Exploiting Dynamic Information in IDEs Improves Speed and Correctness of Software Maintenance Tasks

David Röthlisberger
Software Composition Group
University of Bern, Switzerland
roethlis@iam.unibe.ch

Marcel Härry
Software Composition Group
University of Bern, Switzerland
mhaerry@iam.unibe.ch

Philippe Moret
University of Lugano
Switzerland
philippe.moret@usi.ch

Danilo Ansaloni
University of Lugano
Switzerland
danilo.ansaloni@usi.ch

Alex Villazón
University of Lugano
Switzerland
alex.villazon@usi.ch

Walter Binder
University of Lugano
Switzerland
walter.binder@usi.ch

Oscar Nierstrasz
Software Composition Group
University of Bern, Switzerland
oscar@iam.unibe.ch

ABSTRACT

Modern IDEs such as Eclipse offer static views of the source code, but such views ignore information about the run-time behavior of software systems. Since typical object-oriented systems make heavy use of polymorphism and dynamic binding, static views will miss key information about the run-time architecture. In this paper we show by means of a controlled experiment with 30 professional developers that for typical software maintenance tasks integrating dynamic information into the Eclipse IDE yields a significant 17.5% decrease of time spent while significantly increasing the correctness of the solutions by 33.5%. Furthermore, we describe several enhancements to the Eclipse IDE that integrate static and dynamic information, with the goal of better supporting typical software maintenance activities. We elaborate on a case study which further highlights the usefulness of dynamic information for performance optimizations. We also report on several important efficiency improvements to our dynamic information collection framework, and we present benchmarks evaluating the overhead of our approach.

Keywords

dynamic analysis, development environments, program comprehension, software maintenance, empirical experiments

1. INTRODUCTION

In many object-oriented systems, conceptually related code is scattered over the entire source space. As this space can be very large, developers have difficulties to locate the code relevant for a particular software maintenance task. This problem is further exacerbated by object-oriented language features such as inheritance, interface types or polymorphism that often obscure the actual run-time behavior. The primary tool used by software developers, the IDE, does usually not help them better understand source code relying on polymorphism as it just shows the statically declared types. Moreover, high-level collaborations are often inaccurately presented by the IDE, which is only aware of which classes implement a given interface, not which ones are actually used at run-time.

To improve the understanding for Java-based software systems employing polymorphism or abstract types, we extended the Eclipse IDE to integrate dynamic information. Such information enables the programmer to explore and understand the interprocedural control flow of a system or to see also the run-time types in the source code views that are, for example, sub-types of the statically defined types. Our extensions exploit behavioral information to better understand and navigate the source space in a prototype Eclipse plugin called *Senseo* about which we reported in previous work [20][1].

However, this previous work was considerably limited and had several flaws: (i) dynamic information was just integrated locally in the source code views on a method level, thus there was no overview of the entire system, (ii) dynamic collaboration between source artifacts, such as packages or classes, was not made visible, and (iii) gathering of dynamic information such as run-time types was inefficient. Thus, we contribute in this paper several significant extensions to *Senseo* such as a collaboration view presenting all dynamic collaborators of source artifacts on a package or class level to make explicit the dynamic dependencies between entire source artifacts. Another enhancement, called Calling Context Ring Chart (*CCRC*), a navigable visualization of the system's Calling Context Tree (CCT) [1], is useful for efficiently spotting performance bottlenecks. The CCT is a runtime data structure for calling context profiling. Each node in the CCT conveys the measured dynamic metrics for that calling context. We also significantly reduced the overhead of gathering the dynamic information.

We previously motivated the usefulness of *Senseo* purely with anecdotal evidence. To validate our claims, we conduct a controlled user experiment with 30 professional Java developers to ob-

ICSE 2010 Cape Town, South Africa

[1] http://scg.unibe.ch/download/Senseo.pdf

tain reliable quantitative and qualitative feedback about the impact on developer productivity contributed by *Senseo* and the dynamic information it integrates in Eclipse. We ask the subjects to solve five typical software maintenance tasks in an unfamiliar, medium-sized software system. We split the 30 subjects into two groups, an experimental and a control group. The only difference in treatment between these two groups is that the control group is provided with the standard Eclipse IDE while the experimental group additionally can use the *Senseo* plugin. Analyzed variables are time required to solve the tasks and correctness of the answers. Evaluating the obtained results yields a significant decrease in time of 17.5% and a significant increase in correctness of 33.5%.

In this paper we make the following contributions: (i) validating the impact of the availability of dynamic information and their presentation in Eclipse on developer productivity during software maintenance by conducting a controlled experiment with 30 professional Java developers, (ii) integrating higher-level means in the IDE to visualize and present dynamic information such as the collaboration or the *CCRC* view, (iii) a case study reporting specifically on the usefulness of the *CCRC*, and (iv) improving the efficiency of our aspect-based dynamic information collection framework called *MAJOR* and a thorough performance evaluation.

This paper is structured as follows: In Section 2 we describe the background of our work on integrating dynamic information in IDEs. Section 3 illustrates enhancements to *Senseo*. Section 4 introduces the controlled user experiment, discusses its design, results and threats to validity. In Section 5 we present a case study, validating *Senseo*'s *CCRC* view. Section 6 thoroughly elaborates on the important efficiency issue of the employed dynamic information collection technique. Finally, Section 7 highlights related work in the areas of dynamic analysis, IDEs and empirical studies in software engineering while Section 8 concludes the paper.

2. BACKGROUND

In a previous publication [20], we introduced a prototype of *Senseo* integrating dynamic information in Eclipse about receiver, argument and return value types of message sends. *Senseo* presents this information in tooltips appearing when the mouse hovers over method declarations or invocations in source code. For a method, the tooltip shows the calling methods, the methods called in the method body, the concrete argument, receiver, and return value types. That is, if the statically declared type refers to an abstract or interface type, the tooltip shows the type actually occurred at run-time (see Figure 1, (1)).

Furthermore, the former *Senseo* prototype also visualizes three dynamic metrics[2] [8], the number of method invocations, the number of object allocations, and the allocated bytes. In the *Senseo* prototype we presented such metrics in the ruler columns next to the source code editor (Figure 1, (2) and (3)) and in the package tree (Figure 1, (4)). Source artifacts heavily contributing to the currently selected metric, such as methods creating many objects, are displayed in red in package tree and ruler columns while artifacts with lower activity are colored in blue. This means *Senseo* follows a heatmap coloring scheme with a color gradient from blue to red.

The analyzed application runs in a separate JVM in which *MAJOR* [22] weaves a data gathering aspect in the application code, while the Eclipse IDE including *Senseo* runs in another JVM. The aspect woven with *MAJOR* creates the CCT and collects the dynamic information. Such an aspect-based approach for dynamic information gathering is highly flexible as the aspect code is compact and easy to extend and customize. Furthermore, *MAJOR* enables us to gather dynamic information for the entire bytecode used by a system, including the Java class library and dynamically loaded or generated code.

