

Delft University of Technology
Software Engineering Research Group
Technical Report Series

Trace Visualization for Program Comprehension: A Controlled Experiment

Bas Cornelissen, Andy Zaidman, Bart Van Rompaey, and
Arie van Deursen

Report TUD-SERG-2009-001



TUD-SERG-2009-001

Published, produced and distributed by:

Software Engineering Research Group
Department of Software Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4
2628 CD Delft
The Netherlands

ISSN 1872-5392

Software Engineering Research Group Technical Reports:

<http://www.se.ewi.tudelft.nl/techreports/>

For more information about the Software Engineering Research Group:

<http://www.se.ewi.tudelft.nl/>

Note: Submitted to the 17th International Conference on Program Comprehension (2009).

© copyright 2009, by the authors of this report. Software Engineering Research Group, Department of Software Technology, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology. All rights reserved. No part of this series may be reproduced in any form or by any means without prior written permission of the authors.

Trace Visualization for Program Comprehension: A Controlled Experiment

Bas Cornelissen, Andy Zaidman, Arie van Deursen
Delft University of Technology
The Netherlands

{s.g.m.cornelissen, a.e.zaidman, arie.vandeursen}@tudelft.nl

Bart van Rompaey
University of Antwerp
Belgium

bart.vanrompaey2@ua.ac.be

Abstract

Understanding software through dynamic analysis has been a popular activity in the past decades. One of the most common approaches in this respect is execution trace analysis: among our own efforts in this context is EXTRAVIS, a tool for the visualization of large traces. Similar to other trace visualization techniques, our tool has been validated through anecdotal evidence, but should also be quantitatively evaluated to assess its usefulness for program comprehension.

This paper reports on a first 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.

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 [3, 1]. 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 roughly break 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 [15, 6] and visualization [12, 8] of execution traces. Among our share of these techniques and tools is EXTRAVIS, a tool that offers two interactive views of large execution traces [4]. Through a series of case studies we illustrated 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 paper 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 paper is structured as follows. Section 2 provides a background on dynamic analysis and trace visualization, and motivates our intent to conduct controlled experiments. Section 3 offers a detailed description of the experimental design. Section 4 discusses the results, and threats to validity are treated in Section 5. Section 6 outlines related work, and Section 7 offers conclusions and future directions.

2. Background

Execution trace analysis. The use of dynamic analysis for program comprehension has been a popular research activity in the last decades. In a large survey that we recently performed [5], 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

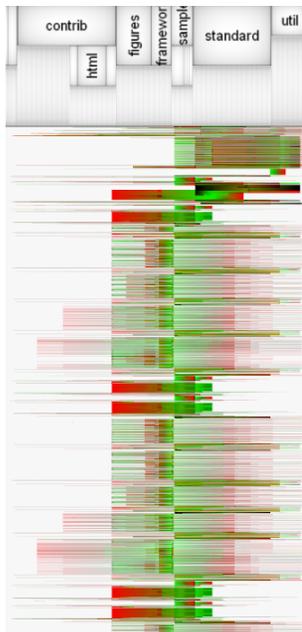


Figure 1. EXTRAVIS' massive sequence view.

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. Reiss and Renieris, for example, report [15] 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., [15, 20, 6]). Another common approach is the *visualization* of execution traces: key contributions on this subject include Insight by De Pauw et al. [12], Scene from Koskimies & Mössenböck [9], ISVis by Jerding et al. [8], and Shimba from Systä et al. [18].

Extravis. Our own contributions to the field of trace understanding include EXTRAVIS, a publicly available¹ tool for the visualization of large execution traces. EXTRAVIS provides two linked, interactive views. The *massive sequence view* is essentially a large-scale UML sequence diagram (similar to Jerding's Information Mural [7]), and offers an overview of the trace and the means to navigate it (Figure 1). The *circular bundle view* hierarchically projects the program's structural entities on a circle and shows their interrelationships in a bundled fashion (Figure 2). We qualitatively evaluated the tool in various program comprehension contexts, including trace exploration, feature location, and top-down program comprehension [4]. The results confirmed EXTRAVIS' benefits in these contexts, the main advantages being its optimal use of screen real estate and the improved insight into

¹EXTRAVIS, <http://swel1.tudelft.nl/extravis>

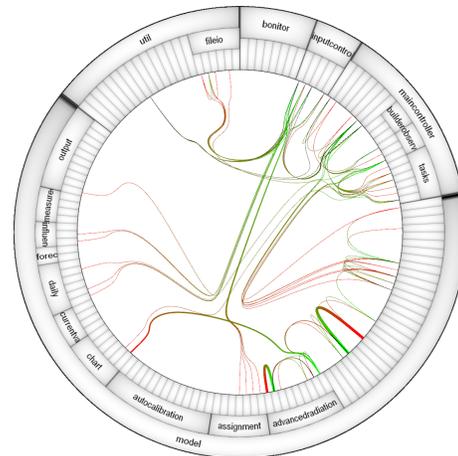


Figure 2. EXTRAVIS' circular bundle view.

a program's structure. However, we hypothesized that the relationships in the circular view may be difficult to grasp.

