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

David Röthlisberger, Marcel Härry, Walter Binder, Philippe Moret,
Danilo Ansaloni, Alex Villazón, Oscar Nierstrasz

Abstract—Modern IDEs such as Eclipse offer static views of the source code, but such views ignore information about the runtime 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 runtime architecture. In this article we present an approach to gather and integrate dynamic information in the Eclipse IDE with the goal of better supporting typical software maintenance activities. By means of a controlled experiment with 30 professional developers we show 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%. We also provide a comprehensive performance evaluation of our approach.

Index Terms—D.2.3.a Object-oriented programming, D.2.6.c Integrated environments, D.2.7.m Restructuring, reverse engineering, and reengineering, D.2.8.a Complexity measures, D.2.8.b Performance measures

◆

1 INTRODUCTION

MAINTAINING object-oriented systems is complicated by the fact that conceptually related code is often scattered over a large source space. The use of inheritance, interface types, and polymorphism leads to code that is hard to understand, as it is unclear which concrete methods are invoked at runtime at a polymorphic call site even in statically-typed languages. As integrated development environments (IDEs) typically focus on browsing static source code, they provide little help to reveal the execution paths a system actually takes at runtime. Being able to reconstruct the execution flow of a system while working in the IDE can, however, lead to better program understanding and to more focused navigation in the source code. In this article we show that developers can more efficiently maintain object-oriented code in the IDE if the static views of the IDE are augmented with dynamic information.

The importance of execution path information becomes clear when inspecting Java applications employing abstract classes or interfaces. The source code of such applications usually refers to these abstract types, while at runtime concrete subtypes are used. However, if the source code just refers to abstract types, it can be difficult to identify the concrete classes actually used at runtime, since there may exist a large number of concrete implementations. Similarly, when examining source code that invokes a particular method, a large list of candidate method implementations may be generated. Static analysis alone will not tell you how frequently, if at all, each of these candidates is actually invoked. Such information is nevertheless crucial to assess the performance impact of particular code statements.

Developers usually resort to debuggers to determine the actual execution flow of an application. Unfortunately, information extracted during a debugging session is volatile, that is, it disappears at the end of the session. Furthermore, such information is bound to a specific execution; in general, it cannot be used to tell which runtime types occur how often at a specific place in source code. To analyze and improve the performance of a system, developers typically use profilers, which suffer from similar drawbacks as debuggers: the collected dynamic information is not integrated in the static source views in the IDE; developers use such tools only occasionally instead of continuously benefiting from dynamic information that is directly available in the static source views.

We present an approach to dynamically analyze systems and to augment the static views of IDEs with dynamic information. We implemented this approach in *Senseo*, an Eclipse plugin that enables developers to dynamically analyze Java applications. *Senseo* enriches the source views of Eclipse with several kinds

- D. Röthlisberger, M. Härry, and O. Nierstrasz are with the Software Composition Group at the University of Bern, Switzerland.
- W. Binder, P. Moret, and D. Ansaloni are with the University of Lugano, Switzerland.
- A. Villazón is with the Centro de Investigaciones de Nuevas Tecnologías Informáticas (CINTI) at the Universidad Privada Boliviana (UPB), Cochabamba, Bolivia. The research presented in this article was conduced while A. Villazón was with the University of Lugano, Switzerland.

of dynamic information such as presenting which concrete methods a particular method invokes how often at runtime, which methods invoke this particular method, and how many objects or how much memory is allocated in particular methods. The gathered information is aggregated over several runs of the subject system; the developer decides which runs to take into account. *Senseo* also contributes two other means to integrate dynamic information: first, a view on the dynamic collaborations between different source artifacts, which illustrates communication at the level of packages, classes and methods, and, second, the Calling Context Ring Chart (*CCRC*) [1], a navigable visualization of the system's Calling Context Tree (CCT) [2].

To validate the practical usefulness of *Senseo*, we conducted a controlled user experiment with 30 professional Java developers to obtain reliable quantitative and qualitative feedback about the impact on developer productivity contributed by *Senseo* and the dynamic information it integrates in Eclipse. The subjects solved five typical software maintenance tasks in an unfamiliar, medium-sized software system. While half the subjects only used the standard Eclipse IDE, the other half additionally used the *Senseo* plugin. The experiment shows that the availability of dynamic information as provided by *Senseo* yields a significant decrease in time of 17.5% and a significant increase in correctness of 33.5%.

We contribute (i) an approach to gather and integrate dynamic information in the Eclipse IDE, (ii) a controlled user experiment to validate the practical usefulness of the approach, and (iii) a detailed performance evaluation of *Senseo*, our implementation of the approach. With respect to our prior work [3], [4], (ii) and (iii) are novel, original contributions.

The article is structured as follows: In Section 2 we present a use case motivating the need for dynamic information within the IDE. Section 3 introduces *Senseo*, a plugin that integrates dynamic information in the Eclipse IDE. Section 4 explains our approach to gather dynamic information from a running application. Section 5 validates the practical usefulness of *Senseo* for software maintenance tasks with a controlled experiment involving 30 professional developers. Section 6 reports on the efficiency of *Senseo*. Section 7 presents related work. Finally, Section 8 concludes the article.

2 MOTIVATION

SENSEO aims at improving understanding and maintenance of object-oriented software systems by providing the developer dynamic information collected from multiple runs of an application, such as from the execution of unit tests. In order to motivate the need for exposing dynamic information in the IDE, we consider the Eclipse JDT[1], a set of plug-

1. http://www.eclipse.org/jdt

ins implementing the Eclipse Java IDE. JDT encompasses interfaces and classes modeling Java source code artifacts, such as classes, methods, fields, or local variables. Clients of this representation usually refer to interface types, such as `IJavaElement` or `IJavaProject`, as the following code snippet found in `JavadocHover` illustrates:

```
IJavaProject javaProject = null;
IJavaElement element = elements[0];
if (element.getElementType()==IJavaElement.FIELD){
   javaProject = element.getJavaProject();
} else if (element.getElementType() ==
         IJavaElement.LOCAL_VARIABLE) {
   javaProject = element.getParent()
                     .getJavaProject();
}
```

This code is difficult to understand due to the lack of information about runtime types of variables and any other dynamic information: (i) it is unclear which `getJavaProject` methods are invoked at runtime; (ii) the variable `javaProject` could still be `null` at the end of the code snippet, as not all possible types of elements might be covered by the conditionals; (iii) the execution frequency of this code and thus its performance impact is unknown.

