	0	2	2	70	13
13	VIT	(Fin)	2.0	4	1	1	0	4	70	22
14	TUD	TUD	1.8	3	3	0	3	0	97	14
15*	TUD	TUD	1.8	3	2	0	2	2	72	22
16	TUD	RUG	1.8	3	2	0	3	1	38	20
17	TUD	TUD	1.8	3	2	0	3	1	36	19
18	TUD	(Spa)	1.8	3	2	0	2	2	58	19
19*	TUD	TUD	1.8	3	2	0	2	2	66	19
20	TUD	UU	1.6	2	1	0	3	2	88	15
21*	TUD	TUD	1.6	3	2	0	2	1	70	18
22	TUD	(Ger)	1.4	3	2	0	2	0	97	7
23	TUD	TUD	1.4	3	2	0	1	1	69	13
24	UA	(Bel)	1.2	2	2	0	1	1	102	5

Table B.1 Characterization of the subjects and their performances, ordered by average expertise. Students are denoted by asterisks.

B.3 Handouts

Introduction

Thank you for your willingness to participate in this experiment! Empirical studies are not very common in the field of software understanding because this field has a strong cognitive aspect that is difficult to measure. This makes controlled experiments (such as the one in which you are now participating) all the more valuable. I hope you will find it an interesting experience.

The context of this experiment concerns a (fictive) developer who is asked to perform certain maintenance tasks on a system, but who is unfamiliar with its implementation. The focus of the experiment is not on *performing* these maintenance tasks, but rather on gaining the necessary *knowledge* and measuring the effort that is involved therein.

The case study in this experiment is Checkstyle. From the Checkstyle site:

> *Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. This makes it ideal for projects that want to enforce a coding standard.*

In short, Checkstyle takes as inputs a Java source file, and an XML configuration file that specifies the coding standards that must be enforced, i.e., the *checks* that are to be used.

Most people are not familiar with Checkstyle's implementation. However, IDEs (such as Eclipse) and effective tools may be able to assist in understanding Checkstyle's inner workings, and most of the source code is fairly documented.

You are given 90 minutes for four comprehension tasks, which have been structured according to their maintenance contexts. Each task involves several related subtasks, some of which designed to help you on your way. The task ordering implies a top-down approach: we start by building a general knowledge of the system and then drill down to a more detailed understanding. For each of the subtasks, you are asked to write down the following items:

- Your answer.
- A motivation of your answer. In most cases it suffices to briefly describe how your answer was obtained.
- The *time* at which you started this subtask (important!).

Furthermore, you are asked (1) to not consult any other participants, (2) to perform the tasks in the order specified, and (3) to *not return to earlier tasks* because it affects the timing. Finally, while there is an online documentation available for Checkstyle, you are kindly requested *not* to use this, because we want to simulate a real life case in which up-to-date documentation is often lacking. Using the Internet is allowed only for Java-related resources (e.g., APIs).

We start off by describing the tools at your disposal. You are then presented with the comprehension tasks, at which point your 90 minutes start ticking. The experiment is concluded with a short questionnaire.

Using the Eclipse IDE

Eclipse is the IDE that you will be using to perform the tasks. You are expected to have a basic knowledge of Eclipse, but a quick "reference chart" is provided nonetheless. While this chart only shows the basic features, of course you are encouraged to use more advanced functionalities if you are familiar with them.

Your Eclipse setup contains a project with Checkstyle's source (and links to its external libraries). Several aspects are worth mentioning:

- Checkstyle's testsuite is not available to you. This reflects real life cases in which the testsuite is not complete, out of date, or non-existent at all.
- The experiment will not be concentrating on Checkstyle's GUI.
- You may compile and run the application if so desired.

Finally, you have at your disposal an input source file, `Simple.java`; and two different configuration XML-files, `many_checks.xml` and `several_checks.xml`. The resulting outputs of running Checkstyle with these inputs are given on the next few pages.

Should you have trouble using Eclipse, please refer to the reference chart, or consult me (Bas).

Using Extravis (optional handout)

In addition to the Eclipse IDE, you will also have access to Extravis during the experiment. Extravis is a dynamic analysis tool, which means it provides information on the system's runtime behavior. In this experiment, this is done through *execution traces*, which were obtained by instrumenting Checkstyle and then having it run a certain execution scenario. Such traces contain:

1. A chronological ordering of all *method* and *constructor calls* that occurred during execution. Typically this amounts to thousands or even millions of events for each scenario.
2. The *actual* class instances (objects) on which these methods and constructors were invoked. This means that (e.g.) if class `A` inherits method `a()` from some superclass `B`, the trace will show the receiver of the call `A.a()` to be `A`, not `B`.

Extravis visualizes these execution traces and the program's package decomposition, and provides means to navigate this information.

A quick reference chart of Extravis has been provided as part of the handouts. In addition to depicting all method and constructor calls that occurred in the scenario, Extravis also shows the *actual parameters* and *actual return values* for those calls. The developer is thus provided with a rich and accurate source of information with respect to the scenario at hand.

You have two execution traces at your disposal: `simple-many_checks` and `simple-several_checks`. The former trace is the result of Checkstyle's execution with `Simple.java` and `many_checks.xml` as its parameters; the latter trace was also obtained using `Simple.java`, but with `several_checks.xml` as the configuration file.

Finally, some aspects that are worth noting:

- Only Checkstyle's *core* functionalities were instrumented, which means that the resulting traces do not contain calls to, or from, external libraries or the JDK.
- Extravis provides two *linked* views: changes made in the one view are propagated toward the other.
- The leftmost view concentrates on visualizing the target system's structure and its (runtime!) interrelationships, whereas the rightmost view focuses more on navigating the trace.

In other words, Extravis answers questions related to a program's actual execution, and aims to provide insight in the interactions that take place.

You are free to use Extravis during your tasks whenever you see fit. Should you have trouble using the tool, please refer to the reference chart, or consult me (Bas).

Checkstyle outputs

```
bas@laptop:˜/checkstyle-4.4$ java -jar checkstyle-all-4.4.jar
                                  -c several_checks.xml Simple.java
Starting audit...
Simple.java:11: Line has trailing spaces.
Simple.java:17:1: Line contains a tab character.
Simple.java:18:1: Line contains a tab character.
Simple.java:19:1: Line contains a tab character.
Simple.java:20:1: Line contains a tab character.
Simple.java:21:1: Line contains a tab character.
Simple.java:23:1: Line contains a tab character.
Simple.java:24:1: Line contains a tab character.
Simple.java:25:1: Line contains a tab character.
Simple.java:26:1: Line contains a tab character.
Simple.java:27:1: Line contains a tab character.
Simple.java:28:1: Line contains a tab character.
Simple.java:29: Line has trailing spaces.
Simple.java:29:1: Line contains a tab character.
Simple.java:30:1: Line contains a tab character.
Simple.java:31:1: Line contains a tab character.
Simple.java:32:1: Line contains a tab character.
Simple.java:33:1: Line contains a tab character.
Simple.java:34:1: Line contains a tab character.
Simple.java:35:1: Line contains a tab character.
Simple.java:36:1: Line contains a tab character.
Simple.java:37:1: Line contains a tab character.
Simple.java:38:1: Line contains a tab character.
Simple.java:40:1: Line contains a tab character.
Simple.java:40:9: Missing a Javadoc comment.
Simple.java:41:1: Line contains a tab character.
Simple.java:42:1: Line contains a tab character.
Simple.java:43:1: Line contains a tab character.
Simple.java:44:1: Line contains a tab character.
Simple.java:45:1: Line contains a tab character.
Simple.java:46:1: Line contains a tab character.
Simple.java:47:1: Line contains a tab character.
Audit done.
```

```
bas@laptop:˜/checkstyle-4.4$ java -jar checkstyle-all-4.4.jar
                                  -c many_checks.xml Simple.java
Starting audit...
/home/bas/checkstyle-4.4/package.html:0: Missing package documentation file.
Simple.java:11: Line has trailing spaces.
Simple.java:17:1: Line contains a tab character.
Simple.java:17:9: Missing a Javadoc comment.
Simple.java:18:1: Line contains a tab character.
Simple.java:18:9: Missing a Javadoc comment.
Simple.java:19:1: Line contains a tab character.
Simple.java:19:9: Missing a Javadoc comment.
Simple.java:19:23: ’<’ is not preceded with whitespace.
Simple.java:19:24: ’<’ is not followed by whitespace.
Simple.java:19:31: ’>’ is not preceded with whitespace.
Simple.java:20:1: Line contains a tab character.
Simple.java:20:9: Missing a Javadoc comment.
Simple.java:21:1: Line contains a tab character.
Simple.java:21:9: Missing a Javadoc comment.
Simple.java:23:1: Line contains a tab character.
Simple.java:24:1: Line contains a tab character.
Simple.java:25:1: Line contains a tab character.
Simple.java:26:1: Line contains a tab character.
Simple.java:27:1: Line contains a tab character.
Simple.java:28:1: Line contains a tab character.
Simple.java:29: Line has trailing spaces.
Simple.java:29:1: Line contains a tab character.
Simple.java:30:1: Line contains a tab character.
Simple.java:31:1: Line contains a tab character.
Simple.java:32:1: Line contains a tab character.
Simple.java:33:1: Line contains a tab character.
Simple.java:34:1: Line contains a tab character.
Simple.java:34:21: Parameter map should be final.
Simple.java:35:1: Line contains a tab character.
Simple.java:36:1: Line contains a tab character.
Simple.java:37:1: Line contains a tab character.
Simple.java:38:1: Line contains a tab character.
Simple.java:40:1: Line contains a tab character.
Simple.java:40:9: Method 'initialize' is not designed for extension - needs
                                    to be abstract, final or empty.
Simple.java:40:9: Missing a Javadoc comment.
Simple.java:41:1: Line contains a tab character.
Simple.java:41:38: '<' is not preceded with whitespace.
Simple.java:41:39: '<' is not followed by whitespace.
Simple.java:41:46: '>' is not preceded with whitespace.
Simple.java:41:47: '>' is not followed by whitespace.
Simple.java:42:1: Line contains a tab character.
Simple.java:43:1: Line contains a tab character.
Simple.java:44:1: Line contains a tab character.
Simple.java:45:1: Line contains a tab character.
Simple.java:46:1: Line contains a tab character.
Simple.java:47:1: Line contains a tab character.
Audit done.
```

Tasks

1. Gaining a general understanding

The first thing the developer might desire is a first impression of how Checkstyle works, especially in case domain knowledge is lacking.

- **Task 1.** [current time: . . : . .]
  Having glanced through the available information for several minutes, which do you think are the main stages in a typical (non-GUI) Checkstyle scenario? Formulate your answer from a *high-level* perspective: refrain from using identifier names and stick to a maximum of six main stages.

2. Identifying refactoring opportunities

In certain cases (not necessarily Checkstyle) it is desirable to modify the program's package hierarchy. Examples include the movement of tightly coupled classes to the same package, and the movement of classes with high fan-in and (almost) no fan-out to a utility package.

The *fan-in* of a class is defined as the number of distinct methods/constructors directed toward that class, not counting self-calls. Its *fan-out* is defined as the number of distinct methods/constructors directed toward other classes.