Senseo aggregates the collected dynamic information of multiple executions of the program, thus the information presented in the IDE is not bound to a specific execution as for instance in a debugger or a profiler. In comparison with debuggers or profilers, *Senseo* supports a wider range of software maintenance tasks and the aggregated information is not volatile, but rather permanent and aggregated. Hence *Senseo*'s approach to dynamic analysis suffers less from typical problems of behavioral analysis, such as coverage or completeness of the information, as the dynamic information stemming from various different system executions is typically more reliable and complete, depending on the executed scenarios.

While the former *Senseo* prototype and the dynamic information it provided were already helpful to better understand systems heavily relying on polymorphic message sending or abstract interface types [20], we realized that there were several types of software maintenance tasks these extensions could not well support:

i. Understanding higher-level concepts, such as application layers, models, or separation of concerns.
ii. Identifying collaboration patterns, that is, how various source artifacts communicate with each other at run-time.
iii. Locating design flaws, design "smells", performance bottlenecks, and other code quality issues, such as classes heavily coupled to classes in other packages or classes residing in wrong packages.
iv. Gaining an overview of control flow and execution complexity, for instance to quickly locate performance bottlenecks.

In the next section we discuss how we enhanced *Senseo* to support these kind of tasks by providing visualizations or other means to use various dynamic information directly in Eclipse. The user experiment presented in Section 4 shows how these enhancements actually help developers performing such tasks.

3. ENHANCEMENTS TO SENSEO

This section discusses enhancements and new features integrated in *Senseo*, which were not available in the prototype presented in [20]. Whereas this earlier prototype of *Senseo* showed dynamic information only close to the source code and locally to specific source artifacts (for instance, by extending the tooltip appearing when the mouse hovers over methods), the new version greatly improves the integration of behavioral information with additional means such as the collaboration view which makes explicit dependencies between different source entities or the *CCRC* which gives an overview of the system's calling context tree in order to support developers in software maintenance tasks, such as those mentioned in Section 2. *Senseo* now supports an additional metric, the number of executed bytecodes, which is useful for program optimizations (see Section 5)

Collaboration View. In a separate view next to the source code editor (Figure 1, (6)), *Senseo* presents all dynamic collaborators for the currently selected artifact. For instance, if a method has been selected, the collaboration view shows the collaborators at the package, class, or method level; that is, it lists all packages or classes invoking methods of the package or class in which the selected method is declared (callers). Furthermore, the collaboration view shows all packages or classes with which the package or class declaring the method is actively communicating (callees). For

[2]In this paper, the term *dynamic metrics* refers to numerically represented dynamic information, such as the number of method invocations, the number of object allocations, or the number of executed bytecodes. The term *dynamic information* subsumes dynamic metrics as well as information on run-time types.

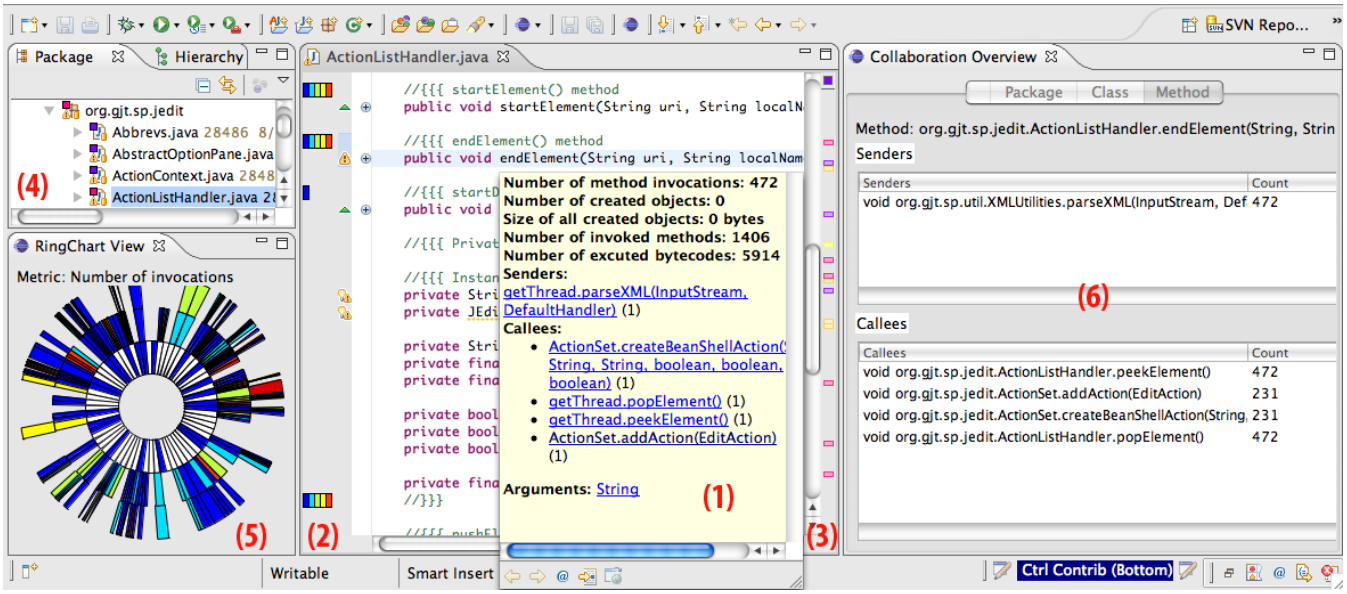

Figure 1: Overview of *Senseo* and all its techniques to integrate dynamic information in Eclipse

the method itself, the collaboration view lists all direct callers and callees.

Calling Context Ring Chart (CCRC). The CCRC, first introduced in [15][3], offers a condensed visualization of a Calling Context Tree (CCT) and provides navigation mechanisms to locate and explore subtrees of interest for the software maintenance task at hand. In a CCRC, the CCT root is represented as a circle in the center. Callee methods are represented by ring segments surrounding the caller's ring segment. A CCRC can display all calling contexts of a CCT in a single view, correctly preserving the caller/callee relationships conveyed in the CCT. For a detailed analysis of certain calling contexts, CCT subtrees can be visualized separately and the number of displayed tree layers can be limited. In order to reveal hot calling contexts with respect to a chosen dynamic metric, ring segments can be sized proportional to the aggregated metric contribution of the corresponding CCT subtree.

Senseo integrates a CCRC view of the CCT (Figure 1, (5)), which is interlinked with the static source view. The chart in the figure shows a subtree whose root has been selected by double-clicking on a calling context. For a selected calling context in the CCRC, the developer can switch to the corresponding method source. Vice versa, for a method in the source view, the methods' occurrences in the CCT (if any) can be highlighted and automatically selected one after the other.

While the CCRC implementation described in [15] visualizes only a single, aggregated dynamic metric, the new CCRC version integrated in *Senseo* supports visualization of a second dynamic metric (which may be aggregated for CCT subtrees or not) by coloring each ring segment according to the calling context's "hotness" with respect to the selected metric, as seen in Figure 1, (5) (note the CCRC displayed in this figure does not use proportional ring segment sizing). For instance, if the chosen dynamic metric for coloring is the number of method invocations (not aggregated), the most frequently invoked calling contexts are colored in red. Visualizing two different dynamic metrics within the same CCRC is particularly useful for program optimization. For example, sizing ring segments proportional to the number of executed bytecodes in a CCT subtree and coloring them according to the number of method invocations, reveals hot calling contexts (*i.e.*, large ring segments) and highlights whether a calling context is hot because the corresponding method is invoked frequently (*i.e.*, hot color red) or because of the method's (or its callees') internal complexity (*i.e.*, cold color blue). In the former case, optimizations should aim at reducing the number of method invocations, whereas in the latter case, optimizations should try reducing the complexity of the computation represented by the calling context. A concrete optimization performed with *CCRC* is illustrated as a case study in Section 5.