These questions cannot be easily answered using only the IDE's static source views because there are more than ten different implementations of the method `getJavaProject` in the JDT, thus, we do not know which implementations are actually used. Furthermore, JDT contains many interfaces and classes implementing `IJavaElement`, therefore, we cannot statically determine which types of elements are used at runtime in this code.

Using a debugger, we find out that `element` is of type `SourceField` in one scenario. However, we know that debuggers focus on specific runs, thus we still cannot know all the different types `element` has in this code. To reveal all types of `element` and all `getJavaProject` methods invoked by this polymorphic call site, we would have to debug many more scenarios, which is very time-consuming as this code is executed many times for each system run.

For all these reasons, it is much more convenient for a developer if the IDE itself could show dynamic information aggregated over several runs within the static source views, that is, Eclipse's source code viewer should show precisely which methods are invoked at runtime, including detailed runtime types for receiver, arguments, and return values. In addition, information about the number of method invocations or object allocations helps developers identify performance bottlenecks in an application. If developers are interested in a specific execution, *Senseo* also allows them to just analyze the information from this single scenario. If source code enriched with dynamic information changes, the recorded dynamic information about this piece of code is invalidated. For instance, if a method changes, we invalidate all

dynamic information about this method itself and all its directly or indirectly invoked methods.

3 INTEGRATING DYNAMIC INFORMATION IN IDEs

I N this section we present an approach to augment IDEs with dynamic information, towards the goal of supporting the understanding of runtime behavior of applications. First, we present the architecture of *Senseo*, an Eclipse plugin implementing our approach. Second, we discuss different kinds of dynamic information that can support program understanding. Third, we illustrate how *Senseo* integrates and visualizes such dynamic information within Eclipse. More information about *Senseo* is available in a master's thesis [5] concerned with this project and on its website[2] where it is also available for download.

3.1 Architecture

Dynamic information can be collected using a modified Java Virtual Machine (JVM), with a profiling agent in native code using the standard JVM Tool Interface (JVMTI), or with the aid of program transformation or bytecode instrumentation techniques. For portability and compatibility reasons, we chose the last approach. Instead of using a low-level bytecode engineering library to instrument code, we use high-level aspect-oriented programming (AOP) [6] to specify instrumentation as an aspect. This approach not only results in a compact implementation, but it also ensures ease of maintenance and extension.

Because it is important that the gathered dynamic information covers the complete program execution, *Senseo* uses *MAJOR* [7], [8], an aspect weaver that supports comprehensive aspect weaving into every class linked in a JVM, including the standard Java class library and dynamically loaded or generated classes. *MAJOR* is based on the standard AspectJ [9] compiler and weaver and uses advanced bytecode instrumentation techniques to ensure portability [10].

The application to be analyzed is executed in a separate application JVM where *MAJOR* weaves the data-gathering aspect into every loaded class, while the Eclipse IDE with the *Senseo* plugin runs in a standard JVM to avoid perturbations. While the subject system is still running, the gathered dynamic data is periodically transmitted from the application JVM to Eclipse using a socket. We do not have to halt the application to obtain its dynamic data. *Senseo* receives the transfered data, processes it, and stores the aggregated information in its own storage system which is optimized for fast access from the IDE (see Fig. 1).

To analyze the application dynamically within the IDE, developers have to execute it with *Senseo*. Before
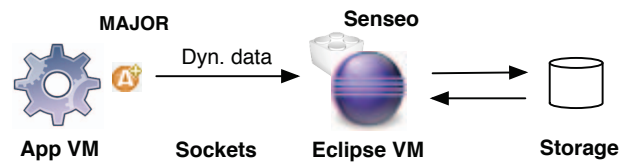
2. http://scg.unibe.ch/research/senseo

Fig. 1. Setup to gather dynamic information.

starting the application, developers can define what kind of dynamic information should be gathered at runtime. By default, all packages and classes of the application and the Java class library are dynamically analyzed. However, developers can restrict the analysis to specific classes or even methods to reduce the analysis overhead if only specific areas need to be observed. *Senseo* aggregates dynamic information over all application runs executed with it, but developers can clear the store and start afresh.

3.2 Dynamic Information

The *Senseo* plugin integrates the following dynamic information into the IDE: (i) method invocation, that is, which method was invoked with which receiver and argument types. Additionally, return types are recorded as well. (ii) The number of invocations is recorded to ease locating frequently invoked methods or unused code. (iii) Information about number of created objects helps developers to locate expensive code while (iv) *allocated memory* informs about the actual size of the created objects. The last three kinds of dynamic information are particularly well suited to identify and optimize inefficient source artifacts.

To gather such dynamic information, *Senseo* relies on the CCT. The CCT [2] allows dynamic information to be collected separately for each calling context. A calling context is a stack of methods that have been invoked but have not yet completed. The CCT helps the dynamic inter-procedural control flow of an application to be analyzed.

3.3 Enhancements to the IDE

We now describe how these different kinds of dynamic information are presented in Eclipse by *Senseo*.
Source code enhancements. We use *tooltips*, small windows that pop up when the mouse hovers over a source element to complement source code without impeding its readability. *Senseo* tooltips are interactive; that is, the developer can open the class of a receiver type by clicking on it.
Method header tooltip. When the mouse hovers over the method name in a method header, the tooltip shows (i) all callers invoking that particular method, (ii) all callees of the method, and, optionally, (iii) all argument and return value types. For each piece of information we also show how often a particular invocation occurred. Fig. 2 (1) shows a tooltip for the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