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 [5], i.e., they rely on UML, graph, or tree notations, to which presumably most software engineers are accustomed. In contrast, EXTRAVIS uses non-traditional visualization techniques, and Storey argues [17] that advanced visual interfaces are not often used in development environments because they tend to require complex user interactions. These reasons have motivated us to empirically validate EXTRAVIS through a controlled experiment: we seek to assess its added value in concrete maintenance contexts.

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 IDE². Similar to related efforts (e.g., [10, 14]) 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.

3.1. Research questions & Hypotheses

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

²Eclipse IDE, <http://www.eclipse.org>

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 artefacts
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 1. Pacione's nine principal activities.

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:

- $H1_0$: The availability of EXTRAVIS does not impact the time needed to complete typical comprehension tasks.
- $H2_0$: 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:

- $H1$: The availability of EXTRAVIS reduces the time needed to complete typical comprehension tasks.
- $H2$: 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 3.2 through 3.6 provide a detailed description of the experiment.

3.2. Object & Task design

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.

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 2. Descriptions of the comprehension tasks.

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 authors of this paper are familiar with its internals as a result of earlier experiments [21, 16, 4]. 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 this end, we use the framework by Pacione et al. [11], 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 1). The latter aspect significantly reduces any 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 2 provides outlines of the tasks and shows how

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

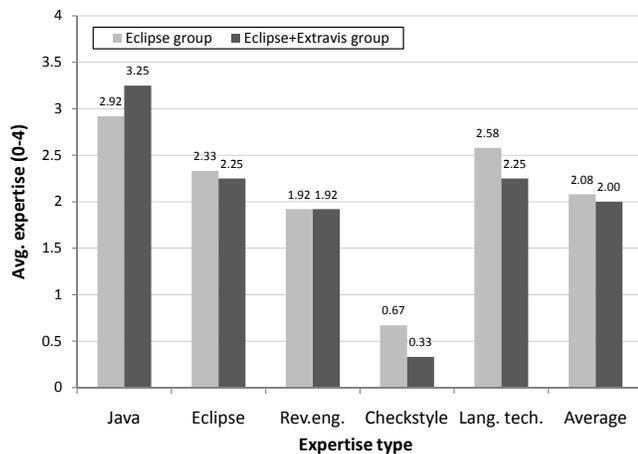


Figure 3. Average expertises of the subject groups.

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, T4.1, and T4.2; and activity A4, “*Investigating dependencies between artefacts*“, 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 authors of this paper 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 paper 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 at the end of this report. Following the experiment, the first two authors of this paper 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.

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 largely mitigates concerns from Di Penta et al., who argue [13] that “*a subject group made up entirely of students might not adequately represent the intended user population*” (discussed in further detail in Section 5.2). Participation is on a vol-

untary 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*”). A characterization of the subjects is provided at the end of this report.

The assignments to the control and experimental group were conducted manually to evenly distribute the available knowledge. This is illustrated by Figure 3: 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.

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 earlier experiments [4]. 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 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.

Group	Mean	Diff.	Min	Max	Median	Stdev.	one-tailed Student's t-test					one-tailed M.-W.		
							K.-S. Z	Lev. F	df	t	p-val.	U	p-val.	
<i>Time</i>														
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	
<i>Correctness</i>														
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 3. Descriptive statistics of the experimental results.

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 between 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. In our analysis of the results, we shall first test these requirements and then 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% ($\alpha=0.05$), 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.

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 paper, 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 3.2. Fur-

thermore, the studies lead to the refinement of several tasks in order to make the questions clearer. Other than these unclaritys, the tasks were found to be sufficiently feasible in both the Eclipse and the Ecl+Ext pilot.

4. Results & Discussion

This section describes our interpretation of the results. We first discuss the time and correctness aspects in Section 4.1 and 4.2, and then take a closer look at the scores from a task perspective in Section 4.3.