Dynamic Bytecode Metric. The new version of *Senseo* gathers an additional dynamic metric, the number of executed bytecodes in each calling context. The aggregation of this metric for a CCT subtree relates to the complexity of the corresponding computation; it is well suited for sizing ring segments in the CCRC visualization. In contrast to CPU time, the number of executed bytecodes is a largely platform-independent metric that can be easily collected without significant measurement perturbations [3, 4, 8].

In order to count the number of executed bytecodes without modifying the JVM, it is necessary to intercept the execution of each basic block of code, increasing a bytecode counter by the number of bytecodes in each executed basic block [3]. Because prevailing aspect frameworks such as AspectJ do not support pointcuts at the level of basic blocks of code, we extended *MAJOR* with a new basic block pointcut designator.

4. CONTROLLED USER EXPERIMENT

We conducted a controlled experiment with 30 professional Java developers to measure the expected impact of *Senseo* [20] enhanced with the contributions discussed in Section 3. We now describe the experimental design, the subjects, the evaluation procedure and the final results (including qualitative feedback) as well as threats to validity.

4.1 Experimental Design

This experiment aims at quantitatively evaluating the impact of *Senseo* and the dynamic information it integrates in the Eclipse IDE on developer productivity in terms of efficiently and correctly solving typical software maintenance tasks. We therefore analyze two

[3] http://inf.usi.ch/projects/ferrari/ccrc.pdf

variables in this experiment: *time spent* and *correctness*. This experiment also reveals which kind of tasks benefit the most from the availability of dynamic metrics in the IDE. The experimental design we opted for is similar to the one applied in the study of Cornelissen *et al.* [5] which evaluated a trace visualizing tool called *EXTRAVIS*.

Study Hypotheses. We claim that the availability of the *Senseo* plugin reduces the amount of time it takes to solve software maintenance tasks and that it increases the correctness of the solutions. Accordingly, we formulate the following two null hypotheses:

- $H1_0$: Having available *Senseo* does not impact the time for solving the maintenance tasks.
- $H2_0$: Having available *Senseo* does not impact the correctness of the task solutions.

Congruently, we formulate these two alternative hypotheses:

- H1: Having available *Senseo* reduces the time for solving the maintenance tasks.
- H2: Having available *Senseo* increases the correctness of the task solutions.

We test the two null hypotheses by assigning each subject to either a control group or an experimental group. While the experimental group has *Senseo* available for answering typical software maintenance tasks and questions, the control group uses a standard, unextended Eclipse IDE; otherwise there is no difference in treatment between the two subject groups. As both groups have nearly equal expertise, differences in time or solution correctness can be attributed to the availability of *Senseo*.

Study Participants. We asked 30 software developers working in industry (24) or with former industrial experience in software development (6) to participate in our experiment. Participation was voluntary and unpaid. All subjects answered a questionnaire asking for their expertise with Java, Eclipse and specific skills in software engineering, such as how often they work with unfamiliar code or how often they apply dynamic analysis. Most subjects (25) are mainly working with Java on their job, the others (5) mainly use another language but rely on Java at least in some of their professional projects. All participants are familiar with the Eclipse IDE.

The subjects have between one and 25 years of professional experience as a software engineer (average 4.8 years, median 4 years). 27 subjects have a university degree in computer science (Bachelors or Masters from 18 different universities) while three subjects either studied in another area or learned software engineering on the job. The subjects are very heterogeneous and thus fairly representative (seven different nationalities, working for eight different companies). In a Likert scale from 0 (no experience) to 4 (expert) subjects rated themselves on a level of 2.93 for Java experience, 2.73 for Eclipse experience and 2.72 for experience in working with unfamiliar code. All these ratings refer to "very experienced". With an average rating of 2.20, experience in applying dynamic analysis is slightly lower, but this rating is still considered as "quite experienced". Note that no subject claimed to have no experience in any of these four areas.

To assign the 30 subjects to either the experimental or the control group, we used the obtained expertise information to build two groups with equal expertise. To assess the expertise we considered four variables as given by the subjects: number of years of professional experience in software engineering, experience with Java, Eclipse and with maintaining unfamiliar code. For each subject we searched for a pair with similar expertise concerning these variables and then randomly assigned these two persons to either of the two groups. This leads to a very similar overall expertise in both groups as shown in Table 1.

Table 1: Average expertise in control and experimental group

Expertise variable	*Control group*	*Exper. group*
Years of experience	4.73	4.40
Java experience [0..4]	2.93	2.80
Eclipse experience [0..4]	2.80	2.67
Unfamiliar code exp. [0..4]	2.73	2.73
Mean	3.30	3.15

Subject System and Tasks. As a subject system we have chosen *jEdit*[4], an open-source text editor written in Java. JEdit consists of 32 packages with 5275 methods in 892 classes totaling more than 100 KLOC. We opted for jEdit as a subject system as it is medium-sized and representative for many software projects found in industry. JEdit has a long history of development spanning nearly ten years and involving more than ten developers. Even though it has been refactored several times, a careful analysis of the code quality revealed several design flaws, such as the use of deprecated code, tight coupling of many source entities to package-external artifacts, or lack of cohesion in almost all packages, which makes jEdit hard to understand. We expect many industrial systems to have similar quality problems, thus we consider jEdit to be a well-suited subject application fairly typical for many industrial systems developers come across on their job. Furthermore, the domain of a text editor is familiar to everyone, thus no special domain-knowledge is required to understand jEdit.

The tasks we gave the subjects are concerned with analyzing and gaining an understanding for various features of jEdit. While choosing the tasks, our main goal was to select tasks representative for real maintenance scenarios. Furthermore, these tasks must not be biased towards dynamic analysis. To assure that these criteria are met we selected the tasks according to the framework proposed by Pacione *et al.* [16]. They identified nine principal activities for reverse engineering and software maintenance tasks covering both static and dynamic analysis. Based on these activities they propose several characteristical tasks including all identified activities. We thus design our tasks following this framework to respect all nine principal activities, which avoids a potential bias towards *Senseo*.

This leads us to the definition of five tasks, each divided into two subtasks, resulting in ten different questions we asked to the subjects. Table 2 outlines all five tasks and their subtasks and explains which of Pacione's activities they cover. Task five is a special case since we use it as a "time sink task" to avoid ceiling effects [2]. Subjects that can answer the questions quickly might spend considerably more time on the last task when they notice that there is still much time available, thus the addition of a time-consuming task at the end which is not considered in the evaluation makes sure that subjects have a constant time pressure for all relevant tasks. The first four tasks also cover all of Pacione's activities.