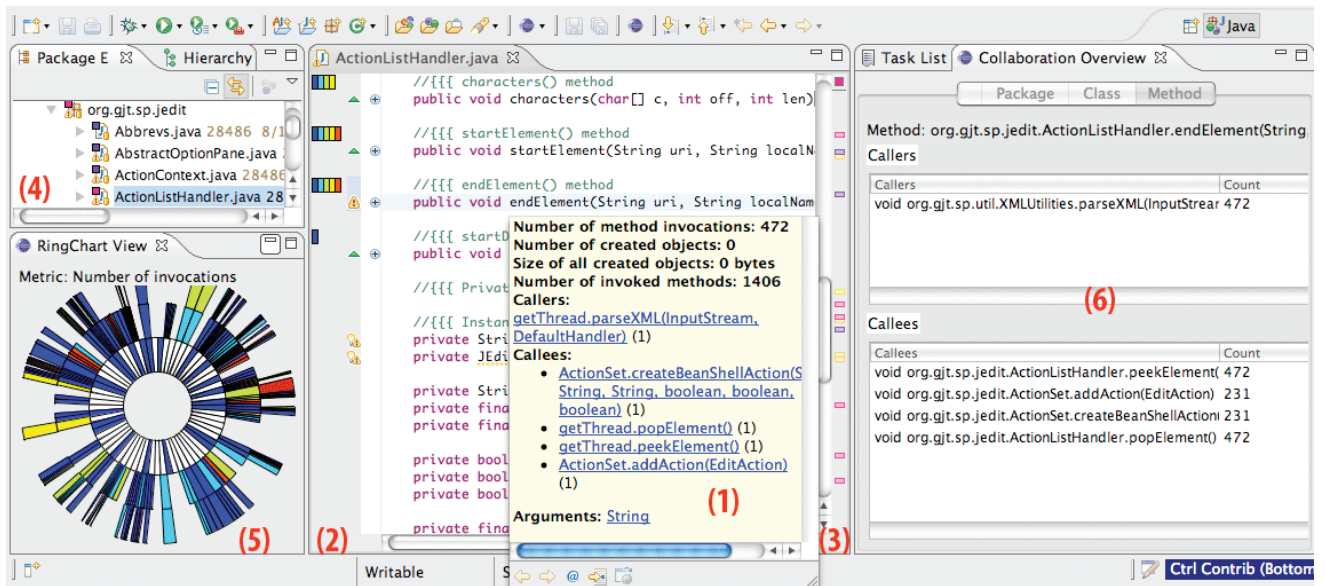JOURNAL OF TRANSACTIONS ON SOFTWARE ENGINEERING 4

Fig. 2. All six interactive views of *Senseo*.

concrete method `endElement`. If available, we also show information about argument and return types when the mouse is over the declared arguments of a method or the declared return type. These tooltips also display how often specific argument and return value types occurred at runtime.

Method body tooltip. Source elements in the method body also support tooltips. For each call site of the method, we provide the dynamic callee information as for the method name, namely concretely invoked methods, optionally along with argument or return types that occurred in this method for that particular method invocation at runtime.

Ruler columns. In Eclipse, the source code editor comes with two ruler columns: The left one shows local annotations (errors, warnings, *etc.*) while the right one presents an overview of all annotations from the entire document. We extended these two rulers to also display dynamic information. For every executed method in a Java source file, the overview ruler (Fig. 2 (3)) presents how often it has been executed on average per system run using three different icons colored using a heat scheme: *blue* means only a few, *yellow* several, and *red* many invocations [11]. Clicking on such an annotation icon triggers a jump to the declaration of the method in the file. The ruler on the left (Fig. 2 (2)) shows the frequency of invocation of a particular method on a scale from 1 to 6, compared to all other invoked methods. A completely filled bar for a method denotes methods that have been invoked the most in this application. These two rulers allow developers to quickly identify hot spots in their code.

Developers can choose between different kinds of dynamic information to be visualized in the rulers, such as the number of objects a method creates or the amount of memory it allocates, either on average or in total over all executions. Such metrics allow developers to quickly assess the runtime complexity of specific methods and thus to locate candidate methods for optimization.

Package Explorer. The package explorer is the main tool in Eclipse used to locate packages and classes of an application. *Senseo* augments the package explorer with dynamic information to guide the developer at a high level to the source artifacts of interest, see Fig. 2 (4). For this purpose, we annotate packages and classes in the explorer tree with icons denoting the degree to which they contribute to the selected dynamic metric such as amount of allocated memory. Thus a class aggregates the metric value of all its methods, a package the value of all its classes. Similar to the overview ruler the metric values are mapped to *blue*, *yellow*, and *red* package explorer icons representing a heat coloring scheme [11].

Collaboration View. In a separate view next to the source code editor (Fig. 2 (6)), *Senseo* presents all dynamic collaborators for the currently selected artifact. For instance, if a method has been selected, the collaboration view shows all packages or classes invoking methods of the package or class in which the selected method is declared (callers). The collaboration view also shows all packages or classes with which the package or class declaring the method is actively communicating (callees). For the method itself, the collaboration view lists all direct callers and callees.

Calling Context Ring Chart (CCRC). The *CCRC* [1] offers a compact visualization of a CCT and provides navigation mechanisms to locate and explore subtrees of interest for the software maintenance task at hand (Fig. 2 (5)). Like the Sunburst visualization [12], *CCRC* uses a circular layout. The CCT root is represented as a circle in the center. Callee methods are represented by

ring segments surrounding the caller's ring segment. For a detailed analysis of certain calling contexts, CCT subtrees can be visualized separately and the number of displayed tree layers can be limited.

4 COLLECTING DYNAMIC INFORMATION

IN this section we explain our approach to collecting dynamic information using AOP.

Senseo requires flexible support to aggregate dynamic information. For instance, runtime type information is needed separately for each pair of caller and callee methods, while memory allocation metrics need to be aggregated for the whole execution of a method (including direct and indirect callees). In order to support different ways of aggregating metrics, a data structure is needed to store dynamic information separately for each executed calling context. The CCT [2] perfectly fits this requirement.

Our CCT representation is designed for extensibility so that additional metrics can be easily integrated. Each CCT node stores dynamic information and refers to an identifier of the target method for which the metrics have been collected. It also links to the parent and child nodes for navigation in the CCT.

Our implementation leverages *MAJOR* [7], [8], an aspect weaver with two distinguishing features. First, *MAJOR* supports complete method coverage. Method invocations through reflection and callbacks from native code into bytecode are correctly handled. Second, *MAJOR* provides efficient access to complete calling context information through customizable, thread-local shadow stacks. Using the pseudo-variables `thisStack` and `thisSP`, the aspect gets access to the array holding the current thread's shadow stack, respectively to the array index (shadow stack pointer) corresponding to the currently executing method.

Fig. 3 illustrates three advices[3] of our aspect for CCT construction and dynamic information collection. In the `CCTAspect`, each thread generates a separate, thread-local CCT. The shadow stack is an array of `CCTNode` instances, representing nodes in the thread-local CCT. A special root node is stored at position zero. Periodically, after a configurable number of profiled method calls, each thread integrates its thread-local CCT into a shared CCT in a synchronized manner. This approach reduces contention on the shared CCT, yielding significant overhead reduction in comparison with an alternative solution where all threads directly update a shared CCT upon each method invocation.