Table 3 shows descriptive statistics of the measurements, aggregated over all tasks. The measurements themselves are available as a spreadsheet.⁴

Wohlin et al. [19] 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 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 2).

4.1. Time results

We start off by testing null hypothesis H1₀, which states that the availability of EXTRAVIS does not impact the time that is needed to complete typical comprehension tasks.

⁴Results spreadsheet, <http://www.st.ewi.tudelft.nl/~cornel/results.xlsx>

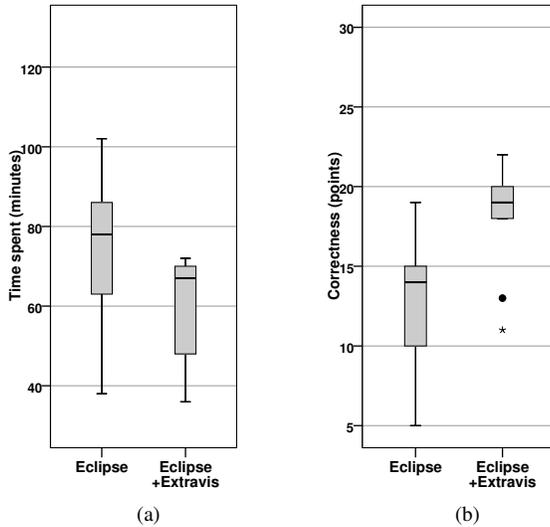


Figure 4. Box plots for time spent and correctness.

Figure 4(a) shows a box plot for the total time that the subjects spent on the first six tasks. Table 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 H_{10} . As shown in Table 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 H_{10} can be rejected in favor of the alternative hypothesis H_1 , 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 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 EXTRAVIS' 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.

4.2. Correctness results

We now test null hypothesis H_{20} , which states that the availability of EXTRAVIS does not impact the correctness of answers given during typical comprehension tasks.

Figure 4(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 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 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 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 H_{20} can be easily rejected in favor of our alternative hypothesis H_2 , 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 4.3 discusses this in more detail through an example task.

Second, the results of the debriefing questionnaire (Table 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 infor-

	Eclipse		Eclipse+Extravis	
	Mean	Stdev.	Mean	Stdev.
Misc.				
Time pressure (0-4)	2.17	1.27	2.08	0.51
Dynamic analysis (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 features used			6.42	2.68
No. of tasks used (0-8)			6.75	1.14
No. of tasks success (0-8)			5.08	1.31
% of time spent (est.)			60.00	26.71
Speed (0-2)			1.42	0.51

Table 4. Results of the debriefing questionnaire.

mation” 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 [14], we could not identify a correlation between the subjects’ performances and their (subjective) expertise levels.

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 5 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 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

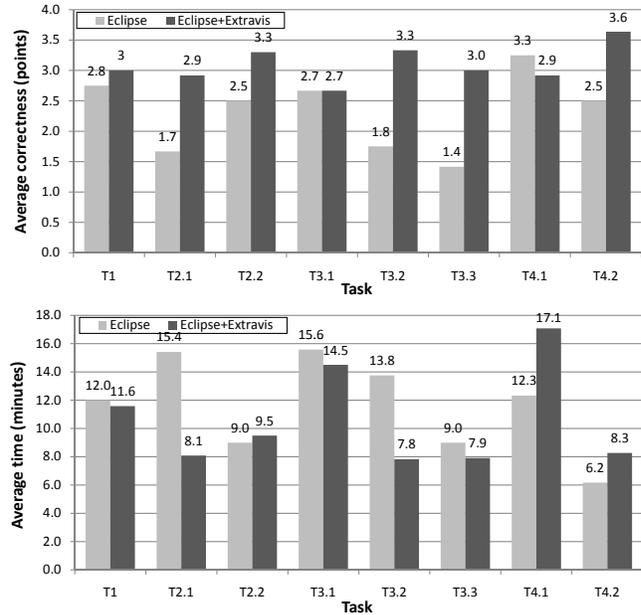


Figure 5. Averages per task.

“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 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 5, 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 sev-

eral 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.

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.

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 paper, 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 [11] 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 Eclipse+Extravis 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 paper 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 lead 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 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.

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 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 [11], 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. 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. [2] 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 [14] 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 location⁵ tasks for two subject systems. The use of DOPGs by his experimental group lead 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 [6] 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.

7. Conclusion

In this paper, 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

⁵Feature location is a reverse engineering activity that concerns the establishment of relations between concepts and source code.