- **Task 2.1.** [current time: . . : . .]
  Name three classes in Checkstyle that have a high fan-in and (almost) no fan-out.

Assume that a tight coupling is characterized by a relatively large number of *different* method calls between two structural entities (e.g., classes or packages).

- **Task 2.2.** [current time: . . : . .]
  Name a class in the default package (i.e., classes not in any package) that could be a candidate for movement to the `api` package because of its tight coupling with classes therein.

3. Understanding the checking process

Checkstyle's purpose is the application of *checks* on its input source file. These checks each have their own class and are located in the `checks`-package. They can be written by a developer and contributed to the Checkstyle package: For example, one could write a check to impose a limit on the number of methods in a class. If our developer wants to write a new check, one way to gain the necessary knowledge is to study existing checks (i.e., learning by example).

Let's assume that we want to know how `checks.whitespace.TabCharacterCheck` interacts with the rest of the program, and that this check is part of the current configuration (and will therefore be applied).

- **Task 3.1.** [current time: . . : . .]
  In *general* terms, describe the life cycle of this check during execution: when is it created, what does it do and on whose command, and how does it end up?
  Do not go into details yet, and use no more than five sentences.

The `TreeWalker` class plays an important role in Checkstyle's inner workings and interacts extensively with the checks. We now take a closer look at the protocol between `TreeWalker` and the various checks.

- **Task 3.2.** [current time: . . : . .]
  List the identifiers of all method/constructor calls that typically occur between `TreeWalker` and a `checks.whitespace.TabCharacterCheck` instance, and the order in which they are called. Make sure you also take inherited methods/constructors into account.

- **Task 3.3.** [current time: . . : . .]
  In comparison to the calls listed in Task 3.2., which *additional* calls occur between `TreeWalker` and `checks.coding.IllegalInstantiationCheck`? Can you think of a reason for the difference?

4. Understanding the violation reporting process

Once the developer has written a new check, he/she would like to know if it works, i.e., whether it reports warnings when appropriate. Consider the situation in which some check has encountered a violation.

- **Task 4.1.** [current time: . . : . .]
  How is the check's warning handled, i.e., where/how does it originate, how is it internally represented, and how is it ultimately communicated to the user?

Verifying whether a check actually found violations is not trivial: most of the warnings that are reported in Checkstyle's output (provided a few pages back) cannot be traced back directly to the checks from which those warnings originate. Some reported warnings may even be quite confusing.

- **Task 4.2.** [current time: . . : . .]
  Given `Simple.java` as the input source and `many_checks.xml` as the configuration, does
  `checks.whitespace.WhitespaceAfterCheck` report warnings? Specify how your answer was obtained.

Debriefing Questionnaire [current time: . . : . .]

The experiment is concluded with a short questionnaire in which we ask for your opinions on several experimental aspects. You may fill in your answers on the handouts themselves.

- On a scale of 1 to 5, how did you feel about the time pressure?
  1. too much time pressure: could not cope with it, regardless of task difficulty
  2. fair amount of time pressure: could certainly have done better with more time
  3. not so much time pressure: hurried a bit, but it was OK
  4. very little time pressure: felt quite comfortable
  5. no time pressure at all
- Regardless of the time given, how difficult would you rate the tasks? Please mark the appropriate difficulty for each of the tasks:

	impossible	difficult	intermediate	simple	trivial
Task 1					
Task 2.1 **Task 2.2**					
Task 3.1 **Task 3.2** **Task 3.3**					
Task 4.1 **Task 4.2**					

- Which particular Eclipse features did you frequently use?
  - Package explorer
  - Open declaration
  - Open type hierarchy
  - Open call hierarchy
  - Text search
  - other:

    ...

    ...

- Do you feel that additional Eclipse plugins (that you know of) could have helped during the experiment? If so, please name those plugins and briefly explain how they would have assisted you.

  ...

  ...

- "Dynamic analysis" is the study of a program through its runtime behavior. Are you familiar with dynamic analysis and its benefits?
  1. Never heard of it
  2. I know what it is, more or less
  3. I'm familiar with it and could name one or two benefits
  4. I know it quite well and performed it once or twice
  5. I've used it on multiple occasions

And finally, several questions on your use of Extravis:

- During which tasks did you use Extravis, and in which of these tasks did you actually find it helpful? Please fill in the following table:

	used it	used it successfully
Task 1		
Task 2.1		
Task 2.2		
Task 3.1		
Task 3.2		
Task 3.3		
Task 4.1		
Task 4.2		

- Can you give a rough estimate of the percentage of time that you spent on using Extravis?
  ...

- Which particular Extravis features did you use more than once? Please mark those features on your reference chart.

- Which (types of) Extravis features did you feel were missing?
  - Compare multiple traces side-by-side
  - Interactiveness in terms of the input (e.g., support for creating and visualizing own traces)
  - More readable / intuitive views of detailed interactions
  - Direct link from actual calls to their source code locations
  - Search capabilities
  - other:
    ...
    ...

- On a scale of 1 to 3, how did Extravis perform in terms of speed and responsiveness?
  1. Quite sluggishly: I got impatient very often
  2. Pretty OK: occasionally had to wait for information
  3. Very quickly: the information was shown instantly

- Did you experience a certain "resistance" to using Extravis and, instead, stuck to Eclipse as much as possible? If so, how would you explain this tendency? (multiple answers possible)
  - Time pressure

- I'm so comfortable or skilled with Eclipse that I prefer to use Eclipse whenever I can
- I felt that the tasks simply did not require more than Eclipse / source code
- I'm not sufficiently familiar with Extravis (tutorial was too short)
- I'm not (sufficiently) familiar with the benefits of using runtime information in general
- Extravis is standalone rather than embedded in Eclipse, and I'm not comfortable with context switching
- other:
  ...
  ...

B.4 Answer model

Task 1
The following stages largely capture a typical Checkstyle scenario, and represent the minimum for this question. We can be somewhat flexible since the task deals from a very high level perspective.

Assign one point for each stage that is contained by the given answer. Additional stages are permitted but do not yield additional points. Motivations are not necessary if the answer is meaningfully-named or self-explanatory.

- Initialization: command line parsing, or config reading, or environment setup (creation of checkers, listeners etc.)
- Source parsing: source input file is read/parsed, or AST construction
- Checking: input file is checked, or AST traversal
- Error reporting/Termination/results logging: conveyance of warnings, and teardown of application

Task 2.1
The second list below is alphabetically ordered and shows all classes of which the fanin is higher than the fanout. (Classes not in this list are obviously incorrect.) We were looking for classes with a (relatively) high fanin and a low fanout, so award points in case of appropriate proportions between the two. Examples include fanin 5 + fanout 0, fanin 10 + fanout 1, fanin 15 + fanout 2, etc. (flexible scale).

Each correct class receives one point; award four points in case all classes have appropriate fanins and fanouts of 0 or 1. In case no plausible motivation is given, award no points.

Task 2.2
The first list below shows all classes in the default package, and their degrees of coupling with the api package. The coupling values in this list were statically derived; it is defined as the sum of the no. of distinct calls from, and the no. of calls to, classes in the api package.

TreeWalker and Checker receive 4 points because in the context of the question (i.e., our definition of "coupling") they have the strongest api-coupling by far. The next four classes are awarded 2 points; all others receive none.

For certain alternative classes it can be argued that they have a strong coupling with the api because they communicate exclusively with the api (and with external libraries). While this is reasonable, it is not according to the given definition of coupling, and therefore receives only 2 points. Award one point in case one went looking in the api-package.

Task 3.1
The following elements largely capture the lifecycle of a TabCharacterCheck instance during the specified execution scenario. Again we can be a bit flexible due to the high-level perspective, but the elements below must be mentioned because they answer the four implicit sub-questions (where does it originate, what does it do, on whose command, how does it end up). Assign one point for each element that is contained in the given answer.

- The check is created/configured during config reading / init. / environment setup / in setupChild() / by ModuleFactory / by PackageObjectFactory.
- The check scans (the file contents of) the input source for tab character occurrences (which may lead to the creation of warning messages).
- The above happens at the command of TreeWalker (as it commences processing the input source). (mentioning beginTree is sufficient)
- The check is explicitly destroyed (by TreeWalker).

Task 3.2
Listed below are the eight calls that occur between TreeWalker and TabCharacterCheck during this specific scenario. Assign one point for every two correct calls (in the right order). Subtract one point for every two incorrect calls (and in case of one single incorrect call).

Note that in fact there is one more call, destroy(), but a bug in Extravis prevents this call from being shown in the MSV, even at 100visibility. Therefore, the reviewer should not take this call into consideration in either of the two subject groups.

Note that if the answer specifies nested calls within correct calls, they may be ignored by the reviewer as long as it is clear that they are nested.

contextualize
configure
init
getTokenNames
getDefaultTokens
setFileContents
beginTree
finishTree
destroy (not considered due to Extravis bug)

Task 3.3

visitToken
leaveToken

One correct: 2 points. Both correct: another point.
Subtract one point for each incorrect call. Also allow implementation-based reasoning.
Award 1 ″bonus″ point in case the motivation specifies that IllegalInstantionCheck actually visits/checks for tokens (in the AST), whereas TabCharacterCheck does not (because it processes the file contents directly) – regardless of (in)correctness of the abovementioned identifiers.

Task 4.1
The following elements largely capture the essence of the error handling process. Assign one point for each element that is contained in the given answer. Note that mentioning the check's ″mMessages″-field implies knowledge of both 2. and 3., and therefore yields points for both of these elements.

- A violation results in a call to log(), or in a read of message.properties for a human-readable format.
- The warning is internally stored/represented as an api.LocalizedMessage.
- The LocalizedMessage is added to the api.LocalizedMessages field (called ″mMessages″) of its Checker (in this case, TreeWalker) – note that there are multiple such repositories, not a global one!
- At the *end* of execution, the messages are relayed to the listeners (which each convert the messages to different output types – human readable format / xml / etc.). Mentioning fireErrors() and its effect is also sufficient.

Task 4.2
The answer is no, which yields two points. The remaining points depend on the soundness of the motivation. The following are correct examples:

- Look for communication between the check and api.LocalizedMessage(s) in an execution trace.
- or: look for communication between the check and api.DetailAST.
- or: investigate the actual effect of visitToken().
- or: find out what kind of human-readable message should result from violations in this check, and match this message with Checkstyle's (example) output. The true conclusion should be that there is no match: none of the warnings in this scenario's output relate to the check at hand.
- or: run and debug the application (e.g., using print statements, breakpoints, etc.).

If the answer is solely based on the interpretation of the ws.notFollowed file, the answer is partly correct because this trail will run cold – award two points in case the conclusion was "no", one point in case of a "yes".

Only two points are awarded if the reasoning is based on an understanding of what the check is looking for, and on the fact that the input source file contains no occurrences of whitespaces after tokens. No full score here because through this reasoning it cannot be determined that the check actually works.

Coupling measurements

Acquired through an automated analysis of the static call graph.

```
43 TreeWalker
35 Checker
------------------------
8  XMLLogger
6  CheckStyleTask
6  DefaultLogger
6  DefaultConfiguration
------------------------
4  ConfigurationLoader$InternalLoader
3  PackageNamesLoader
2  ConfigurationLoader
2  DefaultContext
2  Main
1  PackageObjectFactory
1  PropertyCacheFile
```

Fanin/Fanout measurements

This list is left to the technical report (Cornelissen et al., 2009c).