All questions are open, that is, subjects cannot select from multiple choices but have to write a text in their own words. We graded the subjects answers by assigning scores from zero to four for each question. Before starting with the experiments, the two experimenters (who are also authors of this paper) answered all prepared questions. We compared and combined both solutions to form an answer model which we then used to grade the subjects' answers.

Experimental Procedure. We conducted the experiment with up to four subjects at the same time. We gave the subjects a short five minute introduction to the experiment setup. Subjects from the experimental group additionally received an introduction to the *Senseo* plugin lasting for 20 minutes. This introduction followed a script we prepared to ensure that every subject receives the same

[4] `http://www.jedit.org/`

Table 2: The five software maintenance tasks

Task	*Activities*	*Description*
1.1	A 1, 9	Feature understanding on a high architectural level
1.2	A 1, 4 ,5	Describing package collaborations in this feature
2.1	A 8	Assessing quality of three classes in terms of their external dependencies
2.2	A 4, 5, 6, 8	Describing coupling between the packages of these three classes
3.1	A 1, 3, 4, 5	Reporting about message sending and control flow in a class
3.2	A 1, 3, 5, 7	Locating clients of this class and analyzing their communication paths
4.1	A 4, 5, 8, 9	Comparing two features on a fine-grained method level to locate a defect
4.2	A 2	Correcting this defect by changing one feature to work similar as the other
5.1	A 4, 5, 6, 7	Exploring an algorithm in a specific class and report on its execution patterns
5.2	A 5, 6, 7, 8	Comparing this algorithm to another, similar algorithm in terms of efficiency

information about *Senseo*. Furthermore, we provided the *Senseo* subjects with a short description and a screenshot highlighting and explaining the core features of *Senseo*. This documentation served as a reference during the experiment.

Afterwards, we started the experiment. We supervised all subjects during the entire experiment and recorded the time they took to answer each question. Concerning infrastructure, each subject obtained the same pre-configured Eclipse installation we distributed in a virtual image. The only difference between the control group and the experimental group was the availability of *Senseo*, otherwise the Eclipse IDE was configured in exactly the same way. We provided the *Senseo* group with pre-recorded dynamic information obtained by running several features of jEdit, including all features and parts of the system covered by the five tasks. For the experiment the subjects used computers that meet the following minimum hardware requirements: 2.16 GHz Intel Core 2 Duo processors, 2 GB RAM, screen resolution of at least 1280x800.

Variables and Evaluation. The two dependent variables we study in this experiment are *time* the subjects spend to answer the questions, and *correctness*, that is, how correct are their answers to the tasks we pose. Keeping track of the answer time is straightforward as we prohibited going back to previously answered questions. We simply record the time span between the starting time of one question and the next. Correctness is measured using a score from 0 to 4 that expresses how closely the subject's answer tallies with the model answer.

The only independent variable in our experiment is whether the *Senseo* plugin is available in the Eclipse IDE to the subjects during the experiment.

We apply the parametric, one-tailed Student's t-test to test our two hypotheses at a confidence level of 95% (α=0.05). To validate that the t-test can be used, we first apply the Kolmogorov-Smirnov test to verify normal distribution and then Levene's test to verify equality of variances in the sample.

4.2 Results and Discussion

In this section we analyze the results obtained in the experiment. First, we evaluate the results for time and correctness. Second, we identify for which types of tasks the availability of dynamic information in the IDE is most useful. Finally, we evaluate the qualitative feedback we gathered by means of a debriefing questionnaire.

Table 3: Statistical evaluation of the experimental results

Group	*Mean*	*Stdev.*	*Diff*	*K.-S.*	*Lev F*	*t*	*p*
Time [m]:							
Eclipse	114.80	20.62		0.27			
Senseo	94.73	12.41	-17.5%	0.18	3.06	3.23	0.0016
Correctness (points):							
Eclipse	11.33	2.58		0.31			
Senseo	15.13	2.10	+33.5%	0.24	0.22	4.42	0.0001

Three subjects could not complete the time sink task (task 5) in the two hours we allotted, but all subjects started with this task, thus everybody finished the four relevant tasks.

Time Results. On average, the *Senseo* group spent 17.5% less time solving the maintenance tasks. The time spent by the two groups is visualized as a box plot in Figure 2, (1).

To statistically verify whether *Senseo* has an impact on the time to answer the questions, we test the null hypothesis $H1_0$ which says that there is no impact. We successfully applied the Kolmogorov-Smirnov and the Levene test on the time data (see Table 3), thus we are able to apply Student's t-test to evaluate $H1_0$. The application of the t-test allows us to reject the null hypothesis and instead accept the alternative hypothesis, which means that the time spent is statistically significantly reduced by the availability of *Senseo* as the p-value is with 0.0016 considerably lower than α=0.05 (see Table 3).

We attribute this result to several factors: (i) the availability of dynamic information in the source code views helps developers to more quickly gain an understanding how source artifacts communicate with each other, (ii) the visualizations of dynamic metrics such as number of method invocations shown in ruler columns and package tree enable developers to quickly spot which source elements are executed and how often, and (iii) as the collaboration view accurately presents all source artifacts that are related or collaborating to a selected source entity such as a package, class or method, developers can more quickly navigate to code relevant for a specific task. Note that *Senseo* was an unfamiliar plugin for all subjects, thus the results would be even better if participants had used *Senseo* in their daily work before doing the experiment.

Correctness Results. The *Senseo* group's answers for the four maintenance question are 33.5% more correct, which is also shown in the box plot in Figure 2, (2).

To test the null hypothesis $H2_0$, which suggests that there is no effect of the availability of *Senseo* on answer correctness, we are also allowed to use the Student's t-test as the Kolmogorov-Smirnov and the Levene test succeeded for the correctness data (compare Table 3). As the t-test gives a p-value of 0.0001 which is clearly below α=0.05, we reject the null hypotheses and accept the alternative hypothesis H2, which means that having available *Senseo* during software maintenance activities supports developers to more correctly solve maintenance tasks.

The availability of *Senseo* increases the correctness of answers probably due to the following reasons : (i) accurate overviews of collaborating artifacts or of calling contexts supports developers in exploring all relevant parts of the system to completely address a task, (ii) precise information about run-time collaboration or execution paths enables developers to accurately navigate to dependent artifacts, and (iii) information about execution complexity (number of method calls or number and size of created objects) eases the

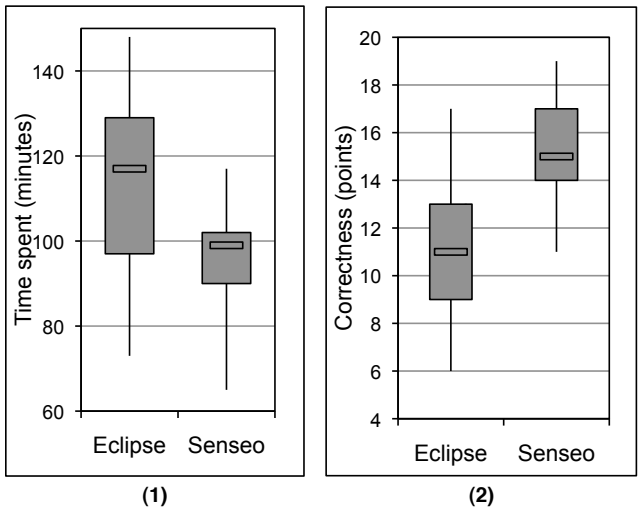

Figure 2: Box plots comparing time spent and correctness between control and experimental group

Table 4: Task individual performance concerning time required and correctness.