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 paper demonstrates the potential of trace visualization for program comprehension, and paves the way for other researchers to conduct similar experiments. The work in this paper 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

This research is sponsored by NWO via the Jacquard Reconstructor project. We would like to thank the 24 subjects for their participation, Danny Holten for his implementation of EXTRAVIS, and Cathal Boogerd for performing one of the pilot studies and for proofreading this paper. Also, many thanks to CHECKSTYLE's lead developer, Oliver Burn, who assisted in the design of our task review protocol.

References

- [1] V. R. Basili. Evolving and packaging reading technologies. *J. Syst. Softw.*, 38(1):3–12, 1997.
- [2] C. Bennett, D. Myers, D. Ouellet, M.-A. Storey, M. Salois, D. German, and P. Charland. A survey and evaluation of tool features for understanding reverse engineered sequence diagrams. *J. Softw. Maint. Evol.*, 20(4):291–315, 2008.
- [3] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.
- [4] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *J. Syst. Softw.*, 81(11):2252–2268, 2008.
- [5] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. Technical Report TUD-SERG-2008-033, Delft University of Technology, 2008.
- [6] A. Hamou-Lhadj and T. C. Lethbridge. Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In *Proc. Int. Conf. on Program Compr. (ICPC)*, pages 181–190. IEEE CS, 2006.
- [7] D. F. Jerding and J. T. Stasko. The information mural: A technique for displaying and navigating large information spaces. *IEEE Trans. Vis. Comput. Graph.*, 4(3):257–271, 1998.
- [8] D. F. Jerding, J. T. Stasko, and T. Ball. Visualizing interactions in program executions. In *Proc. Int. Conf. on Softw. Eng. (ICSE)*, pages 360–370. ACM, 1997.
- [9] K. Koskimies and H. Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proc. Int. Conf. on Softw. Eng. (ICSE)*, pages 366–375. IEEE CS, 1996.
- [10] C. F. J. Lange and M. R. V. Chaudron. Interactive views to improve the comprehension of UML models - an experimental validation. In *Proc. Int. Conf. on Program Compr. (ICPC)*, pages 221–230. IEEE CS, 2007.
- [11] M. J. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In *Proc. Working Conf. on Reverse Eng. (WCRE)*, pages 70–79. IEEE CS, 2004.
- [12] W. De Pauw, R. Helm, D. Kimelman, and J. M. Vlissides. Visualizing the behavior of object-oriented systems. In *Proc. Conf. on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 326–337. ACM, 1993.
- [13] M. Di Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In *Proc. Int. Conf. on Program Compr. (ICPC)*, pages 281–285. IEEE CS, 2007.
- [14] J. Quante. Do dynamic object process graphs support program understanding? – a controlled experiment. In *Proc. Int. Conf. on Program Compr. (ICPC)*, pages 73–82. IEEE CS, 2008.
- [15] S. P. Reiss and M. Renieris. Encoding program executions. In *Proc. Int. Conf. on Softw. Eng. (ICSE)*, pages 221–230. IEEE CS, 2001.
- [16] B. Van Rompaey and S. Demeyer. Estimation of test code changes using historical release data. In *Proc. Working Conf. on Reverse Eng. (WCRE)*, pages 269–278. IEEE CS, 2008.
- [17] M.-A. Storey. Theories, methods and tools in program comprehension: past, present and future. In *Proc. Int. Workshop on Program Compr. (IWPC)*, pages 181–191. IEEE CS, 2005.
- [18] T. Systä, K. Koskimies, and H. A. Müller. Shimba: an environment for reverse engineering Java software systems. *Softw., Pract. Exper.*, 31(4):371–394, 2001.
- [19] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslen. *Experimentation in software engineering - an introduction*. Kluwer Acad. Publ., 2000.
- [20] A. Zaidman and S. Demeyer. Managing trace data volume through a heuristical clustering process based on event execution frequency. In *Proc. European Conf. on Softw. Maint. and Reeng. (CSMR)*, pages 329–338. IEEE CS, 2004.
- [21] A. Zaidman, B. Van Rompaey, S. Demeyer, and A. van Deursen. Mining software repositories to study co-evolution of production & test code. In *Proc. Int. Conf. on Softw. Testing (ICST)*, pages 220–229. IEEE CS, 2008.

Subjects & Results

Subject no.	Affiliation	M.Sc.	Expertise (0-4)						Performance (T1-T3)	
			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	(Dui)	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 5. Characterization of the subjects and their performances, sorted by average expertise. Students are denoted by asterisks.