The first advice in Fig. 3 intercepts method entries and pushes the `CCTNode` representing the invoked method onto the shadow stack. To this end, it gets the caller's `CCTNode` instance from

```
before(): execution(* *(..)) {
  CCTNode[] ss = thisStack; int sp = thisSP;
  ss[sp] = ss[sp-1].
          profileCall(thisJoinPointStaticPart);
  ss[sp].storeRcvArgsRuntimeTypes(thisJoinPoint);
}

after() returning(Object o): execution(* *(..)) {
  CCTNode[] ss = thisStack; int sp = thisSP;
  ss[sp].storeRetRuntimeType(o);
  ss[sp] = null;
}

after() returning(Object o): call(*.new(..)) {
  CCTNode[] ss = thisStack; int sp = thisSP;
  ss[sp].storeObjAlloc(o);
}
...
}
```

Fig. 3. Simplified excerpt of the CCTAspect

the shadow stack (i.e., at position `sp-1`) and invokes the `profileCall` method, which takes as argument an identifier of the callee method. We use static join points, accessed through AspectJ's `thisJoinPointStaticPart` pseudo-variable, to uniquely identify method entries; they provide information about the method signature, modifiers, *etc.* The `profileCall` method returns the callee's `CCTNode` instance and increments its invocation counter; if the same callee has not been invoked in the same calling context before, a new `CCTNode` instance is created as child of the caller's node.

The second advice in Fig. 3 deals with normal method completion, popping the method's entry from the shadow stack. For simplicity, here we do not show cleanup of the shadow stack in the case of a method completing abnormally by throwing an exception [8].

The third advice intercepts object creation to keep track of the number of created objects and the memory allocated for each calling context. The method `storeObjAlloc(Object)` uses the object size estimation functionality of the `java.lang.instrument` API to update the memory allocation statistics in the corresponding `CCTNode` instance.

`CCTAspect` collects the receiver, argument and result runtime types using dynamic join points.

During execution, the aspect code periodically sends the collected metrics to the *Senseo* plugin in the IDE. Upon metrics transmission, thread-local CCTs of terminated threads are first integrated into the shared CCT. Afterwards, the shared CCT is traversed to aggregate the metrics as required by *Senseo*. Finally, the aggregated metrics are sent to the plugin through a socket. Metrics aggregation and serialization may proceed in parallel with the program threads, since they operate on thread-local CCTs most of the time.

5 VALIDATION

WE conducted a controlled experiment with 30 professional Java developers to evaluate the

---

3. Aspects specify *pointcuts* to intercept selected *join points* in the execution of programs, such as method calls. *Advices* adapt join points with code to be executed before, after or around them.

benefits of *Senseo* [3] for software maintenance. We now describe the experimental design, the subjects, the evaluation procedure, the final results (including qualitative feedback), as well as threats to validity.

5.1 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 variables in this experiment: *time spent* and *correctness*. This experiment also reveals which kind of tasks benefit the most from the availability of dynamic information in the IDE. The experimental design we opted for is similar to the one applied in the study of Cornelissen *et al.* [13] which evaluated a trace visualization tool called *EXTRAVIS*.

Study Hypotheses. We claim that the availability of *Senseo* 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 *Senseo* available does not impact the time for solving the maintenance tasks.
- $H2_0$: Having *Senseo* available does not impact the correctness of the task solutions.

Congruently, we formulate these two alternative hypotheses:

- H1: Having *Senseo* available reduces the time for solving the maintenance tasks.
- H2: Having *Senseo* available 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 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 the *Senseo* plugin.

Study Participants. We asked 30 software developers working in industry (24) or with former industrial experience (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. All participants are familiar with Java and the Eclipse IDE.

The subjects have between one and 25 years of professional experience as software engineers (average 4.8 years, median 4 years). 27 subjects have a university degree in computer science while three subjects either studied in another area or learned software

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

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 [14] from 0 (no experience) to 4 (expert) subjects rated themselves on average 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".

To assign the 30 subjects to either the experimental or the control group, we used the obtained expertise information. 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.

Subject System and Tasks. As a subject system we have chosen *jEdit*[4], version 4.2, 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 of 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, and 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.* [15].

4. http://www.jedit.org/

TABLE 2
The five software maintenance tasks

Task	Activities	Description
1.1	A 1, 9	Locating in the code the feature "indent selected lines" and naming the packages and architectural layers in which it is implemented
1.2	A 1, 4 ,5	Describing package collaborations in this feature
2.1	A 8	Comparing fan-in, fan-out of three classes used in various features
2.2	A 4, 5, 6, 8	Describing coupling between the packages of these three classes
3.1	A 1, 3, 4, 5	Analyzing the order in which methods of a class used in the "folding" feature are invoked
3.2	A 1, 3, 5, 7	Locating clients of this class and analyzing the communication patterns between the class and its clients
4.1	A 4, 5, 8, 9	Comparing on a fine-grained method level the two features "shift indent left" and "remove trailing whitespaces" to locate a defect in the first feature
4.2	A 2	Correcting this defect by comparing it to the other, flawless feature
5.1	A 4, 5, 6, 7	Exploring an algorithm in a specific class used in the "spaces to tabs" feature and analyzing its performance
5.2	A 5, 6, 7, 8	Comparing this algorithm to another, similar algorithm in terms of efficiency

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, including the jEdit features covered, and explains which of Pacione's activities they cover. A more detailed description of the tasks can be found in Marcel Härry's master's thesis [5]. The five tasks are independent of each other; the order in which the tasks are solved does not matter. Task five is special since we use it as a "time sink task" to avoid ceiling effects [16]. 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, so the addition of a time-consuming task at the end which is not considered in the evaluation ensures that subjects have a constant time pressure for all relevant tasks. The first four tasks still 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. Beforehand, the experimenters solved all tasks individually to prepare an answer model according to which the subjects' answers were corrected. The answer model is the combination of the individual solutions of the experimenters.

Experimental Procedure. We gave the subjects a short five minute introduction to the experiment setup. Subjects from the experimental group additionally received a 20 minute introduction to *Senseo*, following a prepared script to ensure that every subject receives the same information. We provided the *Senseo* subjects with a short description and a screenshot highlighting and explaining the core features of *Senseo*, to serve 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 the *Senseo* plugin, otherwise the Eclipse IDE was configured in exactly the same way.

We provided the *Senseo* group with pre-recorded dynamic information obtained by executing all actions from the menu bar of jEdit to make sure that the pre-recorded information is not biased towards the experiment tasks. Recording these execution traces took around ten minutes; most time was spent manually triggering the menu items while the actual recording was efficient, that is, not much slower than without dynamic data recording activated. We provided pre-recorded dynamic information to control the variable of tracing the appropriate software features. Although it does not take much time to gather dynamic information with *Senseo*, freeing subjects from this task makes sure that the subjects' performance in the experiment is only dependent on how *Senseo* presents the information and not on which information has been recorded. As the control group did not receive any dynamic information, we clearly stated in the task descriptions how to run and analyze the feature under study with the conventional debugger in Eclipse.