Bibliography

Andrews, J. (1997). Testing using log file analysis: tools, methods, and issues. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 157–166. IEEE Computer Society Press. (Cited on page 3.)

Antoniol, G. and Di Penta, M. (2004). A distributed architecture for dynamic analyses on user-profile data. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 319–328. IEEE Computer Society Press. (Cited on page 156.)

Antoniol, G., Di Penta, M., and Zazzara, M. (2004). Understanding web applications through dynamic analysis. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 120–131. IEEE Computer Society Press. (Cited on page 156.)

Antoniol, G. and Guéhéneuc, Y.-G. (2006). Feature identification: An epidemiological metaphor. *IEEE Transactions on Software Engineering*, 32(9):627–641. (Cited on pages 16, 32, and 156.)

Ball, T. (1996). Software visualization in the large. *IEEE Computer*, 29(4):33–43. (Cited on page 156.)

Ball, T. (1999). The concept of dynamic analysis. In *Proc. European Software Engineering Conference & ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 216–234. Springer. (Cited on pages 2, 3, 14, 71, 73, and 156.)

Basili, V. R. (1997). Evolving and packaging reading technologies. *Journal of Systems & Software*, 38(1):3–12. (Cited on pages 71, 91, and 127.)

Basili, V. R., Caldiera, G., and Rombach, H. D. (1994). The goal question metric approach. In *Encyclopedia of Software Engineering*, pages 528–532. Wiley & Sons. (Cited on page 74.)

Beck, K. (1999). *Extreme Programming Explained - Embrace Change*. Addison-Wesley. (Cited on page 37.)

Beck, K. (2003). *Test-Driven Development: By Example*. Addison-Wesley. (Cited on page 37.)

Beck, K. and Gamma, E. (1998). Test infected: Programmers love writing tests. *Java Report*, 3(7):51–56. (Cited on page 37.)

Beecham, S., Baddoo, N., Hall, T., Robinson, H., and Sharp, H. (2008). Motivation in software engineering: A systematic literature review. *Information & Software Technology*, 50(9-10):860–878. (Cited on page 17.)

Benedikt, M., Freire, J., , and Godefroid, P. (2002). VeriWeb: Automatically testing dynamic web sites. In *Proc. International Conference on World Wide Web (WWW)*, pages 654–668. ACM Press. (Cited on page 29.)

Bennett, C., Myers, D., Ouellet, D., Storey, M.-A., Salois, M., German, D., and Charland, P. (2008). A survey and evaluation of tool features for understanding reverse engineered sequence diagrams. *Journal of Software Maintenance & Evolution*, 20(4):291–315. (Cited on pages 6, 31, 33, 144, and 156.)

Bennett, K. H. (1995). Legacy systems: Coping with success. *IEEE Software*, 12(1):19–23. (Cited on page 1.)

Biermann, A. W. (1972). On the inference of turing machines from sample computations. *Artificial Intelligence*, 3(1-3):181–198. (Cited on page 14.)

Biggerstaff, T. J., Mitbander, B. G., and Webster, D. (1993). The concept assignment problem in program understanding. In *Proc. International Conference on Software Engineering (ICSE)*, pages 482–498. IEEE Computer Society Press. (Cited on page 1.)

Binder, R. (2000). *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley. (Cited on page 47.)

Bojic, D. and Velasevic, D. M. (2000). A use-case driven method of architecture recovery for program understanding and reuse reengineering. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 23–32. IEEE Computer Society Press. (Cited on page 156.)

Brereton, P., Kitchenham, B. A., Budgen, D., Turner, M., and Khalil, M. (2007). Lessons from applying the systematic literature review process within the software engineering domain. *Journal of Systems & Software*, 80(4):571–583. (Cited on pages 5, 17, 18, 33, and 34.)

Briand, L. C., Labiche, Y., and Leduc, J. (2006). Toward the reverse engineering of UML sequence diagrams for distributed Java software. *IEEE Transactions on Software Engineering*, 32(9):642–663. (Cited on pages 6, 15, 26, 67, 94, 119, and 156.)

Chan, A., Holmes, R., Murphy, G. C., and Ying, A. T. T. (2003). Scaling an object-oriented system execution visualizer through sampling. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 237–244. IEEE Computer Society Press. (Cited on pages 73, 76, and 156.)

Cilibrasi, R. and Vitányi, P. (2005). Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545. (Cited on pages 78, 86, and 87.)

Cook, J. E. and Du, Z. (2005). Discovering thread interactions in a concurrent system. *Journal of Systems & Software*, 77(3):285–297. (Cited on pages 15 and 156.)

Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306. (Cited on pages 2, 11, 71, and 127.)

Cornelissen, B. (2007). Dynamic analysis techniques for the reconstruction of architectural views. In *Proc. Working Conference on Reverse Engineering (WCRE), Doctoral Symposium*, pages 281–284. IEEE Computer Society Press. (Cited on page 10.)

Cornelissen, B., Graaf, B., and Moonen, L. (2005). Identification of variation points using dynamic analysis. In *Proc. Workshop on Reengineering Towards Product Lines (R2PL)*, pages 9–13. Special Report CMU/SEI-2006-SR-002, Carnegie Mellon University. (Cited on page 10.)

Cornelissen, B., Holten, D., Zaidman, A., Moonen, L., van Wijk, J. J., and van Deursen, A. (2007a). Understanding execution traces using massive sequence and circular bundle views. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 49–58. IEEE Computer Society Press. (Cited on pages 10 and 80.)

Cornelissen, B. and Moonen, L. (2007). Visualizing similarities in execution traces. In *Proc. Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 6–10. Tech. report TUD-SERG-2007-022, Delft University of Technology. (Cited on pages 10 and 81.)

Cornelissen, B., Moonen, L., and Zaidman, A. (2008a). An assessment methodology for trace reduction techniques. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 107–16. IEEE Computer Society Press. (Cited on pages 10, 71, and 156.)

Cornelissen, B., van Deursen, A., and L.Moonen (2006). Aiding in the comprehension of testsuites. In *Proc. Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 17–20. Tech. report 2006-11, University of Antwerp. (Cited on page 10.)

Cornelissen, B., van Deursen, A., Moonen, L., and Zaidman, A. (2007b). Visualizing testsuites to aid in software understanding. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 213–222. IEEE Computer Society Press. (Cited on pages 10, 37, 72, 73, and 156.)

Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., and van Wijk, J. J. (2008b). Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems & Software*, 81(11):2252–2268. (Cited on pages 10, 80, 91, 131, and 156.)

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2008c). A systematic survey of program comprehension through dynamic analysis. Technical Report TUD-SERG-2008-033, Delft University of Technology. (Cited on pages 19 and 26.)

Cornelissen, B., Zaidman, A., van Deursen, A., Moonen, L., and Koschke, R. (2009a). A systematic survey of program comprehension through dynamic analysis. To appear. (Cited on pages 10, 11, and 18.)

Cornelissen, B., Zaidman, A., Van Rompaey, B., and van Deursen, A. (2009b). Trace visualization for program comprehension: A controlled experiment. In *Proc. Int. Conf. on Program Comprehension (ICPC)*. IEEE Computer Society Press. To appear. (Cited on pages 10 and 127.)

Cornelissen, B., Zaidman, A., Van Rompaey, B., and van Deursen, A. (2009c). Trace visualization for program comprehension: A controlled experiment. Technical Report TUD-SERG-2009-001, Delft University of Technology. (Cited on page 173.)

Dalton, A. R. and Hallstrom, J. O. (2008). A toolkit for visualizing the runtime behavior of TinyOS applications. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 43–52. IEEE Computer Society Press. (Cited on page 156.)

De Pauw, W., Helm, R., Kimelman, D., and Vlissides, J. M. (1993). Visualizing the behavior of object-oriented systems. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 326–337. ACM Press. (Cited on pages 6, 14, 33, 119, 122, 127, and 128.)

De Pauw, W., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. M., and Yang, J. (2001). Visualizing the execution of Java programs. In *Proc. ACM 2001 Symposium on Software Visualization (SOFTVIS)*, pages 151–162. ACM Press. (Cited on pages 15, 38, 44, 67, and 157.)

De Pauw, W., Kimelman, D., and Vlissides, J. M. (1994). Modeling object-oriented program execution. In *Proc. European Object-Oriented Programming Conference (ECOOP)*, pages 163–182. Springer. (Cited on pages 14 and 157.)

De Pauw, W., Krasikov, S., and Morar, J. F. (2006). Execution patterns for visualizing web services. In *Proc. ACM 2006 Symposium on Software Visualization (SOFTVIS)*, pages 37–45. ACM Press. (Cited on page 157.)

De Pauw, W., Lorenz, D., Vlissides, J., and Wegman, M. (1998). Execution patterns in object-oriented visualization. In *Proc. USENIX Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 219–234. USENIX. (Cited on pages 14, 38, 67, 73, and 122.)

De Roover, C., Michiels, I., Gybels, K., Gybels, K., and D'Hondt, T. (2006). An approach to high-level behavioral program documentation allowing lightweight verification. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 202–211. IEEE Computer Society Press. (Cited on page 157.)

de Sousa, F. C., Mendonça, N. C., Uchitel, S., and Kramer, J. (2007). Detecting implied scenarios from execution traces. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 50–59. IEEE Computer Society Press. (Cited on page 158.)

Demeyer, S., Ducasse, S., and Nierstrasz, O. (2003). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann. (Cited on pages 30, 37, and 38.)

Deprez, J. and Lakhotia, A. (2000). A formalism to automate mapping from program features to code. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 69–78. IEEE Computer Society Press. (Cited on page 156.)

Di Penta, M., Stirewalt, R. E. K., and Kraemer, E. (2007). Designing your next empirical study on program comprehension. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 281–285. IEEE Computer Society Press. (Cited on pages 5, 133, and 153.)

Ducasse, S., Lanza, M., and Bertuli, R. (2004). High-level polymetric views of condensed run-time information. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 309–318. IEEE Computer Society Press. (Cited on pages 7, 123, and 156.)

Dugerdil, P. (2007). Using trace sampling techniques to identify dynamic clusters of classes. In *Proc. Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 306–314. (Cited on page 73.)

Dyba, T. and Dingsøyr, T. (2008). Empirical studies of agile software development: A systematic review. *Information & Software Technology*, 50(9-10):833–859. (Cited on page 17.)

Eaddy, M., Aho, A. V., Antoniol, G., and Guéhéneuc, Y.-G. (2008). CERBERUS: Tracing requirements to source code using information retrieval, dynamic analysis, and program analysis. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 53–62. IEEE Computer Society Press. (Cited on pages 32 and 156.)

Edwards, D., Simmons, S., and Wilde, N. (2006). An approach to feature location in distributed systems. *Journal of Systems & Software*, 79(1):457–474. (Cited on page 156.)

Eisenbarth, T., Koschke, R., and Simon, D. (2001). Feature-driven program understanding using concept analysis of execution traces. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 300–309. IEEE Computer Society Press. (Cited on page 106.)