Task	Time [m]		Correctness (points)	
	Eclipse	*Senseo*	*Eclipse*	*Senseo*
Task 1	511	425 (-16.8%)	38	53 (+39.5%)
Task 2	388	340 (-12.4%)	58	79 (+36.2%)
Task 3	437	291 (-33.4%)	52	69 (+32.7%)
Task 4	386	365 (-5.4%)	22	26 (+18.2%)

correct identification of inefficient code.

Task-dependent Results. We also analyzed the two variables, time spent and correctness, for each task individually to reveal which kind of task benefit most from dynamic information integrated in Eclipse. Table 4 presents the aggregated results for time spent and correctness for each subject group and each task individually. Tasks 1, 2 and 3 benefit significantly from the availability of *Senseo* both in terms of time required to solve them and the correctness of the solution. However, for task 4 the benefit of *Senseo* is less pronounced. Coming back to the kind of tasks introduced in Section 2 that we wanted to support with *Senseo*, we can conclude that we successfully enhanced *Senseo* to aid developers performing such tasks. The experimental task 1 refers to task type (i), task 2 to type (ii) and (iii), and task 3 to type (iv), while for task 4 we consider lower level information as more relevant, for instance information on a method body level.

Qualitative Feedback. In the experiment we also collected qualitative feedback by means of a questionnaire to for instance evaluate the impact of particular parts of *Senseo* on specific kinds of maintenance tasks. This evaluation yields answers to the question which kind of dynamic information is actually relevant or useful in what kind of software maintenance tasks.

In Table 5 we illustrate for which tasks the subjects used which dynamic information integrated by *Senseo*, and Table 6 presents how useful subjects rated each technique of *Senseo* in a Likert scale from 0 (useless) to 4 (very useful).

From the evaluation asking for the use of dynamic information in specific tasks, we draw the conclusion that there are basically three kinds of tasks whose solving process is very well supported by the availability of dynamic information in IDEs: (i) tasks requiring developers to understand how different source artifacts collaborate or

Table 5: Percentage of subjects using specific dynamic information in particular tasks

Dynamic Information	*Task 1*	*Task 2*	*Task 3*	*Task 4*
Run-time types (Tooltip)	33%	47%	47%	20%
Number of invocations	53%	67%	40%	27%
Number of created objects	33%	47%	27%	13%
Number of exec. bytecodes	27%	33%	20%	7%
CCRC	7%	7%	0%	0%
Dynamic collaborators (callers, callees)	53%	80%	73%	33%

Table 6: Mean ratings of the subjects for each feature of *Senseo*

Dynamic Information	*Mean rating [0..4]*
Tooltip showing run-time types	3.6
Ruler column incl. dynamic metrics	3.2
Overview ruler column incl. dyn. metrics	3.0
Package tree incl. dynamic metrics	2.4
CCRC	2.1
Collaboration view	3.7

depend on each other, (ii) tasks in which developers have to assess how often code is executed or how complex it is, and (iii) tasks that require the developer to understand which code is related to a given feature. This conclusion agrees with the quantitative results discussed earlier where we revealed that task 1 (feature and collaboration understanding), task 2 (quality assessment) and task 3 (control flow understanding) benefited most from the availability of *Senseo* while for task 4 (low level defect correction) dynamic information was less useful.

From the results evaluating the different *Senseo* concepts, we conclude that developers particularly benefit from the availability of the collaboration views and run-time type information in source code. Also considered useful are visualizations of dynamic metrics in the source code columns such as the presentation of number of invoked methods in a method or class. The aggregated dynamic metrics presented in the package tree are perceived as less useful by the developers, probably because it is not meaningful to study run-time complexity on a high package level. The subjects also could not benefit from the *CCRC* as this visualization serves the rather specialized task of performance optimization which has not been directly covered by the maintenance tasks of the experiment. Thus we elaborate in a separated case study in Section 5 on the *CCRC*'s usefulness for performance optimizations.

4.3 Threats to Validity

In this section we discuss several threats to validity concerning this experiment. We distinguish between (i) construct validity, that is, threats due to how we operationalized the time and correctness measures, (ii) internal validity, that is, threats due to inferences between treatment and effect during the analysis, and (iii) external validity which refers to threats concerning the generalization of the experiment results.

Construct Validity. Due to the operationalization of the time and correctness variables, the results might not hold in real, non-experimental situations. For instance, subjects could have been more attentive than they would be in their daily job, or they might have guessed the experimental goal and acted accordingly, or were more anxious as they were observed and could have assumed that their personal performance was evaluated. In general, the testing of

the treatment, the (un)availability of *Senseo*, could have influenced the outcome of the experiment. However, we consider this threat to be negligible as the experimental goal was not revealed to subjects. At the same time we made clear that we do not evaluate their personal performance (we anonymised their answers), and tried to use a familiar, non-artificial atmosphere by conducting the experiment with most subjects in their own office using their own computer if it fulfilled the requirements for the experiment, see Section 4.1.

Internal Validity. Some threats to internal validity originate from the subjects. First, subjects might not have the required expertise to properly solve the maintenance tasks. This threat is largely eliminated by preliminary assessing subject's expertise concerning their Java, Eclipse, or software maintenance skills. Additionally, we required them to not have expert knowledge in developing *jEdit*. Second, the experimental group might have had more knowledge than the control group. This threat is tackled by assigning the subjects in a randomized manner to the two groups in a way that both groups have nearly equal expertise (see Table 1).

Other threats to internal validity stem from the maintenance tasks we prepared. First, the tasks could have been too difficult or time-consuming to solve. This threat is refuted by the fact that nearly all subjects from both groups could solve all tasks in time (except two from the control group and one from the *Senseo* group). Moreover, each question was answered fully correctly by at least one person from each group. Additionally, we asked subjects in the questionnaire directly how they judged the time pressure and the difficulty. On average, the ratings were 2.8 for time pressure (representing "felt no time pressure") and 3.1 for average difficulty of all tasks (which means "appropriately difficult"). Second, the threat that we formulated tasks favoring *Senseo* is largely limited as we used Pacione's established framework [16] to find the tasks used in the experiment. Third, a threat for the correctness evaluation is that the experimenters might have favored *Senseo* while grading subjects' answers. By initially building an answer model according to which the subjects answers were graded, we mitigated this threat. For the obtained answers the experimenters gave points as pre-defined in the answer model which in turn has been formulated and validated by two persons individually.

External Validity. Generalizing the results of the experiment could be unjustified due to the selection of tasks, subjects, or the application used in the experiment. This threat is mitigated since we selected the maintenance tasks carefully to follow Pacione's framework [16] of representative maintenance tasks. We furthermore asked open questions to the subjects to better model industrial reality than would be possible with multiple choice questions.