We expect that executing appropriate scenarios of a software system to gather dynamic data is not difficult in practice. Usually, software maintenance tasks are expressed in terms of features that need to be changed or corrected, thus it is clear to developers which features to record. Often industrial systems comprise a set of unit or component tests exercising particular system features, thus execution scenarios are already encoded in these tests. If such tests are not available, developers can, as illustrated in the case of jEdit, in reasonable time manually exercise system features as its end-users would do.

Variables and Evaluation. The two dependent variables we study in this experiment are *time* the subjects spend to answer the questions, and *correctness* of the answers. 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

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

JOURNAL OF TRANSACTIONS ON SOFTWARE ENGINEERING   8

TABLE 3
Statistical evaluation of the experimental results

Group	Mean	Stdev.	K.-S.	Lev F	t	p
Time [m]:						
Eclipse	114.80	20.62	0.27			
Senseo	94.73 (-17.5%)	12.4	0.18	3.06	3.23	.0016
Correctness (points):						
Eclipse	11.33	2.58	0.31			
Senseo	15.13 (+33.5%)	2.10	0.24	0.22	4.42	.0001

question and the next. Correctness is measured using a score from 0 to 4 according to the overlap with the answer model, which forms a set of expected answer elements. For instance, for task 1.1, the correct answer lists packages `jedit.textarea`, `jedit.indent`, `jedit.buffer`, and `jedit`. For each correctly named package, the subject's answer receives one point. All answers were corrected by at least two experiments to ensure double-check the correction.

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 check for equality of variance in the sample.

5.2 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.

Only three subjects could not complete the time sink task (task 5) in the two hours we allotted, but everybody finished the four relevant tasks.

Time. 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 Fig. 4.

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).

From the observations of subjects during the experiment, from their informal feedback during the

debriefing interviews, and particularly from the formal questionnaires (see below), we could conclude that subjects using *Senseo* were more efficient due to the following reasons: (i) the availability of dynamic information in the source code tooltips helps developers to more quickly gain an understanding how source artifacts communicate with each other, (ii) the visualizations of dynamic information 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 collaborate with a selected source entity such as a package, class or method, developers can more quickly navigate to code relevant for a specific task. Due to space restrictions, we omit a discussion of observational and informal data we gathered during the experiment.
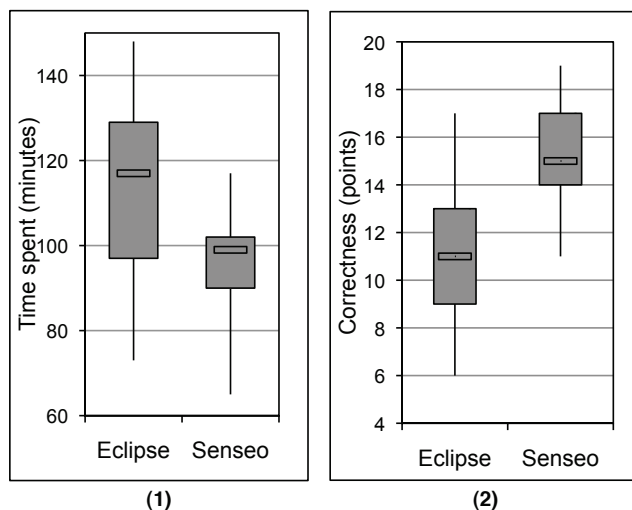

Fig. 4. Box plots comparing time spent and correctness between control and experimental group

Correctness. The *Senseo* group's answers for the four maintenance question are 33.5% more correct, which is also shown in the box plot in Fig. 4.

To test the null hypothesis $H2_0$, which suggests that there is no effect of the availability of *Senseo* on answer correctness, we can also 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 *Senseo* available during software maintenance activities helps developers to more correctly solve maintenance tasks.

The evaluation of the questionnaire, the observations during and the informal interviews after the experiment allowed us to attribute the improvements in correctness to the same techniques of *Senseo* that also improved the efficiency: (i) precise information about runtime collaboration or execution paths as

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%)

TABLE 5
Percentage of subjects using specific dynamic information in particular tasks

Dynamic Information	Task 1	Task 2	Task 3	Task 4
Runtime 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 runtime types	3.6
Ruler column incl. dynamic info	3.2
Overview ruler column incl. dyn. info	3.0
Package tree incl. dynamic info	2.4
CCRC	2.1
Collaboration view	3.7

highlighted in the extended source tooltips enables developers to accurately navigate to dependent artifacts, (ii) information about execution complexity (number of method calls or number and size of created objects shown in ruler columns or package tree) eases the correct identification of inefficient code, and (iii) accurate overviews of collaborating artifacts given by the collaboration view supports developers in exploring all relevant parts of the system to completely address a task.

Task-dependent Results. We also analyzed the two variables, time spent and correctness, for each task individually to reveal which kinds of tasks 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.

Qualitative Feedback. We also collected qualitative feedback using a questionnaire to evaluate the impact of particular parts of *Senseo* on specific kinds of maintenance tasks. This evaluation yields answers to the question which *Senseo* feature and which kind of dynamic information is actually relevant or useful in what kind of maintenance tasks.

In Table 5 we list for each task the percentage of subjects that used a specific kind of dynamic information integrated by *Senseo* ("Did you use dynamic information X in task Y?"), and Table 6 presents how useful subjects rated each *Senseo* technique on a Likert scale from 0 (useless) to 4 (very useful).

From the evaluation, we draw the conclusion that there are basically three kinds of tasks whose solution 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 depend on each other, (ii) tasks in which developers have to assess how often code is executed or how complex its execution 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 (Table 6), we conclude that developers particularly benefit from the availability of the collaboration views and runtime type information in source code. Also considered useful are visualizations of dynamic information in the source code columns such as the presentation of number of invoked methods in a method or class. The aggregated dynamic information presented in the package tree are perceived as less useful by the developers, probably because it is not meaningful to study runtime complexity at 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.

Subjects in both the control group and the *Senseo* group used various standard tools of Eclipse. Table 7 lists for each tool how many subjects used it during the experiment and how they rated on average its usefulness on a scale from 0 (useless) to 4 (very useful). The results show that *Senseo* subjects used the standard Eclipse less and also considered them to be less useful than subjects of the control group. Particularly the Eclipse debugger and inspector haven been used less as *Senseo* already embeds similar information in the source perspectives.