Eisenbarth, T., Koschke, R., and Simon, D. (2003). Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224. (Cited on pages 15, 106, 121, and 156.)

Eisenberg, A. D. and Volder, K. D. (2005). Dynamic feature traces: finding features in unfamiliar code. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 337–346. IEEE Computer Society Press. (Cited on page 156.)

El-Ramly, M., Stroulia, E., and Sorenson, P. G. (2002). From run-time behavior to usage scenarios: an interaction-pattern mining approach. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 315–324. ACM Press. (Cited on page 156.)

Ernst, M. D. (2003). Static and dynamic analysis: synergy and duality. In *Proc. ICSE International Workshop on Dynamic Analysis (WODA)*, pages 25–28. IEEE Computer Society Press. (Cited on page 27.)

Fischer, M., Oberleitner, J., Gall, H., and Gschwind, T. (2005). System evolution tracking through execution trace analysis. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 237–246. IEEE Computer Society Press. (Cited on page 156.)

Fisher II, M., Elbaum, S. G., and Rothermel, G. (2007). Dynamic characterization of web application interfaces. In *Proc. International Conference on Fundamental Approaches to Software Engineering*, pages 260–275. Springer. (Cited on page 156.)

Fjeldstad, R. K. and Hamlen, W. T. (1979). Application program maintenance study: Report to our respondents. In *Proc. GUIDE*, volume 48. (Cited on pages 2 and 11.)

Forward, A. and Lethbridge, T. C. (2002). The relevance of software documentation, tools and technologies: a survey. In *Proc. Symposium on Document Engineering*, pages 26–33. ACM Press. (Cited on pages 37 and 38.)

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley. (Cited on pages 49 and 117.)

Gargiulo, J. and Mancoridis, S. (2001). Gadget: A tool for extracting the dynamic structure of Java programs. In *Proc. International Conference on Software Engineering & Knowledge Engineering (SEKE)*, pages 244–251. (Cited on pages 73 and 156.)

Gold, N., Mohan, A., Knight, C., and Munro, M. (2004). Understanding service-oriented software. *IEEE Software*, 21(2):71–77. (Cited on page 29.)

Greevy, O. (2007). *Enriching Reverse Engineering with Feature Analysis*. PhD thesis, Universität Bern. (Cited on pages 5 and 16.)

Greevy, O., Ducasse, S., and Gîrba, T. (2005). Analyzing feature traces to incorporate the semantics of change in software evolution analysis. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 347–356. IEEE Computer Society Press. (Cited on pages 15 and 156.)

Greevy, O., Ducasse, S., and Gîrba, T. (2006a). Analyzing software evolution through feature views. *Journal of Software Maintenance & Evolution*, 18(6):425–456. (Cited on pages 15 and 156.)

Greevy, O., Lanza, M., and Wysseier, C. (2006b). Visualizing live software systems in 3D. In *Proc. ACM 2006 Symposium on Software Visualization (SOFTVIS)*, pages 47–56. ACM Press. (Cited on pages 123 and 156.)

Gschwind, T., Oberleitner, J., and Pinzger, M. (2003). Using run-time data for program comprehension. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 245–250. IEEE Computer Society Press. (Cited on page 156.)

Guéhéneuc, Y.-G. (2004). A reverse engineering tool for precise class diagrams. In *Proc. Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 28–41. IBM Press. (Cited on page 156.)

Guéhéneuc, Y.-G., Douence, R., and Jussien, N. (2002). No Java without Caffeine: A tool for dynamic analysis of Java programs. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 117–126. IEEE Computer Society Press. (Cited on page 156.)

Hamou-Lhadj, A., Braun, E., Amyot, D., and Lethbridge, T. C. (2005). Recovering behavioral design models from execution traces. In *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*, pages 112–121. IEEE Computer Society Press. (Cited on pages 73 and 124.)

Hamou-Lhadj, A. and Lethbridge, T. C. (2002). Compression techniques to simplify the analysis of large execution traces. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 159–168. IEEE Computer Society Press. (Cited on pages 56 and 73.)

Hamou-Lhadj, A. and Lethbridge, T. C. (2004). A survey of trace exploration tools and techniques. In *Proc. Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 42–55. IBM Press. (Cited on pages 5, 15, 16, 68, and 156.)

Hamou-Lhadj, A. and Lethbridge, T. C. (2005). Measuring various properties of execution traces to help build better trace analysis tools. In *Proc. International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 559–568. IEEE Computer Society Press. (Cited on page 90.)

Hamou-Lhadj, A. and Lethbridge, T. C. (2006). Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 181–190. IEEE Computer Society Press. (Cited on pages 15, 31, 73, 79, 127, 128, 145, and 156.)

Hamou-Lhadj, A., Lethbridge, T. C., and Fu, L. (2004). Challenges and requirements for an effective trace exploration tool. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 70–78. IEEE Computer Society Press. (Cited on pages 15 and 156.)

Hendrickson, S. A., Dashofy, E. M., and Taylor, R. N. (2005). An architecture-centric approach for tracing, organizing, and understanding events in event-based software architectures. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 227–236. IEEE Computer Society Press. (Cited on page 156.)

Heuzeroth, D., Holl, T., Högström, G., and Löwe, W. (2003). Automatic design pattern detection. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 94–103. IEEE Computer Society Press. (Cited on pages 15, 28, and 156.)

Heuzeroth, D., Holl, T., and Löwe, W. (2002). Combining static and dynamic analyses to detect interaction patterns. In *Proc. International Conference on Integrated Design & Process Technology (IDPT)*. Society for Design and Process Science. (Cited on page 15.)

Holten, D. (2006). Hierarchical Edge Bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization & Computer Graphics*, 12(5):741–748. (Cited on pages 7, 94, and 150.)

Holten, D., Cornelissen, B., and van Wijk, J. J. (2007). Visualizing execution traces using Hierarchical Edge Bundles. In *Proc. International Workshop on Visualizing Software for Understanding & Analysis (VISSOFT)*, pages 47–54. IEEE Computer Society Press. (Cited on pages 10 and 99.)

Huang, H., Zhang, S., Cao, J., and Duan, Y. (2005). A practical pattern recovery approach based on both structural and behavioral analysis. *Journal of Systems & Software*, 75(1-2):69–87. (Cited on page 156.)

Israr, T., Woodside, M., and Franks, G. (2007). Interaction tree algorithms to extract effective architecture and layered performance models from traces. *Journal of Systems & Software*, 80(4):474–492. (Cited on pages 28 and 156.)

Jacobson, I. (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley. (Cited on pages 14 and 122.)

Jerding, D. F. and Rugaber, S. (1997). Using visualization for architectural localization and extraction. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 56–65. IEEE Computer Society Press. (Cited on pages 14 and 156.)

Jerding, D. F. and Stasko, J. T. (1998). The information mural: A technique for displaying and navigating large information spaces. *IEEE Transactions on Visualization & Computer Graphics*, 4(3):257–271. (Cited on pages 7, 14, 22, 95, and 129.)

Jerding, D. F., Stasko, J. T., and Ball, T. (1997). Visualizing interactions in program executions. In *Proc. International Conference on Software Engineering (ICSE)*, pages 360–370. ACM Press. (Cited on pages 14, 67, 99, 122, 127, and 128.)

Jiang, J., Koskinen, J., Ruokonen, A., and Systä, T. (2007). Constructing usage scenarios for API redocumentation. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 259–264. IEEE Computer Society Press. (Cited on page 156.)

Jiang, M., Groble, M., Simmons, S., Edwards, D., and Wilde, N. (2006). Software feature understanding in an industrial setting. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 66–67. IEEE Computer Society Press. (Cited on page 156.)

Jiang, Z. M., Hassan, A., Hamann, G., and Flora, P. (2008). An automated approach for abstracting execution logs to execution events. *Journal of Software Maintenance & Evolution*, 20(4):249–267. (Cited on page 156.)

Joshi, S. and Orso, A. (2007). SCARPE: A technique and tool for selective record and replay of program executions. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 234–243. IEEE Computer Society Press. (Cited on page 31.)

Kelsen, P. (2004). A simple static model for understanding the dynamic behavior of programs. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 46–51. IEEE Computer Society Press. (Cited on page 156.)

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. (2001). An overview of AspectJ. In *Proc. European Conference on Object-Oriented Programming*, pages 327–353. Springer. (Cited on pages 2 and 42.)

Kitchenham, B. A. (2004). Procedures for performing systematic reviews. In *Technical Report TR/SE-0401, Keele University, and Technical Report 0400011T.1, National ICT Australia*. (Cited on pages 18, 33, and 34.)

Kleyn, M. F. and Gingrich, P. C. (1988). Graphtrace - understanding object-oriented systems using concurrently animated views. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 191–205. ACM Press. (Cited on pages 14 and 156.)

Koenemann, J. and Robertson, S. P. (1991). Expert problem solving strategies for program comprehension. In *Proc. SIGCHI Conference on Human Factors in Computing Systems*, pages 125–130. ACM Press. (Cited on page 3.)

Kollmann, R. and Gogolla, M. (2001a). Capturing dynamic program behaviour with UML, collaboration diagrams. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 58–67. IEEE Computer Society Press. (Cited on page 67.)

Kollmann, R. and Gogolla, M. (2001b). Capturing dynamic program behaviour with UML collaboration diagrams. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 58–67. IEEE Computer Society Press. (Cited on page 156.)

Kollmann, R., Selonen, P., Stroulia, E., Systä, T., and Zündorf, A. (2002). A study on the current state of the art in tool-supported UML-based static reverse engineering. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 22–32. IEEE Computer Society Press. (Cited on page 67.)

Korel, B. and Rilling, J. (1998). Program slicing in understanding of large programs. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 145–152. IEEE Computer Society Press. (Cited on page 156.)

Koschke, R. and Quante, J. (2005). On dynamic feature location. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 86–95. ACM Press. (Cited on page 157.)

Koskimies, K. and Mössenböck, H. (1996). Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proc. International Conference on Software Engineering (ICSE)*, pages 366–375. IEEE Computer Society Press. (Cited on pages 6, 14, 33, 128, and 157.)

Koskinen, J., Kettunen, M., and Systä, T. (2006). Profile-based approach to support comprehension of software behavior. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 212–224. IEEE Computer Society Press. (Cited on pages 15 and 157.)

Kothari, J., Denton, T., Shokoufandeh, A., and Mancoridis, S. (2007). Reducing program comprehension effort in evolving software by recognizing feature implementation convergence. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 17–26. IEEE Computer Society Press. (Cited on pages 15 and 157.)

Kuhn, A. and Greevy, O. (2006). Exploiting the analogy between traces and signal processing. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 320–329. IEEE Computer Society Press. (Cited on pages 73, 75, 106, 123, and 157.)

Lange, C. F. J. and Chaudron, M. R. V. (2007). Interactive views to improve the comprehension of UML models - an experimental validation. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 221–230. IEEE Computer Society Press. (Cited on page 129.)

Lange, D. and Nakamura, Y. (1997). Object-oriented program tracing and visualization. *IEEE Computer*, 30(5):63–70. (Cited on page 157.)