The literature suggests avoiding experimental groups consisting of only students [17]. We therefore selected subjects who all have professional experience in industry as software developers as mentioned in Section 4.1. As the subjects also work for different companies and have a high variety of education profiles, the study participants should be fairly representative for professional software developers and thus not impose a threat to generalization.

In Section 4.1 we described several reasons why jEdit is representative for many industrial systems. Additionally, we asked subjects at the end of the experiment how comparable in terms of maintainability they consider *jEdit* to be to systems they daily work with. On average, they gave on a Likert scale from 0 (totally different) to 4 (very representative) a rating of 3.1, which refers to "many similarities". Hence we are confident to have found with *jEdit* a system representative for most industrial applications.

5. CCRC CASE STUDY

The CCRC is especially useful for locating performance bottlenecks. As such a task was not directly part of our user experiment, we present here a case study inspired by [4], concerned with a performance optimization task using CCRC. In this case study, we illustrate how CCRC helps locate a performance problem in the lexical analyzer generator JLex [11], which is afterwards optimized. We use the aggregated number of executed bytecodes as dynamic metric for sizing ring segments; in the CCRC, the ring segments are ordered counterclockwise by their corresponding metric contribution.

Figure 3 (top) shows a CCRC representing the execution of the original, unmodified JLex for a sample grammar included in the JLex distribution. Looking at the aggregated bytecodes and navigating through the different elements of the call stack, we immediately locate a hot calling context, an invocation of the `sortStates(...)` method. This method executes a high number of bytecodes, contributing 23.6% of the overall executed bytecodes. The large ring segment corresponding to this calling context is highlighted in Figure 3 (top).

The `sortStates(...)` method uses a primitive selection sort algorithm of complexity $O(n^2)$. In order to optimize the code, we replace it with the merge sort algorithm of complexity $O(n\ log\ n)$. Figure 3 (bottom) shows the resulting CCRC after the optimization. The number of executed bytecodes in the previously hot calling context representing the `sortStates(...)` method is reduced to 10.86% of the overall executed bytecode. Comparing the top and bottom CCRCs in Figure 3 shows that the ring segment corresponding to the sorting functionality, which before had the largest contribution of executed bytecodes, is shifted counterclockwise after the optimization, since another ring segment has a higher contribution. In contrast to the primitive selection sort used in Figure 3 (top), the merge sort algorithm visualized in Figure 3 (bottom) uses recursion, which explains the different shape of the corresponding sub-segments of the `sortStates(...)` method.

In order to confirm that our optimization based on the number of executed bytecodes also results in a speedup in execution time, we run both versions of JLex (original and optimized) on an Intel Core 2 Duo 2.33 Ghz computer with 2 GB RAM (Linux Fedora 10), using Sun JDK 1.6.0_12. Regarding the execution time, we take the median of 15 runs. The original JLex executes 20,393,685 bytecodes in 37.04ms, whereas the optimized JLex executes 16,955,185 bytecodes in 27.7ms. In the bytecode metric, the optimized version is 20.3% "faster", while in the CPU time metric, the optimized version is 33.7% faster. While the number of executed bytecodes is exactly reproducible in each run, the measured execution time varies considerably, but the optimized version is consistently faster.

6. PERFORMANCE

The prototype of *Senseo* [20] suffered from excessive overhead, because it used a naive, non-optimized aspect for collecting dynamic information and always transmitted the complete CCT to the Eclipse plugin with Java's standard serialization mechanism. Already for medium-sized applications, serialization introduced long latencies and yielded transmission data of several gigabytes.

The new version of *Senseo* includes two essential optimizations. First, we optimized the aspect that gathers dynamic information. Wherever run-time type information can be statically inferred, the new aspect avoids expensive access to dynamic context information through AspectJ's reflection API. For example, if all formal method arguments are of a primitive or final type, the actual argument types cannot vary at run-time and therefore need not be collected.

Second, we hand-crafted an optimized serialization mechanism that transmits the CCT in an incremental way, sending only those nodes where some dynamic information has changed since the pre-

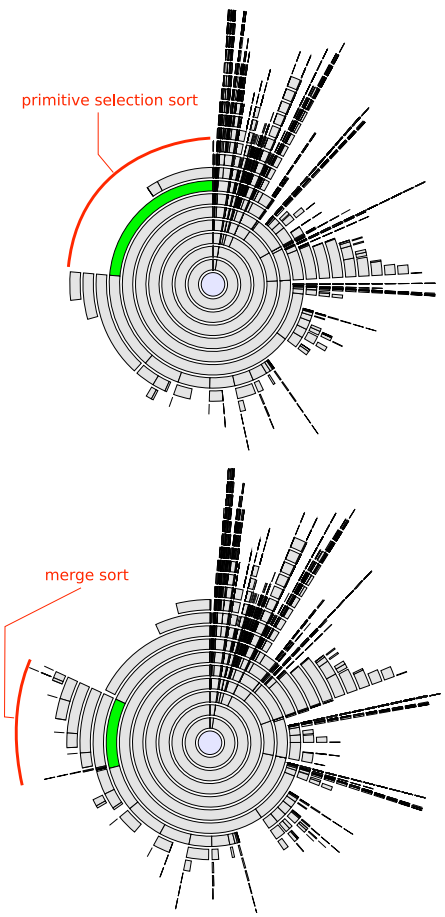

Figure 3: CCRCs for JLex with the original (top) and optimized (bottom) sort algorithm

vious transmission. In addition, we optimized the data structures that store dynamic information, since they are accessed extremely frequently. Thanks to the principle of locality, typically only a small subset of the CCT nodes is transmitted. Thus, it is now possible to frequently update the dynamic information in the Eclipse plugin, such as once per second. Our new serialization format includes a name table (types, methods, signatures), as well as compact representations of the CCT nodes and the gathered dynamic information using only integer arrays.

In order to validate that the new *Senseo* version offers sufficient performance for coping with real-world workloads, we evaluated the different sources of overhead and analyzed the amount of transmitted data for the DaCapo benchmarks[5]. For our measurements, we use *MAJOR*[6] version 0.6 with AspectJ[7] version 1.6.5 and the SunJDK 1.6.0_13 Hotspot Server Virtual Machine. We execute the benchmarks on a quadcore machine running CentOS Enterprise Linux 5.3 (Intel Xeon, 2.4GHz, 16GB RAM).

Figure 4 shows the overhead for CCT creation, collection of dynamic information (including the number of method invocations, the number of object allocations, the estimated allocated bytes, the number of executed bytecodes, and the run-time receiver, argument, and return value types), as well as serialization and data transmission to the Eclipse plugin, including processing of the received data by the plugin. In this measurement setting, each benchmark is executed 15 times and the median execution time is taken for computing the overhead. For each run of each benchmark, the CCT and the gathered dynamic information are serialized and transmitted once upon benchmark completion. To this end, we modify the DaCapo benchmark harness in order to delay the end of a measurement until the transmitted data have been received and processed by the Eclipse plugin. Figure 4 also shows the average overhead (geometric mean) for the DaCapo suite.