5.3 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,

TABLE 7
Users and usefulness rating for standard Eclipse tools.

Eclipse tool	Control group		*Senseo* group	
	Users	*Rating*	*Users*	*Rating*
Package explorer	14	2.4	10	2.1
Source code editor	15	2.8	13	2.7
Debugger	10	1.8	4	1.3
Inspector	6	1.5	0	–
Declaration view	7	2.1	8	2.1
Type hierarchy	14	1.9	8	2.1
Call hierarchy	12	1.7	4	2.5
Reference view	5	2.0	2	1.5
Search	15	2.7	11	1.8

(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 could have been more anxious as they were observed and assumed that their performance was being evaluated. However, we consider this threat to be negligible as we made clear that subjects' performance is not evaluated. Furthermore, this threat is likely to affect both the control and the experimental group, equally.

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 assessment of the subjects' expertise concerning their Java, Eclipse and 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 mitigated 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 [15] 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 [15] 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.

As the subjects 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 5.1 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.
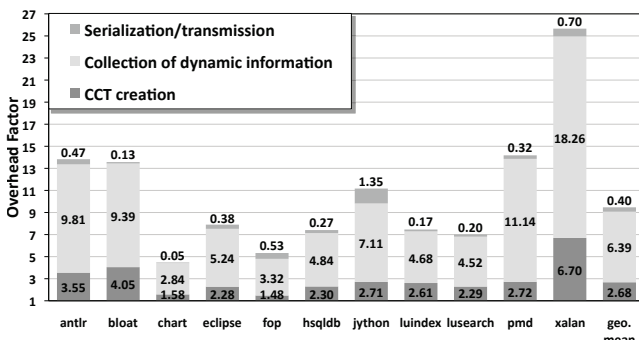
6 PERFORMANCE

IN order to validate that *Senseo* offers sufficient performance to cope with real-world workloads, we evaluated the different sources of overhead and analyzed the amount of transmitted data for the DaCapo benchmarks[5] [17]. 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).

Fig. 5 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, and the runtime 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

---

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

Fig. 5. *Senseo* overhead for the DaCapo benchmarks

Fig. 6. Size of transmitted data packets for "eclipse". Serialization/transmission rate: 1.25 packets per second

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. Fig. 5 also shows the average overhead (geometric mean) for the DaCapo suite.

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 runtime type information is particularly expensive. Serialization/transmission causes only minor overhead, because in these measurement settings serialization/transmission happens only once upon benchmark completion.

Senseo features an optimized serialization mechanism that transmits the CCT in an incremental way, sending only those nodes where some dynamic information has changed since the previous transmission. Thanks to the principle of locality, typically only a small subset of the CCT nodes is transmitted. Thus, it is possible to frequently update the dynamic information in the Eclipse plugin, such as once per second.

Fig. 6 illustrates the size of successively transmitted data packets for a single run of DaCapo's "eclipse" benchmark with a serialization/transmission rate of 1.25 packets per second.[8] 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 runtime of "eclipse" is about 296s in this setting (causing an overhead factor of 14.8, whereas a single serialization/transmission upon benchmark comple-

tion induces an overhead of factor 7.9 as shown in Fig. 5). For each packet, Fig. 6 differentiates between the size of the transmitted CCT nodes 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 Fig. 6 (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.

As *Senseo* can be used to gather dynamic information from all applications used in the DaCapo suite in reasonable time, we conclude that *Senseo* 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. Even though the overall overhead is high when gathering dynamic information, we do not consider this as a major issue, as the application does not need to run at productive speed while analyzing it.

7 RELATED WORK

I N this section we present related work in the context of dynamic information collection and development environments.

Dynamic Analysis for Program Comprehension. JFluid exploits dynamic bytecode instrumentation and code hotswapping to collect dynamic information [18]. JFluid uses a hard-coded, low-level instrumentation to collect gross time for a single code region

---

8. We chose the "eclipse" benchmark for this measurement, since it has the longest execution time in the DaCapo suite in our measurement environment.
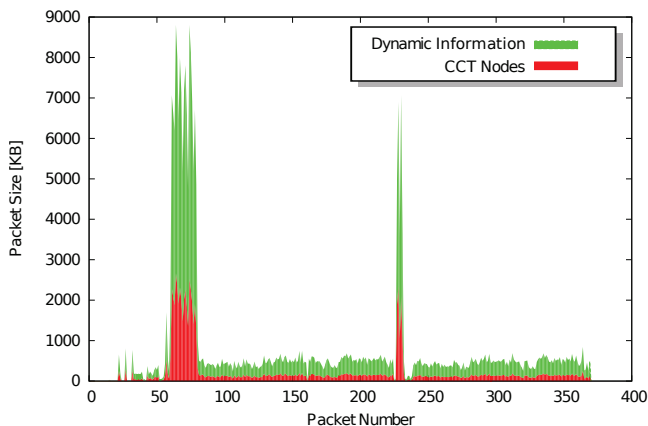
and to build a CCT augmented with accumulated execution time for individual methods. In contrast, we use a flexible, high-level, aspect-based approach to specify CCT construction and dynamic information collection, which eases customization and extension. Similar to *Senseo*, JFluid runs the application 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. The JFluid technology is integrated into the NetBeans Profiler [19]. For Eclipse, the Test & Performance Tools Platform (TPTP)[9] is available, which allows developers to build performance measuring and profiling tools.

Dufour *et al.* [20] present a variety of dynamic information for Java programs. They introduce a tool called *J [21] for metrics measurement. In contrast to our fully portable approach, *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.

Sampling-based profiling techniques, which are often used for feedback-directed optimizations in dynamic compilers [22], help significantly reduce the overhead of metrics collection. However, sampling produces incomplete and possibly inaccurate information, whereas *Senseo* requires complete and exact metrics for all executed methods. Hence, we rely on *MAJOR* to comprehensively weave our aspect into all methods in the system.

With static analysis, especially with static type inference [23], it is possible to gain insights into the types that variables assume at runtime. However, static type inference is computationally expensive and cannot always provide precise results in the context of object-oriented languages [24]. Furthermore, static analysis does not cover dynamically generated code, although dynamic bytecode generation is a common technique.