Lange, D. B. and Nakamura, Y. (1995a). Interactive visualization of design patterns can help in framework understanding. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 342–357. ACM Press. (Cited on pages 14 and 157.)

Lange, D. B. and Nakamura, Y. (1995b). Program Explorer: A program visualizer for C++. In *Proc. USENIX Conference on Object-Oriented Technologies & Systems (COOTS)*, pages 39–54. USENIX. (Cited on pages 14 and 122.)

LaToza, T. D., Venolia, G., and DeLine, R. (2006). Maintaining mental models: a study of developer work habits. In *Proc. International Conference on Software Engineering (ICSE)*, pages 492–501. ACM Press. (Cited on pages 71 and 91.)

Lehman, M. M. and Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press. (Cited on page 1.)

Li, M., Chen, X., Li, X., Ma, B., and Vitányi, P. (2004). The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264. (Cited on page 78.)

Licata, D. R., Harris, C. D., and Krishnamurthi, S. (2003). The feature signatures of evolving programs. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 281–285. IEEE Computer Society Press. (Cited on page 157.)

Lienhard, A., Greevy, O., and Nierstrasz, O. (2007). Tracking objects to detect feature dependencies. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 59–68. IEEE Computer Society Press. (Cited on page 157.)

Lientz, B. P. and Swanson, E. B. (1980). *Software Maintenance Management*. Addison-Wesley. (Cited on page 1.)

Liu, D., Marcus, A., Poshyvanyk, D., and Rajlich, V. (2007). Feature location via information retrieval based filtering of a single scenario execution trace. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 234–243. ACM Press. (Cited on pages 26 and 157.)

Lo, D. and Khoo, S. (2006a). QUARK: Empirical assessment of automaton-based specification miners. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 51–60. IEEE Computer Society Press. (Cited on page 157.)

Lo, D. and Khoo, S. (2006b). SMArTIC: Towards building an accurate, robust and scalable specification miner. In *Proc. European Software Engineering Conference & ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 265–275. ACM Press. (Cited on page 157.)

Lo, D., Khoo, S., and Liu, C. (2008). Mining temporal rules for software maintenance. *Journal of Software Maintenance & Evolution*, 20(4):227–247. (Cited on pages 15, 33, and 157.)

Lucca, G. A. D., Fasolino, A. R., Tramontana, P., and de Carlini, U. (2003). Abstracting business level UML diagrams from web applications. In *Proc. International Workshop on Web Site Evolution (WSE)*, pages 12–19. IEEE Computer Society Press. (Cited on page 157.)

Lukoit, K., Wilde, N., Stowell, S., and Hennessey, T. (2000). TraceGraph: Immediate visual location of software features. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 33–39. IEEE Computer Society Press. (Cited on page 157.)

Maletic, J. I., Marcus, A., and Collard, M. L. (2002). A task oriented view of software visualization. In *Proc. International Workshop on Visualizing Software for Understanding & Analysis (VISSOFT)*, pages 32–40. IEEE Computer Society Press. (Cited on page 92.)

Marick, B. (2004). Agile methods and agile testing. http://testing.com/agile/agile-testing-essay.html (accessed October 9th, 2006). (Cited on pages 37 and 38.)

Martin, L., Giesl, A., and Martin, J. (2002). Dynamic component program visualization. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 289–298. IEEE Computer Society Press. (Cited on page 157.)

Mesbah, A., Bozdag, E., and van Deursen, A. (2008). Crawling Ajax by inferring user interface state changes. In *Proc. International Conference on Web Engineering (ICWE)*, pages 122–134. IEEE Computer Society Press. (Cited on page 29.)

Miller, K. W. (2004). Test driven development on the cheap: text files and explicit scaffolding. *Journal of Computing Sciences in Colleges*, 20(2):181–189. (Cited on pages 37 and 38.)

Moe, J. and Carr, D. A. (2001). Understanding distributed systems via execution trace data. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 60–67. IEEE Computer Society Press. (Cited on page 29.)

Moe, J. and Sandahl, K. (2002). Using execution trace data to improve distributed systems. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 640–648. IEEE Computer Society Press. (Cited on page 157.)

Murphy, G. C., Notkin, D., and Sullivan, K. J. (2001). Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380. (Cited on pages 14, 119, and 123.)

North, C. (2006). Toward measuring visualization insight. *IEEE Computer Graphics and Applications*, 26(3):6–9. (Cited on page 131.)

Oechsle, R. and Schmitt, T. (2001). JAVAVIS: Automatic program visualization with object and sequence diagrams using the Java Debug Interface (JDI). In *Proc. ACM 2001 Symposium on Software Visualization (SOFTVIS)*, pages 176–190. ACM Press. (Cited on page 157.)

OMG (2003). UML 2.0 infrastructure specification. Object Management Group, http://www.omg.org/. (Cited on pages 6, 38, and 45.)

Pacione, M. J. (2005). *A Novel Software Visualisation Model to Support Object-Oriented Program Comprehension*. PhD thesis, University of Strathclyde. (Cited on page 16.)

Pacione, M. J., Roper, M., and Wood, M. (2003). Comparative evaluation of dynamic visualisation tools. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 80–89. IEEE Computer Society Press. (Cited on pages 5, 16, 68, and 157.)

Pacione, M. J., Roper, M., and Wood, M. (2004). A novel software visualisation model to support software comprehension. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 70–79. IEEE Computer Society Press. (Cited on pages 132, 142, 144, and 157.)

Pankratius, V., Schaefer, C., Jannesari, A., and Tichy, W. F. (2008). Software engineering for multicore systems: an experience report. In *Proc. ICSE International Workshop on Multicore Software Engineering (IWMSE)*. ACM Press. (Cited on page 29.)

Pheng, S. and Verbrugge, C. (2006). Dynamic data structure analysis for Java programs. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 191–201. IEEE Computer Society Press. (Cited on page 157.)

Pigoski, T. M. (1997). *Practical software maintenance. Best practices for managing your investment*. Wiley & Sons. (Cited on pages 2 and 11.)

Poshyvanyk, D., Guéhéneuc, Y.-G., Marcus, A., Antoniol, G., and Rajlich, V. (2007). Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432. (Cited on pages 16, 32, 106, and 157.)

Qingshan, L., Chu, H., Hu, S., Chen, P., and Yun, Z. (2005). Architecture recovery and abstraction from the perspective of processes. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 57–66. IEEE Computer Society Press. (Cited on page 157.)

Quante, J. (2008). Do dynamic object process graphs support program understanding? – a controlled experiment. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 73–82. IEEE Computer Society Press. (Cited on pages 31, 129, 139, 144, and 157.)

Quante, J. and Koschke, R. (2007). Dynamic protocol recovery. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 219–228. IEEE Computer Society Press. (Cited on pages 15, 33, and 157.)

Reiss, S. P. (2003a). Event-based performance analysis. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 74–83. IEEE Computer Society Press. (Cited on page 157.)

Reiss, S. P. (2003b). Visualizing Java in action. In *Proc. ACM 2003 Symposium on Software Visualization (SOFTVIS)*, pages 57–65. ACM Press. (Cited on pages 31 and 123.)

Reiss, S. P. (2006). Visualizing program execution using user abstractions. In *Proc. ACM 2006 Symposium on Software Visualization (SOFTVIS)*, pages 125–134. ACM Press. (Cited on page 157.)

Reiss, S. P. (2007). Visual representations of executing programs. *Journal of Visual Languages & Computing*, 18(2):126–148. (Cited on pages 5, 16, and 157.)

Reiss, S. P. and Renieris, M. (2001). Encoding program executions. In *Proc. International Conference on Software Engineering (ICSE)*, pages 221–230. IEEE Computer Society Press. (Cited on pages 6, 15, 72, 73, 123, 127, 128, and 157.)

Renieris, M. and Reiss, S. P. (1999). Almost: Exploring program traces. In *Proc. Workshop on New Paradigms in Information Visualization & Manipulation*, pages 70–77. ACM Press. (Cited on pages 91 and 157.)

Richner, T. and Ducasse, S. (1999). Recovering high-level views of object-oriented applications from static and dynamic information. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 13–22. IEEE Computer Society Press. (Cited on pages 11, 14, and 157.)

Richner, T. and Ducasse, S. (2002). Using dynamic information for the iterative recovery of collaborations and roles. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 34–43. IEEE Computer Society Press. (Cited on pages 15, 123, and 157.)

Rilling, J. (2001). Maximizing functional cohesion of comprehension environments by integrating user and task knowledge. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 157–165. IEEE Computer Society Press. (Cited on page 157.)

Ritsch, H. and Sneed, H. M. (1993). Reverse engineering programs via dynamic analysis. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 192–201. IEEE Computer Society Press. (Cited on page 157.)

Riva, C. (2000). Reverse architecting: An industrial experience report. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 42–50. IEEE Computer Society Press. (Cited on page 157.)

Riva, C. and Rodriguez, J. V. (2002). Combining static and dynamic views for architecture reconstruction. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 47–55. IEEE Computer Society Press. (Cited on pages 15, 44, 68, and 73.)

Riva, C. and Yang, Y. (2002). Generation of architectural documentation using XML. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 161–179. IEEE Computer Society Press. (Cited on pages 15 and 157.)

Rohatgi, A., Hamou-Lhadj, A., and Rilling, J. (2008). An approach for mapping features to code based on static and dynamic analysis. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 236–241. IEEE Computer Society Press. (Cited on page 157.)

Röthlisberger, D., Greevy, O., and Nierstrasz, O. (2008). Exploiting runtime information in the IDE. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 63–72. IEEE Computer Society Press. (Cited on pages 30, 31, and 157.)

Rountev, A. and Connell, B. H. (2005). Object naming analysis for reverse-engineered sequence diagrams. In *Proc. International Conference on Software Engineering (ICSE)*, pages 254–263. ACM Press. (Cited on pages 44, 67, and 76.)

Rountev, A., Volgin, O., and Reddoch, M. (2005). Static control-flow analysis for reverse engineering of UML sequence diagrams. In *Proc. Workshop on Program Analysis for Software Tools & Engineering (PASTE)*, pages 96–102. ACM Press. (Cited on page 67.)

Rumbaugh, J., Jacobson, I., and Booch, G. (1998). *The Unified Modeling Language Reference Manual*. Addison-Wesley. (Cited on page 38.)

Safyallah, H. and Sartipi, K. (2006). Dynamic analysis of software systems using execution pattern mining. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 84–88. IEEE Computer Society Press. (Cited on pages 73 and 157.)

Salah, M. and Mancoridis, S. (2003). Toward an environment for comprehending distributed systems. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 238–247. IEEE Computer Society Press. (Cited on page 157.)

Salah, M. and Mancoridis, S. (2004). A hierarchy of dynamic software views: From object-interactions to feature-interactions. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 72–81. IEEE Computer Society Press. (Cited on pages 73 and 157.)

Salah, M., Mancoridis, S., Antoniol, G., and Di Penta, M. (2006). Scenario-driven dynamic analysis for comprehending large software systems. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 71–80. IEEE Computer Society Press. (Cited on page 158.)

Sartipi, K. and Dezhkam, N. (2007). An amalgamated dynamic and static architecture reconstruction framework to control component interactions. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 259–268. IEEE Computer Society Press. (Cited on page 158.)