[5] http://dacapobench.org/
[6] http://www.inf.usi.ch/projects/ferrari/
[7] http://www.eclipse.org/aspectj/

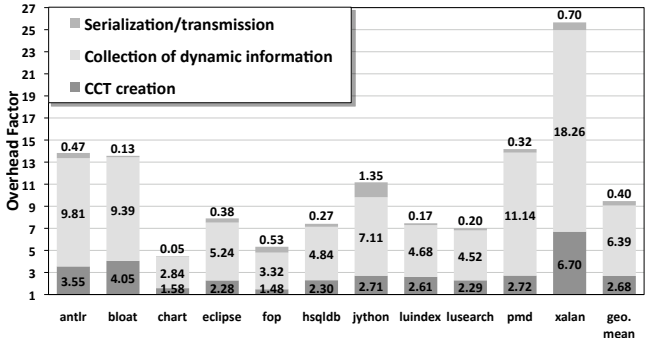

Figure 4: *Senseo* overhead for the DaCapo benchmarks

On average (geometric mean), CCT creation alone causes an overhead of factor 2.68. CCT creation and collection of dynamic information result in an overhead of factor 9.07. The total overhead, including serialization/transmission, is of factor 9.47. For all benchmarks, the larger part of the overhead is due to the collection of dynamic information, where the collection of run-time type information is particularly expensive. Serialization/transmission causes only minor overhead, because in these measurement settings serialization/transmission happens only once upon benchmark completion.

We also measured the speedup of the new *Senseo* version relative to the former prototype presented in [20] in the same settings used for Figure 4 (i.e., collection of all dynamic information, one serialization/transmission upon benchmark completion).[8] On average (geometric mean for DaCapo), the new *Senseo* version is 31 times faster than the previous prototype.

Thanks to the incremental serialization mechanism in the new *Senseo* version, it is possible to frequently transmit CCT and dynamic information from a running application to the Eclipse plugin. Figure 5 illustrates the size of successively transmitted data packets (including name table, CCT nodes, and dynamic information) for a single run of DaCapo's 'eclipse' benchmark with a serialization/transmission rate of 1.25 packets per second.[9] Such a high serialization/transmission rate ensures that the developer always sees up-to-date dynamic information in the IDE, refreshed more than once per second, while the application under maintenance is running in the *MAJOR* JVM. In total, 370 packets are sent, that is, the total run-time of 'eclipse' is about 296s in this setting (causing an overhead factor of 14.8, whereas a single serialization/transmission upon benchmark completion induces an overhead of factor 7.9 as shown in Figure 4). For each packet, Figure 5 differentiates between the size of the transmitted CCT nodes (including the name

[8] Note that the measurements of the former prototype reported in [20] are not comparable, because they excluded the collection of run-time type information as well as serialization/transmission.
[9] Here we evaluate the size of transmitted packets only for 'eclipse', which has the longest execution time in the DaCapo suite in our measurement environment. Due to space limitations, figures showing packet sizes for the other benchmarks cannot be presented here.

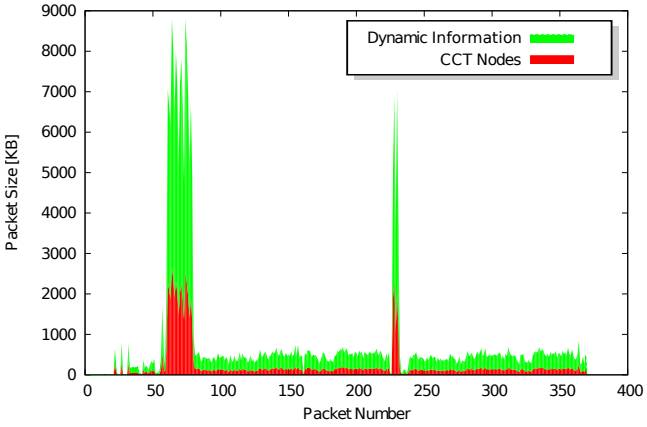

Figure 5: Size of transmitted data packets for 'eclipse'. Serialization/transmission rate: 1.25 packets per second

table) and the size of the sent dynamic information.

While most packets are rather small, below 1MB, some packets are considerably larger, reaching up to 9MB. The packets 60–79 appear as a major peak in the figure. We found that these packets convey dynamic information collected while the 'eclipse' benchmark is compiling some projects. The minor peak in Figure 5 (packets 227–232) corresponds to some XML data processing. The initial packets, collecting during the startup phase of 'eclipse', are very small. This can be explained by the fact that the startup phase is IO-intensive and involves much class-loading and just-in-time compilation by the JVM, which are mostly implemented in native code and are therefore not amenable to *MAJOR*'s instrumentation.

We conclude that the new *Senseo* version is fast enough to cope even with large-sized applications, and it is possible to frequently transmit the collected dynamic information to the Eclipse plugin, continuously providing up-to-date dynamic information to the software developer.

7. RELATED WORK

First, we report on related work in the area of dynamic analysis. Second, several IDE enhancements to improve software navigation are not based on dynamic analysis but employ different techniques such as relating source artifacts to each other based on navigation patterns. We briefly describe and compare these related works to our approach. Third, we discuss controlled experiments conducted by other researchers in a similar area as ours.

Dynamic information gathering. In [8], Dufour *et al.* present a variety of dynamic metrics for Java programs. They introduce a tool called *J [9] for metrics measurement. *J relies on the Java Virtual Machine Profiler Interface (JVMPI), which is known to cause high performance overhead and requires profiler agents to be written in native code. In contrast, our aspect-based approach enables high-level specification of instrumentations, which is flexible, fully portable and introduces moderate overhead.

Profiling capabilities have been integrated in IDEs such as the NetBeans Profiler[10] and Eclipse's Tracing and Profiling Project (TPTP)[11]. The NetBeans Profiler uses JFluid technology [7]. JFluid exploits dynamic bytecode instrumentation and code hotswapping to collect dynamic metrics. JFluid uses a hard-coded, low-level instrumentation to collect gross time for a single code region and to build a Calling Context Tree (CCT) augmented with accumulated execution time for individual methods. In contrast, we use a flexible, high-level, aspect-based approach to specify CCT construction and dynamic metrics collection, which eases customization and extension. Similar to *Senseo*, JFluid runs the application under instrumentation in a separate JVM, which communicates with the visualization part through a socket and also through shared memory. JFluid is a pure profiling tool, whereas *Senseo* was designed to support program understanding and maintenance.

Dynamic analyses based on tracing mechanisms traditionally focus on capturing a call tree of message sends, but existing approaches do not bridge the gap between dynamic behavior and the static structure of a program [10, 23]. Our work aims at incorporating the information obtained through dynamic analyses into the IDE and thus connecting the static structure with the dynamic behavior of the system. At the same time, we aggregate the information of multiple program executions and feed back the results in the IDE, corresponding to inductive program analysis in [24].

IDE enhancements. Other works also visualize software dynamics, but usually these approaches are integrated in a separate tool and not directly in the IDE. Reiss [19] for instance developed a tool visualizing the behavior of Java programs in real-time. Löwe *et al.* [14] merges both, information from static and dynamic analysis to generate visualizations in a dedicated tool. Lanza *et al.* [13] contributed CodeCrawler, a stand-alone tool to analyze statics and dynamics of programs.