Dynamic analyses based on tracing mechanisms traditionally focus on capturing a method call tree, but existing approaches usually do not bridge the gap between dynamic behavior and the static structure of a program or present the analysis results in tools separated from the IDE [25], [26], [27]. Jinsight [28] for example is a visualization tool providing several views analyzing the running of Java programs to detect performance issues. Jinsight's support to gain an understanding for the program execution is limited and its views are separated from the IDE. Collaboration Browser [29] recovers objects collaboration from execution traces and identifies collaboration patterns. A pattern is displayed as a UML sequence diagram or in another high-level view in a tool separated from the IDE; this approach requires detailed knowledge about the system implementation to reduce the amount of

information displayed in the diagrams, which renders the approach less usable for unfamiliar systems.

Senseo differs from these related works by integrating the dynamic information in the IDE locally to specific static system artifacts instead of providing a general overview in a separated tool. Such a local integration particularly recognizes the conceptual relation between static and dynamic aspects of software systems. Ferret [30] follows a similar approach by integrating a query tool into Eclipse to allow developers executing conceptual queries about source artifacts directly in the IDE. An example of such a query is "callers of method x". Ferret focuses on querying static information, but is also able to take into account dynamic information to obtain more precise results. In contrary to *Senseo*, Ferret does not aim at giving an overview of the system or enriching the static IDE perspectives with dynamic information.

Integration of Dynamic Analysis in IDEs. Seesoft [31] is a software visualization system that eases software analysis by mapping each line of code to a colored row. The color indicates an interest metric using a heat map approach: red lines are for instance most recently changed lines and blue lines least recently changed. Seesoft explores different data sources, namely static or dynamic analysis (mainly profiling), but also version control information such as age or author(s) of source artifacts. The Seesoft approach can easily be embedded in an IDE. As Seesoft, however, focuses on single lines of code, it does not scale for large object-oriented software systems.

Ferret [30] recognizes the conceptual relation between static and dynamic aspects of software systems by integrating a query tool into Eclipse to allow developers to execute conceptual queries about source artifacts directly in the IDE. An example of such a query is "callers of method x". Ferret focuses on querying static information, but is also able to take into account dynamic and evolutionary information to obtain more precise results [30]. Ferret implements 36 conceptual queries of which five also consider dynamic information, such as which methods are actually invoked and what other methods these methods invoked at runtime. Ferret just gathers information about method invocations and does not integrate such information in the static source perspectives of IDEs like *Senseo*. Instead developers have to query for particular information.

Reiss [32] visualizes the dynamics of Java programs in real time, *e.g.*, the number of method invocations. Löwe *et al.* [33] follows a similar approach by merging information from static analysis with information from dynamic analysis to generate visualizations. The visualizations of these two approaches are not tightly integrated in an IDE though, but are provided by a separated tool. Thus, it is not directly possible to use these analyses while working with source code. We consider it as crucial to incorporate knowledge

---

9. http://www.eclipse.org/tptp

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.

JOURNAL OF TRANSACTIONS ON SOFTWARE ENGINEERING
13

about the dynamics of programs into the IDE to ease navigating within the source space.

Prior Empirical Evaluations. Other researchers also conducted controlled experiments to validate tools supporting software maintenance tasks. Bennet *et al.* [34] empirically evaluates sequence diagram tools with developers to reveal which features of these tools are useful in practice for software understanding. The study results are particularly useful to identify which features of sequence diagram tools could be integrate into IDEs to improve software maintenance.

Cornelissen *et al.* [13] evaluated a trace visualizing tool with 24 student subjects. They present the design of a controlled experiment for the quantitative evaluation of their tool *Extravis* for program comprehension. They report a 22% decrease in time and 43% increase in correctness of solving various typical software maintenance tasks.

Quante *et al.* [35] evaluated with 25 students 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 but describe the control flow of an application from the perspective of a single object. The involved students had to perform a series of feature location tasks in two systems. The use of DOPGs by the 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 and Quante *et al.* suggest to perform further evaluation with more than one system.

8 CONCLUSION

In this article we presented *Senseo*, an approach for gathering and integrating various kinds of dynamic information from running Java applications within the Eclipse IDE. The provided dynamic information includes callers, callees, runtime type information, method invocation counters, and object allocation metrics. *Senseo* integrates dynamic information in the package tree, the ruler columns, and in the source editor tooltips of the Eclipse IDE. In addition, *Senseo* offers a condensed and interactive visualization of the CCT and provides a navigable view on all dynamic collaborators of a source artifact (package, class, or method). The dynamic information is continuously updated in the IDE while an application is running.

An important issue of dynamic analysis is the selection of execution scenarios from which the dynamic data stems. The information integrated by *Senseo* into the IDE is as complete as the analyzed execution scenarios; the encoding of such scenarios, for instance with a set of test cases, is hence crucial to benefit from *Senseo* in practice.

A controlled experiment with 30 professional developers confirms that the dynamic information provided by *Senseo* significantly improves correctness and reduces the time needed for various software maintenance tasks. A performance evaluation shows that our approach is practical and able to visualize dynamic information in the IDE that is updated more than once per second.

REFERENCES

[1] 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: IEEE Computer Society, Sep. 2009, pp. 33–36.

[2] 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*. ACM Press, 1997, pp. 85–96.

[3] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret, "Augmenting static source views in IDEs with dynamic metrics," in *Proceedings of the 25th International Conference on Software Maintenance (ICSM 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 253–262. [Online]. Available: http://scg.unibe.ch/archive/papers/Roet09bDynamicInfoEclipse.pdf

[4] ——, "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: IEEE Computer Society, 2009, pp. 383–384, tool demo. [Online]. Available: http://scg.unibe.ch/archive/papers/Roet09cSenseo.pdf

[5] M. Haerry, "Augmenting eclipse with dynamic information," Master's Thesis, University of Bern, May 2010. [Online]. Available: http://scg.unibe.ch/archive/masters/Haer10a.pdf

[6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of European Conference on Object-Oriented Programming*, M. Akşit and S. Matsuoka, Eds. Berlin, Heidelberg, and New York: Springer-Verlag, 1997, vol. 1241, pp. 220–242.

[7] 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*. New York, NY, USA: ACM, Sep. 2008, pp. 159–167.

[8] A. Villazón, W. Binder, and P. Moret, "Flexible Calling Context Reification for Aspect-Oriented Programming," in *AOSD '09: Proceedings of the 8th International Conference on Aspect-oriented Software Development*. Charlottesville, Virginia, USA: ACM, Mar. 2009, pp. 63–74.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP-2001)*, ser. Lecture Notes in Computer Science, J. L. Knudsen, Ed., vol. 2072, 2001, pp. 327–353.