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

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

	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		

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?
- 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:
 - ...
 - ...

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

Acquired through an automated analysis of the static call graph.

```

api.AbstractFileSetCheck 32 11
api.AbstractLoader 8 0
api.AbstractViolationReporter 123 6
api.AuditEvent 25 6
api.AuditListener 5 0
api.CheckstyleException 11 0
api.Comment 20 0
api.Configurable 4 0
api.Configuration 10 0
api.Context 2 0
api.Contextualizable 3 0
api.DetailAST 375 0
api.FileContents 23 3
api.FileSetCheck 3 0
api.Filter 3 0
api.FilterSet 9 5
api.FullIdent 55 2
api.LocalizedMessage 14 0
api.LocalizedMessages 9 0
api.MessageDispatcher 14 0
api.Scope 11 0
api.ScopeUtils 36 3

```

```
api.SeverityLevel 14 0
api.SeverityLevelCounter 8 2
api.StrArrayConverter 1 0
api.TextBlock 19 0
api.TokenTypes 10 0
api.Utills 37 0
checks.AbstractFormatCheck 32 1
checks.AbstractOption 10 7
checks.AbstractOptionCheck 16 1
checks.AbstractTypeAwareCheck$ClassAlias 4 4
checks.AbstractTypeAwareCheck$ClassInfo 10 0
checks.AbstractTypeAwareCheck$RegularClass 5 3
checks.AbstractTypeAwareCheck$Token 10 3
checks.ArrayTypeStyleCheck 4 4
checks.BlockFrame 1 1
checks.blocks.AvoidNestedBlocksCheck 4 4
checks.blocks.BlockOption 1 1
checks.blocks.LeftCurlyOption 1 1
checks.blocks.NeedBracesCheck 4 3
checks.blocks.RightCurlyOption 1 1
checks.CheckUtils 15 7
checks.ClassFrame 1 1
checks.ClassResolver 2 0
checks.coding.AbstractIllegalCheck 4 0
checks.coding.AbstractSuperCheck$MethodNode 4 0
checks.coding.AvoidInlineConditionalsCheck 6 4
checks.coding.DeclarationOrderCheck 5 5
checks.coding.DoubleCheckedLockingCheck 4 4
checks.coding.EmptyStatementCheck 4 3
checks.coding.FinalLocalVariableCheck 8 8
checks.coding.HiddenFieldCheck$FieldFrame 6 0
checks.coding.IllegalThrowsCheck 6 6
checks.coding.InnerAssignmentCheck 4 4
checks.coding.ModifiedControlVariableCheck 8 7
checks.coding.MultipleStringLiteralsCheck$stringInfo 3 0
checks.coding.MultipleVariableDeclarationsCheck 4 4
checks.coding.NestedIfDepthCheck 5 5
checks.coding.NestedTryDepthCheck 5 4
checks.coding.PackageDeclarationCheck 8 3
checks.coding.ParameterAssignmentCheck 8 6
checks.coding.ReturnCountCheck 9 7
checks.coding.SimplifyBooleanExpressionCheck 7 4
checks.coding.SimplifyBooleanReturnCheck 4 4
checks.coding.StringLiteralEqualityCheck 4 3
checks.coding.SuperCloneCheck 1 0
checks.coding.SuperFinalizeCheck 1 0
checks.DescendantTokenCheck 16 7
checks.design.FinalClassCheck$ClassDesc 7 0
checks.design.InterfaceIsTypeCheck 6 4
checks.design.ThrowsCountCheck 6 6
checks.duplicates.ChecksumInfo 3 0
checks.duplicates.StrictDuplicateCodeCheck$ChecksumGenerator 1 0
checks.duplicates.StrictDuplicateCodeCheck$JavaChecksumGenerator 1 1
checks.duplicates.StrictDuplicateCodeCheck$TextfileChecksumGenerator 2 1
checks.FileContentsHolder 6 1
checks.FrameStack 5 2
checks.GlobalFrame 1 1
checks.header.CrossLanguageRegexpHeaderCheck$FileSetCheckViolationMonitor 2 1
checks.header.HeaderInfo 9 1
checks.header.HeaderViolationMonitor 2 0