Schmerl, B. R., Aldrich, J., Garlan, D., Kazman, R., and Yan, H. (2006). Discovering architectures from running systems. *IEEE Transactions on Software Engineering*, 32(7):454–466. (Cited on pages 11, 15, 28, and 158.)

Schwaber, K. and Beedle, M. (2001). *Agile Software Development with Scrum*. Prentice Hall. (Cited on page 37.)

Sefika, M., Sane, A., and Campbell, R. H. (1996). Architecture-oriented visualization. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 389–405. ACM Press. (Cited on pages 14, 28, and 158.)

Sharp, R. and Rountev, A. (2005). Interactive exploration of UML sequence diagrams. In *Proc. International Workshop on Visualizing Software for Understanding*

& Analysis (VISSOFT), pages 8–15. IEEE Computer Society Press. (Cited on page 44.)

Shevertalov, M. and Mancoridis, S. (2007). A reverse engineering tool for extracting protocols of networked applications. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 229–238. IEEE Computer Society Press. (Cited on page 158.)

Shneiderman, B. (1996). The eyes have it: A task by data type taxonomy for information visualizations. In *Proc. Symposium on Visual Languages (VL)*, pages 336–343. IEEE Computer Society Press. (Cited on page 121.)

Sim, S. E., Easterbrook, S. M., and Holt, R. C. (2003). Using benchmarking to advance research: A challenge to software engineering. In *Proc. International Conference on Software Engineering (ICSE)*, pages 74–83. IEEE Computer Society Press. (Cited on pages 8, 32, and 74.)

Simmons, S., Edwards, D., Wilde, N., Homan, J., and Groble, M. (2006). Industrial tools for the feature location problem: an exploratory study. *Journal of Software Maintenance & Evolution*, 18(6):457–474. (Cited on pages 31 and 158.)

Sjøberg, D. I. K., Hannay, J. E., Hansen, O., Kampenes, V. B., Karahasanovic, A., Liborg, N.-K., and Rekdal, A. C. (2005). A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753. (Cited on page 17.)

Smit, M., Stroulia, E., and Wong, K. (2008). Use case redocumentation from gui event traces. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 263–268. IEEE Computer Society Press. (Cited on page 158.)

Souder, T. S., Mancoridis, S., and Salah, M. (2001). Form: A framework for creating views of program executions. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 612–620. IEEE Computer Society Press. (Cited on page 158.)

Staples, M. and Niazi, M. (2007). Experiences using systematic review guidelines. *Journal of Systems & Software*, 80(9):1425–1437. (Cited on page 34.)

Stapleton, J. (1997). *Dynamic Systems Development Method: The method in practice*. Addison-Wesley. (Cited on page 37.)

Storey, M.-A. (2005). Theories, methods and tools in program comprehension: past, present and future. In *Proc. International Workshop on Program Comprehension (IWPC)*, pages 181–191. IEEE Computer Society Press. (Cited on pages 129 and 150.)

Stroulia, E., El-Ramly, M., Kong, L., Sorenson, P. G., and Matichuk, B. (1999). Reverse engineering legacy interfaces: An interaction-driven approach. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 292–302. IEEE Computer Society Press. (Cited on page 158.)

Systä, T., Koskimies, K., and Müller, H. A. (2001). Shimba: an environment for reverse engineering Java software systems. *Software: Practice & Experience*, 31(4):371–394. (Cited on pages 15, 44, 67, 73, 123, 128, and 158.)

Tonella, P. and Potrich, A. (2002). Static and dynamic C++ code analysis for the recovery of the object diagram. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 54–63. IEEE Computer Society Press. (Cited on page 158.)

van Deursen, A. (2001). Program comprehension risks and benefits in extreme programming. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 176–185. IEEE Computer Society Press. (Cited on page 37.)

Van Geet, J. and Zaidman, A. (2006). A lightweight approach to determining the adequacy of tests as documentation. In *Proc. International Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 21–26. (Cited on page 70.)

Van Rompaey, B. and Demeyer, S. (2008). Estimation of test code changes using historical release data. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 269–278. IEEE Computer Society Press. (Cited on page 131.)

Vasconcelos, A., Cepêda, R., and Werner, C. (2005). An approach to program comprehension through reverse engineering of complementary software views. In *Proc. Workshop on Program Comprehension through Dynamic Analysis (PCODA)*, pages 58–62. Tech. report 2005-12, University of Antwerp. (Cited on page 73.)

Voets, R. (2008). JRET: A tool for the reconstruction of sequence diagrams from program executions. Master's thesis, Delft University of Technology. (Cited on pages 10, 54, 58, 59, and 65.)

von Mayrhauser, A. and Vans, A. M. (1995). Program comprehension during software maintenance and evolution. *IEEE Computer*, 28(8):44–55. (Cited on pages 71 and 110.)

Walker, R. J., Murphy, G. C., Freeman-Benson, B. N., Wright, D., Swanson, D., and Isaak, J. (1998). Visualizing dynamic software system information through high-level models. In *Proc. Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA)*, pages 271–283. ACM Press. (Cited on pages 11, 14, and 122.)

Walker, R. J., Murphy, G. C., Steinbok, J., and Robillard, M. P. (2000). Efficient mapping of software system traces to architectural views. In *Proc. Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 12–21. IBM Press. (Cited on page 158.)

Walkinshaw, N., Bogdanov, K., Holcombe, M., and Salahuddin, S. (2008). Reverse engineering state machines by interactive grammar inference. *Journal of Software Maintenance & Evolution*, 20(4):269–290. (Cited on pages 15 and 158.)

Wang, L., Cordy, J. R., and Dean, T. R. (2005). Enhancing security using legality assertions. In *Proc. Working Conference on Reverse Engineering (WCRE)*, pages 35–44. IEEE Computer Society Press. (Cited on page 158.)

Wilde, N., Buckellew, M., Page, H., and Rajlich, V. (2001). A case study of feature location in unstructured legacy fortran code. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 68–76. IEEE Computer Society Press. (Cited on page 14.)

Wilde, N., Buckellew, M., Page, H., Rajlich, V., and Pounds, L. (2003). A comparison of methods for locating features in legacy software. *Journal of Systems & Software*, 65(2):105–114. (Cited on pages 14, 31, and 32.)

Wilde, N. and Casey, C. (1996). Early field experience with the Software Reconnaissance technique for program comprehension. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 312–318. IEEE Computer Society Press. (Cited on page 14.)

Wilde, N., Gomez, J. A., Gust, T., and Strasburg, D. (1992). Locating user functionality in old code. In *Proc. International Conference on Software Maintenance (ICSM)*, pages 200–205. IEEE Computer Society Press. (Cited on pages 106 and 110.)

Wilde, N. and Scully, M. C. (1995). Software Reconnaissance: Mapping program features to code. *Journal of Software Maintenance*, 7(1):49–62. (Cited on pages 14, 15, and 158.)

Wohlin, C., Runeson, P., Höst, M., Ohlesson, M. C., Regnell, B., and Wesslen, A. (2000). *Experimentation in software engineering - an introduction*. Kluwer Acad. Publ. (Cited on pages 5 and 135.)

Wong, W. E. and Gokhale, S. S. (2005). Static and dynamic distance metrics for feature-based code analysis. *Journal of Systems & Software*, 74(3):283–295. (Cited on page 158.)

Wong, W. E., Gokhale, S. S., and Horgan, J. R. (2000). Quantifying the closeness between program components and features. *Journal of Systems & Software*, 54(2):87–98. (Cited on page 15.)

Xu, G., Rountev, A., Tang, Y., and Qin, F. (2007). Efficient checkpointing of Java software using context-sensitive capture and replay. In *Proc. European Software Engineering Conference & ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 85–94. ACM Press. (Cited on page 31.)

Yan, H., Garlan, D., Schmerl, B. R., Aldrich, J., and Kazman, R. (2004). Discotect: A system for discovering architectures from running systems. In *Proc. International Conference on Software Engineering (ICSE)*, pages 470–479. IEEE Computer Society Press. (Cited on page 15.)

Yang, S., Burnett, M. M., DeKoven, E., and Zloof, M. (1997). Representation design benchmarks: a design-time aid for VPL navigable static representations. *Journal of Visual Languages & Computing*, 8(5-6):563–599. (Cited on pages 41, 92, and 95.)

Zaidman, A. (2006). *Scalability Solutions for Program Comprehension through Dynamic Analysis*. PhD thesis, University of Antwerp. (Cited on pages 3, 31, 38, 42, 71, and 92.)

Zaidman, A., Calders, T., Demeyer, S., and Paredaens, J. (2005). Applying webmining techniques to execution traces to support the program comprehension process. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 134–142. IEEE Computer Society Press. (Cited on pages 42, 44, 73, 80, and 92.)

Zaidman, A. and Demeyer, S. (2004). Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 329–338. IEEE Computer Society Press. (Cited on pages 27, 73, 123, 128, and 158.)

Zaidman, A. and Demeyer, S. (2008). Automatic identification of key classes in a software system using webmining techniques. *Journal of Software Maintenance & Evolution*, 20(6):387–417. (Cited on page 158.)

Zaidman, A., Demeyer, S., Adams, B., Schutter, K. D., Hoffman, G., and Ruyck, B. D. (2006). Regaining lost knowledge through dynamic analysis and aspect orientation. In *Proc. European Conference on Software Maintenance & Reengineering (CSMR)*, pages 91–102. IEEE Computer Society Press. (Cited on pages 30 and 158.)

Zaidman, A., Van Rompaey, B., Demeyer, S., and van Deursen, A. (2008). Mining software repositories to study co-evolution of production & test code. In *Proc. International Conference on Software Testing (ICST)*, pages 220–229. IEEE Computer Society Press. (Cited on page 131.)

Summary

Evaluating Dynamic Analysis Techniques for Program Comprehension
Bas Cornelissen

Introduction

Software evolution has become an increasingly important aspect of the software development process. As software systems grow larger and their development becomes more expensive, they are constantly modified rather than rebuilt from scratch. As a result, a great deal of resources is spent on performing software *maintenance*. An important prerequisite for maintenance is a sufficient level of understanding of the software at hand. Unfortunately, this knowledge is not always readily available because documentation is often lacking or outdated. Thus, the necessary knowledge must be gathered through the analysis of the software itself, also known known as *program comprehension*. This activity is the main focus of this dissertation.

Program comprehension typically comprises the study of static artifacts such as source code, also known as *static analysis*. However, as dealing with source code involves a mental mapping between a system's code and its functionality, it is often difficult to interpret code directly because of its size or complexity, and the maintainer may have difficulties in coping with the resulting cognitive overload. As a consequence, program comprehension is a rather time-consuming activity, and may take up to 60% of the time spent on software maintenance tasks.

Since program comprehension is so expensive, the development of techniques and tools that support this activity can significantly increase the overall efficiency of software development. While most of these techniques traditionally focus on static analysis, the use of *dynamic analysis* has been relatively underemphasized. Dynamic analysis studies a software system's actual behavior rather than its source code, and has the important advantage that it provides rather precise information on a system's runtime functionality. This type of analysis is central to this dissertation.