[10] W. Binder, J. Hulaas, and P. Moret, "Advanced Java Bytecode Instrumentation," in *PPPJ'07: Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*. New York, NY, USA: ACM Press, 2007, pp. 135–144.

[11] D. Röthlisberger, O. Nierstrasz, S. Ducasse, D. Pollet, and R. Robbes, "Supporting task-oriented navigation in IDEs with configurable HeatMaps," in *Proceedings of the 17th International Conference on Program Comprehension (ICPC 2009)*. Los Alamitos, CA, USA: IEEE Computer Society, 2009, pp. 253–257. [Online]. Available: http://scg.unibe.ch/archive/papers/Roet09aHeatMapsICPC2009.pdf

[12] J. Stasko, "An evaluation of space-filling information visualizations for depicting hierarchical structures," *Int. J. Hum.-Comput. Stud.*, vol. 53, no. 5, pp. 663–694, 2000.

[13] 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)*. IEEE Computer Society, 2009, pp. 100–109.

[14] R. Likert, "A technique for the measurement of attitudes," *Archives of Psychology*, vol. 22, no. 140, pp. 1–55, 1932.

[15] 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*. IEEE Computer Society, Nov. 2004, pp. 70–79.

[16] 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*, vol. 33, no. 2, pp. 65–86, 2007.

[17] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.

[18] 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*. ACM Press, 2004, pp. 139–150.

[19] NetBeans, "The NetBeans Profiler Project," Web pages at http://profiler.netbeans.org/.

[20] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic metrics for Java," *ACM SIGPLAN Notices*, vol. 38, no. 11, pp. 149–168, Nov. 2003.

[21] 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*. New York, NY, USA: ACM Press, 2003, pp. 306–307.

[22] M. Arnold and B. G. Ryder, "A framework for reducing the cost of instrumented code," in *SIGPLAN Conference on Programming Language Design and Implementation*, 2001, pp. 168–179.

[23] J. Pleviak and A. A. Chien, "Precise concrete type inference for object-oriented languages," in *Proceedings of OOPSLA '94*, 1994, pp. 324–340.

[24] P. Rapicault, M. Blay-Fornarino, S. Ducasse, and A.-M. Dery, "Dynamic type inference to support object-oriented reengineering in smalltalk," pp. 76–77, 1998, proceedings of the ECOOP '98 International Workshop Experiences in Object-Oriented Reengineering, abstract in Object-Oriented Technology (ECOOP '98 Workshop Reader forthcoming LNCS). [Online]. Available: http://scg.unibe.ch/archive/famoos/Rapi98a/type.pdf

[25] A. Hamou-Lhadj and T. Lethbridge, "A survey of trace exploration tools and techniques," in *Proceedings IBM Centers for Advanced Studies Conferences (CASON 2004)*. Indianapolis IN: IBM Press, 2004, pp. 42–55.

[26] 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)*. ACM Press, 2000, pp. 467–476.

[27] N. Wilde and R. Huitt, "Maintenance support for object-oriented programs," *IEEE Transactions on Software Engineering*, vol. SE-18, no. 12, pp. 1038–1044, Dec. 1992.

[28] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the behavior of object-oriented systems," in *Proceedings of International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'93)*, Oct. 1993, pp. 326–337.

[29] T. Richner and S. Ducasse, "Using dynamic information for the iterative recovery of collaborations and roles," in *Proceedings of 18th IEEE International Conference on Software Maintenance (ICSM'02)*. Los Alamitos CA: IEEE Computer Society, Oct. 2002, p. 34. [Online]. Available: http://scg.unibe.ch/archive/papers/Rich02aRolesExtractionICSM2002.pdf

[30] B. de Alwis and G. C. Murphy, "Answering conceptual queries with ferret," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*. New York, NY, USA: ACM, 2008, pp. 21–30.

[31] S. G. Eick, J. L. Steffen, and S. Eric E., Jr., "SeeSoft—a tool for visualizing line oriented software statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, Nov. 1992, depth.

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

[33] W. Löwe, A. Ludwig, and A. Schwind, "Understanding software – static and dynamic aspects," in *17th International Conference on Advanced Science and Technology*, 2001, pp. 52–57.

[34] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland, "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams," *Journal of Software Maintenance and Evolution*, vol. 20, no. 4, pp. 291–315, 2008.

[35] 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)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 73–82.

David Röthlisberger holds a MSc. and PhD. in Computer Science from the University of Bern, Switzerland. He is currently a postdoctoral student at the University of Bern. His research focus is on development environments, program analysis, in particular dynamic analysis, and information visualization.

Marcel Härry holds a BSc. degree in Computer Science from the University of Bern, Switzerland. He is currently a Master student at the University of Bern. Senseo is part of his master thesis with the title "Augmenting Eclipse with Dynamic Information", which is supervised by David Röthlisberger.

Walter Binder is senior assistant professor at the Faculty of Informatics, University of Lugano, Switzerland. He holds a MSc., a PhD., and a venia docendi from the Vienna University of Technology, Austria. Before joining the University of Lugano, he was senior researcher at the Artificial Intelligence Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland. His main research interests are in the area of dynamic program analysis, virtual execution environments, aspect-oriented programming, resource management, and service-oriented computing.

Philippe Moret holds a MSc. degree in Computer Science from the Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland. He is now a PhD. student at the University of Lugano, Switzerland. His research interests include dynamic program analysis, programming languages, and concurrency.

Danilo Ansaloni holds BSc. and MSc. degrees in Computer Science from the University of Modena and Reggio Emilia, Italy. He is now a PhD. student at the University of Lugano, Switzerland. His research interests include parallel computing, programming languages, and dynamic program analysis.

Alex Villazón is associate professor at the Universidad Privada Boliviana (UPB) in Cochabamba, Bolivia. He is also director of the *Centro de Investigaciones de Nuevas Tecnologías Informáticas* (CINTI) at the same University. He holds a MSc. and a PhD. from the University of Geneva, Switzerland. After his PhD., he joined the Distributed and Parallel Systems Group (DPS) at the University of Innsbruck, Austria, where he was lecturer and researcher. He also worked as senior researcher at the Faculty of Informatics, University of Lugano, Switzerland. His main research interests are in the area of aspect-oriented programming, virtual execution environments, distributed computing, grid and cloud computing.

Oscar Nierstrasz is full Professor of Computer Science at the Institute of Computer Science (IAM) of the University of Bern, where he founded the Software Composition Group in 1994. He holds a BMath from the University of Waterloo (1979), and a MSc (1981) and PhD (1984) in Computer Science from the University of Toronto. His current research focuses on various aspects of software evolution.