checks.header.RegexpHeaderCheck$CheckViolationMonitor 2 1
checks.header.RegexpHeaderInfo 5 2
checks.imports.AccessResult 1 0
checks.imports.AvoidStarImportCheck 4 4
checks.imports.Guard 3 0
checks.imports.PkgControl 4 2
checks.indentation.ExpressionHandler 129 41
```

```
checks.indentation.FinallyHandler 1 1
checks.indentation.HandlerFactory 4 1
checks.indentation.IndentLevel 24 0
checks.indentation.LineSet 9 0
checks.indentation.PrimordialHandler 6 2
checks.indentation.StaticInitHandler 1 1
checks.indentation.SwitchHandler 6 6
checks.indentation.TryHandler 3 3
checks.j2ee.AbstractInterfaceCheck 5 1
checks.j2ee.AbstractJ2eeCheck 17 5
checks.j2ee.BeanManagedMethodChecker 9 9
checks.j2ee.BeanMethodChecker 8 7
checks.j2ee.EntityBeanMethodChecker 16 10
checks.j2ee.HomeInterfaceMethodChecker 5 5
checks.j2ee.LocalInterfaceMethodChecker 2 2
checks.j2ee.MessageBeanMethodChecker 9 8
checks.j2ee.MethodChecker 32 16
checks.j2ee.PersistenceOption 1 1
checks.j2ee.RemoteInterfaceMethodChecker 2 2
checks.j2ee.SessionBeanMethodChecker 9 9
checks.j2ee.ThisParameterCheck 6 6
checks.j2ee.ThisReturnCheck 6 6
checks.j2ee.Utills 33 5
checks.javadoc.HtmlTag 7 0
checks.javadoc.JavadocMethodCheck$ExceptionInfo 5 5
checks.javadoc.JavadocTag 14 0
checks.javadoc.Point 3 0
checks.LexicalFrame 6 0
checks.LineSeparatorOption 3 1
checks.MethodFrame 1 1
checks.metrics.AbstractComplexityCheck 13 4
checks.metrics.BooleanExpressionComplexityCheck 8 7
checks.metrics.ClassDataAbstractionCouplingCheck 3 2
checks.metrics.ClassFanOutComplexityCheck 3 1
checks.metrics.CyclomaticComplexityCheck 5 2
checks.metrics.JavaNCSSCheck 9 7
checks.metrics.JavaNCSSCheck$Counter 2 0
checks.modifier.ModifierOrderCheck 4 3
checks.naming.ConstantNameCheck 4 4
checks.naming.LocalFinalVariableNameCheck 4 4
checks.naming.LocalVariableNameCheck 4 4
checks.naming.MethodNameCheck 3 1
checks.naming.ParameterNameCheck 4 2
checks.naming.StaticVariableNameCheck 4 4
checks.naming.TypeNameCheck 3 1
checks.sizes.AnonInnerLengthCheck 4 4
checks.sizes.ExecutableStatementCountCheck$Context 4 0
checks.sizes.FileLengthCheck 4 2
CheckStyleTask$Property 2 0
checks.UpperEllCheck 4 3
checks.whitespace.NoWhitespaceAfterCheck 5 5
checks.whitespace.OperatorWrapOption 1 1
checks.whitespace.PadOption 1 1
checks.whitespace.TabCharacterCheck 4 2
checks.whitespace.TypecastParenPadCheck 6 5
DefaultConfiguration 13 2
DefaultContext 5 0
DefaultLogger 8 6
doclets.CheckDocsDoclet 1 0
doclets.TokenTypesDoclet 0 0
filters.IntFilter 2 0
filters.IntMatchFilter 2 0
filters.IntRangeFilter 2 0
filters.SuppressionCommentFilter$Tag 5 2
grammars.CommentListener 2 0
grammars.GeneratedJavaLexer 4 4
grammars.GeneratedJavaRecognizer 3 2
```

```
gui.AbstractTreeTableModel 6 2
gui.FileDrop 2 1
gui.FileDrop$Listener 1 0
gui.JTreeTable$ListToTreeSelectionModelWrapper 3 0
gui.JTreeTable$TreeTableCellRenderer 3 1
gui.ParseTreeInfoPanel$FileDropListener 2 1
gui.ParseTreeInfoPanel$FileSelectionAction 1 1
gui.ParseTreeInfoPanel$ReloadAction 1 1
gui.ParseTreeModel 8 8
gui.TreeTableModel 7 0
ModuleFactory 2 0
PackageObjectFactory 3 1
PropertiesExpander 3 0
PropertyCacheFile 4 1
PropertyResolver 1 0
StringArrayReader 1 0
TreeWalker$SilentJavaRecognizer 1 1
```


TUD-SERG-2009-001
ISSN 1872-5392