Challenges

The use of dynamic analysis for program comprehension poses several challenges. In this dissertation, we identify and address four such issues.

Existing literature. In the past decades, the field of program comprehension through dynamic analysis has witnessed the development of numerous techniques every year, which has led to a large research body with multiple subfields. The absence of a broad overview of the *state of the art* has made it increasingly difficult for researchers to put new techniques into perspective. An inventory and characterization of the research efforts to date would enable the comparison of existing work, and would assist the community in such tasks as finding related work and identifying new research opportunities.

Traditional visualization. Visualization is a popular means to convey information to the user. Among the traditional visualization techniques in our context are UML sequence diagrams, which have been proposed in the dynamic analysis literature on several occasions. Associated with this approach are three important issues. First, one must come up with appropriate execution scenarios for dynamic analysis data to be useful. Second, considering UML sequence diagrams are prone to scalability problems, the large amounts of data (typically running in the thousands of events) often cannot be visualized efficiently. Third, for lack of thorough empirical studies in the literature, there exists little convincing evidence of the applicability of such diagrams during actual program comprehension tasks. For these reasons, we argue that the use of UML sequence diagrams in this context should (1) involve efficient *abstraction techniques*, and (2) be rigorously evaluated.

Advanced visualization. In contrast to traditional visualizations, the literature also offers a number of *advanced* visualization techniques. Most of these visualizations are quite scalable, which renders them useful for dynamic analysis. On the other hand, the intended target audience may not be so easily accustomed to advanced visualizations as it is to traditional ones. Moreover, particularly on the subject of execution trace visualization, the existing work mostly provides anecdotal evidence of their usefulness, rather than evaluations that involve human subjects and actual comprehension tasks. Therefore, the focus of a novel visualization technique should not be merely on the development of the technique but also on its thorough empirical validation.

Evaluation. Finally, we observe that the *evaluation quality* of approaches on our topic could be subject to improvement. In order to raise the bar in this respect, our own manners of validation must meet certain standards. For example, we opt for multiple case studies that each involve different software systems and program comprehension activities. Moreover, if we are to compare our techniques to existing solutions and if fellow researchers are to compare their future techniques to ours, we need to introduce methodologies and reusable experimental designs to facilitate the comparison process.

Results & Implications

Literature survey. In this dissertation, we started off by reporting on a literature survey on program comprehension and dynamic analysis. The survey involved 172 articles that were selected from 4,795 articles on this topic and the references therein. The characterization of this research body resulted in an overview that is useful as a reference for researchers in the field of program comprehension through dynamic analysis, and that helps them identify related work and new research opportunities in their respective subfields.

Our study of the results revealed an apparent lack of comparisons and benchmarks in most subfields, and the fact that very few articles offer publicly available tools, which hinders the use of existing solutions in current developments and the adoption of such tools in industry. We also expressed our concerns about the lack of controlled experiments, which we feel are crucial in the field of program comprehension. Finally, we identified four types of target applications that have been underemphasized in literature, being web applications, distributed systems, multithreaded applications, and legacy systems. We stressed the importance of the above aspects, discussed the issues involved, and offered a series of potential solutions and recommendations.

Sequence diagram reconstruction. Our next step was the design of a framework for the reconstruction of UML sequence diagrams. We used test cases as execution scenarios, as these effectively decompose a system's functionality into tractable parts. The approach was initially implemented in our Sdr tool and was evaluated on a small subject system involving two concrete maintenance tasks. In close collaboration with an M.Sc. student we then proposed JRET, a more mature tool implementation featuring several key improvements, and evaluated it on a medium-scale open source application.

The use of test cases as scenarios and their visualization as sequence diagrams proved useful as a starting point for program comprehension. However, our approach makes extensive use of abstraction techniques to counter scalability issues, and we opted for a more thorough investigation into the abstractions used in literature because their effectiveness and applicability in practice remained unclear.

To enable side-by-side comparisons of the numerous abstraction techniques offered in literature, we proposed an assessment methodology that distinguishes six steps: context, criteria, metrics, test set, application, and interpretation. The use of our framework was demonstrated through an experiment that involved a test set of seven large traces – three of which were larger than one million events in size – and the implementation of four different abstraction techniques from the literature. The use of this framework enables the proper evaluation of abstraction techniques and provides a deeper understanding of the applicability of such techniques in specific contexts.

Extravis. Another manner in which we sought to tackle the scalability issues associated with dynamic analysis was the design and evaluation of more advanced visualizations, i.e., techniques not based on traditional visualizations such as trees and UML. To this end, we employed two novel views in the context of large execution traces. The circular bundle view hierarchically projects a program's structural entities on a circle and shows their interrelationships in a bundled fashion. The massive sequence view provides a navigable overview of a trace by chronologically visualizing each event as a colored line. The views are linked and fully interactive, fit on a single screen, and are implemented in a tool called Extravis, which is the result of a collaboration with visualization experts from Eindhoven University of Technology.

We initially evaluated the tool through a series of three case studies. The results provided a strong indication as to Extravis' benefits, the main advantages being its optimal use of screen real estate and the easy insight into a program's structure. However, we did hypothesize that the relationships in the circular view could be difficult to grasp: indeed, whereas the use of traditional visualizations such as UML is widespread, more advanced techniques warrant extensive validations to assess their understandability and usefulness.

For this reason, we conducted a *controlled experiment* to measure Extravis' added value to an existing solution, the Eclipse IDE. We designed eight typical tasks aimed at gaining an understanding of a representative subject system, and measured how a control group (using the Eclipse IDE) and an experimental group (using both Eclipse and Extravis) performed in terms of correctness and time spent. The results showed a statistically significant decrease in time and increase in correctness for the latter group.

The design and execution of this experiment have demonstrated the potential of trace visualization for program comprehension, and have paved the way for other researchers to conduct similar experiments.

Emphasis on evaluation quality. The evaluation quality has been an important point of interest in our experiments. This objective can be traced back to each of the core chapters in this dissertation.

Our literature survey concerned a systematic approach to ensure the reproducibility of the results. The use of explicit article selection criteria and the derivation of a reusable article characterization framework enables fellow researchers to replicate our findings and compare existing work. Both of these tasks involved pilot studies to reduce the risk of author bias.

The reconstruction of UML sequence diagrams was performed using two novel tools, which we evaluated in case studies. While the subject system in our initial study is rather small, the second study involved a more representative program. Both studies revolved around comprehension tasks that stem from realistic software maintenance problems. Our second tool was implemented as an Eclipse plugin and is publicly available online for fellow researchers to download and experiment with.

Our assessment framework for abstraction techniques is fully documented, and the traces and abstraction technique implementations used in our example experiment are publicly available online. This ensures the reproducibility of our findings, and supports fellow researchers in performing similar assessments of different abstraction techniques.

Extravis was thoroughly evaluated through a series of three extensive case studies. Each of these studies involved a distinct program comprehension context and a distinct subject system that is representative in terms of size, and of which one was industrial. Furthermore, the tool is documented and available online to enable its involvement in similar experiments.

Finally, our controlled experiment was designed to be reusable. We designed eight comprehension tasks that were based on a framework in the literature, and have taken several measures to reduce any bias as much as possible. We provided all the necessary information for the replication our experiment, which enables similar experiments to be conducted with different visualization techniques in the future.

Samenvatting

Evalueren van Dynamische Analysetechnieken voor Softwarebegrip
Bas Cornelissen

Introductie

Software-evolutie wordt een steeds belangrijker aspect van het softwareontwikkelingsproces. Naarmate softwaresystemen groter worden en hun ontwikkeling duurder wordt, worden zij niet langer telkens opnieuw gebouwd maar continu aangepast. Als gevolg hiervan wordt veel tijd en geld geïnvesteerd in het uitvoeren van *onderhoud*. Een belangrijke vereiste voor het plegen van softwareonderhoud is dat er voldoende kennis beschikbaar is van het systeem in kwestie. Helaas is deze informatie niet altijd voorhanden, bijvoorbeeld doordat documentatie vaak ontbreekt of is verouderd. De benodigde kennis zal in zulke gevallen moeten worden verkregen via een analyse van de software, een proces dat bekendstaat *softwarebegrip*. Dit is het centrale thema van dit proefschrift.

Softwarebegrip omvat meestal het bestuderen van statische artefacten zoals broncode, ook wel *statische analyse* geheten. Bij het interpreteren van broncode staat het maken van een vertaalslag van broncode naar functionaliteit centraal. Dit is een lastige taak: hoe omvangrijker en complexer een softwaresysteem is, des te moeizamer wordt het om deze vertaalslag te maken doordat de onderhoudsmonteur meer informatie moet verwerken dan mogelijk is. Om deze reden is softwarebegrip vaak een behoorlijk tijdrovende activiteit die tot 60% van de onderhoudsinspanningen kan vertegenwoordigen.

De hoge prijs van softwarebegrip betekent dat de ontwikkeling van ondersteunende technieken en gereedschappen het softwareontwikkelingsproces als geheel aanzienlijk kunnen versnellen. Hierbij wordt traditioneel gebruikgemaakt van statische analyse. Relatief onderbelicht is *dynamische analyse*, het bestuderen van het daadwerkelijke gedrag van een systeem in plaats van zijn broncode. Dynamische analyse heeft als belangrijk voordeel dat het zeer nauwkeurige informatie verschaft over functionaliteiten terwijl zij worden uitgevoerd. Het is deze soort analyse waarop we ons in dit proefschrift concentreren.

Uitdagingen

Het gebruik van dynamische analyse voor softwarebegrip brengt een aantal uitdagingen met zich mee. In dit proefschrift identificeren en adresseren we vier belangrijke aandachtspunten.

Bestaande literatuur. In de laatste jaren heeft het onderzoek naar softwarebegrip via dynamische analyse een sterke groei doorgemaakt. Dit heeft geleid tot een grote hoeveelheid literatuur waarin meerdere deelgebieden zijn te onderscheiden. Het ontbreken van een breed overzicht van de huidige *staat van de wetenschap* maakt het steeds moeilijker voor onderzoekers om nieuwe technieken in perspectief te plaatsen. Een inventarisatie en karakterisering van alle inspanningen tot nu toe zou onderzoekers in staat stellen om vergelijkingen te maken met bestaande oplossingen, en zou de onderzoeksgemeenschap ondersteunen bij zaken als het vinden van gerelateerde werken en het identificeren van nieuwe kansen.

Traditionele visualisatie. *Visualisatie* is een populaire manier om informatie over te dragen naar de gebruiker. Een van de traditionele oplossingen in dit verband zijn UML sequence diagrammen. Hoewel deze meermaals zijn toegepast in dynamische analysecontexten, zijn er drie belangrijke problemen. Ten eerste moeten er geschikte uitvoeringsscenario's worden gevonden om de dynamisch verkregen informatie nuttig te kunnen maken. Ten tweede wordt dynamische analyse veelal in verband gebracht met (zeer) grote hoeveelheden informatie, wat de visualisatiestap in sterke mate bemoeilijkt, zeker bij UML sequence diagrammen omdat deze toch al gevoelig zijn voor schaalbaarheidsproblemen. Ten derde ontbreekt er uitvoerig empirisch bewijsmateriaal voor de bruikbaarheid van dergelijke diagrammen (verkregen via dynamische analyse) tijdens typische softwareonderhoudstaken. Om deze redenen zou het gebruik van dynamisch verkregen UML sequence diagrammen gepaard moeten gaan met effectieve abstractietechnieken enerzijds, en rigoureuze evaluaties anderzijds.