To help developers navigating software systems, several works rely on other techniques than program analysis. For instance, NavTracks [21] exploits the navigation history of software developers to form associations between related source files and presents a recommendation list of entities related to the current selected source file. Mylyn [12] computes a degree-of-interest value for each source artifact by analyzing navigation history. The relative degree-of-interest of artifacts is highlighted using colors — interesting entities are assigned a "hot" color. We also use heat colors to denote the degree of participation in a particular dynamic metric such as number of invocations. Fluid source code views [6] present related code (for instance an invoked method) directly in the current source code view in an additional widget. Such a view recognizes the separated but linked nature of source artifacts. However, fluid source code views statically link separated source artifacts together and may thus identify wrong or unrelated candidate methods at polymorphic call sites.

Controlled experiments in software engineering. Other researchers also conducted controlled experiments to validate tools supporting software maintenance tasks: Cornelissen *et al.* [5] evaluated a trace visualizing tool with 24 student subjects. Quante *et al.* [18] evaluated with 25 students the benefits of Dynamic Object Process Graphs (DOPGs) for program comprehension.

8. CONCLUSIONS

In this paper we present the Eclipse plugin *Senseo* which integrates various dynamic information in the Eclipse IDE, such as run-time types in the source code views, dynamic metrics (*e.g.*, the number of executed bytecodes) in package tree and ruler columns, a collaboration view presenting all dynamic collaborators of a selected source artifact (package, class, or method), and the *CCRC*, a navigable visualization of the Calling Context Tree enriched with dynamic metrics that help quickly locate hot spots in programs.

We carefully evaluate the impact of *Senseo* on the productivity of developers performing software maintenance tasks by conducting a controlled experiment with 30 professional developers. This experiment reveals that the participants spend 17.5% less time on the maintenance tasks while at the same time providing 33.5% more

[10] `http://profiler.netbeans.org/`
[11] `http://www.eclipse.org/tptp/performance/`

correct answers. Furthermore, we present a case study illustrating how the *CCRC* is used for optimizing an application. A thorough performance evaluation with the DaCapo benchmarks confirms that *Senseo* is able to cope with large-sized applications and enables a high refresh rate for displaying dynamic information on a running application in Eclipse.

While *Senseo* currently focuses on the inter-procedural control flow represented by the Calling Context Tree, our ongoing research aims at capturing also the intra-procedural control flow, which will offer additional support for program optimization. Moreover, we are integrating other dynamic analyses, such as memory leak and data race detectors. Thanks to our aspect-oriented approach, the development and integration of such advanced features will be completed shortly.

Acknowledgments. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "Bringing Models Closer to Code" (SNF Project No. 200020-121594, Oct. 2008 – Sept. 2010) and "FERRARI – Framework for Efficient Rewriting and Reification Applying Runtime Instrumentation" (SNF Project No. 200021-118016/1, Oct. 2007 – Sept. 2010).

9. REFERENCES

[1] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1997.

[2] E. Arisholm, H. Gallis, T. Dyba, and D. I. Sjoberg. Evaluating pair programming with respect to system complexity and programmer expertise. *IEEE Transactions on Software Engineering*, 33(2):65–86, 2007.

[3] W. Binder, J. Hulaas, P. Moret, and A. Villazón. Platform-independent profiling in a virtual execution environment. *Software: Practice and Experience*, 39(1):47–79, 2009.

[4] B. F. Cooper, H. B. Lee, and B. G. Zorn. ProfBuilder: A package for rapidly building Java execution profilers. Technical Report CU-CS-853-98, University of Colorado at Boulder, Department of Computer Science, Apr. 1998.

[5] B. Cornelissen, A. Zaidman, A. van Deursen, and B. van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *Proceedings 17th International Conference on Program Comprehension (ICPC)*, pages 100–109. IEEE Computer Society, 2009.

[6] B. Desmet, J. Vallejos, and P. Costanza. Introducing mixin layers to support the development of context-aware systems. In *3rd European Workshop on Aspects in Sofware*, 2006.

[7] M. Dmitriev. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *WOSP '04: Proceedings of the Fourth International Workshop on Software and Performance*, pages 139–150, 2004.

[8] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. *ACM SIGPLAN Notices*, 38(11):149–168, Nov. 2003.

[9] B. Dufour, L. Hendren, and C. Verbrugge. *J: A tool for dynamic analysis of Java programs. In *OOPSLA '03: Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 306–307, New York, NY, USA, 2003. ACM Press.

[10] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[11] Elliot Berk. JLex: A Lexical Analyzer Generator for Java. Web pages at `http://www.cs.princeton.edu/~appel/modern/java/JLex/`, 2003.

[12] M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for ides. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 159–168, 2005. ACM Press.

[13] M. Lanza, S. Ducasse, H. Gall, and M. Pinzger. Codecrawler — an information visualization tool for program comprehension. In *Proceedings of ICSE 2005 (27th IEEE International Conference on Software Engineering)*, pages 672–673. ACM Press, 2005.

[14] W. Löwe, A. Ludwig, and A. Schwind. Understanding software - static and dynamic aspects. In *17th International Conference on Advanced Science and Technology*, pages 52–57, 2001.

[15] P. Moret, W. Binder, D. Ansaloni, and A. Villazón. Visualizing Calling Context Profiles with Ring Charts. In *VISSOFT 2009: 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, Edmonton, Alberta, Canada, Sep. 2009.

[16] M. Pacione, M. Roper, and M. Wood. A novel software visualisation model to support software comprehension. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 70–79. IEEE Computer Society, 2004.

[17] M. D. Penta, R. E. K. Stirewalt, and E. Kraemer. Designing your next empirical study on program comprehension. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 281–285, Washington, DC, USA, 2007. IEEE Computer Society.

[18] J. Quante. Do dynamic object process graphs support program understanding? - a controlled experiment. In *Proceedings of the 16th International Conference on Program Comprehension (ICPC'08)*, pages 73–82, Washington, DC, USA, 2008. IEEE Computer Society.

[19] S. P. Reiss. Visualizing Java in action. In *Proceedings of SoftVis 2003 (ACM Symposium on Software Visualization)*, pages 57–66, 2003.

[20] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Senseo: Enriching eclipse's static source views with dynamic metrics. In *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*, Los Alamitos, CA, USA, 2009. IEEE Computer Society. To appear.

[21] J. Singer, R. Elves, and M.-A. Storey. NavTracks: Supporting navigation in software maintenance. In *International Conference on Software Maintenance (ICSM'05)*, pages 325–335, Washington, DC, USA, sep 2005. IEEE Computer Society.

[22] A. Villazón, W. Binder, and P. Moret. Aspect Weaving in Standard Java Class Libraries. In *PPPJ '08: Proceedings of the 6th International Symposium on Principles and Practice of Programming in Java*, pages 159–167, 2008. ACM.

[23] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

[24] A. Zeller. Program analysis: A hierarchy. In *Proceedings of the ICSE 2003 Workshop on Dynamic Analysis*, pages 6–9, 2003.