Geavanceerde visualisatie. Naast traditionele visualisaties biedt de literatuur ook verscheidene *geavanceerde* visualisatietechnieken. Het merendeel hiervan is redelijk schaalbaar, wat hen in principe geschikt gemaakt voor dynamische analyse. Aan de andere kant gaan geavanceerde visualisaties gepaard met een zekere leercurve omdat het beoogde publiek er nog geen ervaring mee heeft. Bovendien vinden we in de literatuur voornamelijk indirect bewijsmateriaal van hun bruikbaarheid, en worden er zelden proefpersonen en echte softwareonderhoudstaken bij de evaluatie betrokken, vooral bij de visualisatie van *executie-traces* (een soort logboeken van het daadwerkelijke gedrag van een systeem). Het is dus zaak dat bij het bedenken van nieuwe typen visualisaties de focus niet alleen ligt bij hun ontwerp maar zeker ook bij hun uitvoerige empirische validatie.

Evaluatie. Tot slot vormt de *evaluatiekwaliteit* een belangrijk punt van aandacht in het kader van softwarebegrip via dynamische analyse. Om in dit opzicht de lat hoger te kunnen leggen zullen onze eigen validatiemethoden moeten voldoen aan zekere maatstaven. Zo streven wij ernaar om meerdere case studies uit te voeren waarin meerdere softwaresystemen en onderhoudstaken de revue passeren. Ook zullen we de mogelijkheid moeten bieden om onze technieken te vergelijken met bestaande oplossingen en vice versa. Bij het verwezenlijken van deze ambities is er een behoefte aan herbruikbare experimentele ontwerpen en methodologieën.

Resultaten & Implicaties

Literatuuronderzoek. De aanvankelijke aandacht in dit proefschrift was gevestigd op een literatuuronderzoek. Dit onderzoek betrof 172 wetenschappelijke publicaties die werden geselecteerd uit 4795 artikels op het gebied van softwarebegrip via dynamische analyse. Onze karakterisering van deze literatuur heeft geresulteerd in een beknopt overzicht dat dient als een referentiekader voor onderzoekers op dit gebied, en dat hen ondersteunt bij het zoeken naar gerelateerde onderzoeken en bij het vinden van onderzoekskansen in hun respectievelijke deelgebieden.

Het bestuderen van de resultaten heeft een duidelijk tekort aan vergelijkingen en benchmarks blootgelegd. Noemenswaardig was ook dat de literatuur zelden publiek beschikbare gereedschappen biedt, wat niet alleen het gebruik van dergelijke oplossingen in een onderzoekscontext bemoeilijkt, maar zeker ook het uitproberen ervan door het bedrijfsleven. Verder hebben we onze zorgen uitgesproken over het beperkte aantal gecontroleerde experimenten, die juist cruciaal zijn als het aankomt op softwarebegrip. Tot slot hebben we vier typen applicaties en platformen onderscheiden, t.w. web-applicaties, gedistribueerde systemen, "multithreaded" software, en zogeheten legacy-systemen. We hebben hun belang in de nabije toekomst beargumenteerd, de belangrijke knelpunten besproken, en een reeks mogelijke oplossingen en aanbevelingen gepresenteerd.

Sequence diagram reconstructie. De volgende stap behelsde het ontwerp van een raamwerk voor het reconstrueren van UML sequence diagrammen. We hebben testcases gebruikt als executiescenario's omdat de functionaliteit van een programma zodoende effectief wordt verdeeld. Deze benadering leidde aanvankelijk tot ons Sdr-gereedschap, en werd geëevalueerd op een klein systeem met twee concrete onderhoudstaken. In samenwerking met een afstudeerder presenteerden we vervolgens JRET, een volwassener implementatie met een aantal belangrijke verbeteringen, en haar evaluatie op een applicatie van een realistische omvang.

Het gebruik van testcases en hun visualisatie als UML sequence diagrammen bleek bruikbaar als een startpunt voor softwarebegrip. Echter, we maakten hierin veelvuldig gebruik van abstractietechnieken om de schaalbaarheidsproblemen op te lossen, en concludeerden dat een uitvoeriger onderzoek naar de beschikbare abstracties nodig was omdat hun effectiviteit en toepasbaarheid in de praktijk onduidelijk bleef.

Om de vele abstractietechnieken in de literatuur grondig met elkaar te kunnen vergelijken, presenteerden we een evaluatiemethodologie waaraan zes stappen ten grondslag liggen: context, criteria, metrieken, testomgeving, toepassing, en interpretatie. Het gebruik van dit raamwerk werd gedemonstreerd aan de hand van een experiment met vier bestaande abstractietechnieken en een testomgeving met zeven grote traces, waarvan er drie groter waren een miljoen regels. Dit raamwerk maakt het mogelijk om abstractietechnieken grondig te evalueren, en verschaft zodoende een beter begrip van de toepasbaarheid van zulke technieken in concrete contexten.

Extravis. Een andere manier waarop we de schaalbaarheidsproblemen van dynamische analyse hebben aangepakt was het ontwerp en de evaluatie van meer geavanceerde visualisaties, d.i. technieken die niet zijn gebaseerd op traditionele visualisaties zoals bomen en UML. Hiertoe hebben twee nieuwe aanzichten gebruikt voor het visualiseren van traces. De "circular bundle view" projecteert de hiërarchische structuur van een softwaresysteem op een cirkel en toont de relaties binnen het systeem via gebundelde lijnen. De "massive sequence view" verschaft een navigeerbaar overzicht van een trace dat alle gebeurtenissen in chronologische volgorde visualiseert als gekleurde lijnen. De twee aanzichten zijn met elkaar verbonden en zijn geheel interactief; verder passen ze op één beeldscherm en zijn ze geïmplementeerd in een gereedschap genaamd Extravis. Dit gereedschap is het resultaat van een samenwerking met visualisatie-experts van de Technische Universiteit Eindhoven.

In eerste instantie hebben we Extravis geëvalueerd door middel van een drietal case studies. De resultaten duidden op een aantal belangrijke voordelen, waaronder het optimale gebruik van het beeldscherm en het geboden inzicht in programmastructuren. We stelden echter vast dat de relaties in het cirkelaanzicht mogelijk lastig zijn te begrijpen: immers, waar traditionele visualisaties zoals UML algemeen bekend zijn, roepen meer geavanceerde technieken om uitvoerige validaties om hun begrijpbaarheid en bruikbaarheid te bepalen.

Om deze reden was onze volgende stap om via een *gecontroleerd experiment* de meerwaarde van Extravis te meten ten opzichte van een bestaande oplossing, de Eclipse ontwikkelomgeving. Hiertoe hebben we acht kenmerkende taken ontworpen die zijn gericht op het begrijpen (en onderhouden) van een representatief softwaresysteem. Vervolgens maten we hoe een controlegroep (gebruikmakend van Eclipse) en een experimentele groep (gebruikmakend van zowel Eclipse als Extravis) presteerden in termen van correctheid en tijdsinvestering. De statistisch significante resultaten tonen aan dat de Extravis-gebruikers aanzienlijk minder tijd nodig hadden en nauwkeurigere antwoorden gaven.

Het ontwerp en de uitvoering van dit experiment hebben het potentieel van trace-visualisatie gedemonstreerd en hebben de weg vrijgemaakt voor soortgelijke experimenten door andere onderzoekers.

Nadruk op evaluatiekwaliteit. De evaluatiekwaliteit was in onze experimenten een belangrijk aandachtspunt. Deze doelstelling doorsnijdt elk van de kernhoofdstukken van dit proefschrift.

Ons literatuuronderzoek behelsde een systematische benadering om de reproduceerbaarheid van onze resultaten te waarborgen. Het gebruik van expliciete selectiecriteria en het afleiden van een herbruikbaar raamwerk zorgen ervoor dat collega-onderzoekers onze bevindingen kunnen repliceren en bestaand werk kunnen vergelijken. Om bevooroordeling onzerzijds zoveel mogelijk te vermijden zijn beide onderdelen getoetst door middel van pilot studies.

De reconstructie van UML sequence diagrammen werd bewerkstelligd door middel van twee nieuwe gereedschappen die beide zijn geëvalueerd via case studies. Hoewel het proefsysteem in onze aanvankelijke studie tamelijk klein was, betrof het vervolgexperiment een representatief softwaresysteem. In beide studies draaide het om begripstaken die voortkomen uit realistische softwareonderhoudsproblemen. Ons tweede gereedschap is bovendien geïmplementeerd als een Eclipse-uitbreiding en is online beschikbaar gesteld opdat collega-onderzoekers ermee kunnen experimenteren.

Ons evaluatieraamwerk voor abstractietechnieken is volledig gedocumenteerd, en de traces en technieken die we gebruikten in ons voorbeeldexperiment zijn online beschikbaar. Hiermee is de reproduceerbaarheid van onze bevindingen gewaarborgd, en worden collega-onderzoekers ondersteund in het uitvoeren van soortgelijke experimenten met andere abstractietechnieken.

Extravis hebben we uitvoerig geëvalueerd via drie uitgebreide case studies. Elk van deze studies betrof een verschillende softwarebegripscontext en een eigen softwaresysteem dat qua omvang representatief was, en waarvan er één industrieel was. Ook is het gereedschap gedocumenteerd en online beschikbaar gesteld zodat het bij andere experimenten kan worden betrokken.

Het gecontroleerde experiment, tot slot, was ontworpen met het oog op herbruikbaarheid. We hebben acht taken bedacht die zijn gebaseerd op een raamwerk uit de literatuur, en hebben maatregelen getroffen om de mate van bevooroordeling zoveel mogelijk terug te dringen. We hebben alle informatie beschikbaar gesteld die is vereist voor de replicatie van het experiment, en hebben de onderzoeksgemeenschap zodoende in staat gesteld om soortgelijke experimenten uit te voeren met andere visualisatietechnieken.

Curriculum Vitae

Personal data

Full name: Bas Cornelissen

Date of birth: December 5, 1980

Place of birth: Amsterdam, The Netherlands

Nationality: Dutch

Education

September 1999 – January 2005: B.Sc. and M.Sc. Computer Science from the University of Amsterdam, specialized in Computing, System Architecture & Programming.

September 1993 - August 1999 VWO, Ignatius Gymnasium, Amsterdam. Subjects: Dutch, English, Latin, French, mathematics, physics, chemistry, biology.

Employment

June 2005 - May 2009: PhD candidate at Delft University of Technology, Delft, under the supervision of prof. dr. Arie van Deursen.

February 2005 - May 2005: Web developer at Vergelijk.nl B.V., Utrecht (full time).

July 2003 - January 2005: Web developer at Vergelijk.nl B.V., Utrecht (part time).

Titles in the IPA Dissertation Series since 2005

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue*. Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms*. Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications*. Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures*. Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability*. Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions*. Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments*. Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression*. Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages*. Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations*. Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for Clean*. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13